University of Neuchâtel

Partitioning of Unstructured Meshes for Parallel Processing

by

Noureddine Bouhmala

DISSERTATION

Submitted to the Faculty of Science of the University of Neuchâtel
in fulfillment of the requirements for the degree of "Docteur ès Sciences"

Computer Science Department
Rue Emile-Argand 11, CH-2000, Neuchâtel,
Switzerland.

1998

IMPRIMATUR POUR LA THÈSE

Partitionnement de maillages non structurés pour le calcul parallèle

de M. Noureddine Bouhmala

UNIVERSITÉ DE NEUCHÂTEL

FACULTÉ DES SCIENCES

La Faculté des sciences de l'Université de Neuchâtel sur le rapport des membres du jury,

MM. H.-H. Naegeli (directeur de thèse), K. Stoffel, A. Hertz (EPFL), C. Walshaw (Greenwich, UK) et B. Hendrickson (Sandia Lab., USA)

autorise l'impression de la présente thèse.

Neuchâtel, 18 juin 1998

Le doyen:

F. Stoeckli

F. Stoeckli

DEDICATION

To my parents

for everything

To my wife, Ellen,

for her love and care

To my sons, Zakharia and Mustapha,

for the joy they bring to me

My thanks to Professor H.H.Naegeli for his help in assisting me to improve the writing of this thesis.
I am deeply indebted to Professor A. Hertz, Professor K. Stoffel, Dr. B. Hendrickson, and Dr. C. Walshaw for their interest in my work and for their invaluable help and patience in reviewing this thesis. They suggested many improvements and clarifications that contributed greatly to this thesis.
I would like to express my special thanks to Dr. B. Hendrickson of Sandia Lab, Dr. C. Walshaw of the University of Greenwich, and Professor C. Farhat of the University of Colorado at Boulder who kindly provided the packages CHACO, JOSTLE, and TOP/DOMDEC.
I am also greatful to Dr. Kevin MacManus of the University of Greenwich, and to Michel Pahud of the Swiss Federal Institute of Technology for the nice collaboration in the experiments presented in the chapter 7 of this thesis.
I would like to express my special thanks to Pierre Alain Knutti. I enjoyed very much those days and nights that we worked together. His helpful discussions and friendship are highly appreciated.
I would like to express my appreciation to Yassine Faihe, Belhi Abdelkader, Massoud-Mohamat Alim, and Ait-Salah Said for their support and advice.
Finally, my hearty appreciation to my parents and my wife for their unconditional support which is beyond description.

Contents

1 **Introduction** 1

1.1 Motivation 1

1.2 The Need for Parallel Computers 2

1.3 Parallel Computers 3

1.3.1 Flynn's Taxonomy 3

1.3.2 Shared versus Distributed Memory 4

1.3.3 Interconnection Networks 5

1.3.4 Routing Mechanisms 7

1.4 Numerical Simulations and Parallel Platforms 7

1.5 Parallel Solvers 8

1.5.1 An Algorithmic Model 8

1.5.2 A Performance Model 10

1.5.3 Requirements for the partitioning 11

1.6 Static versus Dynamic Partitioning 13

1.7 Thesis Goal and Contributions 13

1.8 Thesis Organization 14

2 **Problem Formulation** 15

2.1 Basic definitions 15

2.2 Graph Representations of Meshes 18

2.3 Notations and Meanings 19

2.4 Problem Statement and Strategy 20

2.5 Partitioning Phase 22
2.5.1 Partitioning Problem Statement 22
2.5.2 Computational Time 22
2.5.3 Communication Time 23
2.5.4 Subdomain's Shape 24
2.5.5 The Objective Function 25
2.6 Mapping Phase 25
2.6.1 Mapping Problem Statement 25
2.6.2 The Objective Function 26

3 PREVIOUS WORK 27
3.1 Characteristics of Partitioning Algorithms 27
3.2 Geometry-Based Algorithms 28
3.2.1 Recursive Coordinate Bisection 28
3.2.2 Inertial Method 28
3.2.3 Scattered Partitioning 28
3.2.4 Recursive Graph Bisection 30
3.3 Topology-Based Algorithm 30
3.3.1 Recursive Spectral Bisection 30
3.3.2 Reverse Cuthill-McKee 31
3.3.3 Farhat's Algorithm 34
3.3.4 1D Topology Partitioning 36
3.4 Improvement Algorithms 36
3.4.1 The Neighborhood Concept 37
3.4.2 Kernighan-Lin 37
3.4.3 Tabu Search 38
3.4.4 Stochastic Evolution 39
3.5 Multilevel Algorithms 40

4 INITIAL PARTITIONING 42
4.1 A Simple Greedy Algorithm 43

4.2 Partitioning Quality Measures 44
4.3 Versions of the Simple Greedy Algorithm 46
4.4 Test Meshes 47
4.5 Experimental Results 48
4.5.1 Weight Imbalance 48
4.5.2 Edge Cut 48
4.5.3 Subdomain Graph 51
4.5.4 Shape Quality 53
4.5.5 Decomposition Time 55
4.5.6 Sensitivity to the Starting Vertex 57
4.6 Domain Splitting 59
4.6.1 Number of Disconnected Subdomains 59
4.6.2 Avoiding Domain Splitting 60
4.7 Comparison of Different Partitioning Algorithms 64
4.8 Concluding Remarks 66

5 IMPROVEMENT OF PARTITIONS 68
5.1 Iterative Improvement Algorithms 69
5.2 Simulated Annealing 69
5.2.1 Thermodynamics and Optimization Problems 69
5.2.2 The Algorithm 70
5.3 A Variant of Simulated Annealing 71
5.4 Implementation of the Method 73
5.4.1 Initial Temperature 73
5.4.2 Type of Perturbation 74
5.4.3 Improvement Phase 74
5.4.4 Number of Perturbations 75
5.4.5 Deterioration Phase 76
5.4.6 Cooling Rate 76
5.4.7 Stopping Criterion 77
5.5 Experimental Results 78

5.5.1 The Kernighan-Lin Improvement Method 78
5.5.2 Improvement of Partitions 80
5.6 Comparison of Heuristic Techniques 81
5.6.1 Weight Imbalance 82
5.6.2 Boundary Size and Subdomain Graph 83
5.6.3 Decomposition Time 84
5.7 Concluding Remarks 87

6 Graph Coarsening **88**
6.1 Strategy 89
6.2 Coarsening Schemes 90
6.2.1 Coarsening and Matching 90
6.2.2 Heavy Edge Matching 91
6.2.3 Gain Vertex Matching 91
6.2.4 Closest Vertex Matching 92
6.3 Experimental Results 93
6.3.1 Quality of Matching 93
6.3.2 Topology of the Coarse Graph 93
6.3.3 Quality of the Partitions 99
6.4 Influence of the Initial Decomposer 101
6.5 The Advantage of the Coarsening 103
6.6 Direct Coarsening vs Recursive Coarsening 105
6.7 Comparing Different Partitioning Algorithms 111
6.8 Concluding Remarks 116

7 Perspectives **117**
7.1 Need for Parallel Partitioning Algorithms 117
7.2 Parallelization of Partitioning Techniques 118
7.3 Parallel Simulated Annealing 120
7.3.1 Basic Idea 120
7.4 Experimental Results 122

7.5 A Demonstration Application 125
7.5.1 Test Case 125
7.5.2 Transtech Paramid Parallel Machine 125
7.5.3 Impact of the Partitioning and Mapping 126
7.6 Concluding Remarks 128

8 Conclusions and Further Work **129**

List of Figures

1.1 A finite element mesh. 2

1.2 Three different examples of interconnection networks: (a) a bus network; (b) a crossbar switch network; (c) a point-to-point network showing a 2 dimensional grid with no wrap-around connections. 6

1.3 Box-wise partitioning of a square domain into $p^2 = 16$ subdomains. 12

1.4 Stripe-wise partitioning of a square domain into $p^2 = 16$ subdomains. 12

2.1 Graph representation of a finite element mesh:(a) original mesh, (b) node-to-node graph, (c) true communication graph, and (d) dual graph. 19

2.2 Symbols and meanings 20

2.3 A two step paradigm for the mesh distribution problem. . . . 21

3.1 Example illustrating the scattered decomposition method. (a) partitioning the domain into 9 templates, (b) partitioning a template into 16 parts. 29

3.2 Illustrating the basic idea behind the Reverse Cuthill-McKee partitioning algorithm. 31

3.3 Numbering vertices in vertical direction in the Reverse Cuthill-McKee method. 32

3.4 Adjacency matrix of the graph according to the numbering in vertical direction in the Reverse Cuthill-McKee method. . . . 32
3.5 Numbering vertices in horizontal direction in the Reverse Cuthil-McKee method. 33
3.6 Adjacency matrix of the graph according to the numbering in horizontal direction in the Reverse Cuthill-McKee method. . 33

4.1 Metrics 45
4.2 Various graphs used in evaluating the quality of the partitioning. 47
4.3 Comparing different simple greedy partitioning algorithms in terms of edge cut using the graph HAMMOND. 49
4.4 Comparing different simple greedy partitioning algorithms in terms of edge cut using the graph BIG. 49
4.5 Comparing different simple greedy partitioning algorithms in terms of edge cut using the graph SNECMA. 50
4.6 Comparing different simple greedy partitioning algorithms in terms of edge cut using the graph T60K. 50
4.7 Comparing different simple greedy partitioning algorithms in terms of edge cut using the graph NACA. 51
4.8 Comparing different simple greedy partitioning algorithms in terms of the number of edges in the subdomain graph using the graph HAMMOND. 52
4.9 Comparing different simple greedy partitioning algorithms in terms of the number of edges in the subdomain graph using the graph BIG. 52
4.10 Comparing different simple greedy partitioning algorithms in terms of the number of edges in the subdomain graph using the graph SNECMA. 52
4.11 Comparing different simple greedy partitioning algorithms in terms of the number of edges in the subdomain graph using the graph T60K. 53

4.12 Comparing different simple greedy partitioning algorithms in terms of the number of edges in the subdomain graph using the graph NACA. 53

4.13 Comparing different simple greedy partitioning algorithms in terms of subdomain shape using the grapb HAMMOND. . . . 53

4.14 Comparing different simple greedy partitioning algorithms in terms of subdomain shape using the graph BIG. 54

4.15 Comparing different simple greedy partitioning algorithms in terms of subdomain shape using the graph NACA. 54

4.16 Comparing different simple greedy partitioning algorithms in terms of subdomain shape using the graph T60K. 54

4.17 Comparing different simple greedy partitioning algorithms in terms of subdomain shape using the graph SNECMA. 55

4.18 Comparing different simple greedy partitioning algorithms in terms of CPU time using the graph HAMMOND. 55

4.19 Comparing different simple greedy partitioning algorithms in terms of CPU time using the graph BIG. 56

4.20 Comparing different simple greedy partitioning algorithms in terms of CPU time using the graph NACA. 56

4.21 Comparing different simple greedy partitioning algorithms in terms of CPU time using the graph T60K. 56

4.22 Comparing different simple greedy partitioning algorithms in terms of CPU time using the graph SNECMA. 57

4.23 Sensitivity of different greedy algorithms to the starting vertex. 58

4.24 Number of disconnected subdomains generated by the different greedy partitioning algorithms using the graph HAMMOND. 59

4.25 Number of disconnected subdomains generated by the different greedy partitioning algorithms using the graph BIG. . . . 59

4.26 Number of disconnected subdomains generated by the different simple greedy partitioning algorithms using the graph SNECMA. 60

4.27 An example illustrating the domain splitting phenomenon. . . 61

4.28 Impact of the heuristic used to reduce the domain splitting phenomenon on the quality of the partitions using the graph HAMMOND. 63

4.29 Impact of the heuristic used to reduce the domain splitting phenomenon on the quality of the partitions using the graph BIG. 63

4.30 Impact of the heuristic used to reduce the domain splitting phenomenon on the quality of the partitions using the graph SNECMA. 63

4.31 Comparing different partitioning algorithms in terms of edge cut and CPU time using the graph HAMMOND. 65

4.32 Comparing different partitioning algorithms in terms of edge cut and CPU time using the grapb NACA. 65

4.33 Comparing different partitioning algorithms in terms of edge cut and CPU time using the graph T60K. 65

5.1 Analogy between physical systems and optimization problems 69

5.2 Evolution of the cost function for a typical run of the proposed variant of the Simulated Annealing method. 73

5.3 Sensitivity of the Kernighan-Lin improvement algorithm to the starting initial partition qualitywise and timewise using the graph HAMMOND. 78

5.4 Sensitivity of the Kernighan-Lin improvement algorithm to the starting initial partition qualitywise and timewise using the graph BIG. 79

5.5 Sensitivity of the Kernighan-Lin refinement algorithm to the starting initial partition qualitywise and timewise using the graph NACA. 79
5.6 Sensitivity of the Kernighan-Lin improvement algorithm to the starting initial partition qualitywise and timewise using the graph SNECMA. 79
5.7 Improvements made by our modified variant of the Simulated Annealing method using the graph NACA. 80
5.8 Improvements made by our modified variant of the Simulated Annealing using the graph WHEEL. 81
5.9 Improvements made by our modified variant of the Simulated Annealing method using the graph SNECMA. 81
5.10 Load balancing ratio achieved with our multilevel algorithm and TOP/DOMDEC using the graph WHITAKER. 82
5.11 Load balancing ratio achieved with our multilevel algorithm and TOP/DOMDEC using the graph BIG. 83
5.12 Load balancing ratio achieved with our multilevel algorithm and TOP/DOMDEC using the graph NACA. 83
5.13 Comparing our multilevel algorithm and TOP/DOMDEC in terms of boundary size and number of edges in subdomain graphs using the graph HAMMOND. 84
5.14 Comparing our multilevel algorithm and TOP/DOMDEC in terms of boundary size and number of edges in subdomain graphs using the graph BIG. 85
5.15 Comparing our multilevel algorithm and TOP/DOMDEC in terms of boundary size and number of edges in subdomain graphs using the graph NACA. 85
5.16 Comparing our multilevel algorithm and TOP/DOMDEC in terms of boundary size and number of edges in subdomain graphs using the graph WHITAKER. 85

5.17 Comparing our multilevel algorithm and TOP/DOMDEC in terms of CPU time using the graph WHITAKER. 86
5.18 Comparing our multilevel algorithm and TOP/DOMDEC in terms of CPU time using the graph NACA. 86
5.19 Comparing our multilevel algorithm and TOP/DOMDEC in terms of CPU time using the graph BIG. 86

6.1 A simple example illustrating the behavior of the heavy edge and gain vertex matching schemes. 92
6.2 Performance of various coarsening schemes in terms of matching ratio using the graph BIG. 94
6.3 Performance of various coarsening schemes in terms of matching ratio using the graph NACA. 94
6.4 Performance of various coarsening schemes in terms of matching ratio using the graph T60K. 95
6.5 Performance of various coarsening schemes in terms of matching ratio using the graph WHITAKER. 95
6.6 Number of vertices and edges of four coarse graphs produced by the heavy matching coarsening scheme. 96
6.7 Number of vertices and edges of four coarse graphs produced by the gain vertex matching scheme. 96
6.8 Number of vertices and edges of four coarse graphs produced by the closest vertex matching scheme. 97
6.9 Topology of four coarse graphs produced by the heavy edge matching method. 97
6.10 Topology of four coarse graphs produced by the gain vertex matching method. 97
6.11 Topology of four coarse graphs produced by the closest vertex matching method. 98
6.12 Quality of the partitions produced by the various matching schemes using the graph $T60K_{nd}$. 100

6.13 Quality of the partitions produced by the various matching schemes using the graph T60K. 100

6.14 Quality of the partitions produced by the various matching schemes using the graph $NACA_{nd}$. 100

6.15 Quality of the partitions produced by the various matching schemes using the graph NACA. 101

6.16 Impact of the initial decomposition on the overall performance of our multilevel algorithm using the graph BRACK. 102

6.17 Impact of the initial decomposition on the overall performance of our multilevel algorithm using the graph SNECMA. 102

6.18 Impact of the initial decomposition on the overall performance of our multilevel algorithm using the graph WHEEL. 103

6.19 Advantage of the coarsening on the edge cut using the graph BRACK. 104

6.20 Advantage of the coarsening on the CPU time using the graph BRACK. 104

6.21 Performance of direct coarsening and recursive coarsening using the graph BRACK: two and three level of contractions vs one level. 108

6.22 Performance of direct coarsening and recursive coarsening using the graph BRACK: four and five level of contractions vs one level. 108

6.23 Performance of direct coarsening and recursive coarsening using the graph SNECMA: two and three level of contractions vs one level. 108

6.24 Performance of direct coarsening and recursive coarsening using the graph SNECMA: four and five level of contractions vs one level. 108

6.25 Performance of direct coarsening and recursive coarsening using the graph T60K: two and three level contractions vs one level. 109

6.26 Performance of direct coarsening and recursive coarsening using the graph T60K: four and five level of contractions vs one level. 109

6.27 Performance of direct coarsening and recursive coarsening using the graph NACA: two and three levels of contractions vs one level. 109

6.28 Comparing the quality of edge cut produced by direct coarsening and recursive coarsening at the coarse graph using the graph SNECMA. 109

6.29 Comparing the quality of edge cut produced by direct coarsening and recursive coarsening at the coarse graph using the graph BRACK. 110

6.30 Performance of direct coarsening and recursive coarsening using the graph NACA: four and five level of contractions vs one level. 110

6.31 Performance of our multilevel partitioning algorithm against the inertial method combined with the Kernighan-Lin using the graph $T60K_{nd}$. 112

6.32 Performance of our multilevel partitioning algorithm against the inertial method combined with the Kernighan-Lin using the graph SNECMA. 112

6.33 Performance of our multilevel partitioning algorithm against the inertial method combined with the Kernighan-Lin using the graph BRACK. 113

6.34 Performance of various partitioning algorithms using the graph HAMMOND. 113

6.35 Performance of various partitioning algorithms using the graph WHITAKER. 114
6.36 Performance of various partitioning algorithms using the graph BIG. 114
6.37 Comparing a faster version of our multilevel algorithm with other multilevel techniques using the graph BRACK. 114
6.38 Comparing a slower version of our multilevel algorithm with other multilevel techniques using the graph BRACK. 114

7.1 Describing the parallel improvement approach for a pair of working processors. 123
7.2 Comparing the quality of the edge cut produced by the parallel greedy simulated annealing and its sequential counterpart . . . 123
7.3 Speedup Results 124
7.4 Cumulated number of idle processors. 125
7.5 Influence of the partitioning on the performance of the parallel solver. 126
7.6 Influence of the mapping on the performance of the parallel solver. 127

List of Algorithms

1	Pseudo-code of Farhat's algorithm.	35
2	Pseudo-code of the Simple Greedy algorithm.	43
3	Pseudo-code of the heuristic used to minimize the occurrence of domain splitting	62
4	Pseudo-code of the Simulated Annealing method.	71
5	Pseudo-code of the modified variant the Simulated Annealing method.	72
6	Pseudo-code of the Iterative Improvement algorithm.	75
7	Pseudo-code of the Deterioration procedure.	77
8	Pseudo-code of a typical recursive coarsening procedure.	90
9	Pseudo-code of the greedy direct coarsening algorithm.	107

Chapter 1

Introduction

1.1 Motivation

The behavior of many physical systems in real life are governed by a well-defined set of physical principles that can be translated into mathematical statements. These statements often take the form of partial differential equations (PDEs) in which the state of these systems plays the role of the unknown in the problem. Areas of application range from structural analysis and fluid mechanics to solid state physics and quantum mechanics. Often, it is the case that no analytical solution exists for the solution of the problem, and numerical methods are applied to provide an approximate solution. The solution of PDEs involves the resolution of a linear system of equations which arise from the application of appropriate spatial discretization schemes (e.g. *finite elements*, *finite differences*, *finite volumes*) to PDEs [And94] [Far93] [Ham92] [Hsi93] [Ven91] [Wal95b] . The spatial discretization process leads to what is called a finite element mesh or for short a mesh (Figure 1.1). Thus, a mesb describes the nature of a discretization. The complexity of a computational mesh ranges from the simple structured (i.e. the elements are regularly spaced) to fully unstructured (i.e. the elements are arranged in a completely arbitrary manner and may be of widely varying sizes). Typical element geometries are triangles and quadrilateral in two dimensions, and

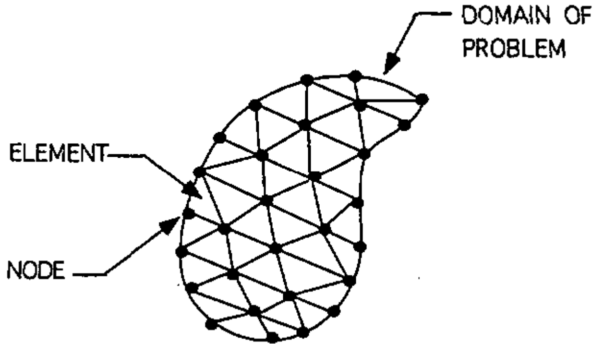


Figure 1.1: A finite element mesh.

tetrahedra, hexahedra, and triangular prisms in three dimensions. The unknown quantities are typically the values located at the nodes of the mesh. Our main motivation is the ability to run realistic numerical simulations on parallel computers. The considerations which motivate the use of such architectures are given in the next section.

1.2 The Need for Parallel Computers

The original need for fast computation have been in a number of applications involving PDEs. These applications demand a vast amount of computer resources in terms of memory requirements and computation time. Running these applications on a sequential machine may not be possible for two significant reasons:

- The problem is too large to fit into the memory of one sequential computer.
- The execution time of the application on a sequential computer may not be feasible.

A common property of problems involving PDEs, as they arise in practice, is that they can be decomposed along spatial dimension, and have therefore been prime candidates for parallelization, where each processor is assigned the task of manipulating a certain region in space. The greater the level parallelism that can be employed, the less time is required to achieve the solution. With this in mind, the advent of massively parallel distributed systems equipped with huge memories and enormous processing power holds the promise of achieving good performance and the capability of solving highly complex problems found in science and in engineering. On the other hand, for decades, we have been used to viewing algorithms in a sequential manner, and designing parallel algorithms for parallel computers is not an easy task. Several issues arise in designing parallel algorithms for parallel computers that do not occur with more traditional machines. One of these issues is how to decompose large computational meshes into parts to be assigned to the different processors. This partitioning problem is central to the effective use of parallel computers for many numerical problems.

1.3 Parallel Computers

There is a variety of models of parallel computers incorporating different assumptions, and the subject of this section is to go through some parameters that have been used to classify or describe parallel computers.

1.3.1 Flynn's Taxonomy

Traditionally, parallel computers are classified according to Flynn's taxonomy [Fly66]. Flynn's classification distinguishes parallel computers according to the number of instructions and data operands being computed on simultaneously. There are three main classifications of interest:

- Single-Instruction-Single-Data (SISD) computers. The SISD model represents the traditional sequential computer. A single program counter

fetches instructions from memory. The instructions are executed on *scalar* operands. There is no parallelism in this model.

- Single-Instruction-Multiple-Data (SIMD) computers. In this model, there is also a single program counter fetching instructions from memory. However, the operands of the instructions can be either scalars or arrays. If the instruction involves only scalar operands, it is executed by the control processor (i.e. the central processing unit fetching instructions from memory). If, on the other hand, the instruction uses array operands, it is broadcast to the processing elements (PEs). The PEs are separate computing devices. The PEs do not have their own program counter. Instead, they rely upon the control processor to determine the instructions they will execute. The parallelism in this model arises from having multiple PEs executing the same instruction, but on different operands.

- Multiple-Instruction-Multiple-data (MIMD) computers. In a MIMD computer, there exist multiple processors each of which has its own program counter. Processors execute independently of each other according to whatever instruction the program counter points to. SIMD and MIMD computers are further subdivided according to whether they all share a global memory or not. The distinction between these two types is illustrated in the next section.

1.3.2 Shared versus Distributed Memory

A significant aspect of parallel computers is the mechanism by which processors communicate. Generally speaking, there are two classes known as *shared memory*, and *distributed memory*:

- In a shared-memory MIMD computer, both the program's instructions and the program's data to be shared exist within a single shared memory that can be accessed by all processors. A processor can communi-

cate with another by writing into the global memory, and then having the second processor reads that same location in the memory. This solves interprocessor communication problem, but introduces the problem of simultaneous accessing of different locations of the memory by several processors, and the need of synchronization.

- The second class does not have a global shared memory, but rather each processor has its *own local* memory. Processors communicate through an interconnection network. Communication and synchronization are handled exclusively through the passing of messages over the interconnection network.

1.3.3 Interconnection Networks

The efficiency of any parallel computer is related to the efficiency of the corresponding interconnection network. This section will now describe some typical network topologies. There are three main types of network topologies:

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

These three types are depicted in Figure 1.2. In a bus network, processors share a single communication channel (the bus) which can only carry one message at a time. The performance of the bus network is limited by the *bus bandwidth* (i.e. the amount of data it can transmit in unit time). This type of network offers the advantage of being easy to extend because additional processors can be connected at any point along the bus. On the other hand, the bus network is a highly nonscalable architecture, because only one processor can send information on the bus at a time.
In point-to-point networks, connections are established between a processor

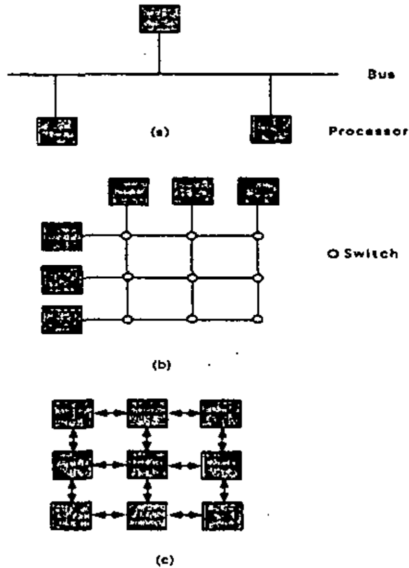


Figure 1.2: Three different examples of interconnection networks: (a) a bus network; (b) a crossbar switch network; (c) a point-to-point network showing a 2 dimensional grid with no wrap-around connections.

and its neighbors. Special routing hardware often takes care of communication destined to others processors (*cut through routing*), so that computation need not be halted in order to handle the message itself. Connections typically consist of two channels, one in each direction. The structure of the network can vary widely, from two and three dimensional arrays to rings or hypercubes.

The switch architecture interconnects processors with direct links that can be reconfigured (switched) dynamically. The most versatile type of switch network is a *crossbar* switch, where every processor can be connected to any other processor. The advantage of a switch architecture is that it avoids

competition for bandwidth. Its disadvantage is that it does not scale well. As the number of processors increases, the complexity and the cost of the switch increase.

1.3.4 Routing Mechanisms

There are two basic strategies for sending data in a network: *store-and-forward* routing and *wormhole* routing. In the store-and-forward routing a packet that must travel from a source processor to a destination processor over a route involving several processors, must have arrived before it is sent further. In wormhole routing, the head of the packet is advanced directly from the incoming channel to the outgoing channel before the whole packet has arrived. In both cases, the completion time is approximately proportional to the number of links it travels from the source processor to the destination processor.

1.4 Numerical Simulations and Parallel Platforms

Let the term *solver* denote the numerical algorithm used for solving a particular problem. When solvers are to be ported onto parallel platforms, the most common approach is to distribute different parts of the mesh over a number of processors. In this thesis, we assume a MIMD system in which processors run the same solver, repeating compute-communicate cycles. At the start, the data set corresponding to the mesh is partitioned into a number of data subsets which are distributed over the processors and the computation on a data subset is carried out by a processor. The partial solutions of all processors are combined over several iterations, to deliver a solution of the global problem within the desired accuracy. Processors communicate when non-local information is needed. This distribution results in communication operations between processors. Minimizing the amount of such operations to-

gether with making sure that each processor gets approximately equal share of work, are two important factors in ensuring efficient execution, as we shall see.

1.5 Parallel Solvers

1.5.1 An Algorithmic Model

To highlight the need for partitioning, we will examine the *conjugate gradient method* (CG)[Jen77] used to solve system of linear equations on a distributed memory parallel computer. The choice for this particular method is motivated by the fact that it appears in one way or another in many PDE solvers, and it is a typical model for the interaction of computation and communication on a distributed memory MIMD system. Let $\mathbf{K}\Phi = \mathbf{f}$ be the system to be solved. The matrix of coefficients $\mathbf{K}$ is commonly referred to as the "stiffness" matrix. The matrix $\mathbf{K}$ in a PDE problem is sparse (i.e. contains much more zero entries than non-zero entries). The interaction between nodes of the mesh is determined by the non-zero entries in the stiffness matrix. The vector Φ denotes the unknowns. From an initial vector Φ^0, CG generates a sequence of vectors $\Phi^0, \cdots \Phi^i$, each of which has a lower norm residual vector $\mathbf{r}^i = \mathbf{f} - \mathbf{K}\Phi^i$, than the last. By successively moving "downhill" and "orthogonally" to the former steps, the method converges quickly towards the desired solution. Before describing the various operations of CG that can be carried out in parallel, we begin by defining the quantities used in this algorithm. The stiffness matrix $\mathbf{K}$ and the vector $\mathbf{f}$ are assumed to have been already constructed, and will remain unchanged by the algorithm. The remaining quantities are three vectors and two scalars whose values are changed in each iteration. The three vectors are Φ^k, the approximation vector, $\mathbf{r}^k$, the residual vector, and $\mathbf{p}^k$, the descent direction. The CG algorithm starts by initializing these vectors: $\mathbf{r}^0 = \mathbf{p}^0 = \mathbf{f} - \mathbf{K}\Phi^0$, then, the following iterative loop labeled by the subscript $k \geq 0$ is performed until a chosen

convergence criterion is met:

- $\alpha_k = \mathbf{r}^k\mathbf{r}^k/\mathbf{p}^k\mathbf{K}\mathbf{p}^k$
- $\Phi^{k+1} = \Phi^k + \alpha_k \mathbf{p}^k$
- $\mathbf{r}^{k+1} = \mathbf{r}^k - \alpha_k \mathbf{K}\mathbf{p}^k$
- $\beta_k = \mathbf{r}^{k+1}\mathbf{r}^{k+1}/\mathbf{r}^k\mathbf{r}^k$
- $\mathbf{p}^{k+1} = \mathbf{r}^{k+1} + \beta_k \mathbf{p}^k$

The implementation of CG on a parallel computer requires the distribution of the components of the vectors $\mathbf{r}^k$, $\mathbf{p}^k$, and Φ^k over the processors. The iterative loop includes two basic operations: the scalar product (such as $\mathbf{r}^k\mathbf{r}^k$), and the matrix-vector product $\mathbf{Kp}^k$, which both can be performed in parallel.

For the scalar product, the processors concurrently perform a local inner product on the components they own. Thereafter, the processors sum their local contributions, which needs a global communication operation, at the end of which they all have the total.

For the matrix-vector multiplication $\mathbf{q} = \mathbf{Kp}$, a component q_i is computed as $\sum_j K_{ij}p_j$. One notices that:

- Most of the p_j needed to perform the required operation are located on the same processor as q_i.

- But some p_j are located on different processors given a certain distribution of the nodes over the processors. We call a node a *border node* if the corresponding p_j contributes to the computation of a component q_i not located on the same processor. For the matrix-vector multiplication operation $\mathbf{q} = \mathbf{Kp}$, the values attached to border nodes have thus to be sent from one processor to the other. Consequently, the

calculation of the Kp product results for each processor in a *neighbor-to-neighbor* communication operation followed by a *local matrix-vector* multiplication.

1.5.2 A Performance Model

We express the effort required by an iteration of the CG algorithm using a simplified performance model. This model will be useful for deducing the requirements guiding the development of a mesh partitioning algorithm. The local matrix-vector multiplication requires a time proportional to $n \times l$, where n is the number of components of a vector owned by the processor under consideration, and l is the average number of non-zero elements in a column of the sparse matrix K. The local scalar product requires a time proportional to n. For the neighbor-to-neighbor communication, we assume that its time can be expressed as $a + b \times m \times f(\Delta)$, where m denotes the size of the exchanged data, a is a constant usually called *latency* or *startup time* (i.e. the time needed to initiate the communication), b is a constant called the *reciprocal bandwidth*, and $f(\Delta)$ is a positive nondecreasing function of the physical distance Δ between the communicating processors. The function $f(\Delta)$ is highly system dependent: on many systems, on can assume $f(\Delta) = c \times \Delta$, for a constant c, whereas on other systems $f(\Delta)$ is almost constant. Finally, we assume that the time needed by an all-to-all communication operation is negligible as it involves only a scalar value. Its main effect on the execution of the CG algorithm is to impose a *barrier synchronization* (i.e. each processor has to wait for data from each other processor). This model is by no means exact. It does not take into account for instance, the fact that the communication operations launched by the different processors tend to take place at the same time leading to *contention* on the network. The contention phenomenon can heavily affect the predicted communication times.

1.5.3 Requirements for the partitioning

Based on the simple performance model described in the previous section, the requirements guiding a partitioning algorithm can be stated as follows:

- As each scalar product in the CG algorithm leads to a barrier synchronization, the number of components owned by the different processors should be as equal as possible; otherwise some processors might have to wait for other overloaded processors. This requirement can be refined by associating to each node a weight equal to the number of non-zero entries in the corresponding column of the matrix **K**.

- As the execution time of the neighbor-to-neighbor communication operations increases with the number of border nodes, their number has to be kept at minimum.

- As each neighbor-to-neighbor communication operation induces a startup time, the number of communicating neighbors should be as small as possible.

- On a system where $f(\Delta)$ increases with Δ, the average physical distance between communicating processors should be as small as possible.

It is to be emphasized that these requirements are conflicting with one another, so that a compromise will have to be found. Consider for instance a square regular 2D mesh with n^2 nodes and p^2 processors. The partition depicted in Figure 1.3 leads to a total border length of $2n(p-1)$, where each processor has at most 4 neighbors. On the other hand, the partition depicted in Figure 1.4 has a total border length of $n(p^2-1)$ but each processor has at most 2 neighbors. The box-wise partitioning is appropriate on systems where the transmission time overwhelms the startup time, whereas on systems where the startup time is the governing parameter, the stripe-wise partitioning is a better choice. Note, that even if one considers other iterative algorithms than CG, the aforementioned requirements remains valid in

most cases, at least qualitatively. One could factor the matrix, which leads to very different algorithmic questions. Partitioning is appropriate for either iterative solvers or explicit methods [Hen96e].

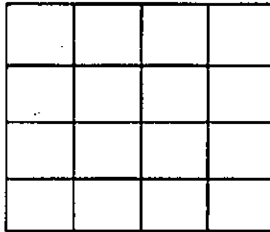


Figure 1.3: Box-wise partitioning of a square domain into $p^2 = 16$ subdomains.

Figure 1.4: Stripe-wise partitioning of a square domain into $p^2 = 16$ subdomains.

There is another class of solvers whose scalability is not governed by the interprocessor communication only [Far95a]. These algorithms are interesting on parallel machines when the number of iterations for convergence grows weakly with the number of subdomains. Their performance is determined by their convergence rate which has to do with the shape of the subdomains.

For these special solvers, a partitioning algorithm should avoid elongated and flat subdomains; thus box-wise partitioning is preferable to stripe-wise *partitioning*.

1.6 Static versus Dynamic Partitioning

In order to find a proper distribution of the mesh across the processors of a parallel machine, one has to resort to *mesh partitioning* techniques. If the computational intensity remains unchanged during the calculation throughout the mesh, the partitioning can be done only once and before the actual execution of the solver. These partitioning strategies are called *static*. Static techniques assume a priori knowledge of the characteristics of the mesh, the algorithm, and the target parallel computer. Partitioning techniques which are applied during the execution of the solver are in contrast called *dynamic*. Such techniques are used for instance, if the mesh undergoes some changes because it has to be refined or derefined in certain areas while the solver is running. The other classical situation where dynamic partitioning is required is a solver for which the computational complexity varies in time across different parts of the mesh, thereby leading to an imbalance of the processors load.

1.7 Thesis Goal and Contributions

The principal objective of this thesis is to develop static mesh partitioning techniques for the distribution of the mesh over the local memories of the processors of a distributed memory computer. The main contributions of this dissertation is the development of:

- New variants of the Greedy Algorithm proposed by Farhat [Far88].
- A simple heuristic for minimizing the occurrence of domain splitting.

- New mesh coarsening schemes.
- A variant of the simulated annealing method for improving mesh partitions.
- A parallel variant of the simulated annealing method for improving mesh partitions on a cluster of workstations..

1.8 Thesis Organization

This dissertation is organized as follows. Chapter 2 describes the problem formulation. Chapter 3 reviews the most popular partitioning algorithms together with three heuristics from the combinatorial optimization field used to improve the quality of a given decomposition. Chapter 4 presents new variants of the greedy partitioning algorithms proposed by Farhat and investigates their properties. In addition, a heuristic which minimizes the occurrence of disconnected subdomains is described. Chapter 5 introduces a variant of the Simulated Annealing method for optimizing mesh partitions. The quality of the partitions produced by this variant is compared against two existing versions of the Tabu Search and Stochastic Evolution methods included in the mesh partitioning package TOP/DOMDEC. In addition, different initial decompositions are used to investigate the performance of the Kernighan-Lin improvement algorithm. Chapter 6 deals with the mesh contraction problem, and provides a comparative experimental evaluation of our multilevel partitioning algorithm against three of the most popular existing methods. Chapter 7 presents a parallel variant of the Simulated Annealing method for improving mesh partitions on a cluster of workstations and explores some of its basic properties. In addition, a practical parallel numerical code is used to present the impact of the partitioning and the mapping on the performance of a parallel solver. Chapter 8 contains conclusions and suggestions for further research.

Chapter 2

Problem Formulation

As has been pointed out in the previous chapter, the implementation of parallel iterative solvers has motivated the need for partitioning. Given a finite element mesh, the mesh distribution problem consists in finding a distribution of the nodes or the elements of the mesh to the processors that leads to the lowest execution time of the parallel solver. The subject of this chapter is to formulate the mesh distribution as an optimization problem. This chapter starts by introducing to the reader some basic concepts and definitions from graph theory, and presents the relationship existing between meshes and graphs. Thereafter, a list of notations adopted throughout this thesis is presented. Finally, a two-step approach is proposed for the mesh distribution problem.

2.1 Basic definitions

A *mesh* is a pair (N, L), where N is a finite set of *nodes* and L is a finite collection of unordered sets of distinct nodes called *elements*. These nodes are points of the problem domain. Graph theory provides an efficient means of describing the topology of the mesh. Before associating a proper graph representation with the mesh, we briefly review some basic definitions.

Definition 2.1 A *graph* G is a pair $G = (V, E)$, where V is a finite set of *vertices* and E is a finite set of unordered pairs of distinct vertices called edges.

Definition 2.2 Two vertices v and w are said to be *neighbors* if there exists an edge denoted by $\langle v, w \rangle$ joining the two vertices. The edge is said to be *incident* upon vertices v and w.

Definition 2.3 A *subgraph* of a graph $G = (V, E)$ is a graph $H = (U, F)$ such that $U \subseteq V$, $F \subseteq E$, and $\langle v, w \rangle \in F$, if and only if $v,w \in U$.

Definition 2.4 Let $G = (V, E)$ be a graph, a *partition* of G into k subgraphs is a set $P_G = \{(V_1, E_1), ..., (V_k, E_k)\}$ of nonempty disjoints subgraphs of G such that $\bigcup_{i=1}^{k} V_i = V$, $V_i \cap V_j = \emptyset$, for $i \neq j$, and $\forall i : \langle v, w \rangle \in E_i$, if $(v, w \in V_i$ and $\langle v, w \rangle \in E)$

Definition 2.5 Let $H = (U, F)$ be a subgraph of a graph $G = (V, E)$, and $\langle v, w \rangle \in E$. A vertex v of the subgraph $H = (U, F)$ is called a *boundary vertex* if there exists an edge $\langle v, w \rangle$ such that $w \notin U$.

Definition 2.6 A *subdomain graph* of a partition $\mathcal{P}_G = \{S_1, \cdots, S_k\}$ of the graph $G = (V, E)$ is a graph $\mathcal{S} = (\mathcal{V}, \mathcal{E})$ *such that:*

- $\mathcal{V} = \{S_1, \cdots, S_k\}$

- $\langle S_i, S_j \rangle \in \mathcal{E}$ if and only if $\exists \langle v, w \rangle \in E$: $v \in V(S_i)$ and $w \in V(S_j)$, where $V(S_i)$ and $V(S_j)$ are the set of vertices of S_i and S_j. A subdomain graph is often referred to as a *quotient graph*.

Definition 2.7 The *degree* $\deg(v)$ of a vertex v is the number of edges incident upon v.

Definition 2.8 Let v and w be two vertices of a graph $G = (V, E)$. These two vertices are connected by a *path* if there exists a sequence $y_0, \cdots y_k$ of $k+1$ vertices where $y_0 = v$, $y_k = w$, $(y_n, y_{n+1}) \in E$, *and,* $0 \leq n \leq k$: $(y_n, y_{n+1}) \in E$. The *shortest path* between v and w is the path with the smallest sequence.

Definition 2.9 The *distance* between two vertices v and w denoted by $\text{dist}(v, w)$, is the number of edges of the shortest path.

Definition 2.10 The *diameter* of a graph is the maximum distance between any pair of vertices.

Definition 2.11 The vertices located at both ends of a diameter are called *peripheral vertices*.

Definition 2.12 A *matching* of a graph $G(V, E)$ is a subset of the edges with the property that no two edges share the same vertex. A *maximal matching* is one that can not be enlarged. That is, all edges are incident to at least one matched vertex.

Definition 2.13 A *weight* is a non-negative number attached to a vertex or to an edge.

Definition 2.14 A *weighted graph* is a graph in which every vertex and/or every edge are assigned a weight.

Definition 2.15 The *processor graph* $\mathsf{P} = (\mathsf{V}, \mathsf{E})$ is an abstraction of the topology of the parallel computer. Its vertex set V is the set of processors, and its edge set E is the set of the physical communication links joining two processors.

Definition 2.16 The *edge cut* of a partition is the set of edges whose vertices belong to different subgraphs of the partitions. In a slight abuse of notation, we use the same term for expressing the cardinality of this set or the sum of the weights of its elements.

2.2 Graph Representations of Meshes

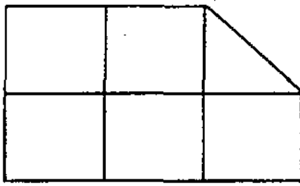
One approach to partition a mesh is to associate a graph with it and to decompose the corresponding graph; the following graphs have been widely used to describe the topology of a finite element mesh:

- The *node-to-node graph*. The vertices of this graph are the nodes of the mesh, and two vertices are connected by an edge if they belong to the same element.

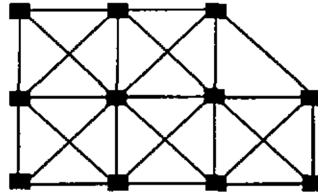
- The *dual graph*. The vertices of this graph are the elements of the mesh. For 3-dimensional, 2-dimensional, or 1-dimensional elements, two vertices are connected by an edge if the corresponding elements share a face, an edge, and a node respectively.

- The *true communication graph* (TCG). *Historically, practitioners used* the dual graph, which does not, however describe the communication pattern in the solver. Therefore Venkatakrishnan et al. [Ven91] proposed the use of the true communication graph, whose vertices are the elements of the mesh, and in which vertices are connected by an edge if the intersection of the corresponding elements is non-empty. Figure 2.1 shows a finite element mesh together with its associated node-to-node, dual, and true communication graphs.

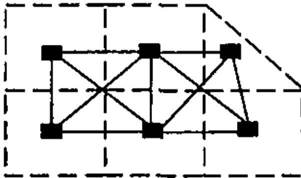
Which type of graph is appropriate for a given application is determined by the manner in which elements interact. If the problem unknowns are associated with the mesh nodes, and any two elements sharing a node must interact, then the true communication graph would be the appropriate choice. If however, the problem unknowns are fluxes (finite volume calculations) through the mesh element faces (three dimension) or edges (in two dimensions), then one would choose the dual graph. In practice, a mesh is represented as a *weighted graph*. The weights attached both to vertices and to



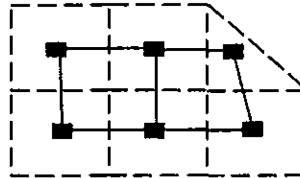
(a)



(b)



(c)



(d)

Figure 2.1: Graph representation of a finite element mesh:(a) original mesh, (b) node-to-node graph, (c) true communication graph, and (d) dual graph.

edges are used in order to take into account factors like computational and communication costs.

2.3 Notations and Meanings

We suppose that a weighted graph has been chosen to represent a finite element mesh. The notations adopted throughout this dissertation are listed in Figure 2.2.

Symbol	Meaning
$G(V(G), E(G))$	Graph with vertex set $V(G)$ and edge set $E(G)$
$C(G)$	Contracted graph of a graph G (section 6.1)
$\langle v_i, v_j \rangle$	Edge joining vertices v_i and v_j
$\|X\|$	Cardinality of a set X
$\overline{F}$	Average of function F taken over a given set
S_i	Subgraph or subdomain i
$\mathcal{S}$	Subdomain graph
N_{sub}	Number of subdomains
P	Processor graph
$\deg_G(v)$	Degree of a vertex v in graph G
$W(v)$	Weight of a vertex v
$W(e)$	Weight of an edge e
$W(V)$	Sum of the weights over the set of vertices V
$W(E)$	Sum of the weights over the set of edges E
$\mathrm{V_{bdry}}(S_i)$	Set of boundary vertices of S_i
$\mathrm{E_{bdry}}(S_i, S_j)$	Set of the edges crossing from S_i to S_j
$\mathrm{E_{bdry}}(S_i)$	Set of the edges crossing the border of S_i
$\mathrm{surf}(S_i)$	Surface of S_i (section 4.2)
$\mathrm{surfcirc}(S_i)$	Surface of circumscribed circle around S_i (section 4.2)
$\mathrm{dist}_G(v_i, v_j)$	Distance between vertex v_i and vertex v_j in graph G.
ISW	Ideal subdomain weight (section 4.1)
MAP	Mapping function (section 2.6.2)
$\mathrm{Adj}_{\mathsf{P}}(\mathsf{p}_i, \mathsf{p}_j)$	Adjacency function in graph P (section 2.6.2)
$\vec{v}$	Coordinates of a vertex v

Figure 2.2: Symbols and meanings

2.4 Problem Statement and Strategy

The *mesh distribution problem* may be stated as follows: given a *mesh* used in an iterative parallel solver for solving a given numerical problem, one seeks for the *distribution* of the nodes or the elements of the mesh over the processors, so that it provides the lowest execution time of the solver on the target distributed memory computer. Finding a mesh distribution that minimizes

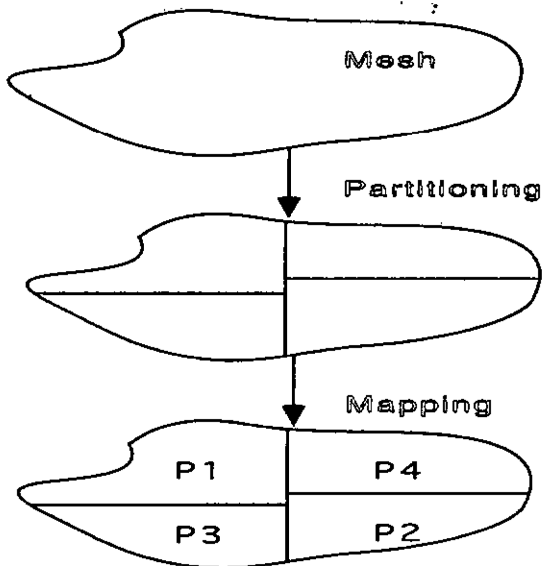


Figure 2.3: A two step paradigm for the mesh distribution problem.

the execution time of a solver on a parallel computer is an optimization problem. The strategy adopted in this thesis is to first find an architecture-independent *partition*, and then, in a second phase, find a proper *mapping* (allocation) of the subdomains onto the processors of the target parallel computer (Figure 2.3). Two schemes are popularly used. One is to use as many subdomains as there are processors, so that each processor is responsible for one subdomain. The other scheme is to have more subdomains than processors, so that each processor is responsible for one or more subdomains. This latter scheme is sometimes feasible, especially in an heterogeneous computing platform such as a network of workstations, in which the processors have different characteristics. In this thesis, we adopt the first scheme.

2.5 Partitioning Phase

2.5.1 Partitioning Problem Statement

Any search for a suitable partitioning has to be guided by some measure of merit. As we consider static partitioning, we use a simple model to approximate the total execution time of the parallel solver, and to estimate a priori the different factors of this model. We call *objective function* the corresponding artificial execution time, as it will be the quantity to be minimized. The purpose of this section is to present an objective function, taking into account the most important factors, which will be used in the partitioning phase.

2.5.2 Computational Time

An iterative solver that operates on a mesh typically involves the repeated application of the same computation at each node or element. We associate with each vertex of the graph describing the mesh, a weight expressing the contribution of the node or the element it represents to the total execution time. Thus, the ideal size of a subdomain is given by: $ISW = \lceil |W(V(G))|/N_{sub} \rceil$ or $ISW = \lfloor |W(V(G))|/N_{sub} \rfloor$. As one aims at an evenly distributed computational load for all processors, one tries to have $W(V(S_1)) = W(V(S_2)) = \cdots = W(V(S_{N_{sub}}))$. In other words, one is interested in minimizing the expression 2.1.

$$\text{WImbalance} = \sum_{k=1}^{N_{sub}} (W(V(S_k)))^2 \tag{2.1}$$

The expression 2.1 is one possible metric among others. The expression 2.2 is another candidate metric:

$$\text{WImbalance} = max_{j=1,\dots,N_{sub}} W(V(S_j)) \tag{2.2}$$

The expression 2.2 measures a fractional deviation from balance. The main motivation behind using the latter is the fact that an iteration of the solver on a parallel computer runs at the speed of the most heavily loaded processor. Thus, any processor with an exceptional work load will cause all other processors to wait, leading to poor performance. On the other hand, should any processor have little work to do, this will not hold up any other processors and have less effect on overall performance. In our implementation, we used the expression 2.1 because it is better suited for a metaheuristic like the Simulated Annealing method: many changes of partition have no effect at all on the expression 2.2.

2.5.3 Communication Time

Communication overhead occurs as a result of a decomposition. The subdomains which are not, in general, independent, must exchange information in order to cooperatively solve the problem. Exchanging information implies sending messages over the network. As has been presented in Section 1.5.3, the communication costs associated with a partition can be analyzed by focusing on two parameters:

1. The size of the edge cut.

2. The number of neighbors for an arbitrary subdomain.

The first parameter affects the volume of communication, whereas the second one determines the total startup time of an arbitrary subdomain. The following expressions are used to approximate the communication time:

$$\text{ECut} = \frac{1}{2} \times \sum_{k,l=1,k\neq l}^{N_{sub}} \left(\sum_{v_i \in V(S_k), v_j \in V(S_l)} W(\langle v_i, v_j \rangle) \right) \tag{2.3}$$

$$\text{DGR} = \frac{1}{2} \times \sum_{k=1}^{N_{sub}} (\text{deg}_{TCG}(S_k) \tag{2.4}$$

The expression 2.3 expresses the size of the total edge cut of the partition, whereas the expression 2.4 expresses the size of the total number of edges in the subdomain graph.

2.5.4 Subdomain's Shape

As has been pointed out in Section 1.5.3, there exist solvers whose effectiveness is determined by the shape of the subdomains [Far95a] [Van95b]. The best convergence rate is achieved for "round" subdomains. Farhat, et al. [Far95a] have observed that the problem of generating subdomains with good shapes is similar to the nearest centroid problem [Mac67]. In the latter problem, the goal is to construct a number of clusters of elements such that any element of a cluster is closer to the center of gravity of that cluster than to the center of gravity of any other cluster. Farhat, et al. [Far95a] have proposed the minimization of the following term, to take into account the effect of subdomain's shape, which we adopt as well:

$$\text{Shape} = \sum_{i=1}^{N_{sub}} \sum_{v \in V(S_i)} \left(\vec{v} - \vec{G}_i\right)^2 \tag{2.5}$$

where $\vec{G}_i = 1/|V(S_i)| \sum_{v \in V(S_i)} \vec{v}$ is the vector of the coordinates of the center of gravity of S_i. The metric used to define the subdomain shape is most appropriate for isotropic problems.

2.5.5 The Objective Function

Combining the expressions 2.1, 2.3, 2.4, and 2.5 yields the following objective function to minimize:

$$\mathrm{OF}_{\mathrm{partitioning}} = \alpha \times \mathrm{WImbalance} + \beta \times \mathrm{ECut} + \gamma \times \mathrm{DGR} + \delta \times \mathrm{Shape} \qquad (2.6)$$

The parameters α, β, γ, and δ are scaling factors expressing the relative importance of the various components of the cost function. As the objective function 2.6 includes different conflicting goals, its minimization is equivalent to finding the best possible compromise mesh partition. So far, there is no automatic strategy for determining these scaling factors. Therefore understanding the behavior of the parallel solver as well as the target parallel computer is necessary in order to take advantage of the parallel processing power.

2.6 Mapping Phase

2.6.1 Mapping Problem Statement

As we already pointed out, the goal in developing *partitioning* algorithms is normally to minimize the total execution time of the chosen solver on a target parallel computer. Most authors up to now used more or less the same strategy as ours to tackle the mesh distribution problem, namely to solve it in two steps which we called partitioning and mapping. Quite often, the second step was dealt with in a very naive way, without taking into account the topology of the parallel computer. It is taken into account, however, in the mesh distribution packages [Hen93a] [Wal95c]. Ignoring the topology of a computer with large number of processors can significantly degrade the performance of a parallel solver. The mapping problem consists in finding

the best possible assignment of the subdomains onto processors in order to minimize the communication overhead.

2.6.2 The Objective Function

The mapping problem may be stated as follows: Let $\mathcal{S} = (\mathcal{V}, \mathcal{E})$ denote the subdomain graph induced by some partition. $\mathcal{V}$ is the set of the subdomains, and each edge $(S_i, S_j) \in \mathcal{E}$ corresponds to at least one edge crossing form S_i to S_j. Let $\mathsf{P} = (\mathsf{V}, \mathsf{E})$ represents the processor graph representing the target parallel computer. The mapping problem consists in finding a one to one function $\mathrm{MAP} : \mathcal{V} \to \mathsf{V}$. Bokhari [Bok81] proposed the maximization of the "adequacy" function 2.7:

$$\mathrm{OF}_{\mathbf{Bokhari}} = \sum_{(S_i,S_j)\in\mathcal{E}} \mathrm{Adj}_{\mathsf{P}}\left(\mathrm{MAP}\left(S_i\right), \mathrm{MAP}\left(S_j\right)\right) \tag{2.7}$$

where $\mathrm{Adj}_{\mathsf{P}}(p_i, p_j) = 1$ if $(p_i, p_j) \in \mathsf{E}$ and $\mathrm{Adj}_{\mathsf{P}}(p_i, p_j) = 0$ otherwise. The cost function 2.7 counts the subdomains graph edges that are mapped onto neighboring vertices in the processor graph. However, subdomains graph edges mapped to paths of length greater than one are not accounted for in this function. The weakness of the function 2.7 is the fact that it expresses only very roughly the effect of the distance between two communicating processors, and that it does not include at all the effect of the communication volume. With this view in mind, we propose the following function as our measure for producing a better mapping:

$$\mathrm{OF}_{\mathbf{mapping}} = \sum_{(S_i,S_j)\in\mathcal{E}} W\left(\mathsf{E}_{\mathsf{bdry}}\left(S_i, S_j\right)\right) \times \mathsf{dist}_{\mathsf{P}}\left(\mathrm{MAP}\left(S_i\right), MAP\left(S_j\right)\right) \tag{2.8}$$

The cost functions 2.6 or 2.8 are by no means exact. For instance, neither of them includes the effects of network congestion or synchronization delays. Nevertheless, they still remain useful to us as a tool to guide the mesh distribution process.

Chapter 3

PREVIOUS WORK

The mesh partitioning problem is NP-hard [Gar79]. As a consequence, no algorithm is able to generate the optimal solution in polynomial time. However, there has been a great amount of research on developing automatic mesh partitioning algorithms for finding good suboptimal partitions. The survey of methods which is the subject of this chapter aims to be as complete as possible. We first present the general characteristics of partitioning algorithms, and then move on to the description of each algorithm separately.

3.1 Characteristics of Partitioning Algorithms

Partitioning algorithms can be divided into three classes. The first class contains methods that decompose the mesh from scratch. The algorithms belonging to this class can be further categorized as either *topology-based-algorithms* or *geometry-based-algorithms*. The first type exploits merely the adjacency information of the nodes or the elements to partition a mesh, whereas the second type uses the coordinates to generate such a partition. The second class of methods contains *improvement techniques*. These techniques work with an initial decomposition and attempt to improve it by changing the assignment of vertices. The third class contains the so-called *multilevel partitioning techniques*. These multilevel techniques reduce the

size of the graph describing the mesh, partition the smallest graph using an algorithm from the first class, and finally proceed with an improvement phase using a technique from the second class.

3.2 Geometry-Based Algorithms

3.2.1 Recursive Coordinate Bisection

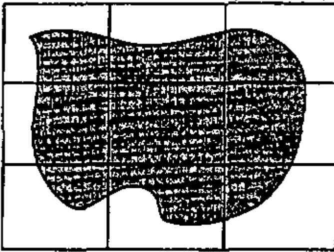
Recursive coordinate bisection is conceptually the easiest partitioning algorithm [Sim91] among all proposed in literature. The algorithm starts by determining the coordinate direction (x, y, or z) corresponding to the longest expansion of the domain, and sorts the vertices according to their coordinate in the selected direction to get a bisection. The same procedure is applied recursively for k steps until the desired number of subdomains is obtained. This technique has the advantage of being inexpensive. A disadvantage of this method is that it does not deliver partitions of good quality.

3.2.2 Inertial Method

The concept of this method [Nou86] is borrowed from mechanics. The mesh is considered as a solid in which the elements or the nodes are considered as points whose mass corresponds to their weights. The elements or nodes are projected onto the axis of the minimal moment of inertia, and the elements or nodes are sorted according to their projection and collected into two subdomains to form a bisection. This procedure is applied recursively to get the required number of subdomains.

3.2.3 Scattered Partitioning

This partitioning method [Fox86] [Mor86] is described using a 2-dimensional domain, and the extension of the procedure to the 3-dimensional case is analogous.



(a)

12	13	14	15
8	9	10	11
4	5	6	7
0	1	2	3

(b)

Figure 3.1: Example illustrating the scattered decomposition method. (a) partitioning the domain into 9 templates, (b) partitioning a template into 16 parts.

The method consists in the following steps:

- Embed the domain into a rectangle.
- Partition this rectangle into n rectangular parts, which we call templates.
- Partition each of these templates into r rectangular subparts, with r being the number of processors, and number these resulting subparts from 0 to $r-1$ adopting the same numbering scheme for all templates.
- All subparts with the label k are collected to form the subdomain k.

It follows that, by increasing the number n of generated parts, and consequently decreasing their areas, a better load balance is achieved. However, this is done at the expense of an increased subdomain scattering leading to a high communication overhead. Figure 3.1 illustrates the concept of the method.

3.2.4 Recursive Graph Bisection

This algorithm [Sim91] [Wil91] starts by calculating the diameter of the graph with two peripheral vertices. Finding the exact diameter of a graph is an expensive procedure, and in practice one usually uses an approximation algorithm [Geo81] to find a pseudo-diameter with two pseudo-peripheral vertices. This approximation algorithm works as follows: it calculates the distance from a given random starting vertex to every vertex in the graph. The vertex at largest distance defines the first pseudo-peripheral vertex. The second pseudo-peripheral vertex is located at the largest distance from the first one. Each of the two pseudo-peripheral vertices serves as a seed of the subdomains. Thereafter, every vertex is assigned to the subdomain whose seed is closer. This procedure is repeated recursively to get the desired number of subdomains.

3.3 Topology-Based Algorithm

3.3.1 Recursive Spectral Bisection

Recursive spectral bisection is the least intuitive partitioning algorithm [Sim91] [Pot90], but it is considered as one of the most effective partitioning methods. Its characteristic feature is the use of the eigenvector corresponding to the second largest eigenvalue of the Laplacian matrix associated with the graph. The Laplacian matrix of a graph G with n vertices is a n×n matrix $L = D - A$, where D is a diagonal matrix whose components $d_{ij} = \deg_G(v_i)$, and A is the adjacency matrix of G. In [Fie75], Fiedler gives theoretical justification that the second eigenvector, called the *Fiedler vector*, gives some directional information on the graph. The components of the eigenvector yield a weighting for the vertices. Sorting the vertices according to their weights provides a means to partition the graph. This technique generates partitions of high quality in terms of edge cut at the expense of large computational time due to the evaluation of the Fiedler vector. To decrease the computational cost of

this technique, a faster implementation is adopted [Bar94]. In this implementation, the graph is contracted using a contraction procedure. The coarsest graph is partitioned using recursive spectral bisection. At every contraction level, the eigenvector obtained for the coarser graph is used to build a starting vector for the computation of the finer graph. Hendrickson and Leland [Hen93c][Hen95d] proposed a new variant of the recursive spectral bisection. Their approach allows at each recursive stage for the decomposition into four or eight subdomains by calculating multiple eigenvectors at the same time. This variant generates partitions of better quality at a lower cost than its original counterpart.

3.3.2 Reverse Cuthill-McKee

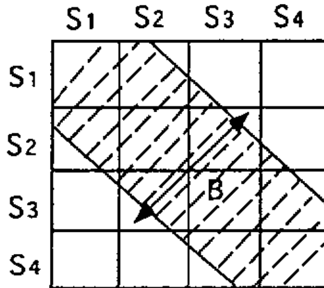


Figure 3.2: Illustrating the basic idea behind the Reverse Cuthill-McKee partitioning algorithm.

The basic idea behind this algorithm [Mal88] is to minimize the *bandwidth* B of the sparse adjacency matrix (Figure 3.2).

In the example depicted in Figure 3.2, a reduction of the bandwidth could avoid communication between S_1 and S_3 as well as between S_2 and S_4. With the intention of doing so, one looks for a numbering of the vertices of the graph and to partition them according to this numbering. This method is

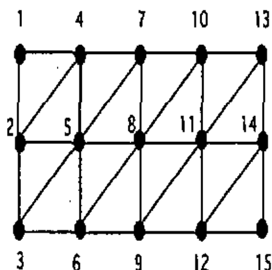


Figure 3.3: Numbering vertices in vertical direction in the Reverse Cuthill-McKee method.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1		1											
2	1	1	1	1	1										
3		1	1		1	1									
4	1	1		1	1		1								
5		1	1	1	1	1	1	1							
6			1		1	1		1	1						
7				1	1		1	1		1					
8					1	1	1	1	1	1	1				
9						1		1	1		1	1			
10							1	1		1	1		1		
11								1	1	1	1	1	1	1	
12									1		1	1		1	1
13										1	1		1	1	
14											1	1	1	1	1
15												1		1	1

Figure 3.4: Adjacency matrix of the graph according to the numbering in vertical direction in the Reverse Cuthill-McKee method.

better illustrated by an example. Let Figures 3.3, and 3.5 show two numberings of the same graph, whereas Figures 3.4, and 3.6 show the communication

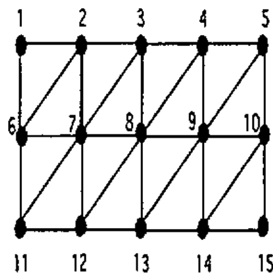


Figure 3.5: Numbering vertices in horizontal direction in the Reverse Cuthil-McKee method.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1				1									
2	1	1	1			1	1								
3		1	1	1			1	1							
4			1	1	1			1	1						
5				1	1				1	1					
6	1	1				1	1				1				
7		1	1			1	1	1			1	1			
8			1	1			1	1	1			1	1		
9				1	1			1	1	1			1	1	
10					1				1	1				1	1
11						1	1				1	1			
12							1	1			1	1	1		
13								1	1			1	1	1	
14									1	1			1	1	1
15										1				1	1

Figure 3.6: Adjacency matrix of the graph according to the numbering in horizontal direction in the Reverse Cuthill-McKee method.

pattern between the subdomains according to these respectively two numberings. Let us assume that the graph is to be decomposed into 4 subdomains. In this case, the last subdomain will have 3 vertices, whereas the others will be having 4 each. If the assignment is done according to the numbering in vertical direction, subdomain S_2, for instance, will have to communicate with subdomains S_1, S_3, and S_4. On the other hand, doing the partitioning according to the numbering in horizontal direction, subdomain S_2 needs to communicate with subdomains S_1 and S_3, but not with S_4. An advantage of this method is that each subdomain has few neighbors which is a desirable property on parallel machines with high startup. On the other hand, the method tends to deliver partitions with quite large edge cut due to the elongated shape of many of the subdomains. It is common to apply the method recursively.

3.3.3 Farhat's Algorithm

Farhat's algorithm [Far88] is a popular greedy partitioning algorithm which works directly on the mesh and provides a partitioning of the set of elements. It enjoys a relatively large user community in the field, because of its good performance/price ratio. The algorithm expands out in layers from a starting node (seed node). A weight is attached to each node in the mesh. This weight is equal to the number of yet unassigned elements incident to this node. This weight is used for selecting the starting node of a subdomain. The starting node of a subdomain is one with minimal non-zero weight among those on the boundary of the previously built subdomain. This node tends to be located in a corner of the yet unassigned part of the mesh. As the algorithm proceeds in a breadth-first fashion, it thus ensures that "remote" parts of the mesh are taken as much as possible in an early stage, thereby reducing the scattering of the subdomains. As the algorithm progresses, a subdomain expands around its initial starting node until it has reached the required number of elements. Once all the elements incident to a node have been assigned to a

Algorithm 1 Pseudo-code of Farhat's algorithm.

```
Procedure Farhat (input M: mesh, output P_M: part);
  for n := 1 to NumberOfNodes do
    weight[n] := deg_M(n);
  end for
  for sub := 1 to N_sub do
    L :={}; Q:= empty queue;
    if sub = 1 then
      choose a boundary node n_i in the mesh with minimal weight;
    else
      choose a boundary node n_i ∈ S_sub-1 with minimal non-zero weight;
    end if
    put all unmasked elements connected to n_i into Q;
    repeat
      if Q is empty then
        choose a boundary node n_i ∈ S_sub-1 with minimal non-zero weight;
        put all unmasked elements connected to n_i into Q;
      end if
      get the first element e_j from Q;
      mask element e_j;
      reduce by one the weight of all nodes attached to e_j;
      put all unmasked neighbors of e_j into Q;
      L := L ∪ {e_j};
    until |L| ≥ ISW
    S_sub:= subdomain whose elements are in L;
    P_M := P_M ∪ {S_sub};
  end for
```

subdomain, the weight of this node becomes zero. Thus, a node lying on the boundary between a subdomain and the set of unassigned elements will have its weight different from zero. Masking the elements during the partitioning process ensures that elements are not reconsidered. Disadvantage of this this algorithm is that it requires large memory because it needs three different relations (the element-to-element adjacency relation, the node-to-element incidence relation, and the element-to-node incidence relation). Experience

has shown that in the case of irregular meshes, this algorithm tends to generate partitions whose last subdomains may be disconnected (i.e. a subdomain consisting of more than one connectivity component). The pseudo-code is given in Algorithm 1.

3.3.4 1D Topology Partitioning

This algorithm has been designed to produce topologically one-dimensional partitioning [Van95a]. The construction of subdomains is done iteratively. The algorithm starts the construction of the first subdomains at a vertex of the graph with minimal degree. Thereafter, vertices are added until the required size is reached. The other subdomains are built as follows: First, the algorithm gets the unassigned vertices that are neighbors to the previous subdomain, thereby enforcing in most cases the one-dimensional property of the partition. Second, neighbors of vertices of the subdomain under construction are added to the subdomain. The strength of this algorithm is that, in most cases, every subdomain has at most two neighboring subdomains which reduces the number of communications.

3.4 Improvement Algorithms

Several methods from the combinatorial optimization field have been used in order to improve the quality of the initial partition provided by any chosen partitioning algorithm from the previous section. Improvement methods such as Tabu Search, Stochastic Evolution, Simulated Annealing, and Kernighan-Lin are very popular. The first three methods are non-deterministic, (i.e. the partitions generated by these methods may vary from one run to another) whereas the last one is deterministic. This section describes these techniques with the exception of Simulated Annealing method which will be dealt with in Chapter 5.

3.4.1 The Neighborhood Concept

We call an *improvement algorithm* an algorithm that works on an existing partition provided by some partitioning algorithm in order to improve its quality. Improving a given partition can be viewed as an iterative procedure performing a number of moves. In most algorithms, a move consists either in the transfer of a boundary vertex from its subdomain to a neighboring one, or in the exchange of pairs of boundary vertices belonging to different subdomains. Two partitions are called *neighbors* if a move leads from one to the other. The *neighborhood* of a partition is the set of its neighbors.

3.4.2 Kernighan-Lin

The Kernighan-Lin (KL) algorithm [Ker70] is essentially a local search algorithm. KL was designed to reduce the edge cut of a *bisection*. The KL algorithm seeks to find a better partition by swapping vertices in an attempt to reduce the edge cut. The algorithm selects the vertices to swap by associating a *gain* value with each vertex and interchanging those yielding the maximum edge cut reduction or the minimum edge cut increase. As the algorithm may accept an exchange with potentially higher edge cut that the previous one, it is said to have a *hill-climbing ability*; the algorithm thus tries to avoid local optima. The main distinction between KL and other improvement techniques is that the former replaces the search for one favorable swap by a search for a favorable sequence of swaps.
The KL algorithm consists of two nested loops. We call a *sweep* a single iteration of the outer loop. In the inner loop, a subset of vertices such that swapping them leads to a partition with a smaller edge cut is identified. Each time a pair of vertices is identified, it can no longer be considered for further exchange within the inner loop. Once a pair of vertices are swapped, the gains of their adjacent vertices is updated. At the end of the inner loop, a sequence of swaps $s_i, \cdots, s_n$ and their corresponding gains $g_i, \cdots, g_n$ is generated. Thereafter, KL finds the index k of the total maximum gain :

$Gain(k) = \sum_1^k g_i$. The outer loop applies successive sweeps, each time using the best partition found on the previous sweep as the starting one for the next sweep. The KL algorithm terminates when no favorable sequence of swaps within the inner loop is determined. Various extensions and improvements of the KL algorithm have been proposed through the years [Fid82] [Hen93a]. The former variant was a very significant advance. Instead of swapping vertices, they move them one at a time. By doing so, they can then make a whole sweep run in linear time.

3.4.3 Tabu Search

The Tabu Search technique was proposed by Glover [Glo85] and further studied in [Her87]. It is a local search strategy for tackling combinatorial optimization problems. The main feature of this algorithm is the ability to avoid returning in a previous state by keeping a trace of the optimization history. A brief description of the method is as follows. First, an initial solution of the problem is constructed. Then, at each iteration, given the current solution, one examines its corresponding neighborhood and chooses to move to the solution that most improves the objective function. To avoid getting stuck in local minima, historical information from the k last iterations is used. The value k may be fixed or a variable that depends on the state of the search, or a particular problem. The set of moves determined by this information forms a *tabu* list. Hence, the method has a short term memory remembering which trajectories have been recently explored. To prevent the method from cycling between the same solutions, one forbids the reverse of any move contained in the tabu list. Often, it can be favorable to override the tabu restrictions. For example, if a tabu move results in a better cost function value, it may be preferable to make the move anyway. Such an overriding of the tabu restriction is called an *aspiration* criterion. When the tabu restrictions are overridden for a hetter value, it is the so called best aspiration criterion that is used. For a more detailed description see [Glo93].

One problem encountered in implementing this method, is the size of the tabu list. If the size of the tabu list is too small, the probability of cycling increases. If it is too large, then there is a chance that all moves from the current solution are tabu, and the algorithm may be trapped.

3.4.4 Stochastic Evolution

The Stochastic Evolution method (SE) [Saa91] is a variant of the Simulated Annealing technique (SA). The similarities between SA and SE are as follows:

- they are stochastic in nature,
- downhill moves are always accepted,
- they have a control parameter that governs the probability of accepting an uphill move.

The key differences between the two techniques are as follows:

- In the SA algorithm, randomness is introduced at two stages, namely move generation, and move acceptance. However, in the SE algorithm, randomness is introduced only during the acceptance of a move. All moves which are possible in a given state are considered. This can be considered as an advantage compared to the Simulated Annealing technique, where most possible moves are just ignored. Since SE examines all possible moves, it is more costly per move.
- In the SA algorithm, the probability of accepting a bad move is monotonically decreasing. In the SE method, however, this probability remains low as long as good moves are possible. As soon as the algorithm gets stuck into a local minimum, this probability is increased.

3.5 Multilevel Algorithms

Multilevel partitioning algorithms have proved to be very efficient especially when the size of the graph is large. The basic idea is quite simple. A sequence of smaller graphs are built from the original graph using a *contraction* procedure. We will also use the term *coarsening* as a synonym of contraction. Thereafter, a partition of the smallest graph is computed using any of the partitioning algorithms described previously. Finally, the partition produced for the smallest graph is projected back towards the original graph, by proceeding with an *improvement phase* at each step. The following multilevel algorithms are considered among the most sophisticated partitioning algorithms in the field:

- CHACO: This is the first efficient multilevel algorithm for the graph partitioning problem [Hen93a]. It uses a *random neighbor contraction* procedure to reduce the size of the graph. This technique contracts the graph by merging an unmatched vertex with a randomly unmatched chosen neighbor. At the coarsest level, spectral bisection is used. During the improvement phase, the KL algorithm is invoked on each level to improve the quality of the partition. In addition, CHACO includes a number of fast partitioning algorithms, and offers the possibility of improving partitions produced by other methods.

- JOSTLE: This graph partitioning package [Wal95a] uses a random neighbor coarsening procedure to reduce the size of the graph. At the coarsest level, a variant of Farhat's algorithm [Far88] is used to deliver an initial partition. A combination of the KL algorithm and a heuristic for improving the shape of the subdomains is used during the improvement phase. The latest version of this package [Wal97] uses a contraction procedure based on the heaviest edge criterion [Kar95a]: this contraction procedure works by merging a vertex v_i and a vertex

v_j provided that the weight of the edge $\langle v_i, v_j \rangle$ is maximum among all valid edges incident to v_i.

- METIS: This graph partitioning package [Kar95a] uses a coarsening scheme based on the heaviest edge criterion, a number of greedy partitioning algorithms and the recursive spectral bisection to produce an initial partition. During the improvement phase, KL's gain criteria is used to move down and stop at the first local minimum.

- TOP/DOMDEC: This partitioning package developed by Nasa [Far95b] includes a synthesis of mesh partitioning techniques. A random neighbor contraction procedure is used to coarsen the mesh, one of the algorithms described in the previous sections is used to generate an initial partition, and finally non-deterministic techniques are invoked to improve the quality of the partition.

Chapter 4

INITIAL PARTITIONING

In the second chapter, an objective function has been designed to deal with the mesh distribution problem during the partitioning phase. To be able to use refinement techniques an initial partition has to be provided somehow. To answer this purpose, topology-based or geometry-based partitioning algorithms such as the ones described in the previous chapter could be applied. In our opinion, these algorithms suffer from at least one major limitation. They all fail to include the various components of the objective function during the partitioning process. After all, this is the goal sought. With the intention of doing so, this chapter is devoted to the development of partitioning algorithms that explicitly incorporate some features of the objective function in order to guide the partitioning. An experimental evaluation of these algorithms on several test cases is presented. All the partitioning algorithms including those developed in this chapter encounter domain splitting in their final partition, and a simple heuristic is proposed to minimize the occurrence of such a phenomenon. Finally, the relative merits of the best partitioning algorithm developed in this chapter is compared to some other partitioning schemes.

4.1 A Simple Greedy Algorithm

As an initial partitioning technique prior to optimization, we have been interested in Farhat's algorithm [Far88]. This interest is motivated by the fact that this algorithm is fast, and delivers decompositions with the following properties [Far93]:

- balanced subdomains;
- subdomains having good shapes;
- subdomains with small perimeter.

Algorithm 2 Pseudo-code of the Simple Greedy algorithm.

Procedure SimpleGreedy (**input** G: graph, **output** $\mathcal{P}$: part)
  $U := G$; $\mathcal{P} := \{U\}$; /* U represents the set of unassigned vertices */
  if $sub = 1$ **then**
    choose a vertex $v \in V(U)$ with minimal degree;
  else
    choose a vertex $v \in \mathrm{V_{bdry}}(U)$;
  end if
  $V(S_{sub}) := \{v\}$;
  $V(U) := V(U) - \{v\}$;
  repeat
    choose a vertex $v \in \mathrm{V_{bdry}}(U)$ such that: $\exists\, w \in V(S_{sub})$: $\langle v, w \rangle \in E(G)$;
    if such a v does not exist **then**
      choose a vertex $v \in \mathrm{V_{bdry}}(U)$;
    end if
    AddVertex(S_{sub},v);
    RemoveVertex(U,v);
  until $|S_{sub}| = ISW$;
  $\mathcal{P} := \mathcal{P} \cup \{S_{sub}\}$;

A disadvantage of Farhat's algorithm is the need to generate three different relations (the element-to-element relation, the node-to-element relation

and the element-to-node relation). The memory on some platforms may be low to store these relations especially for large meshes. In addition, the time required to construct such relations slows down the decomposition process. Therefore, a variant called *Simple Greedy* (SG) is used. This variant is similar to the one used by Walshaw [Wal95a] and differs from its original counterpart in that it works mainly with a graph rather than the nodes and elements of the mesh. Let us assume that a graph G has been selected to represent a given mesh. Let U denote the yet unpartitioned subgraph of G. The SG works by exploiting mainly the vertices adjacency information of G. It starts by choosing a vertex with minimal degree. This criterion is designed in such a way that the first subdomain starts to accumulate vertices at one corner of the graph. Thereafter, unassigned neighbors to the starting vertex are taken and added to the current subdomain. The rest of the subdomain is built by iteratively adding adjacent vertices until its correct size is reached. Once a vertex v is added to a subdomain S_k, the set of edges joining this vertex to those neighbors belonging to S_k are added to $E(S_k)$ and removed from $E(U)$. The starting vertex for a S_k with $k \neq 1$, is an unassigned vertex having a neighbor in previously defined subdomains. We make the assumption that the vertices of the graph describing the mesh have identical weights. This a reasonable assumption as the meshes involved in all our experiments use the same element type. If the number of vertices is a multiple of the number of subdomains, then the correct size of a subdomain is given by: $ISW = |V(G)|/N_{sub}$, otherwise $ISW = \lceil |V(G)|/N_{sub} \rceil$ or $ISW = \lfloor |V(G)|/N_{sub} \rfloor$, depending on the subdomain number. Algorithm 2 describes the pseudo-code of SG.

4.2 Partitioning Quality Measures

Judging the quality of a given partition requires the use of metrics. The first metric that comes into mind is for instance, the use of the value returned by

Symbol	Meaning and Definition
LBR	Load balance ratio: $\frac{\sum_{i=1}^{N_{sub}} W(S_i)}{N_{sub} \times max_{j=1,\ldots,N_{sub}} W(S_j)}$
EC	Edge cut: $1/2 \times \sum_{k,l=1,k\neq l}^{N_{sub}} \sum_{v_i \in S_k, v_j \in S_l} \lvert (v_i, v_j) \rvert$
$\lvert E(S) \rvert$	Number of edges in subdomain graph: $1/2 \times \sum_{i=1}^{N_{sub}} \deg_S(S_i)$
PR	Perimeter balance ratio: $\frac{\sum_{i=1}^{N_{sub}} \lvert E_{bdry}(S_i) \rvert}{N_{sub} \times max_{j=1,\ldots,N_{sub}} \lvert E_{bdry}(S_j) \rvert}$
$\overline{Sh}$	Average subdomain shape: $\frac{1}{N_{sub}} \sum_{i=1}^{N_{sub}} \frac{surf(S_i)}{surfcirc(S_i)}$
$\lvert BV \rvert$	Number of boundary vertices: $1/2 \times \sum_{i=1}^{N_{sub}} \lvert V_{bdry}(S_i) \rvert$
$\lvert DS \rvert$	Number of disconnected subdomains
MR	Matching ratio: $\frac{\lfloor \frac{\lvert V(G) \rvert}{2} \rfloor}{\lvert V(C(G)) \rvert}$
t	Time to partition

Figure 4.1: Metrics

the objective function $OF_{partitioning}$. This value is the sum of four different components, and adopting it as a metric will not provide the user with enough information regarding the behavior of a partitioning algorithm according to a specific criterion. With this view in mind, the metrics listed in Figure 4.1 are used to assess the quality of a given partition. However, we should stress the fact that the ultimately only important measurement is the parallel efficiency of the solver on a given parallel machine. We estimate the shape of a subdomain in a similar fashion as described in [Far95a] [Van95b]. As we will be using two-dimensional meshes as testbeds in this benchmark, the shape of a subdomain is taken as the ratio between the surface of the subdomain and the surface of the circle circumscribed to this subdomain.

4.3 Versions of the Simple Greedy Algorithm

Based on the visual examination of several partitions produced by SG, we observed that the quality of the partitions is reasonable as long as the graph has a relatively regular structure. However, when the graph is highly irregular, we observed that the number of disconnected subdomains becomes higher. The number of disconnected subdomain is a significant parameter because it affects the number of edges in the subdomain graph. In addition, subdomains tend to have their boundaries lying in dense area of the graph, thereby leading to large edge cuts. Therefore, three new versions of SG are introduced. These versions are similar to SG in that they exploit the connectivity of the graph, but differ in the manner vertices are selected. These versions are described as follows:

1. SG_{profit}: This variant attempts to generate partitions whose subdomains have a minimal perimeter. An unassigned vertex v_i from the set U is added to the subdomain under construction S_k, provided the *following two conditions are fulfilled:*

    - v_i is neighbor to at least one boundary vertex of S_k.
    - v_i is the most profitable vertex, (i.e. v_i contributes to the largest decrease or smallest increase in edge cut).

    If, by chance, several candidate vertices satisfy these two conditions, the vertex that improves the shape of S_k will be added first. If there is no neighbor vertex because a domain splitting phenomenon is about to take place, the algorithm proceeds by choosing a boundary vertex from the set U. This variant attempts to minimize the second term of the objective function $OF_{partitioning}$.

2. SG_{sh}: This version attempts to generate subdomains with good shapes. It proceeds as SG_{profit}, except that in the second condition, it uses the

shape (the fourth component of the objective function) to guide the partitioning process.

3. SG_{dist}: This version does not exploit explicitly any specified component of the objective function $OF_{partitioning}$, but rather combines both topological and geometrical information of the graph to guide the partitioning process. This version proceeds as SG_{profit} except that, in the second condition, it uses the euclidean distance from the starting vertex of S_k instead of the edge cut.

As far as the implementation is concerned, all the three variants simply loop over all the boundary vertices of a particular subdomain in order to identify the appropriate unassigned vertex to be inserted.

4.4 Test Meshes

Graph Name	$\|V\|$	$\|E\|$	Description
BIG	15606	91756	2D Mesh
BRACK	62631	366559	3D Mesh
HAMMOND	4720	27444	2D Mesh
$NACA_{nd}$	9170	54636	2D Mesh
NACA	18148	217190	2D Mesh
SNECMA	38589	462876	2D Mesh
$T60K_{nd}$	30570	211720	2D Mesh
T60K	60005	713226	2D Mesh
WHEEL	19620	240692	2D Mesh
WHITAKER	9800	57978	3D Mesh

Figure 4.2: Various graphs used in evaluating the quality of the partitioning.

In the course of this dissertation, we will use a set of realistically meshes to evaluate and compare different partitioning algorithms based on the metrics presented in Section 4.2. In order to be able to draw reasonable conclusions

regarding the behavior of a given partitioning algorithm, the meshes used in the benchmark were selected on the basis of their sizes (small, medium, large) and types (simple and complex geometry). All the graphs representing these meshes are true communication graphs. In addition, we include two node-to-node graphs for the meshes T60K and NACA referred to as $T60K_{nd}$ and $NACA_{nd}$. Figure 4.2 gives an overview over the test meshes.

4.5 Experimental Results

Having described in Section 4.3 the various algorithms we propose to derive an initial partition, we now move on to evaluating their relative merits. All the experiments were performed on an Silicon Graphics with 128 Mbytes of memory and 230 MHz CPU. All times reported are in seconds. The behavior of a partitioning algorithm regarding a specific metric is analyzed using 5 graphs. The number of subdomains ranges from 2 to 256, thus giving rise to a benchmark with 40 test cases per metric.

4.5.1 Weight Imbalance

All the four greedy algorithms assume that the balanced computational time is achieved by balancing the vertices among subdomains. As a result, the number of vertices in any two subdomains differs at most by one, leading to a well balanced weight distribution. The main differences between the different algorithms reside in the shape of the subdomains, the edge cut, and the number of edges connecting the set of the partitions.

4.5.2 Edge Cut

Figures 4.3, 4.4, 4.5, 4.6, and 4.7, show the quality of the partitions generated by the different algorithms in terms of the size of the edge cut.

Based on the obtained results, the following observations are made:

N_{sub}	SG		SG_{profit}		SG_{sh}		SG_{dist}	
	EC	PR	EC	PR	EC	PR	EC	PR
2	401	1	140	1	164	1	178	1
4	415	0.67	362	0.80	441	0.78	425	0.76
8	671	0.77	425	0.78	633	0.63	783	0.72
16	1086	0.64	765	0.56	1190	0.42	989	0.56
32	1514	0.65	1139	0.63	1555	0.33	1380	0.49
64	2167	0.48	1807	0.48	2104	0.41	2298	0.45
128	3017	0.55	2720	0.62	3046	0.40	3111	0.53
256	4261	0.64	3874	0.67	4274	0.48	4450	0.51

Figure 4.3: Comparing different simple greedy partitioning algorithms in terms of edge cut using the graph HAMMOND.

N_{sub}	SG		SG_{profit}		SG_{sh}		SG_{dist}	
	EC	PR	EC	PR	EC	PR	EC	PR
2	1104	1	222	1	289	1	390	1
4	1327	0.76	513	0.77	699	0.63	911	0.79
8	1610	0.70	829	0.75	1261	0.60	1123	0.74
16	2222	0.73	1333	0.59	1673	0.59	2072	0.54
32	2925	0.58	1957	0.64	2397	0.27	2627	0.48
64	4059	0.55	3068	0.56	3664	0.41	3745	0.54
128	5819	0.59	4770	0.57	5539	0.34	5808	0.33
256	8073	0.50	7041	0.64	7929	0.28	8344	0.32

Figure 4.4: Comparing different simple greedy partitioning algorithms in terms of edge cut using the graph BIG.

- The results asserts that SG_{profit} is the clear winner in terms of edge cut compared to the other variants. It produces better partitions for 39 out of 40 test cases. This should not be surprising since SG_{profit} minimize the edge cut increase in each step.

- Based on the visual examination of the partitions produced by SG_{profit}, we observed that the boundaries of the different subdomains tend to be located outwards the dense areas of the graph leading to much smaller

N_{sub}	SG		SG_{profit}		SG_{sh}		SG_{dist}	
	EC	PR	EC	PR	EC	PR	EC	PR
2	1444	1	249	1	290	1	326	1
4	5898	0.64	776	0.63	1578	0.55	1381	0.69
8	9018	0.61	2968	0.67	3438	0.45	4353	0.62
16	13546	0.58	4558	0.65	6703	0.64	5710	0.71
32	17394	0.63	8139	0.48	10094	0.63	10756	0.46
64	22241	0.53	12492	0.53	16860	0.48	17286	0.54
128	28527	0.62	20534	359	26836	0.38	26280	0.44
256	37644	0.59	31538	0.57	39270	0.39	39588	0.36

Figure 4.5: Comparing different simple greedy partitioning algorithms in terms of edge cut using the graph SNECMA.

N_{sub}	SG		SG_{profit}		SG_{sh}		SG_{dist}	
	EC	PR	EC	PR	EC	PR	EC	PR
2	832	1	704	1	1128	1	1145	1
4	2916	0.73	1705	0.74	2569	0.79	2899	0.62
8	5026	0.70	4034	0.79	5338	0.57	5322	0.77
16	8790	0.83	8451	0.78	8797	0.57	8679	0.66
32	13086	0.74	12779	0.59	13831	0.78	13635	0.65
64	19285	0.59	19563	0.66	20136	0.35	20049	0.47
128	29674	0.58	28622	0.60	29723	0.33	30065	0.48
256	43246	0.47	41583	0.56	42775	0.32	43735	0.18

Figure 4.6: Comparing different simple greedy partitioning algorithms in terms of edge cut using the graph T60K.

N_{sub}	SG		SG$_{profit}$		SG$_{sh}$		SG$_{dist}$	
	EC	PR	EC	PR	EC	PR	EC	PR
2	2713	1	852	1	990	1	1267	1
4	3102	0.89	1289	0.80	3207	0.89	3302	0.89
8	5408	0.57	2756	0.74	5220	0.57	5064	0.57
16	6973	0.73	4463	0.74	7073	0.73	7191	0.73
32	9316	0.67	7193	0.57	9516	0.70	9474	0.67
64	12946	0.62	10666	0.61	13183	0.62	12870	0.62
128	18031	0.63	15668	0.56	18160	0.63	19048	0.63
256	24942	0.56	22234	0.60	25064	0.56	26544	0.56

Figure 4.7: Comparing different simple greedy partitioning algorithms in terms of edge cut using the graph NAGA.

edge cut compared to the other variants. On the average, in coarse-grained partitioning ($N_{sub} = 2, 4, 8, 16, 32$), SG$_{profit}$ does 44%, 27%, and 30% better than SG, SG$_{sh}$, and SG$_{dist}$, whereas in fine-grained partitioning ($N_{sub} = 64, 128, 256$) the improvements are 16%, 14%, and 13% respectively. These results indicate that SG$_{profit}$ performs substantially better than the other variants regardless of the number of subdomains.

- The perimeter balance ratio PR metric indicates that all the algorithms fail to generate partitions with balanced perimeters. We notice that this ratio becomes in general worse when the number of subdomains is large.

4.5.3 Subdomain Graph

Figures 4.8, 4.9, 4.10, 4.11, and 4.12, show the quality of the partitions generated by the different algorithms in terms of the number of edges in the subdomain graph. From these results, we see that SG$_{profit}$ generates subdomain graphs having up to 17% less edges than those generated by the other variants.

	SG	SG_{profit}	SG_{sh}	SG_{dist}
N_{sub}	$\|E(S)\|$	$\|E(S)\|$	$\|E(S)\|$	$\|E(S)\|$
64	172	158	165	151
128	347	333	351	326
256	704	695	730	695

Figure 4.8: Comparing different simple greedy partitioning algorithms in terms of the number of edges in the subdomain graph using the graph HAMMOND.

The only notable exception is T60K for which SG_{profit} does up to 7% worse than SG. Finally, we notice that the difference in quality between SG_{profit} and SG_{dist} is within 5% with SG_{profit} doing in general better. The good behavior of SG_{profit} in terms of the number of edges in the subdomain graph is due to the fact that it tends to produce less disconnected subdomains compared to the other variants.

	SG	SG_{profit}	SG_{sh}	SG_{dist}
N_{sub}	$\|E(S)\|$	$\|E(S)\|$	$\|E(S)\|$	$\|E(S)\|$
64	179	148	166	153
128	350	328	344	328
256	729	682	722	686

Figure 4.9: Comparing different simple greedy partitioning algorithms in terms of the number of edges in the subdomain graph using the graph BIG.

	SG	SG_{profit}	SG_{sh}	SG_{dist}
N_{ub}	$\|E(S)\|$	$\|E(S)\|$	$\|E(S)\|$	$\|E(S)\|$
64	182	169	178	161
128	389	359	406	373
256	805	752	819	789

Figure 4.10: Comparing different simple greedy partitioning algorithms in terms of the number of edges in the subdomain graph using the graph SNECMA.

	SG	SG_{profit}	SG_{sh}	SG_{dist}
N_{sub}	$\|E(S)\|$	$\|E(S)\|$	$\|E(S)\|$	$\|E(S)\|$
64	153	165	181	158
128	336	356	376	347
256	724	753	853	748

Figure 4.11: Comparing different simple greedy partitioning algorithms in terms of the number of edges in the subdomain graph using the graph T60K.

	SG	SG_{profit}	SG_{sh}	SG_{dist}
N_{sub}	$\|E(S)\|$	$\|E(S)\|$	$\|E(S)\|$	$\|E(S)\|$
64	191	182	190	201
128	397	374	413	390
256	809	788	827	826

Figure 4.12: Comparing different simple greedy partitioning algorithms in terms of the number of edges in the subdomain graph using the graph NACA.

4.5.4 Shape Quality

Figures 4.13, 4.14, 4.15, 4.16, and 4.17, show the quality of partitions in terms of the average subdomain shape $\left(\overline{Sh} = \frac{1}{N_{sub}} \sum_{i=1}^{N_{sub}} \frac{\text{surf}(S_i)}{\text{surfcirc}(S_i)}\right)$ produced by the different algorithms.

	SG	SG_{profit}	SG_{sh}	SG_{dist}
N_{sub}	$\overline{Sh}$	$\overline{Sh}$	$\overline{Sh}$	$\overline{Sh}$
8	0.29	0.42	0.46	0.44
16	0.30	0.44	0.37	0.33
32	0.29	0.47	0.46	0.38
64	0.36	0.46	0.49	0.40
128	0.41	0.44	0.48	0.40
256	0.42	0.50	0.49	0.40

Figure 4.13: Comparing different simple greedy partitioning algorithms in terms of subdomain shape using the graph HAMMOND.

Examining the relevant figures, we notice that SG_{sh} is the clear winner

	SG	SG_{profit}	SG_{sh}	SG_{dist}
N_{sub}	$\overline{Sh}$	$\overline{Sh}$	$\overline{Sh}$	$\overline{Sh}$
8	0.24	0.48	0.39	0.39
16	0.30	0.44	0.39	0.43
32	0.36	0.43	0.49	0.39
64	0.36	0.44	0.46	0.38
128	0.37	0.44	0.49	0.40
256	0.39	0.46	0.50	0.40

Figure 4.14: Comparing different simple greedy partitioning algorithms in terms of subdomain shape using the graph BIG.

	SG	SG_{profit}	SG_{sh}	SG_{dist}
N_{sub}	$\overline{Sh}$	$\overline{Sh}$	$\overline{Sh}$	$\overline{Sh}$
8	0.35	0.58	0.62	0.55
16	0.36	0.50	0.54	0.47
32	0.41	0.46	0.49	0.45
64	0.39	0.47	0.49	0.41
128	0.37	0.44	0.48	0.39
256	0.41	0.44	0.49	0.37

Figure 4.15: Comparing different simple greedy partitioning algorithms in terms of subdomain shape using the graph NACA.

	SG	SG_{profit}	SG_{sh}	SG_{dist}
N_{sub}	$\overline{Sh}$	$\overline{Sh}$	$\overline{Sh}$	$\overline{Sh}$
8	0.32	0.31	0.47	0.36
16	0.35	0.40	0.46	0.37
32	0.37	0.33	0.49	0.35
64	0.38	0.33	0.50	0.37
128	0.39	0.34	0.55	0.40
256	0.41	0.37	0.56	0.41

Figure 4.16: Comparing different simple greedy partitioning algorithms in terms of subdomain shape using the graph T60K.

	SG	SG_{profit}	SG_{sh}	SG_{dist}
N_{sub}	$\overline{Sh}$	$\overline{Sh}$	$\overline{Sh}$	$\overline{Sh}$
8	0.31	0.41	0.45	0.35
16	0.30	0.35	0.43	0.30
32	0.34	0.37	0.49	0.35
64	0.31	0.37	0.49	0.36
128	0.37	0.39	0.51	0.35
256	0.35	0.44	0.53	0.36

Figure 4.17: Comparing different simple greedy partitioning algorithms in terms of subdomain shape using the graph SNECMA.

when it comes to generate subdomains with good shape. It produces better partitions for 25 out of 30 test cases, followed by SG_{profit} with 5 out of 30 test cases. On the average, SG_{sh} does 27%, 19%, and 16% better than SG, SG_{dist}, and SG_{profit} respectively. Finally, as the experiments show, the performance of SG in terms of the shape is on the average worse than the other variants.

4.5.5 Decomposition Time

Looking at the runtimes for the different algorithms shown in Figures 4.18, 4.19, 4.20, and 4.22, we deduce the following remarks:

	SG	SG_{profit}	SG_{sh}	SG_{dist}
N_{sub}	t	t	t	t
8	0.08	0.16	0.28	0.14
16	0.09	0.18	0.23	0.13
32	0.10	0.15	0.18	0.12
64	0.11	0.13	0.17	0.13
128	0.12	0.17	0.20	0.15
256	0.17	0.24	0.27	0.22

Figure 4.18: Comparing different simple greedy partitioning algorithms in terms of CPU time using the graph HAMMOND.

- SG requires the least amount of time for partitioning. This is not sur-

N_{sub}	SG t	SG_{profit} t	SG_{sh} t	SG_{dist} t
8	0.27	0.66	1.74	0.76
16	0.29	0.63	1.16	0.67
32	0.31	0.57	0.88	0.53
64	0.35	0.55	0.78	0.49
128	0.38	0.57	0.77	0.57
256	0.57	0.72	0.89	0.63

Figure 4.19: Comparing different simple greedy partitioning algorithms in terms of CPU time using the graph BIG.

N_{sub}	SG t	SG_{profit} t	SG_{sh} t	SG_{dist} t
8	0.56	1.58	4.03	1.70
16	0.57	1.59	2.9	1.50
32	0.63	1.43	2.19	1.22
64	0.68	1.45	1.73	1.14
128	0.88	1.47	1.62	1.18
256	1.24	1.72	1.72	1.46

Figure 4.20: Comparing different simple greedy partitioning algorithms in terms of CPU time using the graph NACA.

N_{sub}	SG t	SG_{profit} t	SG_{sh} t	SG_{dist} t
8	1.76	5.48	16.50	6.33
16	1.78	4.24	11.75	5.14
32	1.83	3.88	9.01	4.46
64	1.89	2.56	7.16	3.84
128	1.97	2.02	6.42	3.81
256	2.43	3.10	7.60	4.55

Figure 4.21: Comparing different simple greedy partitioning algorithms in terms of CPU time using the graph T60K.

	SG	SG_{profit}	SG_{sh}	SG_{dist}
N_{sub}	t	t	t	t
8	1.29	3.76	6.84	4.04
16	1.33	3.53	7.26	3.47
32	1.37	3.33	5.63	3.00
64	1.62	3.07	4.76	2.7
128	1.88	3.13	4.71	2.77
256	2.42	3.15	4.35	2.61

Figure 4.22: Comparing different simple greedy partitioning algorithms in terms of CPU time using the graph SNEGMA.

prising since SG exploits only the adjacency information of the graph.

- SG requires as much as 2 times less than either SG_{dist} or SG_{profit}, and as much as 4 times less than SG_{sh}.

- The high runtimes required by SG_{profit}, SG_{dist}, and SG_{sh} are partly due to our inefficient implementation of the search procedure used to select the appropriate vertex to be added.

- A closer examination of the execution time of SG reveals that it increases with the number of subdomains. This observation is not always true for the other variants. Looking at Figure 4.22, we see for instance that in the case of the graph SNECMA, the time required for the 32-partition is higher than the time for the 64-partition. In the variants SG_{profit}, SG_{sh}, and SG_{dist}, the more subdomains are required, the shorter are the boundaries, thus less time is needed to select the appropriate candidate vertex.

4.5.6 Sensitivity to the Starting Vertex

Though the different simple greedy algorithms introduced in this chapter differ in the manner vertices are selected, they all share the common criterion for choosing the very first vertex. Generally speaking, there may be several

vertices satisfying the minimal degree criterion. The subject of this section is to investigate whether the choice of the starting vertex influences the quality of the partitions in terms of edge cut. To this end, the graph NACA is chosen as a testbed together with the algorithms SG, SG_{profit} and SG_{sh}. The graph NACA was chosen because it has the largest number (7) of vertices that meet the requirements of the starting vertex.

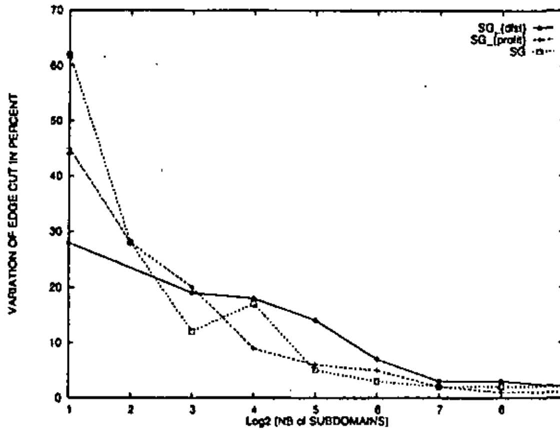


Figure 4.23: Sensitivity of different greedy algorithms to the starting vertex.

In Figure 4.23, the number of subdomains is plotted as a function of the variation of the edge cut; this variation defined as $(100 \times (1 - min/max))$ where min and max represent the best and the worst edge cut obtained after running each algorithm from seven different starting vertices. In Figure 4.23, we observe that the quality of the edge cut is highly sensitive to the choice of the initial vertex for the three algorithms for small and moderate number of subdomains. On the other hand, for large number of subdomains, the influence of the starting vertex becomes less apparent. The difference in quality is on average below 2%. Consequently, it might be worth trying

a small number of vertices in coarse grained partitioning, whereas in fine grained partitioning there is little gain in doing so.

4.6 Domain Splitting

4.6.1 Number of Disconnected Subdomains

Based on the visual examination of the partitions, all the different simple greedy algorithms described in this chapter tend to produce disconnected subdomains (i.e. subdomains consisting of more than one subgraph component).

	SG	SG_{profit}	SG_{sh}	SG_{dist}
N_{sub}	$\|DS\|$	$\|DS\|$	$\|DS\|$	$\|DS\|$
64	7	3	7	21
128	10	7	16	4
256	13	9	27	13
512	34	13	33	21

Figure 4.24: Number of disconnected subdomains generated by the different greedy partitioning algorithms using the graph HAMMOND.

	SG	SG_{profit}	SG_{sh}	SG_{dist}
N_{sub}	$\|DS\|$	$\|DS\|$	$\|DS\|$	$\|DS\|$
64	7	4	8	5
128	12	5	14	9
256	21	18	29	13
512	34	20	58	26

Figure 4.25: Number of disconnected subdomains generated by the different greedy partitioning algorithms using the graph BIG.

A subdomain is likely to be disconnected if its growth is hindered by surrounding subdomains. It is our experience that the more a graph is irregular, the higher the number of disconnected subdomains. Figures 4.24,

N_{sub}	SG $\|DS\|$	SG_{profit} $\|DS\|$	SG_{sh} $\|DS\|$	SG_{dist} $\|DS\|$
64	11	3	7	4
128	13	7	19	8
256	23	11	22	7
512	25	10	44	16

Figure 4.26: Number of disconnected subdomains generated by the different simple greedy partitioning algorithms using the graph SNECMA.

4.25, and 4.26 show the number of disconnected subdomains produced by each algorithm. Based on these results, the following observations are made:

- All the algorithms encounter domain splitting during their partitioning process.
- SG_{profit} produces 9 out of 12 partitions with fewer disconnections than the other variants.
- On the average SG_{profit} produces partitions with as much as 47%, 59%, and 35% less disconnections than those of SG, SG_{sh}, and SG_{dist} respectively.

4.6.2 Avoiding Domain Splitting

As has been noticed earlier, domain splitting is quite frequent with all the algorithms presented in this chapter. To cope with this phenomenon, a simple heuristic is invoked whenever a subdomain is about to be disconnected. This heuristic is better illustrated using the example depicted in Figure 4.27. This figure shows a domain partitioned into 7 subdomains except that the S_7 is still incomplete, and an unpartitioned part of the domain representing the set of yet unassigned vertices which we term U. From Figure 4.27 we see that S_7 becomes trapped as it reaches the periphery of the domain. Thus, any of the simple greedy algorithms will fail to find new unassigned vertices which

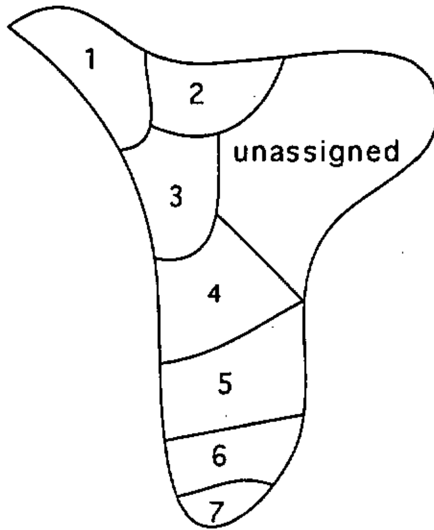


Figure 4.27: An example illustrating the domain splitting phenomenon.

are neighbors to those recently inserted to S_7. Therefore, the next move will be the selection of a vertex belonging to the set U, thereby leading to a domain splitting. In order to prevent this phenomenon from happening, the proposed heuristic is invoked and works in two phases. In the first phase, the shortest path in the subdomain graph between S_7 and the set U is identified. In this example, the shortest path is determined by the sequence U, S_4, S_5, S_6, S_7. In the second phase, vertices are moved along the shortest path ($U \rightarrow S_4 \rightarrow S_5 \rightarrow S_6 \rightarrow S_7$), where the notation $S_i \rightarrow S_j$ means that a transfer is to take place from S_i to S_j. A vertex is allowed to move, for instance from S_i to S_j, provided it meets following requirements:

- Having at least one neighbor in S_j.
- Contributing to the smallest increase or largest decrease in edge cut.

Algorithm 3 Pseudo-code of the heuristic used to minimize the occurrence of domain splitting

```
Procedure PreventSplitting (input/output P: part, input U: graph)
  /* P: The set of the subdomain generated so far */
  /* path: array of the subdomains' indices defining the shortest path */
  /* S_index: Subdomain about to be disconnected */
  U := set of unassigned vertices;
  path := ShortestPath (U,S_index);
  pathlength:= |path|;
  repeat
    for i:= 2 to pathlength do
      if E_bdry(S_path[i],S_path[i-1]) ≠ ∅ then
        v := FindBoundaryVertex (S_path[i],S_path[i-1])
        RemoveVertex(S_path[i-1],v);
        AddVertex(S_path[i],v));
      else
        exit;
      end if
    end for
  until |S_index| = ISW;
```

- Preferably not being an articulation vertex (i.e. a vertex whose removal provokes a disconnection).

The present implementation of the heuristic aborts if by chance under the migration process, an existing boundary does no longer exist between two subdomains involved in the transfer. The pseudo-code of the heuristic is given in Algorithm 3. Figures 4.29, 4.30, and 4.28, show the number of edges in subdomain graphs, the average subdomain shape, the size of the edge cut, and the decomposition time of the partitioning process of SG with and without the use of the heuristic.

Based on the examination of the results, the following observations are made:

- In 10 out of 12 partitions, SG combined with the heuristic performs

N_{sub}	SG without				SG with			
	$\|E(S)\|$	$\overline{Sh}$	EC	t	$\|E(S)\|$	$\overline{Sh}$	EC	t
64	172	0.36	2167	0.14	165	0.35	2333	0.24
128	347	0.41	3017	0.17	329	0.39	3202	0.51
256	704	0.42	4261	0.21	682	0.40	4124	0.85
512	1472	0.43	5915	0.42	1408	0.41	5958	3.29

Figure 4.28: Impact of the heuristic used to reduce the domain splitting phenomenon on the quality of the partitions using the graph HAMMOND.

N_{sub}	SG without				SG with			
	$\|E(S)\|$	$\overline{Sh}$	EC	t	$\|E(S)\|$	$\overline{Sh}$	EC	t
64	179	0.36	4059	0.35	160	0.35	4462	0.78
128	350	0.37	5819	0.38	333	0.34	6292	1.06
256	729	0.39	8073	0.57	682	0.37	6456	2.1
512	1471	0.41	11251	0.94	1413	0.40	11783	5.16

Figure 4.29: Impact of the heuristic used to reduce the domain splitting phenomenon on the quality of the partitions using the graph BIG.

N_{sub}	SG without				SG with			
	$\|E(S)\|$	$\overline{Sh}$	EC	t	$\|E(S)\|$	$\overline{Sh}$	EC	t
64	182	0.33	22241	1.62	179	0.35	23858	6.06
128	389	0.37	28527	1.88	381	0.36	30518	5.76
256	805	0.35	37644	2.42	786	0.33	39583	6.64
512	1652	0.34	38589	4.15	1616	0.32	40093	14.14

Figure 4.30: Impact of the heuristic used to reduce the domain splitting phenomenon on the quality of the partitions using the graph SNECMA.

worse than SG in terms of edge cut. The difference is on the average within 6%. The only exception is the 256-partition using the graph BIG where SG combined with the heuristic does 20% worse than SG.

- The combination of SG and the heuristic produces subdomain graphs

having 4% on the average less edges than those of SG. The best notable case was that of the 64-partition usiog BIG, where up to 11% reduction is obtained.

- The combination of SG and the heuristic yields lower average subdomain shape compared to SG. However, the differences are small.
- Looking at the combined time of SG and the heuristic, we notice that it is up to six times higher than of SG. This is often due to the scattering of the set U over small regions, and the heuristic has to be executed several times before a subdomain gets its correct size.

4.7 Comparison of Different Partitioning Algorithms

To highlight the relative merits of SG_{profit}, its performance is compared with the Inertial Algorithm (IA), the Recursive Spectral Bisection (RSB) using the Lanczos solver, and Farhat's algorithm (GR). We used the graph partitioning package CHACO [Hen93a] to get the results of both IA and RSB, and TOP/DOMDEC [Far95b] to get those of GR. Figures 4.31, 4.32, and 4.33 show the quality of the partitions in terms of the edge cut as well as the running times of each algorithm. All the experiments were performed on a Silicon Graphics with 128 Mbytes of memory and 230 MHz CPU. All times reported are in seconds. Based on these experimental results, the following observations are made:

- SG_{profit} and RSB are the clear winners as far as the size of the edge cut is concerned. Out of 18 partitions, 13 are won by SG_{profit} and 5 by RSB.
- In the 13 cases won by SG_{profit}, the improvement is within 5% on the average, whereas in the 5 cases won by RSB the improvement is within 7% on the average.

	SG$_{profit}$		GR		IA		RSB	
N_{sub}	EC	t	EC	t	EC	t	EC	t
8	425	0.78	611	0.27	880	0.43	468	2.73
16	765	0.56	1045	0.30	1342	0.45	766	3.11
32	1139	0.63	1587	0.34	1791	0.47	1225	3.47
64	1807	0.48	2122	0.40	2392	0.50	1904	3.78
128	2720	0.62	3085	0.47	3285	0.53	2792	4.09
256	3874	0.67	4105	0.57	4471	0.57	3998	4.50

Figure 4.31: Comparing different partitioning algorithms in terms of edge cut and CPU time using the graph HAMMOND.

	SG$_{profit}$		GR		IA		RSB	
N_{sub}	EC	t	EC	t	EC	t	EC	t
8	2756	2.38	4895	0.75	3377	2.04	3065	38.63
16	4463	2.59	6430	0.86	5801	2.16	4888	43.51
32	7193	2.34	8759	0.94	8848	2.28	7371	47.42
64	10666	2.48	12711	1.11	12955	2.42	11231	50.72
128	15568	2.41	17857	1.23	18052	2.58	16182	53.72
256	22234	2.90	24789	1.32	24931	2.75	23179	55.64

Figure 4.32: Comparing different partitioning algorithms in terms of edge cut and CPU time using the graph NACA.

	SG$_{profit}$		GR		IA		RSB	
N_{sub}	EC	t	EC	t	EC	t	EC	t
8	4034	5.48	5121	1.57	5267	10.32	3904	200.22
16	6451	4.24	8689	1.67	8533	10.73	6956	239.45
32	12779	3.88	13086	1.75	13452	11.17	11280	259.95
64	19563	2.56	19285	1.87	19625	11.73	17825	274.78
128	28622	2.02	29304	1.95	28894	12.36	26495	286.16
256	41583	3.10	43146	2.11	41717	13.05	38886	293.85

Figure 4.33: Comparing different partitioning algorithms in terms of edge cut and CPU time using the graph T60K.

- The time of SG_{profit} and RSB vary significantly. For the graph HAMMOND, RSB requires as much as up to 7 more time than SG_{profit} , whereas for the graphs NACA and T60K, RSB requires as much as 19, and 95 more time than SG_{profit} respectively. This high computational time is due to evaluation of the Fiedler vector within the chosen desired accuracy used by RSB.

- The comparison of SG_{profit} and GR reveals that the former consistently performs better than the latter in all but one case. The improvement is on the average 17%. Looking at the running times of the two algorithms, we notice that GR is up to 4 times faster than SG_{profit}.

- SG_{profit} does 19% better on the average than IA. The running times of both SG_{profit} and IA does not vary significantly much for HAMMOND and NAGA. On the other hand, for large problems such as T60K, IA becomes expensive as it requires up to 6 times more time than SG_{profit}.

4.8 Concluding Remarks

In this chapter, new simple greedy partitioning algorithms have been presented. These algorithms exploit the adjacency information of the graph and differ mainly in the manner vertices to be added in a subdomain are selected. The experimental results have shown that SG_{profit} delivers partitions with smaller edge cut and good shapes compared to the other variants. In addition, SG_{profit} generates partitions with smaller number of disconnections than the other variants, thereby reducing the size of subdomain graphs in terms of edges. The quality of the four algorithms studied in this chapter are sensitive to the very first starting vertex. However, this impact seems to be of little significance in fine grained-partitioning. Domain splitting seems to be unavoidable with the four algorithm, and a simple heuristic is proposed to deal with this phenomenon. The main improvement afforded by this heuristic resides in a reduction of the number of edges in subdomain graphs at

the expense of relatively high computational time. Finally, we conclude by saying that SG_{profit} is a good partitioning algorithm as it provides in most cases, partitions at least as good as RSB at a very low price.

Chapter 5

IMPROVEMENT OF PARTITIONS

To derive an initial partition from scratch, any of the partitioning algorithms developed in the previous chapter may be used. All these algorithms share the common feature that once a vertex has been assigned to a specific subdomain, it remains in it until the end of the partitioning process, unless the disconnection reduction algorithm has to be applied. These algorithms generate partitions of reasonable quality, therefore the maximization of the performance of a parallel solver requires the improvement of such partitions via improvement techniques. All the improvement techniques share the same basic idea, namely to modify the structure of the partition through the reassignment of the vertices in order to minimize an objective function. This chapter concerns the improvement of the partitions using a new variant of the simulated annealing method as an optimizer. The quality of the partitions produced by this variant is compared to those provided by the versions of Tabu Search (TS), and Stochastic Evolution (SE) methods which are included in the mesh partitioning package TOP/DOMDEC. Finally, a comparative experimental evaluations of the quality of the partitions produced by SG and SG_{profit} when both are coupled with the Kernighan-Lin improvement algorithm is presented.

5.1 Iterative Improvement Algorithms

One way to improve the quality of a given partition would be the use of iterative improvements algorithms. Keeping the discussion in quite general terms, these algorithms start from any feasible solution and try, step by step, to improve the value of the objective function by moving to a neighboring solution with a better objective function value. Whenever a better solution is reached, the current solution is updated to this new one. When there are no neighbors that have a better objective function value, these algorithms terminate in which case a local minimum is reached. One advantage of these algorithms is that the solution they produce is always feasible if the moves are performed in such a manner that feasibility is maintained. The drawback with these methods is the obvious risk of getting stuck in a local minimum. Therefore, elaborate methods such as Tabu Search, Stochastic Evolution, and Simulated Annealing are usually used to avoiding the local minima pitfalls.

5.2 Simulated Annealing

5.2.1 *Thermodynamics and Optimization Problems*

At the heart of the Simulated Annealing method (SA) [Kir83][Met53], there is an analogy with thermodynamics, specifically with the way that liquids freeze and crystallize, or metals anneal.

Physical systems	Optimization problems
State	Configuration
Energy	Cost
Phase transition	Move generation
Ground state	Optimal solution
Quick cooling(quenching)	Iterative improvement
Slow cooling	Simulated annealing

Figure 5.1: Analogy between physical systems and optimization problems

At high temperatures, the molecules of a liquid move freely. If the liquid is cooled slowly, the atoms are often able to form a pure crystal. This crystal is the state of minimum energy for this system. On the other hand, if a liquid metal is cooled quickly or "quenched", it does not reach the minimum energy, but rather ends up in a polycrystalline state. Thus, the minimum energy state is achieved by a *slow* cooling, allowing ample time for redistribution of the atoms as they lose mobility. The analogy between optimization problems and physical systems is given in Figure 5.1.

5.2.2 The Algorithm

The SA algorithm is a technique that has attracted significant attention as suitable for optimization problems. An iteration of SA starts with proposing a new state by a random perturbation and evaluating the resultant change in the objective function ΔE. If the change is negative, corresponding to a downhill move in the "energy landscape" (i.e. the set of all possible states with the corresponding objective function values), the perturbation is accepted and the new state becomes the starting point for the next perturbation. If ΔE is positive, corresponding to an uphill move, the new state may be accepted according to the law of thermodynamics; at temperature T the probability of an increase in energy ΔE is given by $Pr(\Delta E) = \exp(-\Delta E/kT)$, where k is the Boltzmann's constant (of no interest in combinatorial optimization, and therefore ignored). The higher the temperature, the greater the chance of moving to higher energy state. A temperature reduction function is used to lower the temperature. At low temperatures only moves leading to a lower energy state are accepted leading to a situation equivalent to iterative improvement algorithms. The temperature reduction function together with the initial temperature describes the annealing schedule which dictates how quickly the temperature is to be reduced. Since it is possible that SA finds a good state and then departs to a different region in the energy landscape, the best-so-far is always saved. The method terminates when a chosen conver-

Algorithm 4 Pseudo-code of the Simulated Annealing method.

```
Procedure SA (input S_0: state, input T_0: real, output S: state);
  T := T_0;
  S := S_0;
  C := Cost(S);
  repeat
    repeat
      S' := Perturb(S);
      C' := Cost(S');
      ΔC := C' - C;
      rnd:= random(0,1);
      if exp (-ΔC/T) > rnd then
        S := S' ; C := C';
      end if
    until (some criterion is satisfied);
    Update (T);
  until convergence;
```

gence criterion is fulfilled. The main advantage of SA is that the controlled uphill movements may prevent the search procedure from being trapped in a bad local minimum-energy state. It has been proven that the SA method converges to a global optimal solution under some assumptions. However, these assumptions are too restrictive to be of practical use. The pseudo-code of the Simulated Annealing method is given in Algorithm 4.

5.3 A Variant of Simulated Annealing

This section describes the concept behind a variant of simulated annealing. The heart of this new variant comes from the realization that it would be a great advantage to provide SA with a mean to first reach a locally optimal partition and thereafter use its ability to get out of it to direct the search elsewhere in the search space. The strategy described in [Mar92] uses a combination of SA and KL in order to embed deterministic local search

techniques into SA so that only local optima are subject to the accept/reject verdict. In this section, we propose another variant of SA, henceforth referred to as MSA and describe its concept. The MSA loops over two phases: in

---

Algorithm 5 Pseudo-code of the modified variant the Simulated Annealing method.

Procedure MSA (**input** $\mathcal{P}$: part; **input** T_0: real; **output** $I\mathcal{P}$: part);
  $T := T_0$;
  $\mathcal{P}_{best} := \mathcal{P}$;
  $C_{best} := \mathrm{OF}_{\mathrm{partitioning}}(\mathcal{P}_{best})$;
  repeat
    Improve ($\mathcal{P}$,$\mathcal{P}'$);
    $C' := \mathrm{OF}_{\mathrm{partitioning}}(\mathcal{P}')$;
    if $C' \geq C_{best}$ **then**
      $counter := counter + 1$;
    else
      $counter := 0$; $\mathcal{P}_{best} := \mathcal{P}'$; $C_{best} := C'$;
    end if
    Deteriorate($\mathcal{P}'$,T,$\mathcal{P}$);
    $T := \alpha \times T$;
  until ($counter \geq limit$);
  $I\mathcal{P} := \mathcal{P}_{best}$;

---

the first phase, an iterative improvement algorithm is used to reach a locally optimal partition, whereas in the second phase a deterioration procedure is invoked to worsen the current locally optimal partition by a certain threshold, to let the search proceed each time the algorithm used in the improvement phase is stuck in a locally optimal partition. Our variant samples only locally optimal partitions and retains the best visited one. The pseudo-code for MSA is given in Algorithm 5. The procedures Improve and Deteriorate are given in Algorithms 6 and 7 respectively.
The behavior of MSA for the bisection case using the mesh NACA is illustrated in Figure 5.2. This figure shows 56 partitions and their corresponding edge cut. Even values in the x-axis denote locally optimal partitions.

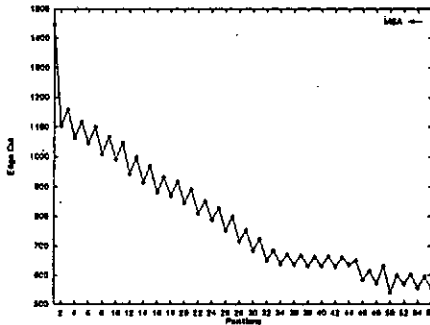


Figure 5.2: Evolution of the cost function for a typical run of the proposed variant of the Simulated Annealing method.

5.4 Implementation of the Method

The difficulty in using stochastic techniques resides in the number of parameters to be determined beforehand. To make use of the MSA method, the following elements have to be specified:

- The starting temperature.
- A method to determine when a locally optimal partition is detected.
- A rule for changing the value of the next temperature (cooling rate).
- A stopping condition.

A choice for these elements is referred to as cooling schedule.

5.4.1 Initial Temperature

The starting temperature is usually chosen to give a high probability of accepting even the worse cost increases. A simple rule to determine the

value of the initial temperature would be to run a series of iterations to determine the average increase in cost $\overline{\Delta C}$ and solve $T_{initial}$ from:

$$T_{init} = \frac{-\overline{\Delta C}}{\log(Pr)} \tag{5.1}$$

where Pr denotes the probability of accepting a move leading to an increase in cost. Note that, the exact starting temperature is not of great importance; it has to be high enough to guarantee a high rate of acceptance, without increasing uselessly the CPU time. From the equation 5.1, the initial temperature is determined using $Pr = 0.5$.

5.4.2 Type of Perturbation

The improvement of a given partitioo consists of perturbing the current partition through a move. Generally speaking, there exist two types of moves:

- Choose a vertex at random, and move it to another subdomain.
- Choose a boundary vertex at random, and move it to the neighboring subdomaio.

The first alternative tends to alter completely the structure of the initial partition, and leads to fragmented subdomains. The second alternative tends to displace the subdomain bouodaries and so is less prone to fragmented subdomains. Io our implementation we adopt the second alternative.

5.4.3 Improvement Phase

The objective of the improvement phase is to move from an initial partition to a locally optimal one through a series of perturbations. The first improvement method that comes immediately to mind is the popular Kernighan-Lin

algorithm (KL) [Ker70]. To improve an existing partition, most authors use KL due to its efficiency and reliability. However, in our work we employ a randomized iterative procedure we term Improve as it has proven to deliver good quality partitions. The pseudo-code of Improve is given in Algorithm 6.

Algorithm 6 Pseudo-code of the Iterative Improvement algorithm.

Procedure Improve (**input** $\mathcal{P}_0$: part, **output** $I\mathcal{P}$: part);
  $\mathcal{P}_{best} := \mathcal{P}_0$;
  $C := \mathrm{OF}_{\mathrm{partitioning}}(\mathcal{P})$;
  $C_{best} := C$;
  repeat
    $counter := 0$;
    for $move$:= 1 **to** max_{moves} **do**
      $\mathcal{P}' := \mathrm{Perturb}(\mathcal{P}_{best})$;
      $C' = \mathrm{OF}_{\mathrm{partitioning}}(\mathcal{P}')$;
      $\Delta := C' - C$;
      if $\Delta < 0$ **then**
        $counter := counter + 1$; $\mathcal{P}_{best} := \mathcal{P}'$; $C_{best} := C'$;
      end if
    end for
  until $(\frac{counter}{max_{moves}} \leq threshold)$;
  $I\mathcal{P} := \mathcal{P}_{best}$;

5.4.4 Number of Perturbations

The number of perturbations in the procedure Improve is the most critical parameter since it directly affects the overall CPU time of MSA. This number should be large enough to secure that most boundary vertices have the chance to be proposed for a move while not spending too much time. In our implementation, the number of iterations of Improve which is denoted by max_{moves} is set equal to twice the total number of boundary vertices. A locally optimal partition is supposed to be reached when the rate of acceptance

is below 5%. From the Algorithm 6, we see that the procedure Improve is equivalent to the simulated annealing method with a temperature equal to zero.

5.4.5 Deterioration Phase

The improvement phase results in a locally optimal partition. The purpose of the deterioration phase is to worsen the reached locally optimal partition by a certain threshold in order to provide the procedure Improve with a new partition to start with. Only uphill moves fulfilling the accept verdict are chosen in the procedure Deteriorate. When the temperature reaches low values, uphill moves are unlikely to be accepted, and much time will be wasted before the desired deterioration is achieved. Therefore, an uphill will be accepted at the latest after *threshold* consecutive rejections. In our implementation, the value of this threshold is set to 10. The pseudo-code of the procedure Deteriorate in given in Algorithm 7.

5.4.6 Cooling Rate

The SA method is sensitive to the manner the temperature is decreased. The cooling rate directly affects the number of iterations of the method. A fast cooling rate enables the method to be extremely fast at the expense of generating poor quality decompositions. There are two classes of cooling schedules. The first class contains fixed (i.e. predefined) schedules. The second class contains adaptive schedules, where the temperature decrement size depends on some observed value such as the average or the variance of the problem's objective function taken over a number of visited states. In our implementation, we use a fixed schedule with a cooling rate of 0.90.

Algorithm 7 Pseudo-code of the Deterioration procedure.

```
Procedure Deteriorate (input P_0: part, input T: real, output WP: part);
  P := P_0;
  C := OF_partitioning(P);
  counter := 0; Nreject := 0;
  repeat
    P' := Perturb (P);
    C' := OF_partitioning(P');
    rnd := random number(0,1);
    Δ := C' − C;
    if rnd < exp(−ΔC/T) then
      P := P'; C := C' ;
    else
      Nreject := Nreject + 1;
    end if
    if Nreject = threshold then
      P := P'; C := C'; Nreject := 0;
    end if
  until (deterioration's criterion reached);
  WP := P;
```

5.4.7 Stopping Criterion

Detecting the convergence of the SA method is a delicate matter. In practical implementations there is always a trade-off between solution quality and speed. Using the value of the temperature to determine the stopping criterion of the method may be inadequate for two reasons:

- A very low temperature leads to high CPU time.
- This high computational time may not be worth spending compared to the improvements made.

Based on the above observations, we assume that convergence is attained when no further progress is made during five consecutive runs of Improve.

5.5 Experimental Results

5.5.1 The Kernighan-Lin Improvement Method

As has been said in the third chapter, the Kernighan-Lin (KL) algorithm seeks to improve a given partition by moving vertices between subdomains in an effort to reduce the size of the edge cut. The subject of this section is to investigate whether KL is sensitive to the starting initial partition from the point of view of the achieved quality and from the point of view of the execution time. To this end, the KL algorithm is applied on different initial partitions provided by two greedy partitioning algorithms presented in the previous chapter, namely SG and SG_{profit}. In this experiment, our initial partitions were fed into the graph partitioning package CHACO [Hen93a] and refined using KL. However, the use of this improvement capability in CHACO is restricted to 8 subdomains at most. Figures 5.3, 5.4, 5.5, and 5.6 show the quality of the partitions in terms of edge cut produced by SG and SG_{profit} before and after application of KL, and the time invested during the refinement. All the experiments were performed on an Silicon Graphics with 128 Mbytes of memory and 230 MHz CPU. All times reported are in seconds.

	Initial Partitions		After Improvement			
N_{sub}	SG	SG_{profit}	SG + KL	t	SG_{profit} + KL	t
2	401	300	98	0.09	120	0.04
4	415	338	255	0.30	255	0.37
8	671	425	429	0.77	384	0.46

Figure 5.3: Sensitivity of the Kernighan-Lin improvement algorithm to the starting initial partition qualitywise and timewise using the graph HAMMOND.

Based on the obtained results, the following observations are made:

- SG_{profit} + KL produces better partitions than SG + KL for 10 out of 12 test cases. The only notable exceptions is the bisection case using the

N_{sub}	Initial Partitions		After Improvement			
	SG	SG_{profit}	SG + KL	t	SG_{profit} + KL	t
2	1104	573	270	0.62	202	0.41
4	1327	513	470	1.91	431	0.99
8	1610	829	875	5.9	789	3.88

Figure 5.4: Sensitivity of the Kernighan-Lin improvement algorithm to the starting initial partition qualitywise and timewise using the graph BIG.

N_{sub}	Initial Partitions		After Improvement			
	SG	SG_{profit}	SG + KL	t	SG_{profit} + KL	t
2	2713	853	1310	0.61	506	0.31
4	3102	1289	1694	2.67	1218	2.92
8	5408	2756	2823	9.06	2602	4.98

Figure 5.5: Sensitivity of the Kernighan-Lin refinement algorithm to the starting initial partition qualitywise and timewise using the graph NACA.

N_{sub}	Initial Partitions		After Improvement			
	SG	SG_{profit}	SG + KL	t	SG_{profit} + KL	t
2	1444	253	234	0.73	233	0.71
4	5898	796	1179	3.32	699	1.99
8	9018	2968	3264	22.69	2693	14.50

Figure 5.6: Sensitivity of the Kernighan-Lin improvement algorithm to the starting initial partition qualitywise and timewise using the graph SNECMA.

graph HAMMOND for which SG_{profit} coupled with KL does 18% worse than SG coupled with KL, and the quadrisection case using the same graph for which SG_{profit} + KL and SG + KL performs similarly.

- The difference in the quality of the partitions produced by SG_{profit} + KL and SG + KL is significant. On the average, this difference is as high as 21%. In some cases, the difference in quality is up to 41%, and 61%.
- The amount of time required by KL when coupled with SG is signifi-

cantly higher than the time it requires when coupled with SG_{profit}. On the average, KL requires 37% more time when combined with SG. The only exception is the quadrisection case using the graph BIG for which KL combined with SG_{profit} requires 9% more time.

- SG_{profit} produces better partitions than SG + KL for 8 out of 12. The difference in quality is up to 35%.

- Based on the above observations, the performance of the KL depends highly on the given initial partition.

5.5.2 Improvement of Partitions

The following experiments concern the improvements in quality in terms of edge cut achieved during the improvement phase after resorting to the heuristic MSA, and the amount of CPU time invested. The values of the parameters α, β, γ, and δ in the objective function $OF_{partitioning}$ are set to 0.5, 0.5, 0, and 0 respectively throughout the remainder of this dissertation.

	SG			SG_{profit}		
N_{sub}	$EC_{initial}$	EC_{opt}	t	$EC_{initial}$	EC_{opt}	t
16	1897	1502	1.06	1204	1166	0.43
32	2584	2059	1.51	2076	1870	0.83
64	3470	2782	2.08	2932	2741	1.28
128	4722	3925	3.54	4366	3920	2.75
256	6545	5535	6.24	6184	5481	5.25

Figure 5.7: Improvements made by our modified variant of the Simulated Annealing method using the graph NACA.

Three graphs of different sizes are used. Each graph has been decomposed using SG and SG_{profit}. The notations $EC_{initial}$ and EC_{opt} denote respectively the edge cut before and after improvements. Figures 5.7, 5.8, and 5.9 show the results obtained. When SG is used as an initial decomposer, we observe

	SG			SG_{profit}		
N_{sub}	$EC_{initial}$	EC_{opt}	t	$EC_{initial}$	EC_{opt}	t
16	8134	6781	1.35	7600	7160	0.65
32	11350	9678	2.96	10762	9987	1.95
64	15521	13156	3.91	13782	12684	3.71
128	20839	17737	8.01	19105	17455	6.66
256	28452	24136	14.02	25705	23577	11.11

Figure 5.8: Improvements made by our modified variant of the Simulated Annealing using the graph WHEEL.

	SG			SG_{profit}		
N_{sub}	$EC_{initial}$	EC_{opt}	t	$EC_{initial}$	EC_{opt}	t
16	13546	8702	4.83	8139	7722	1.41
32	17394	12269	6.88	9626	8936	2.48
64	22241	16023	9.55	13838	12748	4.04
128	28527	22248	12.73	20922	19750	6.85
256	37644	31507	19.24	31950	30082	11.61

Figure 5.9: Improvements made by our modified variant of the Simulated Annealing method using the graph SNECMA.

on the average an improvement of 18%, 15%, and 26% for NACA, WHEEL, and SNECMA. On the other hand, when MSA is combined with SG_{profit}, the improvement is on the average 8%, 7%, and 6% for the three graphs respectively. The time invested during the improvement phase is still reasonable. MSA requires less time when combined with SG_{profit} owing to the good initial partition. Finally, we conclude by saying that we have experienced no single case where MSA was unable to improve a given initial partition.

5.6 Comparison of Heuristic Techniques

The subject of this section is the quality of the partitions produced by our algorithm and those of TOP/DOMDEC [Far95b]. TOP/DOMDEC and our

multilevel partitioning algorithm (MLA) resort to graph reduction techniques to do the partitioning. This topic will be dealt with in detail in the next chapter. Briefly, MLA starts by contracting the size of the graph describing the mesh. Thereafter, a partition is found at the smallest graph followed by an optimization phase at each level of the reduction. MLA uses our new graph contraction procedure we term *gain vertex matching*. This contraction procedure merges two vertices provided the perimeter of the new formed supervertex is the smallest possible. As an optimizer, the MSA method is used at every intermediate level of the contraction. The software TOP/DOMDEC works with its own contraction procedure and uses Farhat's algorithm [Far88] to generate an initial partition. As improvement methods, we selected Tabu Search (TS) and Stochastic Evolution (SE) which are already integrated in the package. All the experiments were performed on an Silicon Graphic with 128 MBytes of memory and 230 MHz CPU. Since the three heuristics are stochastic, the quality of the decompositions varies from one run to another. Therefore, the values of the different metrics used in these experiments are the average values calculated for 10 trials. The objective function used in TOP/DOMDEC is fed with the same parameters as those presented in section 5.5.2.

5.6.1 Weight Imbalance

	MLA	TOP/DOMDEC/TS	TOP/DOMDEC/SE
N_{sub}	LBR	LBR	LBR
16	0.999	0.998	0.999
32	0.998	0.999	0.998
64	0.996	0.998	0.996
128	0.996	0.997	0.997
256	0.994	0.995	0.995

Figure 5.10: Load balancing ratio achieved with our multilevel algorithm and TOP/DOMDEC using the graph WHITAKER.

Figures 5.10, 5.11, and 5.12 give the load balancing ratio for the graphs WHITAKER, BIG, and NACA. The results indicate that both MLA and TOP/DOMDEC deliver partitions whose LBR are above 0.9. For large subdomains, this ratio seems to decrease very slowly.

	MLA	TOP/DOMDEC/TS	TOP/DOMDEC/SE
N_{sub}	LBR	LBR	LBR
16	0.998	0.999	0.998
32	0.997	0.999	0.997
64	0.997	0.999	0.998
128	0.996	0.997	0.997
256	0.994	0.996	0.995

Figure 5.11: Load balancing ratio achieved with our multilevel algorithm and TOP/DOMDEC using the graph BIG.

	MLA	TOP/DOMDEC/TS	TOP/DOMDEC/SE
N_{sub}	LBR	LBR	LBR
16	0.998	0.999	0.998
32	0.997	0.999	0.997
64	0.997	0.999	0.998
128	0.996	0.997	0.997
256	0.995	0.995	0.994

Figure 5.12: Load balancing ratio achieved with our multilevel algorithm and TOP/DOMDEC using the graph NACA.

5.6.2 Boundary Size and Subdomain Graph

The TOP/DOMDEC package gives the number of boundary vertices as a metric to quantify the volume of communications which we adopt as well in this experiment. Figures 5.13, 5.14, 5.15, and 5.16 show the quality of the partitions in terms of the total number of boundary vertices ($|BV|$) and the number of edges in the subdomain graph.

Based on the obtained results, we observe that MLA produces better decompositions compared to TOP/DOMDEC. A comparison of the number of boundary vertices reveals that on the average, MLA does 17%, 24%, 21% and 8% better than TOP/DOMDEC/TS, and 29%, 34%, 29% 13% better than TOP/DOMDEC/SE. In addition, MLA generates subdomains graphs whose number of edges is on average 7%, and 12% less than those produced by TOP/DOMDEC/TS and TOP/DOMDEC/SE respectively. In the cases where TOP/DOMDEC outperforms MLA in terms of number of edges in the subdomain graph, the improvements is only marginally better. As all the different heuristics depend on many parameters, changing the values of the parameters might modify the hierarchy between these heuristics, therefore, we would not draw firm conclusions about their merit.

5.6.3 Decomposition Time

Figures 5.17, 5.18, and 5.19 give the partitioning time for both MLA and TOP/DOMDEC for the graphs WHITAKER, NACA, and BIG.

As far as the running time is concerned, we notice that it increases as the number of subdomain gets higher. This increase is due to an increase in the number of boundary vertices which has an impact on the the number of perturbations performed. From the relevant figures, we see that MLA requires

	MLA		TOP/DOMDEC/TS		TOP/DOMDEC/SE	
N_{sub}	$\|BV\|$	$\|E(S)\|$	$\|BV\|$	$\|E(S)\|$	$\|BV\|$	$\|E(S)\|$
16	163	36	228	35	280	35
32	254	69	368	85	422	91
64	398	148	467	156	544	169
128	603	317	666	330	762	361
256	889	669	916	715	1079	814

Figure 5.13: Comparing our multilevel algorithm and TOP/DOMDEC in terms of boundary size and number of edges in subdomain graphs using the graph HAMMOND.

N_{sub}	MLA $\|BV\|$	MLA $\|E(S)\|$	TOP/DOMDEC/TS $\|BV\|$	TOP/DOMDEC/TS $\|E(S)\|$	TOP/DOMDEC/SE $\|BV\|$	TOP/DOMDEC/SE $\|E(S)\|$
16	273	35	461	44	562	43
32	443	73	649	77	755	80
64	712	149	903	164	1051	168
128	1080	307	1311	342	1481	364
256	1613	667	1810	695	2036	755

Figure 5.14: Comparing our multilevel algorithm and TOP/DOMDEC in terms of boundary size and number of edges in subdomain graphs using the graph BIG.

N_{sub}	MLA $\|BV\|$	MLA $\|E(S)\|$	TOP/DOMDEC/TS $\|BV\|$	TOP/DOMDEC/TS $\|E(S)\|$	TOP/DOMDEC/SE $\|BV\|$	TOP/DOMDEC/SE $\|E(S)\|$
16	288	40	440	42	483	44
32	400	83	598	94	677	94
64	643	176	778	186	862	194
128	943	367	1081	388	1203	421
256	1344	737	1459	781	1650	864

Figure 5.15: Comparing our multilevel algorithm and TOP/DOMDEC in terms of boundary size and number of edges in subdomain graphs using the graph NACA.

N_{sub}	MLA $\|BV\|$	MLA $\|E(S)\|$	TOP/DOMDEC/TS $\|BV\|$	TOP/DOMDEC/TS $\|E(S)\|$	TOP/DOMDEC/SE $\|BV\|$	TOP/DOMDEC/SE $\|E(S)\|$
16	303	31	341	35	351	34
32	454	74	508	73	528	76
64	678	163	728	168	775	177
128	968	342	1051	352	1139	384
256	1382	702	1433	724	1593	801

Figure 5.16: Comparing our multilevel algorithm and TOP/DOMDEC in terms of boundary size and number of edges in subdomain graphs using the graph WHITAKER.

N_{sub}	MLA t	TOP/DOMDEC/TS t	TOP/DOMDEC/SE t
16	3.62	2.88	4.64
32	5.38	5.53	7.48
64	6.34	6.72	11.93
128	9.31	14.13	21.87
256	18.43	25.37	37.17

Figure 5.17: Comparing our multilevel algorithm and TOP/DOMDEC in terms of CPU time using the graph WHITAKER.

N_{sub}	MLA t	TOP/DOMDEC/TS t	TOP/DOMDEC/SE t
16	16.95	15.48	16.95
32	22.17	23.14	23.60
64	30.98	31.11	40.35
128	35.45	39.23	49.14
256	41.21	48.10	59.10

Figure 5.18: Comparing our multilevel algorithm and TOP/DOMDEC in terms of CPU time using the graph NACA.

N_{sub}	MLA t	TOP/DOMDEC/TS t	TOP/DOMDEC/SE t
16	4.45	4.13	5.13
32	6.67	6.34	7.89
64	8.55	9.10	9.20
128	14.19	16.23	19.21
256	23.61	27.10	35.89

Figure 5.19: Comparing our multilevel algorithm and TOP/DOMDEC in terms of CPU time using the graph BIG.

the least amount of time than TOP/DOMDEC/TS and TOP/DOMDEC/SE for large number of subdomains. Looking at the runtimes of MLA and TOP/DOMDEC/TS, we observe that on the average MLA requires 12%

less time than TOP/DOMDEC/TS in 11 out of 15 partitions, and 10% more time than TOP/DOMDEC/TS in 4 partitions. Finally, we notice that TOP/DOMDEC/SE requires on the average 28% more time than MLA.

5.7 Concluding Remarks

In this chapter, a variant of the Simulated Annealing method for improving the quality of a given partition has been presented. This variant combines an iterative improvement algorithm and a deterioration procedure to conduct the search. Experimental results have shown that this variant delivers better decompositions compared to the versions of Tabu Search and Stochastic Evolution methods included in the mesh partitioning package TOP/DOMDEC, while requiring in most cases the least amount of CPU time. These stochastic search strategies include a number of adjustable parameters in their definition, which is in fact an obstacle to performing a fair comparative study, since the efficiency of any search method may usually be dramatically affected by the choice of parameter values, therefore one should refrain from drawing firm conclusions about the merits of these heuristics. Finally, the refinement of the partitions provided by SG and SG_{profit} reveals that when KL is combined with the latter, high quality partition are obtained at a low cost.

Chapter 6

Graph Coarsening

The success of a decomposition algorithm resides in its short runtime and high quality partition. The search space over which the objective function $OF_{partitioning}$ is defined is a discrete one, but very large. For large graphs, the perturbation which is defined as the transfer of a boundary vertex from its subdomain to a neighboring one risks to slow down the improvement phase owing to the size of the search space. Moreover, since the displacements affect only the boundaries of subdomains, they will have, in a large mesh, only a marginal effect on the improvement of the partition quality. This fact motivates the use of *multilevel partitioning algorithms* These algorithms include the following three steps:

- Coarsen the original graph using a coarsening procedure. During this step, a sequence of smaller graphs are constructed.
- Partition the smallest graph.
- The partition found at the smallest graph is projected back to the finer graph (original graph) by going through an improvement phase at each intermediate level.

Thus, coarsening allows the displacement of packets of vertices instead of individual vertices at every intermediate level of the coarsening. At the original graph, only single vertices are considered.

In this chapter, we experiment with these three steps and their impact on the quality of partitions. Two new coarsening schemes are introduced. Their performance is compared to the popular *heavy edge matching* coarsening heuristic. In addition, a comparative experimental evaluation of two different types of coarsening called *recursive coarsening* and *direct coarsening* is presented. Finally, our multilevel partitioning algorithm is compared to three of the most established partitioning algorithms in the field.

6.1 Strategy

Recently, a new class of algorithms called *multilevel partitioning algorithms* was introduced by Bui & Jones [Bui93] and Hendrickson & Leland [Hen93a] [Hen93b] and further adopted in [Kar95a][Kar95b][Kar95c][Van95b] [Wal95a] [Wal97]. These algorithms resort to graph coarsening in order to partition a graph. This strategy proved to be very efficient especially when the size of the graph is large. The basic idea is quite simple. Coarsening aims at building a sequence of smaller graphs $C^1(G)$, $C^2(G)$, $\cdots$ $C^k(G)$ from the original graph C using a coarsening scheme such that $|G| > |C^1(G)| > |C^2(G)| > \cdots > |C^k(G)|$. Thereafter, a partition of the smallest graph $C^k(G)$ is computed, and finally this coarse partition is projected back towards the original graph, by proceeding with an improvement phase at each step. Given a graph G, a coarser graph $C(G)$ can be obtained by merging adjacent vertices. Thus, the edge linking two vertices is collapsed and a supervertex consisting of these two vertices is formed. Supervertices are weighted by the sum of the weights of the merged vertices. In order to preserve the connectivity information, a superedge between two supervertices u and v is the union of the edges between the vertices of u and those of v. The weight of the superedge is the sum of the weights of the edges it represents. Thus, the quality of the partition in terms of edge cut of the coarser graph is equal to the edge cut of the same partition in the original graph. This type of coarsening

will be referred to as *recursive coarsening* and is widely adopted by most multilevel partitioning algorithm. After one coarsening level, we have roughly $|V(C(G))| = |V(G)|/2$.

6.2 Coarsening Schemes

6.2.1 Coarsening and Matching

Given a graph $G = (V(G), E(G))$, a coarser graph can be obtained by collapsing adjacent vertices. Thus, the edge joining two vertices is collapsed and a supervertex consisting of these two vertices is generated. This edge collapsing can be formally defined in terms of matchings.

Algorithm 8 Pseudo-code of a typical recursive coarsening procedure.

Procedure GraphCoarsening (**input** G: graph, **output** $C(G)$: graph);
  $V(C(G)) := \emptyset$;
  $E(C(G)) := \emptyset$;
  for $v := 1$ to $|V(G)|$ do
    $label[v] := unmarked$;
  end for
  repeat
    choose an unmarked $v_i \in V(G)$;
    if $\exists\ v_j \in V(G))$ such that: v_j is unmarked and $\langle v_i, v_j \rangle \in E(G)$ **then**
      form supervertex $w = \{v_i, v_j\}$;
      $label[v_i] := marked$;
      $label[v_j] := marked$;
    else
      form supervetex $w = \{v_i\}$;
      $label[v_i] := marked$;
    end if
    $V(C(G)) := V(C(G)) \cup \{w\}$;
    $E(C(G)) := E(C(G)) \cup \{\langle w, y\rangle \,|\, y \in V(C(G)), \exists\ v' \in w, v'' \in y :$
        $\langle v', v'' \rangle \in E(G)\}$;
  until all vertices of $V(G)$ are marked;

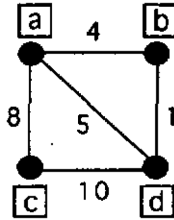
Consequently, a coarser graph $C(G)$ is constructed from G by finding a matching of the elements of $V(G)$ and collapsing the matched vertices into supervertices. The unmatched vertices are simply copied over to $C(G)$. A pseudo-code for a typical sequential coarsening procedure is given in Algorithm 8.

6.2.2 Heavy Edge Matching

Heavy edge matching (HEM) was first introduced in [Man92], and further studied in [Kar95a]. This scheme aims at generating a coarse graph in which vertices connected by a "heavy" edge, i.e. an edge with a high weight, are collapsed to form supervertices. The vertices are visited in random order. An unmerged vertex v_i is collapsed with a vertex v_j provided that the weight of the edge $\langle v_i, v_j \rangle$ is maximum over all edges incident to v_i. Thus, "heavy" edges are made intra-supervertex edges, ensuring that most of the inter-supervertex edges have relatively small weights. Experimental results conducted in [Kar95a] have shown that HEM is an excellent coarsening scheme.

6.2.3 Gain Vertex Matching

While HEM uses the criterion of profitable edges to collapse vertices, our new matching scheme which we term *gain vertex matching* (GVM) exploits the criterion of profitable vertices to do the matching. Vertices are again visited in random manner. An unmerged vertex v_i is matched with one of its unmatched neighbors that lead to the smallest weight sum over the edges incident to the supervertex. The main idea behind GVM is to compute a matching that attempts to generate coarse graphs in which edges joining the different supervertices have the smallest weights. The advantage of using GVM against HEM is better illustrated through the example shown in Figure 6.1. This example shows a graph of four vertices together with the weight of the edges. Let us assume that all the four vertices are still unmatched. To find a mate for the vertex a for instance, HEM will choose the vertex c,



HEM

w(<a,b>)=4

w(<a,c>)=8 max

w(<a,d>)=5

GVM

Collapsing a and b:Boundary{a,b}=14 min

Collapsing a and c:Boundary{a,c}=19

Collapsing a and d:Boundary{a,d}=23

Figure 6.1: A simple example illustrating the behavior of the heavy edge and gain vertex matching schemes.

whereas GVM will choose the vertex b instead. The resulting supervertex produced by HEM will have its boundary equal to 19, while the one produced by GVM will have its boundary equal to 14.

6.2.4 Closest Vertex Matching

The closest vertex matching (CVM) is a coarsening scheme based on a geometric criterion: it uses the coordinates of the vertices to compute the matching. Vertices are again traversed in random order. A yet unmatched

vertex v_i is matched with the unmatched neighbor which is closest in geometric space. The resulting supervertex is then assigned the coordinates of the center of gravity of the collapsed vertices. This coarsening procedure has been first introduced by Boman and Hendrickson [Bom96] in the context of reducing the envelope of sparse matrices.

6.3 Experimental Results

All the experiments conducted in this chapter were performed on an Silicon Graphics with *128* Mbytes of memory and *230* MHz CPU. *All* times reported are in seconds.

6.3.1 Quality of Matching

In this section, we present results regarding the quality of the matching produced by the three coarsening schemes for the graphs BIG, NACA, T60K$_{nd}$ and WHITAKER. All the coarsening schemes do 5 coarsening levels. Figures 6.2, 6.3, 6.4, and 6.5 show the matching ratio $MR = \frac{\lfloor |V(C^k)|/2 \rfloor}{|V(C^{k+1})|}$ at successive coarsening levels for HEM, CVM and CVM.

In terms of the quality of the matching, we observe that GVM produces the highest matching ratio compared to the other two coarsening schemes which makes it the clear matching winner. On the average, HEM, CVM are 4.7%, and 4.5% below the maximum matching respectively, whereas the matching generated by GVM is only 1.4% below the maximum. In addition, we observe that the matching ratio for the three coarsening schemes decreases at successive coarsening levels. We notice also, that this decrease is slower when GVM is used.

6.3.2 Topology of the Coarse Graph

The experiments conducted in the previous section showed the superiority of GVM compared to the other methods in terms of the quality of matching.

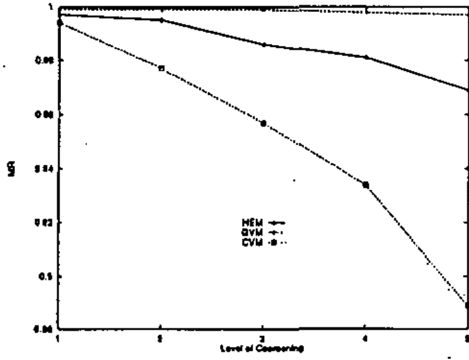


Figure 6.2: Performance of various coarsening schemes in terms of matching ratio using the graph BIG.

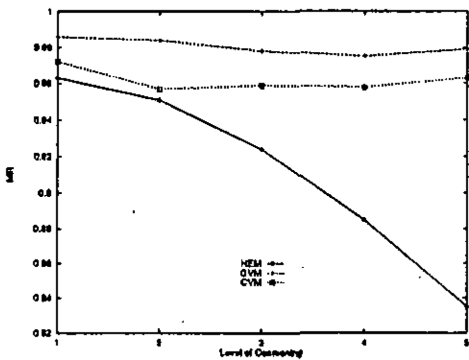


Figure 6.3: Performance of various coarsening schemes in terms of matching ratio using the graph NACA.

As each coarsening scheme is distinguished from the others by the criterion it exploits to merge vertices, the structure of their corresponding coarse graphs

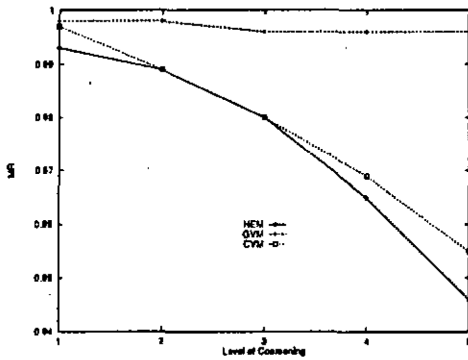


Figure 6.4: Performance of various coarsening schemes in terms of matching ratio using the graph T60K.

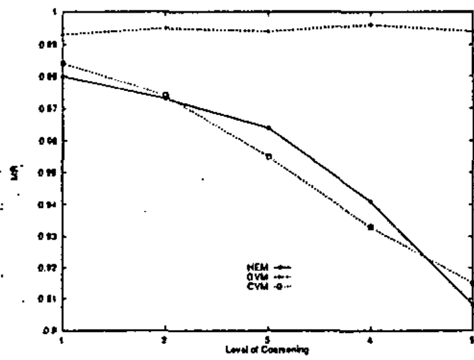


Figure 6.5: Performance of various coarsening schemes in terms of matching ratio using the graph WHITAKER.

Graphs	HEM	
	$\|V(C^5(G))\|$	$\|E(C^5(G))\|$
BIG	523	1415
NACA	904	2813
T60K	1084	2995
WHITAKER	389	1040

Figure 6.6: Number of vertices and edges of four coarse graphs produced by the heavy matching coarsening scheme.

Graphs	GVM	
	$\|V(C^5(G))\|$	$\|E(C^5(G))\|$
BIG	490	1291
NACA	625	2153
T60K	967	2758
WHITAKER	314	888

Figure 6.7: Number of vertices and edges of four coarse graphs produced by the gain vertex matching scheme.

is obviously different. In this section, we take a closer look at the topology of some coarse graphs produced by the three coarsening schemes. Figures 6.6, 6.7, and 6.8 show the number of vertices and edges of four coarse graphs. Figures 6.9, 6.10, and 6.11 provide information regarding the average vertex weight, average vertex degree, and average edge weight of the same four coarse graphs.

A number of interesting observations are summarized below:

- GVM produces coarse graphs having on average 16% and 15% less vertices than those of HEM and CVM respectively. HEM and CVM need to do extra levels of coarsening in order to obtain coarse graphs having less number of vertices than those produced by GVM.
- GVM produces coarse graphs having on average 13% and 25% less edges than those of HEM and CVM. The only exception is the graph

Graphs	CVM	
	$\|V(C^5(G))\|$	$\|E(C^5(G))\|$
BIG	631	1647
NACA	688	4063
T60K	1066	2746
WHITAKER	391	967

Figure 6.8: Number of vertices and edges of four coarse graphs produced by the closest vertex matching scheme.

CoarseGraphs	HEM		
	$\overline{W(V(C^5(G)))}$	$\overline{\deg}$	$\overline{W(E(C^5(G)))}$
BIG	29.83	2.70	8.46
NACA	20.07	3.11	14.87
T60K	28.20	2.76	8.29
WHITAKER	25.19	2.67	7.93

Figure 6.9: Topology of four coarse graphs produced by the heavy edge matching method.

CoarseGraphs	GVM		
	$\overline{W(V(C^5(G)))}$	$\overline{\deg}$	$\overline{W(E(C^5(G)))}$
BIG	31.84	2.63	9.11
NACA	29.03	3.44	19.97
T60K	31.61	2.85	8.71
WHITAKER	31.21	2.82	8.83

Figure 6.10: Topology of four coarse graphs produced by the gain vertex matching method.

T60K whose coarse graph when using CVM has 1% less edges than GVM.

- On the average, the coarse graphs delivered by GVM have supervertices whose weight vary less than those delivered by HEM and CVM. This

CoarseGraphs	CVM		
	$\overline{W(V(C^5(G)))}$	$\overline{\deg}$	$\overline{W(E(C^5(G)))}$
BIG	24.73	2.61	13.52
NACA	26.37	5.90	17.67
T60K	28.67	2.57	15.75
WHITAKER	25.06	2.47	13.78

Figure 6.11: Topology of four coarse graphs produced by the closest vertex matching method.

is not surprising, since GVM produces larger matchings compared to the other two methods.

- CVM delivers coarse graphs whose supervertices have an average degree which is 12% below that obtained with HEM, and 7% below that obtained by GVM. The only exception is the coarse graph NACA whose average degree obtained by CVM is 47% and 42% higher than that of to HEM and GVM.

- CVM produces coarse graphs whose average edge weight is 47% and 45% above those obtained by HEM and GVM respectively. The only exception is the graph NACA for which the average edge weights produced by GVM is 12% higher than that of CVM. This result indicates that the strategy of using CVM as a coarsening scheme followed for instance by SG at the lowest level of the coarsening, leads to partitions having larger edge cut than those of HEM and GVM.

- There are several advantages associated with the larger matching ratio produced by GVM. First, the graphs get smaller more quickly with GVM compared to HEM and CVM using the same number of coarsening levels. Second, the heterogeneity produced by unmatched vertices is reduced. Third, the coarse graphs produced by GVM tend to lead to smaller partitioning runtimes than those produced by HEM and CVM. For example, the coarse graph BRACK takes 1.67 s to partition when

HEM is combined with SG, and only 0.16 s when GVM is selected.

6.3.3 Quality of the Partitions

The questions to be answered in this section is how much the edge cut of the coarse graph exceeds that of the original graph. In addition we are interested to see the impact of different coarsening algorithms on the final quality of the partition in terms of the edge cut. We evaluated the efficiency of the three coarsening procedures using the graphs $T60K_{nd}$, T60K, $NACA_{nd}$, and $NACA$. The multilevel graph partitioning algorithm used in this experiment starts by contracting the graph using either coarsening technique. Thereafter, the algorithm SG is used to deliver an initial partition. Finally, the procedure Improve presented in the previous chapter is used as an optimizer at each intermediate level, with the slight modification that it terminates when a certain number of iterations are done (we took 5.10^5 iterations). Thus, all the coarsening schemes are subject to the same amount of CPU time during the improvement phase. The notations $EC0$ denotes the size of the edge cut of the original graph, whereas $EC5$ denotes the size of the edge cut at the coarse graph. Both graphs have been decomposed using SG. Finally, EC_{opt} denotes the size of the edge cut after resorting to improvement at every intermediate level of the coarsening. Figures 6.12, 6.13, 6.14, and 6.15 report the results of our experiments.

Based on the examination of these results, the following observations are made:

- On the average, the edge cut produced at the coarse graph is 27%, 26%, and 45% worse than that of the original graph for HEM, GVM, and CVM respectively.

- After improvement, HEM produces the best results for six partitions, GVM for twelve partitions, and CVM for two partitions. The difference in quality between GVM and HEM is within 7% in favor of the

		HEM		GVM		CVM	
N_{sub}	$\|EC0\|$	$\|EC5\|$	$\|EC_{opt}\|$	$\|EC5\|$	$\|EC_{opt}\|$	$\|EC5\|$	$\|EC_{opt}\|$
20	2559	3784	2289	3654	2335	4200	2698
40	4125	5424	3470	5488	3648	6961	4411
60	5125	7542	4531	6951	4540	9338	4977
80	5950	8927	5354	8210	5248	10750	6095
100	6656	9304	5901	9484	5986	12078	6962

Figure 6.12: Quality of the partitions produced by the various matching schemes using the graph $T60K_{nd}$.

		HEM		GVM		CVM	
N_{sub}	$\|EC0\|$	$\|EC5\|$	$\|EC_{opt}\|$	$EC5$	EC_{opt}	$\|EC5\|$	$\|EC_{opt}\|$
20	9585	13923	8984	12769	8485	21840	10347
40	15539	20850	13499	20675	13592	32755	16100
60	19515	26669	17453	25101	17215	41188	19988
80	22879	31463	20538	29560	20310	50689	24083
100	25416	35187	23287	34067	22861	56853	27771

Figure 6.13: Quality of the partitions produced by the various matching schemes using the graph T60K.

		HEM		GVM		CVM	
N_{sub}	$\|EC0\|$	$\|EC5\|$	$\|EC_{opt}\|$	$\|EC5\|$	$\|EC_{opt}\|$	$\|EC5\|$	$\|EC_{opt}\|$
20	2055	2401	1296	2613	1219	3284	1351
40	2737	3537	1986	3418	1989	4177	1978
60	3316	4218	2487	4157	2448	5069	2523
80	3829	4816	2950	4776	2915	5879	3004
100	4154	5504	3336	5216	3320	6338	3393

Figure 6.14: Quality of the partitions produced by the various matching schemes using the graph $NACA_{nd}$.

N_{sub}	$\|EC0\|$	HEM		GVM		CVM	
		$\|EC5\|$	$\|EC_{opt}\|$	$\|EC5\|$	$\|EC_{opt}\|$	$\|EC5\|$	$\|EC_{opt}\|$
20	7975	10645	5870	10572	5475	14747	5808
40	10635	14545	8317	14526	7988	18933	8533
60	12545	16618	10198	17916	10487	23497	10438
80	14262	19301	12083	20530	11933	26126	11732
106	15737	21140	13446	22951	13181	29863	13413

Figure 6.15: Quality of the partitions produced by the various matching schemes using the graph NACA.

former. In the cases where HEM does better than GVM, the difference in quality is within 5%. Finally, we notice that the quality of the edge cut when CVM is used as a coarsening scheme is on average 8% worse than of HEM and CVM.

- Looking at the graph NACA in Figure 6.15, the initial partition provided by CVM for 80 subdomains is 21% and 26% worse than that of CVM and HEM respectively. However, after refinement, the final partition of CVM in terms of edge cut is 2% and 3% better than that of GVM and HEM respectively. This particular example shows that when an initial partition is better than another one, its optimized version is not necessarily better than the optimized version of that other one.

6.4 Influence of the Initial Decomposer

This section is about the issue of whether a good initial partitioning algorithm is worth using at the coarse graph in order to speed up the partitioning process.

As pointed out earlier, a number of algorithms can be used to partition the coarse graph. We evaluated the performance of our multilevel algorithm from the point of view of quality and time using SG and SG_{profit} as initial decomposers. In the rest of this section, we present results for the graphs

N_{sub}	SG $\lvert EC3\rvert$	SG $\lvert EC_{opt}\rvert$	SG $t(s)$	SG_{profit} $\lvert EC3\rvert$	SG_{profit} $\lvert EC_{opt}\rvert$	SG_{profit} $t(s)$
20	55208	19469	22.62	23355	16676	13.95
40	64208	27534	30.55	34278	24518	19.41
60	72151	31623	37.10	42011	29869	24.72
80	79206	38548	44.08	47244	34173	31.05
100	85093	41402	49.28	52897	38051	36.26

Figure 6.16: Impact of the initial decomposition on the overall performance of our multilevel algorithm using the graph BRACK.

N_{sub}	SG $\lvert EC3\rvert$	SG $\lvert EC_{opt}\rvert$	SG $t(s)$	SG_{profit} $\lvert EC3\rvert$	SG_{profit} $\lvert EC_{opt}\rvert$	SG_{profit} $t(s)$
20	19945	9492	9.3	7154	5146	4.22
40	23581	10973	9.94	12246	8702	6.55
60	28192	14337	12.65	16407	11751	9.06
80	30482	15904	14.18	20144	14341	9.87
100	32848	18236	15.25	24024	17083	12.45

Figure 6.17: Impact of the initial decomposition on the overall performance of our multilevel algorithm using the graph SNECMA.

BRACK, WHEEL, and SNECMA. The graph BRACK was contracted using HEM, SNECMA was contracted using GVM, and finally WHEEL was contracted using CVM. Improvement is carried out at each level of the contraction using MSA. Three level of contractions were performed. The notations $EC3$, and EC_{opt} denote the size of the edge cut at the coarse graph and after improvement respectively. Based on the experimental results shown in Figures 6.16, 6.17, and 6.18 the following observations are made:

- The edge cut for the three coarse graphs varies quite significantly depending on the algorithm used. On the average, SG_{profit} produces partitioos of the coarse grapbs with an edge cut which is 45%, 43%, and 33% better than that of SG for BRACK, SNECMA, and WHEEL respectively. After improvement, the difference in quality is reduced to

N_{sub}	SG			SG_{profit}		
	$\|EC3\|$	$\|EC_{opt}\|$	$t(s)$	$\|EC3\|$	$\|EC_{opt}\|$	$t(s)$
20	20817	7671	3.49	14153	7678	3.52
40	27364	10417	8.66	18602	10568	6.03
60	32685	13069	9.7	21795	12343	8.82
80	38292	14429	11.70	24919	14181	7.79
100	42390	16472	11.46	27458	15730	9.28

Figure 6.18: Impact of the initial decomposition on the overall performance of our multilevel algorithm using the graph WHEEL.

10%, 20%, and 4%. This comparison is of interest wbenever, for the sake of reduction of the partitioning time, no improvement is invoked.

- The time required during the improvement phase varies significantly. We observe that MSA requires less time when coupled with SG_{profit} than with SG in all cases but one. On the average, MSA requires 33%, 32%, and 23% less time when combined with SG_{profit} than with SG for BRACK, SNECMA, and WHEEL respectively. This seems quite reasonable giving the fact that MSA converges rapidly owing to the good quality of the projected partitions.

6.5 The Advantage of the Coarsening

The contraction schemes introduced in this chapter have been tested on a wide range of graphs. For every test case, we notice an improvement of the quality of the partition together with a reduction of the CPU time. Figures 6.19 and 6.20 show the gain afforded by the coarsening from the point of view of quality and computation time using the graph BRACK and performing three level of contractions using HEM.

In Figure 6.19, we represent the edge cut of the partitions obtained with and without coarsening as a function of the number of subdomains. We notice that coarsening leads to better partitions. For this particular example, the

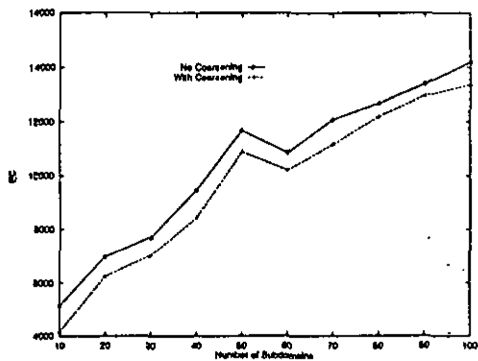


Figure 6.19: Advantage of the coarsening on the edge cut using the graph BRACK.

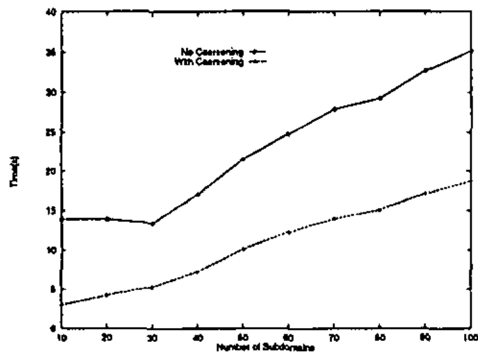


Figure 6.20: Advantage of the coarsening on the CPU time using the graph BRACK.

improvement afforded when resorting to contraction ranges between 3% and 19%. Coarsening seems to be an excellent way used to get better partitions because of the following three facts:

1. Coarsening leads to an important reduction of the search space.

2. A consequence of the above property is that packets of vertices are moved to neighboring subdomains rather than single vertices. Thus, only a small number of transfer leads to a significant edge cut reductions.

3. Improvement is carried out in a more efficient manner owing to the variable size of packet of vertices moved at each coarsening level.

In Figure 6.20, we represent the CPU time of different partitions with and without coarsening as a function of the number of subdomains. We see that coarsening speeds up the partitioning process. In this example, coarsening is on the average twice as fast for this particular choice of the parameters.

6.6 Direct Coarsening vs Recursive Coarsening

*Implementing a contraction procedure requires that the level of the coars*ening and the size of the supervertices should be determined a priori. In previous experiments, coarsening was performed by specifying the desired level of the contraction with the constraint that each supervertex of the new coarser graph is the result of collapsing at most two vertices of the graph of the previous level. Thus, a coarsening on three levels reduces the size of the original graph by at most a factor of 2^3. This type of coarsening was previously termed recursive coarsening. Another way to reduce the size of the *original graph by approximately the same factor would* be to use one level of contraction where each supervertex having at most 2^3 vertices. This type

of coarsening will be referred to as *direct coarsening*. As the size of the supervertices differ whether direct or recursive coarsening is performed, we will use the term supervertex in case of recursive coarsening, and use the term *cluster* in case of direct coarsening. The strategy followed in direct coarsening is similar to the strategy used to get an initial partition using greedy partitioning algorithms developed in Chapter 4. The algorithm starts by choosing the very first vertex satisfying the minimal degree criterion. Thereafter, the cluster is constructed by iteratively adding adjacent vertices of the graph until the required size of the cluster is reached. However, it may occur that during the process of building a particular cluster, the algorithm fails to detect additional adjacent vertices and the resulting cluster is smaller than it should be. If such situation arises, the cluster is taken as it is and the algorithms simply moves on to the construction of the next cluster. The pseudo-code of the greedy coarsening algorithm is given in Algorithm 9. This direct coarsening algorithm exploits mainly the adjacency information of the graph. The algorithm can easily be modified to build clusters according to the criteria used in the partitioning algorithms SG_{profit}, SG_{sh}, and SG_{dist}.
We compare the performance of both direct and recursive coarsening using the graphs SNECMA, T60K, BRACK, and NACA. In all our experiments, the load balance ratio after improvement never dropped below 0.995, therefore we will not include this metric in our results. In this experiment, the multilevel partitioning algorithm ususes GVM as a coarsening scheme either recursively or directly, SG_{profit} as an initial decomposer, and MSA as an optimizer. The procedure Improve used in MSA assumes that a locally optimal partition is reached when the rate of acceptance is below 5%, whereas the convergence criterion of MSA is attained when no further progress is observed during 5 consecutive runs of the procedure Improve. In the case of direct coarsening, MSA is applied only twice (i.e. at the coarse graph and at the finer graph). The notation EC_x^y denotes the edge cut after improvement, x being the number of levels of contraction performed, y the maximum size

Algorithm 9 Pseudo-code of the greedy direct coarsening algorithm.

```
Procedure Direct(input G:graph,input size:integer,output C(G):graph);
  V(C(G)) := ∅;
  E(C(G)) := ∅;
  U := G; /* U denotes the graph with the unassigned vertices */
  cl:= 0;
  repeat
    cl := cl + 1 ;
    if cl = 1 then
      choose a vertex v ∈ V(U) with minimal degree;
    else
      choose a vertex v ∈ V_bdry(U);
    end if
    V(cluster_cl) := {v};
    V(U) := V(U) − {v};
    stop:= false;
    repeat
      choose a vertex v ∈ V_bdry(U) such that:
        ∃ ⟨v, w⟩ ∈ E(G): w ∈ V(cluster_cl);
      if such w does not exist then
        stop:= true;
      else
        AddVertex(cluster_cl,w);
        RemoveVertex(U,w);
        /* remove from U vertex w with its incidents edges */
      end if
    until (|cluster_cl| = size or stop);
    V(C(G)) := V(C(G)) ∪ {cluster_cl};
    E(G(G)) := E(C(G)) ∪ {⟨cluster_cl, y⟩ | y ∈ V(C(G)),
        ∃ v' ∈ cluster_cl, v'' ∈ y : ⟨v', v''⟩ ∈ E(G)};
  until V(U) := ∅;
```

N_{sub}	recursive coarsening				direct coarsening			
	$\lvert EC_2^2 \rvert$	t	$\lvert EC_3^2 \rvert$	t	$\lvert EC_1^4 \rvert$	t	$\lvert EC_1^8 \rvert$	t
32	23061	25.69	22278	19.20	22600	22.11	21042	13.04
64	31079	24.56	30634	22.95	31298	26.67	31811	25.99

Figure 6.21: Performance of direct coarsening and recursive coarsening using the graph BRACK: two and three level of contractions vs one level.

N_{sub}	recursive coarsening				direct coarsening			
	$\lvert EC_4^2 \rvert$	t	$\lvert EC_5^2 \rvert$	t	$\lvert EC_1^{16} \rvert$	t	$\lvert EC_1^{32} \rvert$	t
32	21597	20.09	21311	20.79	21404	18.42	22302	20.63
64	30977	24.28	31287	20.21	31914	22.20	32207	20.92

Figure 6.22: Performance of direct coarsening and recursive coarsening using the graph BRACK: four and five level of contractions vs one level.

N_{sub}	recursive coarsening				direct coarsening			
	$\lvert EC_2^2 \rvert$	t	$\lvert EC_3^2 \rvert$	t	$\lvert EC_1^4 \rvert$	t	$\lvert EC_1^8 \rvert$	t
32	7604	8.61	7061	9.50	7976	8.39	7235	7.15
64	12622	12.98	12192	12.53	12490	11.02	12118	11.87

Figure 6.23: Performance of direct coarsening and recursive coarsening using the graph SNECMA: two and three level of contractions vs one level.

of either supervertices or clusters. This benchmark comprises a total of 32 test cases.

From Figure 6.21 through Figure 6.30, we observe that 22 out of 32

N_{sub}	recursive coarsening				direct coarsening			
	$\lvert EC_4^2 \rvert$	t	$\lvert EC_5^2 \rvert$	t	$\lvert EC_1^{16} \rvert$	t	$\lvert EC_1^{32} \rvert$	t
32	7741	12.81	8068	12.26	7022	9.36	7734	10.67
64	12048	17.07	12536	18.27	12299	12.50	12361	12.30

Figure 6.24: Performance of direct coarsening and recursive coarsening using the graph SNECMA: four and five level of contractions vs one level.

N_{sub}	recursive coarsening				direct coarsening			
	$\|EC_2^2\|$	t	$\|EC_3^2\|$	t	$\|EC_1^4\|$	t	$\|EC_1^8\|$	t
32	15376	13.64	12621	13.61	12966	16.34	12005	13.18
64	21125	18.10	19404	18.25	18460	21.13	17492	17.06

Figure 6.25: Performance of direct coarsening and recursive coarsening using the graph T60K: two and three level contractions vs one level.

N_{sub}	recursive coarsening				direct coarsening			
	$\|EC_4^2\|$	t	$\|EC_5^2\|$	t	$\|EC_1^{16}\|$	t	$\|EC_1^{32}\|$	t
32	13228	14.37	11936	14.55	11781	12.50	11856	13.05
64	18240	19.17	18015	19.60	17848	16.50	17617	16.20

Figure 6.26: Performance of direct coarsening and recursive coarsening using the graph T60K: four and five level of contractions vs one level.

N_{sub}	recursive Coarsening				direct Coarsening			
	$\|EC_2^2\|$	t	$\|EC_3^2\|$	t	$\|EC_1^4\|$	t	$\|EC_1^8\|$	t
32	7280	7.41	6587	7.44	7016	5.17	6953	5.65
64	10414	10.88	10564	10.52	10420	8.59	10284	7.68

Figure 6.27: Performance of direct coarsening and recursive coarsening using the graph NACA: two and three levels of contractions vs one level.

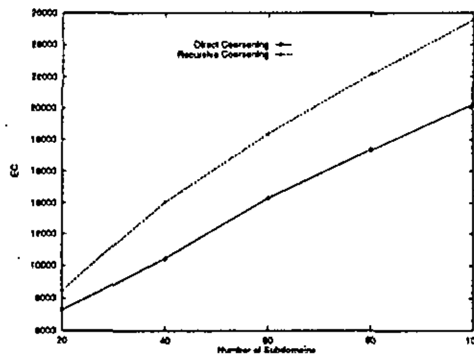


Figure 6.28: Comparing the quality of edge cut produced by direct coarsening and recursive coarsening at the coarse graph using the graph SNECMA.

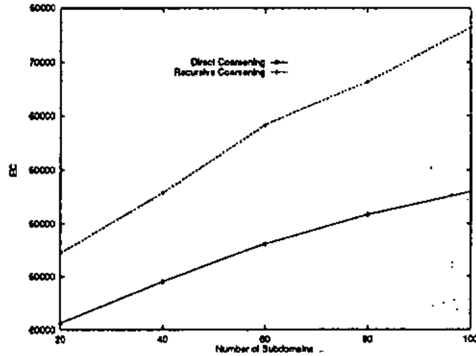


Figure 6.29: Comparing the quality of edge cut produced by direct coarsening and recursive coarsening at the coarse graph using the graph BRACK.

	recursive coarsening				direct coarsening			
N_{sub}	$\lvert EC_4^2\rvert$	t	$\lvert EC_5^2\rvert$	t	$\lvert EC_1^{16}\rvert$	t	$\lvert EC_1^{32}\rvert$	t
32	6874	7.47	6827	8.16	6719	5.70	6566	5.94
64	10386	10.36	10305	13.94	9894	8.63	10058	10.87

Figure 6.30: Performance of direct coarsening and recursive coarsening using the graph NACA: four and five level of contractions vs one level.

test cases are won by direct partitioning. The improvement is up to 16%. In the cases where recursive coarsening outperforms direct coarsening, the improvement is only up to 5%.
The advantage of direct coarsening can be analyzed by looking at Figures 6.28 and 6.29. In these two figures we represent the value of the edge cut prior to refinement as a function of the number of subdomains for the graphs BRACK, and SNECMA. We see that direct coarsening does better than recursive coarsening for both graphs. On the average, the difference in quality ranges from 37% to 40% for BRACK, and from 14% to 26% for SNECMA. Recursive coarsening delivers initial partitions of lower quality compared to those of direct coarsening. Thus, on the absence of a refinement phase, direct coarsening is definitively a better graph coarsening choice than its recursive counterpart. This significant difference in quality at the coarse graphs leads to the fact that MSA is often unable to catch up with the overall quality produced by the combination of direct coarsening and MSA. As far as the time invested during the refinement phase is concerned, we observe that direct coarsening leads in most cases to much cheaper refinement compared to recursive coarsening. On the average, MSA requires 17% more time when combined with recursive coarsening than with direct coarsening.

6.7 Comparing Different Partitioning Algorithms

The subject of this section is to evaluate the performance of our multilevel partitioning algorithm (MLA) against a range of existing methods. As the CHACO partitioning package [Hen93a] offers the possibility of choosing several methods designed to partition graphs, we used it to compare MLA to the inertial partitioning algorithm (IA) [Nou86] coupled with KL. MLA resorts to direct coarsening using GVM with 4 being the maximum size of a cluster. MLA performs as many levels as possible with the constraint that the

N_{sub}	MLA		IA + KL	
	EC	t	EC	t
32	3811	2.61	5890	4.45
64	5639	4.12	7767	5.34
128	7686	6.48	10698	6.23
256	10911	9.52	13472	7.06

Figure 6.31: Performance of our multilevel partitioning algorithm against the inertial method combined with the Kernighan-Lin using the graph $T60K_{nd}$.

N_{sub}	MLA		IA + KL	
	EC	t	EC	t
32	7976	7.21	20249	10.59
64	12956	10.35	26678	12.76
128	20585	15.98	31315	13.92
256	31155	29.90	40756	15.65

Figure 6.32: Performance of our multilevel partitioning algorithm against the inertial method combined with the Kernighan-Lin using the graph SNECMA.

smallest graph should have at least $20 \times N_{sub}$ vertices. The smallest graph is decomposed with SG_{profit}, and the improvement is carried out using MSA, with the modification that at the finer level only the procedure Improve is used. The iteration stops when the rate of acceptance of the procedure Improve is below 5%. This is adopted in order to cut down the execution time of the refinement at the finer level to obtain a reasonable CPU time compared to the other algorithms. All the experiments were performed on an Silicon Graphics with 128 Mbytes of memory and 230 MHz CPU. This benchmark includes the graphs BRACK, SNECMA, and $T60K_{nd}$.

From Figures 6.31, 6.32, and 6.33, we see that the inertial method combined with KL delivers decompositions of lower quality than those produced by MLA. The improvements ranges from 19% to 35%, 24% to 61%, and 18% to 49% for $T60K_{nd}$, SNECMA, and BRACK respectively. This can be explained by the fact that unless a good initial partition is fed into KL, an

N_{sub}	MLA EC	MLA t	IA + KL EC	IA + KL t
32	20908	9.31	40894	14.71
64	30962	16.84	48722	16.62
128	43223	26.79	59976	18.02
256	61138	37.78	74626	21.67

Figure 6.33: Performance of our multilevel partitioning algorithm against the inertial method combined with the Kernighan-Lin using the graph BRACK.

N_{sub}	JOSTLE EC	JOSTLE t	CHACO EC	CHACO t	METIS EC	METIS t	MLA EC	MLA t
32	1069	0.18	1098	1.33	1087	0.20	997	1.51
64	1706	0.34	1759	1.73	1725	0.35	1615	2.40
128	2709	0.71	2644	2.17	2619	0.55	2427	4.45
256	3790	2.25	3863	2.61	3977	0.58	3598	6.30

Figure 6.34: Performance of various partitioning algorithms using the graph HAMMOND.

improvement of high quality is difficult to reach especially for large graphs. Similar observations have been reported [Hen93b]. The comparison of the CPU time reveals that the time of MLA can be as high as 48% compared to that of (IA + KL). This happens especially for a large number of subdomains (128,256) where the number of perturbations performed by the procedure Improve increases owing to an increase of the total number of boundary vertices. On the other hand, the time of (IA + KL) is on the average 30% higher than of MLA below 128 subdomains in most cases. The second benchmark includes a comparison between our MLA and three of the most popular multilevel partitioning schemes METIS [Kar95a], JOSTLE [Wal95a], and CHACO [Hen93a]. We used the most recent versions of METIS (KMETIS) and JOSTLE, but not the latest version of CHACO. Our benchmark uses four graphs of different sizes to illustrate the behavior of each algorithm.

N_{sub}	JOSTLE		CHACO		METIS		MLA	
	EC	t	EC	t	EC	t	EC	t
32	1889	0.36	1922	2.65	1875	0.65	1836	2.07
64	2794	0.47	2822	4.0	2781	0.85	2692	3.39
128	4061	0.87	4066	5.06	4008	0.96	3874	6.25
256	5795	2.46	5913	6.10	5762	1.43	5594	9.27

Figure 6.35: Performance of various partitioning algorithms using the graph WHITAKER.

N_{sub}	JOSTLE		CHACO		METIS		MLA	
	EC	t	EC	t	EC	t	EC	t
32	1834	0.41	1880	3.62	1834	0.99	1719	3.90
64	2984	0.71	2967	4.12	2942	1.01	2815	5.83
128	4620	1.35	4586	5.46	4529	1.23	4420	8.37
256	6827	3.73	6919	6.68	6746	1.73	6498	14.91

Figure 6.36: Performance of various partitioning algorithms using the graph BIG.

N_{sub}	JOSTLE		CHACO		METIS		MLA	
	EC	t	EC	t	EC	t	EC	t
32	20272	5.23	19598	11.15	20942	2.71	20908	9.31
64	28688	7.45	29208	14.16	29671	3.10	30962	16.84
128	41136	11.98	42167	17.87	42148	3.42	43223	26.79
256	58292	16.49	60150	22.17	59567	4.47	61138	37.68

Figure 6.37: *Comparing a faster version of our multilevel algorithm with* other multilevel techniques using the graph BRACK.

N_{sub}	JOSTLE		CHACO		METIS		MLA	
	EC	t	EC	t	EC	t	EC	t
32	20272	5.23	19598	11.15	20942	2.71	18964	125.30
64	28688	7.45	29208	14.16	29671	3.10	28132	187.78
128	41136	11.98	42167	17.87	42148	3.42	41096	201.37
256	58292	16.49	60150	22.17	59567	4.47	57389	275. 12

Figure 6.38: Comparing a slower version of our multilevel algorithm with other multilevel techniques using the graph BRACK.

Based on the results presented in Figures 6.34, 6.35, and 6.36 the following observations are made:

- MLA delivers consistently partitions having smaller edge cut than the other multilevel schemes for small and medium sized meshes.
- When MLA is compared with JOSTLE, CHACO, and METIS, the improvement ranges from 3% to 10%, from 4% to 9%, and from 2% to 10% respectively.
- The runtime of MLA can be up to 11 times higher.

For large meshes, MLA does worse than the other schemes in terms of the quality. Here, we outline the results for the graph BRACK (Figure 6.37). In this example, MLA does up to 7% worse compared to the other schemes, while requiring up to 4 more times. The reason behind the poor quality of the partitions delivered by MLA is that improvement of higher rates is not achieved at different levels due to the low runtime of MSA. Thus, at the finer level, the quality of the edge cut is still lower compared to those of the other partitioning schemes. Thus, much time is needed before reaching the quality of the other partitioning packages. Figure 6.38 compares the quality of partitions produced by a slower version of MLA to those of JOSTLE, CHACO, and METIS. This version uses MSA at different levels , and terminates if no progress has been made during 10 consecutive runs of the procedure Improve. The results show that better quality is achieved at the expense of a much higher execution time. The fact that our MLA is not yet optimized, the two versions are somewhat extreme. On the basis of this result, a better strategy is needed. The time devoted to the improvement phase at different level has to be distributed in a more appropriate manner. Finally, we notice that the quality of the edge cut of METIS, JOSTLE and CHACO is within 6%. As far as the time is concerned, all the three packages deliver decompositions at a lower cost with METIS being the fastest of them all.

6.8 Concluding Remarks

Three coarsening schemes have been used to reduce the size of graphs. These coarsening schemes differ in the manner vertices are merged. The GVM scheme generates coarse graphs whose edges joining the different supervertices have the smallest weights. The main advantage of this scheme resides in the fact that it produces higher matching ratios compared to the other schemes. A direct consequence of this desirable property is the fact that graphs get smaller more quickly using the same number of coarsening levels. Experimental results show that the overall quality of the partitions are sensitive to the coarsening scheme employed. Comparing the three coarsening schemes, GVM yields partitions that are up to 7% better than of HEM given the same execution time. The use of a good initial decomposer at the coarse graph such as SG_{profit} is recommended. Such a choice leads to a reduction in CPU time invested during the improvement phase without sacrifying the quality of the decompositions. Graph coarsening appears to be an efficient means to speed up the partitioning process and provides an elegant natural adaptive neighborhood in the search space. Coarsening could be achieved either directly or recursively. On the average, direct coarsening performs better than recursive coarsening while requiring the least amount of time. Finally, our multilevel algorithm produces better partitions than the three partitioning packages for small and medium size graphs in a reasonable amount of time. However, for large graphs, a further tuning is needed.

Chapter 7

Perspectives

This chapter contains two parts that have not yet reached a level of maturity and which represent two open perspectives for our future work. First, a parallel strategy for the improvement of initial mesh partitions is proposed. We view this work as a first step towards a parallel multilevel algorithm for the load balancing problem. Some preliminary experimental results regarding the performance of this parallelization strategy on a cluster of workstations is presented. Second, since the different metrics used in the objective functions $OF_{partitioning}$, and $OF_{mapping}$ are only an approximation of the factors that affect the runtime of a parallel solver, a parallel demonstration application is used to highlight the impact of the partitioning and the mapping phases on the runtime of the parallel solver.

7.1 Need for Parallel Partitioning Algorithms

Unstructured meshes are increasingly solved on parallel machines because of the significant amount of computing time and memory they require. Even though several efficient sequential partitioning algorithms capable of delivering high quality partitions [Kar95a] [Bar94] [Hen93a] [Wal95a] [Van95b] have been developed in recent years, the task of designing fully parallel partitioning algorithms is still a challenging topic because of its significant practical

value. Solving large-scale numerical simulations using sequential partitioning algorithms may not be possible because of the following facts. First, the amount of memory available on a sequential computer may be insufficient to store the entire mesh. Second, the amount of time needed for solving the problem may not be feasible. Third, in many applications the mesh is already distributed among the processors. As the computation progresses on all processors, the discretization of the physical domain undertakes some changes. Some parts of the mesh will have to be refined, others derefined depending on the characteristics of the calculation. Thus, some processors will have a lot of work to do compared to others, leading to an imbalance situation. The resulting load imbalance means that makes the initial partition is no longer efficient, and a repartitioning process is requested. The approach of collecting the global mesh on one processor or all processors, partitioning with a sequential algorithm, however efficient it may be, and then redistributing the data, is not realistic.

7.2 Parallelization of Partitioning Techniques

The problem of partitioning meshes on parallel computers has received increased attention lately. Some authors have investigated parallel geometry-based partitioning algorithms [Din95] [Hea95]. These algorithms tend to be fast but often produce significantly worse partitions compared to multilevel algorithms [Din95]. Others have focused on developing parallel implementations of the recursive spectral bisection (RSB) [Bar95a] [Bar95b] in order to reduce its high serial computational time. Even the parallel implementation of RSB on 128 or 512 processors tends to be slower than a multilevel partitioning algorithm running on a single processor [Bar95a] [Kar96]. Attempts to parallelize the Kernighan-Lin algorithms [Ker70] and its variant that are used during the improvement phase have had moderate success due to their inherent sequential nature [Gil87] [Din95]. Recently, the interest

of some authors has been directed towards the parallelization of multilevel partitioning algorithms [Kar96][Wal97]. Efficient parallel implementation of these multilevel techniques relies on a successful parallelization of the coarsening, uncoarsening, and improvement phases.
Other authors have been interested in the parallelization of the Simulated Annealing (SA) [Kir83] method. This method is sequential in essence because it is based on the evaluation of a global state which has to take place at each step. Attempts to parallelize SA on distributed memory machines are hampered by two facts:

- the partial moves generated by the different processors have to satisfy a consistency constraint; this requires neighbor-to-neighbor communication;

- the evaluation of the global state requires an all-to-all communication.

An introduction to general concepts of parallel simulated annealing techniques can be found in [Aar89]. A number of strategies have been suggested for its parallelization [Bai89] [Egl90][Gre90][Roo91][Wil86]. Probably, the simplest and most straightforward way to parallelize SA is to let all the available processors run a sequential SA on the whole problem with different seeds for the random generation, and then just select the best solution found. This approach is not suitable in our problem, because the mesh is already distributed across the processors. The other strategy is to parallelize SA exploiting the *data parallelism* approach [Ban90][Wil86][Pon93]. In this approach, the global state is split in as many parts as there are processors; these are allowed to perform several moves without any communication, thereby leading to the following two drawbacks:

- the acceptance of a potential (partial) move is decided purely locally;

- the absence of communication leads to inconsistencies [Man92].

It is only after a number of moves that these inconsistencies are removed and that an evaluation of the global state takes place. This has a negative impact on the convergence of the whole algorithm. Therefore, it is important to find an adequate compromise between too frequent communication and too slow convergence. These local contributions which are subject to inconsistencies [Man92] have to be unified to get the real value of the global objective function in order to prevent degeneration. This unifying process involves communications between processors, which reduces the overall performance of the parallel SA. The key success to the efficiency of SA on parallel system resides in finding a compromise between divergence of the algorithm and the cost of communication.

7.3 Parallel Simulated Annealing

7.3.1 Basic Idea

Our parallel approach is inspired from two distinct ideas introduced by Hammond [Ham92] and Walshaw [Wal95a]. The approach is based on improving piece of boundaries in an independent manner, thereby leading to a natural parallelism. This parallelization strategy gives rise to the following three problems:

1. *How to make two different processors agree on the result of the im*provement of their common boundary ?

2. In what order the improvement of the different piece of boundaries of a particular processor is made ?

3. What local improvement algorithm to use during the improvement phase ?

To deal with the aforementioned problems, the following answers are proposed:

1. Duplicate the work and the necessary set of data.

2. The second problem, which is a scheduling problem, is solved as follows. An initial partition of the graph is supposed to have been constructed. This partition induces a subdomain graph whose vertices correspond to subdomains and edges to common boundaries. Since each subdomain will be assigned to one processor, in the sequel we will be using the term processor instead of subdomain. The improvement which is performed in parallel loops over a number of phases. In each phase, a set of identified pairs of processors sharing common boundaries cooperate to improve their edge cut. Identifying this set of pairs of processors is equivalent to finding a matching by coloring the edges in the subdomain graph [Din95]. In our implementation, the set of pairs of processors is constructed using the heavy edge matching procedure which is executed sequentially once prior to the improvement phase. This matching heuristic scans the vertices in the processor graph in random order. An unmatched processor p_i is paired with an unmatched processor p_j (partner) provided that the size of their boundary in term of edge cut is maximum over all valid incident edges. This way of doing the matching ensures that boundaries with large edge cut are given the highest priority to be improved at early stages of the scheduling. Each time a boundary is improved by a pair of processors, it is marked as *treated* and is never considered in the remaining phases of the scheduling.

   The improvement of all the existing boundaries requires that the scheduling is carried out to the end. In this work, we adopt the strategy of improving at most $k \times |P|/2$ boundaries of the scheduling, where k and $|P|$ denote respectively the number of phases and the number of processors. The number of phases used during the improvement is set equal to $min_{i=1,\ldots|V(S)|} deg_S S_i$.

3. The algorithm used during the improvement phase is a variant of the Simulated Annealing method presented in Chapter 5 (MSA) which does not use the procedure Deteriorate, but only the procedure Improve. Before the start of each phase, each processor proceeds by identifying a set of vertices of a predefined size starting from its boundary and moving inwards in levelwise fashion. One this set is identified, each processor sends a copy of it to its partner. Thereafter, each processor executes the procedure Improve on its own selected set of vertices and those received from its partner (Figure 7.1). To guarantee similar results, each working pair of processors uses the same seed for the random number generator. In our implementation, the size of the set of vertices to be selected by each processor is fixed to 20% of its total assigned vertices. The sequential version of the procedure Improve which is in fact simply a Simulated Annealing with a temperature equals to zero, will be referred to as SGSA, whereas its parallel counterpart will be referred to as PGSA.

7.4 Experimental Results

We evaluated the performance of the PGSA using three graphs. The implementation was carried out on a cluster of Sun Spark20 workstations. We used PVM message passing library for communication.

Figure 7.2 shows tbe quality of the partitions produced by the PGSA and its sequential counterpart SGSA which does not work in a pairwise fashion. From this figure, we see that the edge cut produced by the PGSA is worse than that of SGSA. In the case of HAMMOND graph, PGSA leads to an edge cut which is 40% higher than than obtained by SGSA. However, for the other two graphs, the difference in edge cut is is only 11% higher. The degradation in quality of the PGSA compared to SGSA is due to the fact that the schedule is not carried out to the end, so that a number of

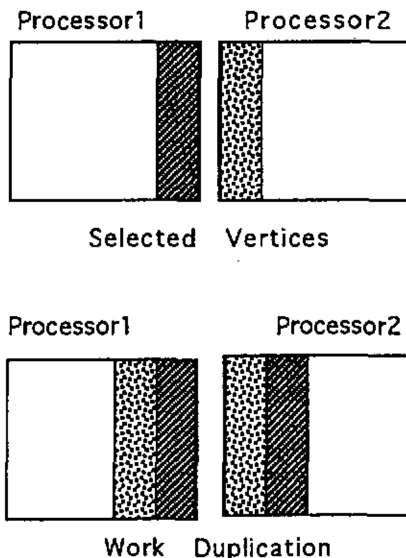


Figure 7.1: Describing the parallel improvement approach for a pair of working processors.

	HAMMOND		NACA		T60K	
N_{sub}	SGSA	PGSA	SGSA	PGSA	SGSA	PGSA
2	95	135	729	815	649	714
4	238	398	1339	1470	1683	1821
8	385	512	2704	3007	3951	4211
16	608	876	4244	4590	6927	7316
32	1012	1412	6860	7543	11492	11965

Figure 7.2: Comparing the quality of the edge cut produced by the parallel greedy simulated annealing and its sequential counterpart

boundaries remain untouched during the improvement phase. Figure 7.3

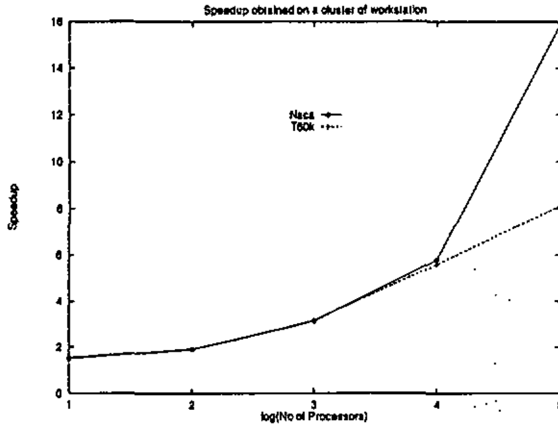


Figure 7.3: Speedup Results

plots the speedup achieved by PGSA for 2 through 32 processors for NACA and T60K. From this figure, we notice that the speedup achieved by PGSA for NACA and T60K are comparable up to 16 processors, whereas, on 32 processors, the difference in speedup is as high as 43%. This difference is due to the cumulated number of processors remaining idle during the execution of PGSA (Figure 7.4). The proposed parallelization strategies offers the following advantages:

- It requires a small amount of communication between the communicating processors during the improvement phase. In addition, no global communication operations among all processors are needed to test the convergence criterion.

- It avoids the inconsistency problems mentioned in [Man92]

Another variant of this parallelization approach would be to design a scheduling strategy where each processor works on a different boundary

rather than having the work duplicated. This will have the advantage of maximizing the number of boundaries to be improved during each phase at the cost of an additional communication operation for each processor in each phase.

$\|P\|$	NACA	T60K
2	0	0
4	0	3
8	3	1
16	3	4
32	11	6

Figure 7.4: Cumulated number of idle processors.

7.5 A Demonstration Application

7.5.1 Test Case

The test case used to highlight the impact of the mesh distribution problem on the CPU time of a parallel application has been developed at the University of Greenwich and is treated in detail in [McM95] [McM96]. Briefly, this application simulates the solidification of liquid gallium in a non-trivial geometry. A fluid-dynamic solver based on the Jacobi method is used in the liquid phase, and a solid mechanics solver based on a preconditioned conjugate gradient method is used in the solid phase.

7.5.2 Transtech Paramid Parallel Machine

The parallel numerical code was run on the Transtech Paramid machine at the University of Greenwich. This machine has 28 *i*860XP-based processor elements (PE), 16 of which are equipped with 32 MBytes and 12 of which are equipped with 16 MBytes of DRAM memory. Each *i*860XP is equipped with a T800 communication co-processor with 8 or 4 MBytes of memory. The PE's

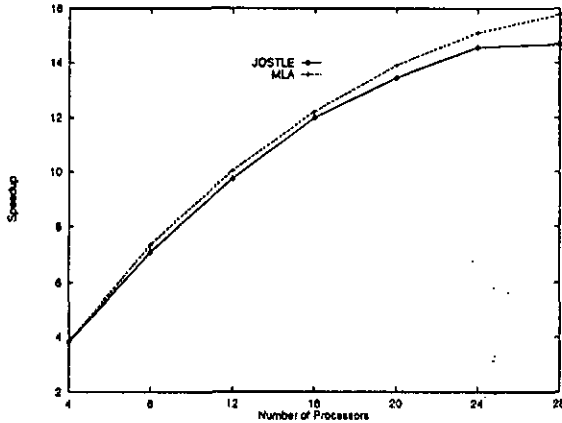


Figure 7.5: Influence of the partitioning on the performance of the parallel solver.

are hard-connected in pairs with Inmos C004 multi-stage crossbar switches providing interconnection between the PE pairs. A simple arrangement for this topology is a $p \times 2$ grid which is the arrangement used for the obtained results. A virtual channel router resident on each processor allows message passing between all the processors in the machine, allowing the machine to be programmed as though the machine had fully connected network.

7.5.3 Impact of the Partitioning and Mapping

Figure 7.5 shows the parallel speedup obtained for various partitioning into 4, 8, 12, 16, 20, 24, and 28 subdomains using JOSTLE and our MLA. The graph used in the test case is T60K with 60,005 vertices and 713,226 edges.

In this experiment, the lowest edge cut is produced by our MLA with a difference that is as high as 5%. The best overall speedup is given by our MLA. This becomes apparent as the number of processors increases. Using

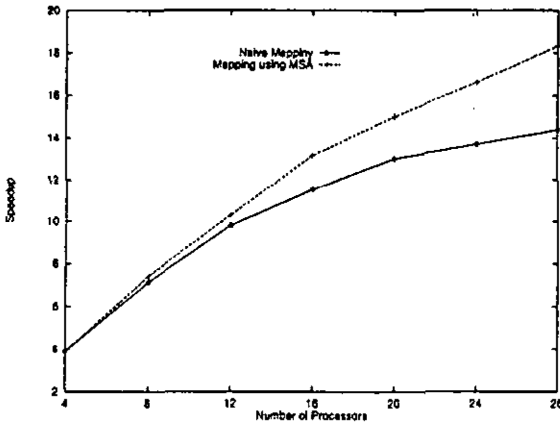


Figure 7.6: Influence of the mapping on the performance of the parallel solver.

28 processors, the time taken by the parallel solver is reduced by 7% using the decomposition produced by MLA. Both partitioning algorithms deliver partitions with a load balance ratio above 0.98. Thus, any difference in performance can not be explained by an effect of a load imbalance. What we can in fact state at this stage is that the average subdomain degree (the number of messages required per processor) and the edge cut (the volume of communication) are the two main factors that explain any difference in the overall runtime. However, several test cases together with a machine equipped with a large number of processors is essential for drawing firm conclusions regarding the impact of the two parameters describing the communication costs. The second issue we have investigated is the impact of the mapping on the performance of the parallel solver. Two mapping strategies are tested for each partition. The first strategy uses a random allocation of subdomains to processors, whereas the second one uses the modified variant of the Simulated Annealing method developed in Section 5.3 to minimize the objective

function $OF_{mapping}$. Figure 7.6 shows the runtime of the application using the two mapping strategies. This result shows the fact that the partitioning coupled with a proper mapping in accordance with the machine topology results in improved performance. The performance difference between the poor and good mappings is insignificant for small number of processors ($P < 16$), and start becoming apparent for large number of processors. In this example, a reduction of up to 22% in CPU time is obtained.

7.6 Concluding Remarks

In this chapter, a parallel variant of Simulated Annealing algorithm for improving mesh partitions is presented. In this parallelization approach, different pairs of processors work simultaneously on the same set of vertices in an effort to reduce the size of their common boundary. The experimental results show that an acceptable speedup is achieved owing to the small communication overhead incurred. However, the quality of the partitions provided by PGSA is worse compared to its sequential counterpart. This degradation in quality is due to the number of boundaries left without being improved because of the low number of phases performed. At the present time, we can not conclude any firm conclusions about the merit of this parallelization strategy before presenting the results for its parallel multilevel implementation.

The last part of this chapter dealt with a demonstration application to see whether there is a little to gain from a proper partitioning and a good mapping. The reduction observed in CPU time is directly related to parameters describing the communication costs. In addition, the acknowledgment of the topology of the machine during the mapping phase has been shown to be of significant importance.

Chapter 8

Conclusions and Further Work

In closing this thesis, we review the various conclusions we arrived at, and draw the attention of the reader to the issues that we shall investigate in the future.

The efficient solution of PDE's which involves the resolution of large sparse linear systems arising from the application of spatial discretization schemes lies at the heart of many scientific computing problems. The discretization process leads to what is defined as a mesh. These meshes are often represented as undirected graphs. In practice, these graphs are large and demand a vast amount of computing resources, in terms of memory and the execution time of the solver. A possible approach for satisfying this high demand is to use parallel computing systems, thus requiring parallel solvers. These parallel solvers rely on a distribution of the graph among the processors.

We used a classical two-step methodology for the mesh distribution problem. In the first step, an initial decomposition of the graph into a specified number of subdomains is generated, which then undergoes some improvement. In the second step, the resulting subdomains are mapped to processors. For both steps, an objective function has been designed to take into account the various parameters having an impact on the execution time of parallel iterative solvers.

Several partitioning algorithms have been developed to derive an initial par-

tition of a graph, and our interest has been directed towards the topology-based Farhat's algorithm. However, this algorithm requires a large amount of memory which may be a limiting factor on some machines. Therefore, four variants of Farhat's algorithm have been developed. These variants differ from Farhat's algorithm in that they work with the true communication graph rather than the mesh itself. These variants share the common feature of exploiting the connectivity of the graph, but differ in the manner vertices are selected for the insertion in the subdomain under construction. Based on the experimental results, the partitions generated by the variant which at each step tries to minimize the edge cut have shorter boundaries, less disconnections, and fewer edges in the subdomain graph compared to the other variants. The only drawback is that it produces subdomains which are not as "round" as those produced by one of the other variants. In all four variants, the edge cut is sensitive to the choice of the starting vertex. This influence is sigoificant for a small and moderate number of subdomains, and becomes negligible for a large number of subdomains. The comparison of our best variant in terms of edge cut against other partitioning algorithms reveals that the former is at least as good as the recursive spectral bisection while requiring less execution time.
The improvement of a partition consists in displacing the boundaries between the subdomains in order to minimize an objective function. To this end, a variant of the Simulated Annealing method (MSA) has been proposed. It consists in an alternation between a pure greedy descent phase and a deterioration phase. The improvement made by this variant is significant regardless of the quality of the initial partition. However, a good initial partition offers the advantage of reducing the time which has to be invested during the improvement phase. Experimental results have shown that MSA delivers better decompositions compared to the versions of Tabu Search and Stochastic Evolutioo methods included in the mesh partitioning package TOP/DOMDEC, while requiring in most cases the least amount of CPU time.

To efficiently decompose large graphs, a multilevel coarsening procedure is used. This procedure consists in reducing recursively the size of the graph, and the decomposition is performed at the coarsest level. Then, an improvement phase is applied at each intermediate level of contraction. Three graph coarsening schemes have been compared. The coarsening scheme which at each step minimizes the "perimeter" of a supervertex produces the smallest coarse graphs and, in most cases, leads to the best final partition in the shortest execution time. Graph coarsening appears to be an efficient means to speed up the partitioning process without sacrifying the quality. A comparison of our multilevel algorithm with CHACO, JOSTLE, and METIS shows that the former produces better partitions for graphs (up to 40,000 vertices) than the three partitioning packages in a reasonable amount of time. However, for large graphs, a further tuning of our implementation is needed. In this research, only one numerical parallel parallel solver is used to compare its performance under two different partitioning algorithms. The results suggest that our partitioning approach, followed by a proper mapping procedure, leads to a moderate reduction of the execution time of the parallel solver.

We conclude this thesis by saying that our future work will include the following parts:

1. The release of an optimized sequential and parallel version of our multilevel partitioning procedure.
2. The measurement of the impact of partitioning and mapping on the execution time for large realistic parallel applications with massively parallel computers.
3. The extension of our partitioning techniques to the dynamic load balancing problem.

Bibliography

[Aar89] Aarts, E., and Korst, J. Simulated Annealing and Boltzman Machines. *New York: John Wiley and Sons Ltd.*, 1989.

[And94] Anderson, W.K., and Bonhaus, D.L. An Implicit Upwind Algorithm for Computing Turbulent Flows on Unstructured Grids. *Computers and Fluids,* Vol. 23, No. 1, pp. 1-21, 1994.

[Bai89] Baiardi, F., and Orlando, S. Strategies for a Massively Parallel Implementation of Simulated Annealing. *Lecture Notes in Computer Science.*, Vol. 366, pp. 273-287, 1989.

[Ban90] Banerjee, P., Jones, M.H., and Sargent, J. Parallel Simulated Annealing for Cell Placement on Hypercube Multiprocessors. *IEE Trans. on Parallel and Distributed Systems,* Vol. 1, No. 1, pp. 91-106, 1990.

[Bar94] Barnard, S.T., and Simon, H.D. A fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems. *Concurrency: Practice and Experience,* Vol. 6, pp.101-107, 1994.

[Bar95a] Barnard, S., and Simon, H.D. Parallel Multilevel Recursive Spectral Bisection. *In Supercomputing 1995*, 1995.

[Bar95b] Barnard, S., and Simon, H.D. A Parallel Implementation of Multilevel Recursive Spectral Bisection for Application to Adaptive Un-

structured Meshes. *In Proceedings of the seventh SIAM conference on Parallel Processing for Scientific Computing*, pp. 627-32, 1995.

[Bok81] Bokhari, S.H. On the Mapping Problem. *IEEE Trans. Comp.*, Vol. 30, No. 3, pp.207-214.

[Bom96] Boman, E., Hendrickson, B. A Multilevel Algorithm for Reducing the Envelope of Sparse Matrices. *Tech Report SCCM-96-14*, Stanford University, 1996.

[Bui93] Bui, T., and Jones, C. A Heuristic for Reducing Fill inSparse matrix Factorization.*In 6th SIAM Conf. Parallel Processing for Scientific Computing*, pp. 445-452, 1993.

[Din95] Diniz, P., Plimpton, S., Hendrickson, B., and Leland, R. Parallel Algorithms for Dynamically Partitioning Unstructured Grids. *Int Proceedings of the seventh SIAM conference on Parallel Processing for Scientific Computing*, pp. 615-20, 1995.

[Egl90] Eglese, R.W. Simulated Annealing: A Tool for Operational Research.*Euro. J. Operational Research*, Vol. 46, pp.271-281.

[Far88] Farhat, C. A Simple and Efficient Automatic Domain Decomposer. *Comp. & Struct.*, Vol. 28, No. 5, pp. 579-602, 1988.

[Far93] Farhat, C., and Lesoine, M. Automatic Partitioning of Unstructured Meshes for the Parallel Solution of Problems in Computational Mechanics. *In J. Number.Meth.Engrg.*, Vol. 36, No.5, pp. 745-764, 1993.

[Far95a] Farhat, C., Naman, N., and Brown, G. Mesh Partitioning for Implicit Computations via Domain Decomposition: Impact and Optimization of the Subdomain Aspect Ratio. *Int. J. Num. Meth. Rngrg.*, 1995.

[Far95b] Farhat, C., Lanter, S., and Simon, H.D. TOP/DOMDEC- A Software Tool for Mesh Partitioning and Parallel Processing. *Computing Systems in Engineering.*, Vol. 6, No. 1, pp.13-26, 1995.

[Fid82] Fiduccia, C.M., and Mattheyses, R.M. A Linear Time Heuristic for Improving Network Partitions. *In 19th Design Automation Conference,* pp. 175-181, 1982.

[Fie75] Fiedler, M. A property of Egenvectors of Non-Negative Symetric Matrices and its Application to Graph Theory. *Czechoslovak Mathematics J.*, Vol. 25, No.100, pp.619-633, 1975.

[Fly66] Flynn, M.J. Very high-speed computing systems.*Proceedings of the IEEE 54, 12*, pp.1901-1909.

[Fox86] Fox, G. Load Balancing and Sparse Matrix-Vector Multiplication on the Hypercube.*Proc. of IMA Workshop,* Galtech report C^3P-327B, 1986.

[Gar79] Garey, M.R., and Johnson, M.R.*Computers and Intractability: A Guide to the NP-Completness.* W.H.Freenam And Co, San Francisco, 1979.

[Geo81] George, A., and Liu., J. Computer Solution of Large Sparse Positive Definite Systems. *Prentice Hall, Inc., Englewood Cliffs,* 1981.

[Gil87] Gilbert, J., and Zmijewski, E. A Parallel Graph Partitioning Algorithm for a Message-Passing Multiprocessor. *In. Journal of Parallel Programming,* Vol.16, pp. 498-13, 1987.

[Glo85] Glover, F., McMillan, C., and Novick, B. Interactive Decision Software and Computer Graphics for Architectural and Space Planning. *Annals of Opns. Res.*, Vol. 5, pp.557-573, 1985

[Glo93] Glover, F., and Laguna, M. Tabu Search. *In C. R. Reeves, editor, Modern Heuristic Techniques for Combinatorial Problems, Blackwell Scientific Publications,* pp. 70-150, 1993.

[Gre90] Greening, D.R. Parallel Simulated Annealing Techniques. *Physica,* Vol. 42, pp. 293-306.

[Ham92] Hammond, S. Mapping Unstructured Grid Computations to Massively Parallel Computers. *PhD Thesis, Rensselaer Polytechnic Institut, Dept. of Computer Science,* Troy, NY, 1992.

[Hea95] Heath, M., and Raghavan, P. A Cartesian Parallel Nested Dissection Algorithm. *SIAM Journal of Matrix Analysis and Applications,* Vol. 16, No. 1, pp. 235-53, 1995.

[Hen93a] Hendrickson, B., and Leland, R. A Multilevel Algorithm for Partitioning Graphs. *Technical Report SAND93-1301,* Sandia National Laboratories, 1993.

[Hen93b] Hendrickson, B., and Leland, R. The Chaco user's guide, Version 2.0. *Technical Report SAND94-2692,* Sandia National Laboratories, 1994.

[Hen93c] Hendrickson, B., and Leland, R. An Improved Spectral Load Balancing Method. *In Proc. 6th SIAM Conf. on Parallel Proc. for Sci. Comput.,* pp. 952-961, 1993.

[Hen95d] Hendrickson, B., and Leland, R. An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations. *SIAM J. Sci. Comput.,* Vol. 16, No.2, pp. 452-469, 1995.

[Hen96c] Hendrickson, B., and Rothberg, E. Effective Sparse Matrix Ordering: Just Around the BEND. *In Proc. Eight SIAM Conf. Parallel processing for Scientific Computing,* 1996.

[Her87] Hertz, A., and de Werra, D. Using Tabu Search Techniques for Graph Coloring. *Computing*, Vol. 39, pp. 345-351, 1987.

[Hsi93] Hsieh, S.H. Parallel Processing for Nonlinear Dynamics Simulations of Structures Including Rotating Bladed-Disk Assemblies. *PhD Thesis*, Cornell University, 1993.

[Jen77] Jennings, A. Matrix Computation for Enginners and Scientists. *John Wiley and Sons, New York* 1977.

[Kar95a] Karypis, G., and Kumar, V. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *Technical Report TR 95-035, Computer Science Department*, University of Minnesota, Minneapolis, MN 55455, 1995.

[Kar95b] Karypis, G., and Kumar, V. Analysis of Multilevel Graph Partitioning. *Technical Report TR 95-037, Department of Computer Science*, University of Minnesota, 1995.

[Kar95c] Karypis, G., and Kumar, V. Multilevel $k-way$ Partitioning Scheme for Irregular Graphs. *Technical Report TR 95-064*, Department of Computer Science, University of Minnesota, 1995.

[Kar96] Karypis, G., and Kumar, V. Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs. *In Supercomputing 1996*, 1996.

[Ker70] Kernighan, B.W., and Lin., S. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell Systems Tech. J.*, Vol. 49, pp.291-307.

[Kir83] Kirkpatrick, S., Gelatt, C., and Vecci, M. Optimization by Simulated Annealing. *Science*, Vol. 220, No. 4598, pp. 671-80,1983.

[Mac67] MacQueen, J.B. Some Methods for Classification and Analysis of Multivariate Observations. *In Proc. of 5th Symp. Math. Stat. and Prob.*, pp. 281-297, 1967.

[Mal88] Malone, J.G. Autamated Mesh Decomposition and Concurrent Finite Element Analysis for Hypercube Multiprocessors Computers. *Comp. Meths. Appl. Mech. Engrg.*, Vol. 70, pp. 27-58, 1988.

[Man92] Mansour, N. Physical Optimization Algorithms for Mapping Data to Distributed Memory Multiprocessors. *PhD Thesis, Syracus University*, 1992.

[McM95] McManus, M., Cross, M., and Johnson, S. Integrated Flow and Stress using an Unstructured Mesh on Distributed Memory Parallel Systems. *In Parallel CFD'94, Elseveir*, 1995.

[McM96] McManus, M. A Strategy for Mapping Unstructured Mesh Computational Mechanics programs onto Distributed Memory Parallel Architectures. *PhD Thesis*, University of Greenwhich, London, UK, 1996.

[Mar92] Martin, O., and Otto, S. Combining Simulated Annealing with Local Search Heuristics.*In G. Laporte and I. Osman, editors, Metaheuristics in Combinatorial Optimization.*, 1992.

[Met53] Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A, and Teller, E. Equation of State Calculation by Fast Computing Mechanics.*J. of Chem. Phys.*, Vol.21, pp. 1087-1091, 1953.

[Mor86] Morrison, R., and Otto, S.W. The Scattered Decomposition for Finite Element Problems. *Journal of Scientific Computing* Vol. 2, Caltech report C3P-286, 1986.

[Nou86] Nour-Omid, B., Raefsky, A., and Lyzenga, G. Solving Finite Element Equations Concurrent Computers.*In Parallel computations and their impact on mechanics, A. K. Noor,ed., American Soc. Mech. Eng.,*New York, pp. 209-227, 1986.

[Pon93] Ponnusamy, R., Nashat, M., Choudary, A., Fox, G. Graph Contraction and Physical Optimization Methods: A Quality-cost tradeoff for Mapping Data on Parallel Computers. *In International conference on Supercomputing*, 1993.

[Pot90] Pothen, A., Simon, H.D., and Liou, K.P. Partitioning Sparse Matrices with Eigenvectors of Graphs. *SIAM J. Mat. Anal. Appl.*, Vol. 11, No.3, pp. 430-452, 1990.

[Roo91] Roussel-Ragot, P., Kouicem, N., and Dreyfus, G. Error-Free Parallel implementation of Simulated Annealing. *Lecture Notes in Computer Science,* Vol. 496, Springer-Verlag, 1991.

[Saa91] Saab., Y.G., and Rao, V.B. Combinatorial Optimization by Stochastic Evolution. *IEEE Trans. on CAD,* Vol. 10, pp. 525-535, 1991.

[Sim91] Simon, H.D. Partitioning of Unstructured Problems for Parallel processing. *Computer Systems in Engineering.*, Vol. 2 pp. 611-614, 1995.

[Van95a] Vanderstraeten, D., and Keunings, R. Optimized Partitioning of Unstructured Finite Element Meshes. *Int. J. Num. Meth. Engrg.*, Vol. 38, pp. 433-450, 1995.

[Van95b] Vanderstraeten, D., and Keunings, R., and Farhat, C. Beyond Conventional Mesh Partitioning Algorithms and the Minimum Edge Cut Criterion: Impact on Realistic Applications. *Proc. of the 7th SIAM Conference on Parallel Processing for Scientific Computing,* 1995.

[Ven91] Venkatakrishnan, V., Simon, H.D., and Barth, T.J. A mimd Implementation of a Parallel Euler Solver for Unstructured Grids. *Report RNR-91-024,* NAS Systems Division, Applied Research Branch, NASA Ames Research Center, Mail Stop T045-1, Moffet Field, CA 94035

[Wal95a] Walshaw, C., Cross, M., Everett, M. A Localised Algorithm for Optimizing Unstructured Partitions. *Int Journal. Supercomputer Appl,* Vol. 9, No. 4, pp. 280-95,1995.

[Wal95b] Walshaw, C., and Berzins, M. Dynamic Load-Balancing for PDE solvers on Adaptive Unstructured Meshes. *Concurrency: Practice and Practice,* Vol. 7, No. 1, pp.17-28, 1995.

[Wal95c] Walshaw, C., Cross, M., Everett, M.G., Johnson, S., and McManus, M. Partitioning and Mapping of Unstructured Meshes to Parallel Machine Toplogies. *Prc. Irregular'95: Parallel Algorithms for Irregularly Structured Problems,* Vol. 980, pp. 121-126, 1995.

[Wal97] Walshaw, C., Cross, M., Everett, M. Mesh Partitioning and Load balancing for Distributed Memory Parallel Systems. *In Proceedings of Parallel and Distributed Computing for Computational Mechanics,* Lochinver, Scotland, 1997.

[Wil86] Williams, R.D. Minimization By Simulated Annealing: Is Detailed Balance necassary?.*Caltech report C3P-354*, 1986.

[Wil91] Williams, R. Performance of Dynamic Load Balancing Algorithms for Unstructured Mesh Calculations. *Concurrency and Experience,* Vol. 3, pp. 457-481, 1991.