Parallel Adaptive Unstructured Computation

Jose Gabriel Castanos
Ph.D. Dissertation

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-00-06
May 2000

Parallel Adaptive Unstructured Computation

by

José Gabriel Castaños

Analista de Systemas,
Facultad de Ciencias Fisicomatemáticas e Ingeniería
Universidad Católica Argentina, 1988
Licenciado en Investigación Operativa,
Facultad de Ciencias Fisicomatemáticas e Ingeniería
Universidad Católica Argentina, 1989
Sc. M. in Computer Science, Brown University, 1996

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2000

This dissertation by José Gabriel Castaños is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date ______________ ____

John E. Savage, Director

Recommended to the Graduate Council

Date ______________ ____

Paul E. Fischer, Reader
Argonne National Laboratories

Date ______________ ____

Maurice Herlihy, Reader

Date ______________ ____

Franco Preparata, Reader

Approved by the Graduate Council

Date ______________ ____

Dean of the Graduate School and Research

Vita

Vitals

José Gabriel Castaños was born in Buenos Aires, Argentina, on March 12, 1967. He studied Systems Analysis and Operations Research at the Universidad Católica Argentina where he graduated in 1988. He entered the Sc.M. program at the Department of Computer Science at Brown University in 1993 and joined its Ph.D. program in 1996.

Education

Ph.D. in Computer Science, May 2000.
Brown University, Providence, RI.

Sc.M. in Computer Science, May 1996.
Brown University, Providence, RI.

Licenciado en Investigación Operativa, December 1989.
Facultad de Ciencias Fisicomatemáticas e Ingeniería.
Universidad Católica Argentina, Buenos Aires, Argentina.

Analista de Systemas, December 1988.
Facultad de Ciencias Fisicomatemáticas e Ingeniería.
Universidad Católica Argentina, Buenos Aires, Argentina.

Acknowledgements

This thesis is the most important accomplishment of my first 33 years of being a student. During this time I have been surrounded by a number of people that have helped to make this process fruitful and enjoyable. I will always be in debt with them because that have had direct or indirect influence in my work and that have helped shape and discover the person that I am today.

I am deeply thankful to my advisor Prof. John Savage for encouraging me to pursue this Ph.D. and teaching me to do research. John had the vision to find a project that perfectly suited my skills and interests. During these years we had the best student-advisor interaction that I can dream of, something I wish every graduate student had the chance to experience. Although I have been many times on the verge of quitting due to stress and frustration, John always found the right words to keep me going. In the following pages you will read the work that we did together. This thesis is the outcome of our many meetings and discussions in which I always found his wisdom priceless. I owe this Ph.D. to him.

My deepest gratitude goes to Paul Fischer, Maurice Herlihy and Franco Preparata for reviewing and evaluating both my Sc.M. and Ph.D. theses. They represent an incredible source of knowledge which was never been more than an e-mail or door knock away. They always found time to answer my questions and to help me solve my problems. I do really cherish their support and guidance.

This thesis would have not been completed without the help of Vaso Chatzi. Our joint work was fundamental on understanding the complex mathematics and numerical analysis required for this work.

While at Brown I have been very fortunate to work with many professors that I deeply respect. I am very grateful to Tom Doeppner for allowing me to assist in his operating systems course. This was a personal challenge for myself and being able to pull through felt like a little victory. I will always treasure my early interactions with Tom Dean and

I am very thankful giving me the responsibility to teach one of our introductory courses. I greatly regret that I did I did not have the opportunity to spend more time with Paris Kanellakis. I am one of the last students that he are left at Brown but I am sure that his example is never forgotten.

My coming to Brown and to a new country could have not been a more pleasant experience thanks to the great gang of Room 402: Al Mamdani, Andy Foersberg, Dawn Garneau, Jon Metzger, Laura Paglione, Madhu Jalan, Rob Mason and especially Sonal Jha who were always finding new ways of keeping me away from the CIT. For two wonderful years they manage to make our office the funniest windowless spot in Providence. All my officemates Sonia Leach, Michael Littman, Song Zhang and David Tucker had to bear my insatiable demands for computing power. Thanks also to my friends Laurent Michel and Michael Benjamin.

Finally, I would like to dedicate this thesis to my family. Without their love and encouragement I would have not been able to come to the USA to pursue my goals.

Contents

List of Tables **xi**

List of Figures **xiii**

1 Introduction **1**

1.1 Motivation 1

1.2 The Challenge of Parallel Adaptive Computation 3

1.3 Contributions of This Thesis 4

1.4 Related Work 6

1.4.1 DIME 6

1.4.2 Scalable Unstructured Mesh Computation (Summa3D) 6

1.4.3 PMDB 7

2 The FEM Adaptive Process **8**

2.1 A Short Introduction to the Finite Element Method 8

2.2 Selection of the Mesh Type 11

2.3 Qualities of Unstructured FEM Meshes 11

2.4 Local Adaptation of FEM meshes 12

3 PARED: An Overview **13**

3.1 What is PARED? 13

3.2 FEM Mesh Representation in Pared 16

3.2.1 Refinement Trees 16

3.2.2 Implementing a Parallel Mesh With Remote References 18

3.3 Using PARED to Solve PDEs . . . 19

4 Local Adaptation of Unstructured Meshes **22**

4.1 Rivara's Longest Edge Bisection in Two Dimensions . . . 22

4.2 Rivara's Longest Edge Bisection in Three Dimensions . . . 23

4.3 Parallel Refinement of Unstructured Meshes . . . 24

4.3.1 The Challenge of Refining in Parallel . . . 27

4.3.2 The Message Model for Refinement . . . 29

4.4 Properties of Parallel Refined Meshes . . . 32

4.5 Mesh Coarsening . . . 36

4.6 Parallel Refinement of Unstructured Meshes: Experimental Results . . . 36

4.6.1 Global Refinement of Regular Meshes . . . 37

4.6.2 Global Refinement of Irregular Meshes . . . 42

5 Mesh Partitioning and Repartitioning **45**

5.1 Introduction . . . 45

5.2 Partitioning Finite Element Meshes . . . 46

5.2.1 Review of Graph Partitioning Methods . . . 48

5.3 The Repartitioning Problem . . . 51

5.4 The Parallel Nested Repartitioning Method (PNR) . . . 52

5.4.1 Repartitioning the Adapted Mesh . . . 53

5.5 Quality of the Partitions Obtained from PNR . . . 55

5.5.1 Competitive Analysis of PNR . . . 57

5.6 The High Migration Cost of RSB and Multilevel-KL . . . 64

5.7 Bounding the Migration Cost . . . 67

5.8 Minimizing the Migration Cost . . . 69

5.9 A Transient Problem . . . 72

6 Mesh Migration **79**

6.1 Introduction . . . 79

6.2 Overview of the Migration Algorithm . . . 81

6.3 The Mesh Migration Procedure used in PARED . . . 83
6.3.1 Element and Vertex Transfer Phase . . . 86
6.3.2 Remote Reference Update Phase . . . 89
6.3.3 Element Deletion, Removal of Remote Vertices, and Vertex Deletion 90

7 The Engineering of PARED **94**
7.1 Introduction . . . 94
7.2 Mesh Data Structures . . . 95
7.2.1 Mesh Classes . . . 96
7.2.2 Element Classes . . . 97
7.2.3 Vertex Classes . . . 99
7.2.4 Parallel Meshes . . . 101
7.3 Classes Controlling the Refinement of Meshes . . . 101
7.4 Partitioning of Meshes . . . 103
7.5 Classes Controlling Mesh Migration . . . 103
7.6 Console Classes . . . 106
7.7 Representing Systems of Equations . . . 107
7.8 Higher Order Polynomials . . . 110
7.9 Specifying Problems in PARED . . . 110
7.9.1 Defining New Problems in PARED . . . 112
7.9.2 Predefined Differential Equations . . . 113
7.9.3 `Problem` and PARED . . . 114
7.9.4 Future Improvements to the `Problem` class . . . 115

8 The Communications Library of PARED **116**
8.1 Motivation . . . 116
8.2 New Classes in Our Communications Library . . . 117
8.2.1 Communicators and Ports . . . 117
8.2.2 Buffers . . . 120
8.2.3 Stream Classes . . . 120
8.3 Using Our Library to Exchange Messages . . . 121

8.4 Performance Analysis 123
8.4.1 Introduction 123
8.4.2 Point-to-Point Communication 125
8.4.3 Ping-Pong Communication 127
8.4.4 All-to-All Communication 130
8.4.5 Ring Communication 133

9 Experimental Results **136**
9.1 Putting it All Together 136
9.2 A Laplace Example 136
9.2.1 Experiments on an IBM SP 139
9.2.2 Experiments on Sun Workstations 142
9.3 A Navier-Stokes Problem 142
9.3.1 Problem Formulation 142
9.3.2 Solution Strategy 145
9.3.3 Numerical Results 147

10 Future Work **159**

Bibliography **162**

List of Tables

4.1 Number of elements and vertices of meshes obtained by the repeated refinement of regular two- and three-dimensional meshes. 39

4.2 Largest number of edge refinements sent by a processor to other processors in each refinement level obtained by performing successive refinements on a regular 2D mesh partitioned randomly and with the Multilevel-KL algorithm. 40

4.3 Largest number of edge refinements sent by a processor to other processors in each refinement level obtained by performing successive refinements on a regular 3D mesh partitioned randomly and with the Multilevel-KL algorithm. 40

4.4 Number of elements and vertices that result from successive refinement of irregular two- and three-dimensional meshes. 43

4.5 Largest number of new edge refinements sent by a processor to other processors in each refinement phase by performing successive refinements on an irregular 2D and 3D meshes (Multilevel-KL partitions). 44

5.1 Mesh sizes of the locally adapted 2D and 3D meshes at each refinement level. 57

5.2 Comparison of the quality of the partitions produced by Multilevel-KL and PNR. The tables show the number of shared vertices obtained by partitioning a sequence of locally adapted meshes with Multilevel-KL and PNR into 4 to 128 subsets. 58

5.3 Migration cost resulting from repartitioning a series of two-dimensional unstructured meshes of increasing size using the RSB algorithm. M^{t-1} is the mesh before refinement and distributed according a balanced partition Π^{t-1} obtained from RSB. M^t is the refined mesh. $\widehat{\Pi}^t$ is a new balanced partition of M^t also produced by the RSB algorithm and $\widetilde{\Pi}^t$ is a permutation of $\widehat{\Pi}^t$ that minimizes data movement. 65

5.4 Migration cost resulting from repartitioning a series of three-dimensional unstructured meshes of increasing size using the RSB algorithm. M^{t-1} is the mesh before refinement and distributed according a balanced partition Π^{t-1} obtained from RSB. M^t is the refined mesh. $\widehat{\Pi}^t$ is a new balanced partition of M^t also produced by the RSB algorithm and $\widetilde{\Pi}^t$ is a permutation of $\widehat{\Pi}^t$ that minimizes data movement. 66

5.5 Migration cost resulting from repartitioning a series of two-dimensional unstructured meshes of increasing size using the PNR algorithm. M^{t-1} is the mesh before refinement and distributed according a balanced partition Π^{t-1} obtained from PNR. M^t is the refined mesh. $\widehat{\Pi}^t$ is a new balanced partition of M^t also produced by the PNR algorithm and $\widetilde{\Pi}^t$ is a permutation of $\widehat{\Pi}^t$. 73

5.6 Migration cost resulting from repartitioning a series of three-dimensional unstructured meshes of increasing size using the PNR algorithm. M^{t-1} is the mesh before refinement and distributed according a balanced partition Π^{t-1} obtained from PNR. M^t is the refined mesh. $\widehat{\Pi}^t$ is a new balanced partition of M^t also produced by the PNR algorithm and $\widetilde{\Pi}^t$ is a permutation of $\widehat{\Pi}^t$. 74

List of Figures

2.1 Decomposition of a two-dimensional domain into a triangular mesh. 10

3.1 A snapshot of PARED's graphical user interface. 14

3.2 The refinement of the mesh in (a) using a nested refinement algorithm creates a forest of trees as shown in (b) and (c). The dotted lines identify the leaf triangles. 17

3.3 Mesh representation in a distributed memory machine using remote references. 19

3.4 Outline of the different procedures executed by PARED to adaptively compute the solution of PDEs. 20

4.1 Longest edge (Rivara) bisection algorithm on triangular mesh. 23

4.2 Rivara's longest edge bisection in two dimensions. 23

4.3 Rivara's longest edge bisection in three dimensions. 24

4.4 Parallel longest edge bisection. 26

4.5 Parallel adaptation collision. 28

4.6 The messaging algorithm executed by the coordinator P_C. 30

4.7 Message model for refinement by processor P_i. 31

4.8 Termination detection of the refinement phase. 32

4.9 A cyclic propagation using the parallel longest edge bisection algorithm. . . 33

4.10 General longest-edge bisection (GLB) algorithm. 35

4.11 Initial two- and three-dimensional regular meshes. 38

4.12 Regular two- and three-dimensional regular meshes partitioned between 32 processors using a partition provided by Multilevel-KL (left) and a random partition (right). 39

4.13 Refinement time for the global successive refinements of a regular two dimensional mesh with a (*a*) Multilevel-KL partition and (*b*) random partition. (*c*) and (*d*) show the same times for the refinement of a regular three-dimensional mesh. 41

4.14 Relative speedups for the refinement of a regular mesh using Multilevel-KL and random partition for a (*a*) two-dimensional regular mesh of 262,144 elements and (*b*) three-dimensional regular mesh of 98,304 elements. 42

4.15 Initial two- and three-dimensional irregular meshes on a square and a cube. 43

4.16 Refinement time for the global successive refinements of a irregular (*a*) two-dimensional mesh and (*b*) three-dimensional irregular mesh when Multilevel-KL is used. 44

5.1 Outline of the Parallel Nested Repartitioning Algorithm. 54

5.2 Irregular two- and three-dimensional regular meshes adaptively refined to solve Laplace's equation of a problem that exhibits high physical activity in one of its corners. 57

5.3 Algorithm to convert a partition Π^t of G^t with balance B to a partition Π^0 of G with balance $B + \gamma$. 60

5.4 An 6-level uniform refinement of a triangle. 61

5.5 Outline of PNR's procedure to repartition the dual graph G in the coordinator P_C. 72

5.6 (*a*) and (*b*) show the computed solution u with $t = -0.5$ and $t = 0.5$. (*c*) and (*d*) illustrate the adapted mesh at these two different time steps. 76

5.7 Quality of the partitions measured by the number of shared vertices produced by RSB and PNR for 4, 8, 16 and 32 processors for each of the 100 time steps between $t = -0.5$ to $t = 0.5$. 77

5.8 Elements moved between time steps of partitions produced by RSB, permuted RSB and PNR for 4, 8, 16 and 32 processors. 78

6.1 A simple migration example. (a) shows the initial mesh. The goal is to move Ω_a from P_0 to P_2. (b) We first copy Ω_a to P_2 and (c) update the references. (d) We finally delete the element Ω_a in P_0. 82

6.2 The mesh shown in (a) has shared vertex V_2 between P_0 and P_1. (b) P_0 sends the vertex to P_2 while P_1 sends the element to P_3. (c) shows an incorrect mesh because there are no references between the copies of V_2 located in P_2 and P_3. (d) shows the correct final mesh. 84

6.3 Migration of elements from an initial partition Π^t (a) to a target partition Π^{t+1} (b) showing the multiple copies and remote references. 85

6.4 Initial distribution of the mesh between four processors for example in Figure 6.3 (a). 86

6.5 Element and vertex transfer phase: procedure executed by processor P_i to send a selected set of elements and vertices to a destination processor P_j. . 87

6.6 Element and vertex transfer phase: state of the mesh after creating the new copies in the destination processors. We highlight the new elements vertices created in this phase in each processor. We only show the references between the multiple copies of vertex V_7 to reduce the complexity of this figure. . . 88

6.7 Element and vertex transfer phase: procedure executed by each processor P_j to receive a transfer message from processor P_i and to create the elements and vertices indicated in the message. 89

6.8 Remote reference update phase: outline of the algorithm executed by each processor P_i to update the references of new shared vertices. 89

6.9 Remote reference update phase: state of the mesh at the end of this phase. The elements that will be deleted in each processor in the next phase are highlighted 90

6.10 Element Deletion: state of the mesh after deleting the elements from the source processors but before removing the unnecessary vertices. We only include the references for vertices that are going to be removed in the next phase 91

6.11 Procedure executed by each processor P_i in the three phases of the migration algorithm. 92

6.12 A migration example: internal representation of the mesh at the end of the migration phase. 93

7.1 FEMesh classes. FEMeshCommon is an actual mesh or a portion of the mesh located in a processor. The concrete classes are FEMesh2D for 2D meshes and FEMesh3D for 3D ones. To simplify the use of parallel meshes we created the class FEMeshStub that implements the same interface (or public methods) as FEMeshCommon. 97
7.2 The Element class hierarchy supports two- and three-dimensional elements. The RootElement classes store references to the elements in the initial mesh. 98
7.3 The Vertex class hierarchy. PARED defines classes for two and three dimensional vertices (Vertex2D and Vertex3D). Vertices in the boundary of the domain (of type BoundaryVertex2D and BoundaryVertex3D) contain references to the Boundary objects. 100
7.4 Mesh refinement classes. RefineMgr uses the classes SharedVertexInfo and SharedVertexNum to manage the parallel refinement of meshes. 102
7.5 Class diagram of the classes used to compute partitions of the mesh. . . . 104
7.6 Mesh migration classes. The manager class MigrateMgr uses a variety of auxiliary classes such as AddRef and DeleteRef. 104
7.7 Users interact with PARED through Console objects. The ConsoleMaster class is used in serial systems. In parallel mode a ConsoleMaster is located in the coordinator processor, while the remaining processors contain a ConsoleSlave object. 106
7.8 Matrix class diagram. PARED supports one- two- and three-dimensional matrices using a variety of storage methods. 108
7.9 The system class combines AbsMatrix2D, AbsMatrix1D and Precondition and provides methods for solving linear systems. 109
7.10 Diagram of the classes used to define higher order approximations. 111
7.11 The Problem class hierarchy defines the equation to solve. 112

8.1 Class diagram of PARED's communications library. 118
8.2 A simple example that communicates a vector of doubles between two processors using our library. 122
8.3 To communicate user defined objects the user must provide an implementation to the input and output stream operators. 124

8.4 Point-to-Point MPI Communication. Bandwidth obtained by sending vectors of 1 to 256K doubles using MPI's blocking, non-blocking and packing function calls in a NOW and IBM SP. 126

8.5 Point-to-Point Stream Communication. Bandwidth obtained by sending vectors of 1 to 256K doubles using the `MPIOStream` and `MPIIStream` with buffers of 4K, 16K and 64K bytes in a NOW and IBM SP. 128

8.6 Point-to-Point MultiStream Communication. Bandwidth obtained by sending vectors of 1 to 256K doubles with the classes `MPIMultiOStream` and `MPIMultiIStream` with buffers of 4K, 16K and 64K bytes in a NOW and IBM SP. 129

8.7 Ping-Pong MPI Communication. Bandwidth obtained by sending vectors of 1 to 256K doubles using MPI's blocking, non-blocking and packing function calls in a NOW and IBM SP. 130

8.8 Ping-Pong Stream Communication. Bandwidth obtained by sending vectors of 1 to 256K doubles using the `MPIOStream` and `MPIIStream` with buffers of 4K, 16K and 64K bytes in a NOW and IBM SP. 131

8.9 Ping-Pong MultiStream Communication. Bandwidth obtained by sending vectors of 1 to 256K doubles with the `MPIMultiOStream` and `MPIMultiIStream` classes with buffers of 4K, 16K and 64K bytes in a NOW and IBM SP. . . 132

8.10 All-to-All communication. Bandwidth obtained by exchanging between all the processors in a NOW and IBM SP for 4 to 32 processors sending and receiving of 1 to 256K doubles to and from all the other processors in a NOW and IBM SP. 134

8.11 Ring communication. Bandwidth obtained by first sending and receiving of 1 to 256K doubles to the next and previous processor respectively in a NOW and IBM SP. 135

9.1 Number of elements and vertices that result from successive local refinements of irregular two- and three-dimensional meshes. 138

9.2 Times for each refinement phases of the locally adapted two-dimensional problem on 4, 8, 16, 32 and 64 processors of an IBM SP parallel computer. 140

9.3 Times for each refinement phases of the locally adapted three-dimensional problem on 4, 8, 16, 32 and 64 processors of an IBM SP parallel computer. 141

9.4 Times for each refinement phases of the locally adapted two-dimensional problem on 4, 8, 16, 32 and 64 processors of a network of workstations. . . 143

9.5 Times for each refinement phases of the locally adapted three-dimensional problem on 4, 8, 16, 32 and 64 processors of a network of workstations. . . 144

9.6 Procedure for adaptively solving the incompressible Navier-Stokes equations. 146

9.7 Initial 2D mesh used to study the flow past a cylinder. 148

9.8 Refined mesh at time $t = 0.5$ secs. (top) and $t = 1.25$ secs. (bottom). . . . 149

9.9 Refined mesh at time $t = 2.5$ secs. (top) and $t = 5$ secs. (bottom). 150

9.10 Refined mesh at time $t = 7.5$ secs. (top) and $t = 10$ secs. (bottom). 151

9.11 Refined mesh at time $t = 12.5$ secs. (top) and $t = 15$ secs. (bottom). . . . 152

9.12 Refined mesh at time $t = 17.5$ secs. (top) and $t = 20$ secs. (bottom). . . . 153

9.13 Refined mesh at time $t = 22.5$ secs. (top) and $t = 25$ secs. (bottom). . . . 154

9.14 Refined mesh at time $t = 27.5$ secs. (top) and $t = 30$ secs. (bottom). . . . 155

9.15 Number of elements, vertices and shared vertices for time step t, $0 \leq t \leq 15,000$, in the adapted meshes used to simulate a turbulent flow. 156

9.16 Fraction of the total time used to compute $\hat{\mathbf{u}}$, solve p, solve $\mathbf{u}$, select elements for refinement and coarsening using higher order polynomials, adapt the mesh and partition and migrate the mesh for all $15,000$ time steps of a turbulent flow simulation. 157

Chapter 1

Introduction

1.1 Motivation

The finite element method (FEM) is a powerful and successful collection of techniques for the numerical solution of partial differential equations (PDEs). The FEM was originally developed to solve structural problems but it is now used to approximate the solution of PDEs in all areas of engineering and applied mathematics. Many static problems have regions of high physical activity embedded in large domains in which the solution is smooth. In transient problems, the regions of interest can appear or vanish, and modify their size, shape or location, as occurs in the study of turbulence in fluid flows. In all cases, it is necessary to *adapt* the mesh to follow the physical anomalies so that regions of high gradients are not under-resolved while maintaining a coarser mesh everywhere else.

Adaptive computation offers the potential to provide large storage and computational savings on problems with dissimilar scales by focusing the available computational resources on the regions where the solution changes rapidly. Adaptive computation can be applied to a wide variety of problems and has been successfully implemented on a wide variety of meshes for finite-difference, finite-element, finite-volume and spectral methods, providing exponential rates of convergence. An overview of problems and relevant adaptation strategies is given by Powell, Roe and Quirk [69]. Unfortunately, adaptivity significantly increases the complexity of algorithms and software. Nevertheless, the gains produced by adaptive schemes greatly justify their extra overhead and the challenging programming task. To cope with this complexity, new design techniques based on object-oriented technology are needed.

In this thesis we study the problems that arise when adaptive schemes are used in a parallel computing environment. We discuss the difficulties of designing parallel adaptation methods and we introduce a new parallel refinement algorithm based on Rivara's bisection algorithm for triangular and tetrahedral meshes [72, 73]. By representing the adapted mesh as a forest of trees of elements we avoid the synchronization problems for which Jones and Plassman [54] use randomization.

The local adaptation of a mesh produces imbalances in the work assigned to processors. Thus, adaptive finite element methods are an excellent example for the study of dynamic load balancing schemes on distributed memory parallel computers. We propose a new *Parallel Nested Repartition* (PNR) algorithm that has its roots in the multilevel algorithms by Barnard and Simon [10] and by Hendrickson and Leland [47]. Our method produces high quality partitions at a low cost, a very important requirement for recomputing partitions at runtime. It has a very natural parallel implementation that allows us to repartition adapted meshes of arbitrary size. The collapsing of the elements is performed locally in each processor using the refinement history and avoiding the communication overhead of other partitioning methods [56]. We have also designed a migration procedure to rebalance the mesh after repartitioning.

The work presented in this thesis assumes the LogP [27] model of computation in which a parallel computer consists of few (much smaller than the problem size) coarse grained processing nodes that communicate by exchanging messages across a high latency network. The LogP model is an accurate characterization of current parallel computers that have a relatively high message startup cost.

To evaluate these ideas we have developed and implemented a parallel object-oriented system called PARED [21]. Earlier work on this system was described in [16, 17, 18]. The constant advances in computer architectures has allowed researchers to increase the range of the problems that can be studied and to improve the quality of their simulations. Massively parallel computers can deliver impressive peak performances but many more gains are still possible by making better use of the current technology. The most salient characteristic of adaptive codes is the high sophistication of their code and data structures. The use of object oriented techniques allowed us to reduce the complexity of the implementation without significantly affecting the performance.

1.2 The Challenge of Parallel Adaptive Computation

A system for the parallel adaptive solution of PDEs must integrate subsystems for solving equations, estimating errors, adapting a mesh, finding a good assignment of work to processors, migrating portions of a mesh according to a new assignment, and handling interprocessor communication. Each of these systems offer challenging problems of their own.

Adaptive schemes start by computing a solution on an initial coarse grid. Based on local and global error estimates, the mesh is locally adapted so that high gradient regions are not under-resolved and low gradient regions are not over-resolved. In many refinement algorithms the refinement of a mesh element might cause the refinement of adjacent elements to maintain the conformality of the mesh. In a parallel environment, where a portion of the given mesh and its corresponding equations and unknowns is assigned to each processor, this propagation of refinement requires synchronization between two or more neighboring processors.

One of the most difficult tasks of parallelizing a refinement algorithm that propagates the refinement is to determine the termination of the refinement phase. A processor might have no more local elements to refine, but it needs to wait for possible propagations from neighbor processors. Only when all the processors agree on the termination of the refinement phase can they proceed to the next phase.

Mesh adaptation produces imbalances in the work assigned to the processors. Because of the irregular load requirements of parallel adaptive computation, a mesh must also be dynamically repartitioned and migrated between processors at runtime. Repartitioning must be interleaved with the numerical simulation. Thus, we cannot afford expensive algorithms that recompute partitions from scratch after each refinement or coarsening. While we have developed partitioning algorithms that run fast in parallel, these algorithms must avoid the movement of large regions of the mesh in response to small changes, which is typical of standard partitioning algorithms.

Finally, we want our project to execute on distributed memory machines that do not share a common address space. Our system must provide support for global data structures that are distributed between processors so that different objects can migrate between processors.

1.3 Contributions of This Thesis

Rather than studying the problems of mesh adaptation, mesh repartitioning and mesh migration independently, in this thesis we put special emphasis on investigating how these different components interact. By considering adaptivity as a whole we obtain new results that are not available when these problems are studied separately. For example, there is extensive literature on graph partitioning methods, but there is almost no work on partitioning, and in particular repartitioning, of adapted meshes. By maintaining the adaptation history of the mesh we can easily cluster adjacent elements, thereby reducing the complexity of partitioning large meshes.

The most important contributions of the work presented in this thesis are the following:

- We have designed a parallel refinement algorithm for unstructured two- and three-dimensional meshes based on the longest edge bisection of triangles and tetrahedra. We have studied the properties of the meshes generated by this parallel algorithm and we have shown the resulting meshes and the ones generated by the serial longest edge algorithm are the same.

- We have developed a new partitioning algorithm for adapted meshes called *Parallel Nested Repartitioning* (PNR). The quality of the partitions obtained using this method is similar to the ones produced by the widely used Recursive Spectral Bisection (RSB) and Multilevel-KL from the Chaco [48] graph partitioning library. Our algorithm takes as an input previously computed partitions and generates partitions that have a much smaller migration cost compared to the ones obtained from standard methods for problems on which the mesh is already allocated to processors.

- We have created new data structures for parallel adaptive meshes. Our environment supports the creation and destruction of mesh structures at runtime in individual processors and the reassignment of portions of the mesh between the processors, all while maintaining a consistent global mesh. In our meshes, the assignment of elements and vertices to processors is not fixed throughout the computation. Instead our design supports dynamically changing connectivity information where references to remote objects are updated as new elements are created, deleted or moved to a new processor to rebalance the work. The complexity of this approach is hidden by the use of a global object space where remote object communication is facilitated by the use of proxies. Cached proxies are also used to reduce latency and communication overhead.

- We have designed and implemented Pared, a system for the parallel adaptive solution of partial differential equations. In [16, 18] we presented an early prototype that supported the parallel manipulation of unstructured two-dimensional meshes. This prototype introduced some of the ideas that were later included in Pared such as refinement trees and remote references but did not include any FEM analysis.

  Pared is based on a serial adaptive project developed in conjunction with Vasiliki Chatzi. This system supports a variety of mesh types, including the use of Crystalline Meshes proposed by Chatzi and Preparata [25, 26].

  Pared is an integrated system that includes modules for adapting the mesh, partitioning and repartitioning the work between processors, mesh migration, error estimation and solution of systems of equations. It runs on distributed memory machines such as the IBM SP and networks of workstations.

- We have designed a high performance library to communicate irregular data structures between processors. This library uses a standard C++ stream interface. Therefore exchanging objects between processors is similar to reading and writing files.

- We have evaluated these ideas in a variety of static and dynamic partial differential equations in an IBM SP and a network of workstations. We have shown that our system can handle very large meshes and can solve complex problems such as the incompressible Navier-Stokes equations at high Reynolds numbers. In this example we have shown that our dynamic meshes can be 4.75 to 18.63 times smaller than equivalent static meshes that achieve the same desired error.

Chapters 2 and 3 present a short introduction to the FEM and an overview of Pared. The parallel mesh refinement algorithm is described in Chapter 4. Chapter 5 provides an overview of graph partitioning algorithms and introduces our new repartitioning heuristics. Dynamic meshes and the parallel mesh migration procedure are the subjects of Chapter 6. Chapter 7 describes the classes that form Pared and Chapter 8 presents a new communications library for object migration between processors. Finally, Chapter 9 shows the solution obtained by our adaptive system to two typical problems: a static problem with a region of high errors and a transient fluid flow.

1.4 Related Work

Due to its importance, parallel adaptive computation has been the subject of several projects. Although all these systems have a common goal, each system proposes different strategies for each phase of the adaptive process. Among the most important projects that are based on unstructured triangular or tetrahedral meshes (such as the ones used in Pared), we mention DIME, Summa3D and PMDB.

1.4.1 DIME

Pared follows in the spirit of the Distributed Irregular Mesh Environment (DIME) [94], an early system developed by Roy Williams at the California Institute of Technology. DIME executes in the Cubix server [1], an old parallel environment system that is not supported by current parallel computers.

The original version of DIME works on 2D unstructured meshes. DIME does not support mesh coarsening so it cannot be used in fluid dynamic problems. DIME uses a variety of mesh partitioning algorithms such as Simulated Annealing and Orthogonal Recursive Bisection (ORB) that do not minimize the movement of data between repartitions.

DIME uses a database of voxels with hash functions to maintain the identity of each mesh structure. The migration procedure implemented in DIME is inefficient because every processor must receive every migration message in order to update references to shared data structures.

1.4.2 Scalable Unstructured Mesh Computation (Summa3D)

In [53, 54], Jones and Plassman describe a method for the parallel refinement of triangular meshes. In this method, the mesh is partitioned by vertices rather than by elements (unlike Pared), although each element is owned by only one processor. To avoid the synchronization problem of refining adjacent elements owned by neighboring processors, the authors rely on the use of heuristics. They first compute an independent set of the elements of the mesh. Elements in the same subset are refined in the same phase so no two adjacent elements can be refined at the same time.

The independent set is approximated by the assignment of random numbers to the elements of the mesh. The authors also propose a strategy on which elements are assigned to subsets according to a graph coloring of the dual graph of the mesh. In a bounded degree graph, as it is the case in the dual graph, the maximum number of colors, and also the number of subsets, is independent of the graph size. Nevertheless, the greedy algorithm proposed to compute the graph coloring still requires an initial random assignment of colors to mesh elements.

Its partitioning heuristic is based on a variation of ORB which is called Unbalanced Recursive Bisection (URB). It is well known that methods that rely on coordinate information of the mesh such as ORB and URB often produce partitions with a high bisection width, that is, they contain many more interprocessor links than is necessary.

1.4.3 PMDB

In [83] an adaptive environment for unstructured problems called PMDB is presented. PMDB refines tetrahedra using a variety of different subdivision patterns. PMDB first marks a set edges for refinement. Then each element is refined into two or more tetrahedra depending on which of its edges are marked. In the case that this procedure creates a tetrahedron with very sharp angles, then some adjacent tetrahedra might be additionally refined but only if they are located in the same processor. PMDB does not propagate the refinement across processor boundaries. Therefore, multiple runs of the same program in different number of processors can generate different results.

PMDB offers several partitioning heuristics. Its diffusion-based method, which moves boundary elements between processors, is slow; it requires multiple iterations to converge to a balanced distribution. Its octree graph partitioning algorithm does not guarantee small bisection widths because it can produce many small disconnected subsets.

Chapter 2

The FEM Adaptive Process

2.1 A Short Introduction to the Finite Element Method

The finite element method (FEM) is used to approximate the solution to a wide range of differential equations. Although the FEM is more difficult to implement than competing techniques such as finite differences, it easily applies to problems with irregular domains, general boundary conditions and non-uniform grids. In this section we present a short overview of the FEM and introduce some of the notation and procedures that we use in the following chapters. For a more extensive treatment of the FEM see [4, 23, 52, 80, 81, 96].

The FEM is actually a class of methods that seek an approximation of an unknown function u from a finite dimensional space S_m. This space (also known the *trial space*) is defined by a set of basis functions $\{\phi_1, \ldots, \phi_m\}$. One important property of basis functions is that any function in the space S_m can be expressed by a unique linear combination of them. Therefore, an approximation $\hat{u}$ in the space S_m to a partial differential equation

$$\mathcal{L}u = q$$

can be expressed in the form

$$u \approx \hat{u} = \sum_{i=1}^{m} a_i \phi_i$$

where $\mathcal{L}$ is a differential operator and the coefficients a_i are unknowns.

For example, consider the two-dimensional Poisson problem

$$\begin{aligned} -\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) &= q(x, y) \qquad \text{in } \Omega \\ u &= 0 \qquad \text{on } \partial\Omega \end{aligned} \tag{2.1}$$

where Ω is a region in the plane and $\partial\Omega$ is its boundary.

Multiplying equation 2.1 by an arbitrary smooth function v that satisfies $v = 0$ on $\partial\Omega$ and integrating in the domain results in the following equation.

$$-\int_{\Omega} v\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) d\Omega = \int_{\Omega} vq\, d\Omega$$

Using Green's formula to integrate by parts we obtain

$$\int_{\Omega}\left(\frac{\partial v}{\partial x}\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\frac{\partial u}{\partial y}\right) d\Omega = \int_{\Omega} vq\, d\Omega. \tag{2.2}$$

This equation is almost equivalent to the original equation but contains lower order derivatives than the original formulation.

The *weak* formulation of Equation 2.2 is: find $u \in H_0'$ such that for all $v \in H_0'$ the equation holds with

$$\int_{\partial\Omega} v\frac{\partial v}{\partial n} \cdot \hat{n} dS = 0 \tag{2.3}$$

The first step in the FEM is to divide the domain Ω into a set of non-overlapping cells $\{\Omega_a, 1 \leq a \leq n\}$ or elements of regular shape such as triangles and quadrilaterals in 2D and tetrahedra and hexahedra in 3D, as suggested in Figure 2.1.

We then define the trial space S_m which is usually chosen from a finite-dimensional subspace H_0' consisting of piecewise polynomials of bases 1. To every node j in the mesh we associate a basis function ϕ_j. For triangular linear elements, these are polynomials of linear order in x and y of the form:

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

These functions ϕ_j are continuous in Ω and piecewise linear. If (x_j, y_j) are the coordinates of node j, then ϕ_j also satisfies the following condition:

$$\phi_j(x_i, y_i) = \delta_{i,j} = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases}$$

and the basis coefficients a_j correspond to the nodal point values.

To make an approximation to u from equation 2.2 we put a set of weighting functions $w_j, 1 \leq j \leq m$, in place of v such that the number of equations is equal to m, the number of unknowns $a_i, 1 \leq i \leq m$. There are several choices of w_j that result in the different

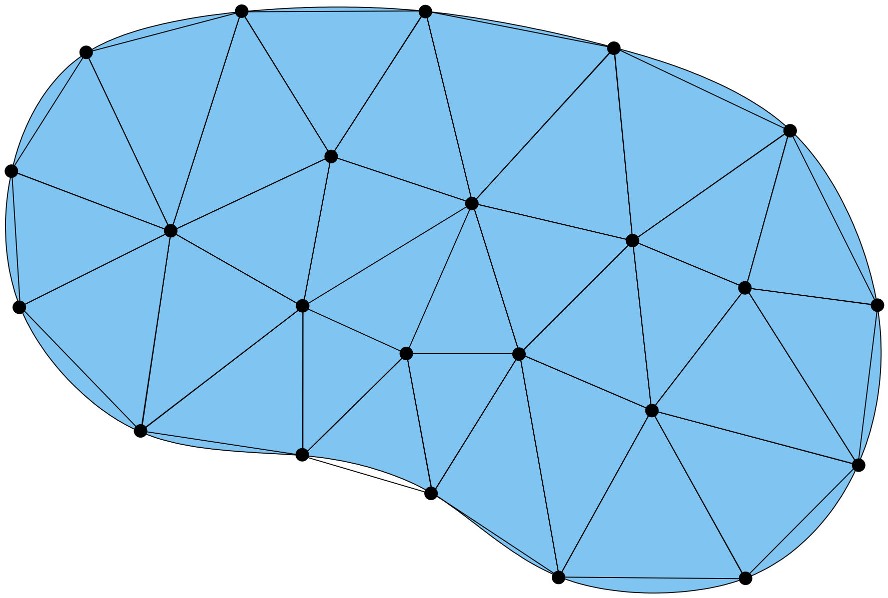


Figure 2.1: Decomposition of a two-dimensional domain into a triangular mesh.

variations of the method of the weighted residuals, a general class of methods that include the FEM. One of its most popular forms, known as the Garlekin method, chooses $w_j = \phi_j$ and results in the system of equations of m unknowns

$$Ka = f \tag{2.4}$$

where

$$k_{j,i} = \int_\Omega \left(\frac{\partial \phi_j}{\partial x}\frac{\partial \phi_i}{\partial x} + \frac{\partial \phi_j}{\partial y}\frac{\partial \phi_i}{\partial y} \right) d\Omega \tag{2.5}$$
$$f_j = \int_\Omega \phi_j q \, d\Omega.$$

The coefficient matrix K (usually referred as the *stiffness matrix*) and the *force vector* f are easily obtained by summing the contributions of the individual elements.

The matrix K in equation 2.5 is sparse because $k_{j,i} = 0$ unless there is an overlap on the regions in which ϕ_i and ϕ_j are nonzero. K is also symmetric and positive definite so the linear systems of equations 2.4 has a unique solution. On meshes composed of triangles or tetrahedra its non-zero structure is usually irregular. The system of equations can be solved using Gaussian elimination without pivoting or by most iterative methods, such as Conjugate Gradient.

2.2 Selection of the Mesh Type

The selection of the mesh type depends on the problem to be studied since there is no strategy that is considered best for every problem. Most finite element meshes belong to two categories:

- Structured meshes: there is a mapping from the physical space to the computational space. In the computational space the elements appear as squares (in two dimensions) or cubes (in three dimensions). Each vertex of the mesh, except the boundary vertices, has an isotropic neighborhood. The neighbors and vertices of an element are easily calculated using array based data structures. For example, the neighbors of a grid point (i, j) in a two dimensional space using a natural ordering are $(i-1, j), (i+1, j), (i, j-1), (i, j+1)$. Elements are also defined using a similar ordering. Thus, it is not necessary to reserve additional storage to hold the identity of neighbors and vertices of any particular element.

- Unstructured meshes: in this case the elements store explicit connectivity information to determine their neighbors and vertices. The data structures in this case are more complex than in structured meshes but it is easier to represent irregular geometries.

Each type of mesh has its advantages and disadvantages. Structured meshes require simpler codes with less overhead but are more limited in the representation of complex domains. Although a number of techniques have been developed to find appropriate mappings from the physical domain to the computational domain [22, 90, 91], it is generally not possible to find a transformation that fits irregular domains. Unstructured meshes, such as the ones used in PARED, are more complex, require more storage and overhead per element but can easily represent arbitrary domains. For that reason, unstructured meshes are usually used on problems with complex geometries, such as wings or machine parts. Some techniques implement the meshes as a combination of both approaches [95, 44]. For example, block structured or semistructured methods, decompose the domain into a set of unstructured super-elements, where each super-element is a structured grid.

2.3 Qualities of Unstructured FEM Meshes

The compatibility of a mesh to a problem topology and correct treatment of the boundaries are not the only requirements for high-quality meshes. In addition, the mesh must also satisfy element size and shape constraints, which vary over the domain.

The rate of convergence and quality of the solutions provided by the FEM depends heavily on the number, size and shape of the mesh elements. For a given shape, the approximation error increases with element size (h), which is usually measured by the length of the longest edge of an element. Smaller elements result in larger systems of equations which increase the complexity of the method.

The condition number of the matrices used in the FEM and the approximation error are related to the minimum and maximum angle of all the elements in the mesh [7]. In three dimensions, the solid angle of all tetrahedra and their ratio of the radius of the circumsphere to the inscribed sphere (which implies a bounded minimum angle) are usually used as measures of the quality of the mesh [60, 67]. For some flow problems, the orientation of the elements is also important.

In addition, the basis functions must satisfy certain continuity conditions at the element boundaries. A mesh is *conforming* if neighbor elements intersect at a common vertex, edge or face. The mathematical formulation only admits conforming meshes. Although the FEM has also been applied to non-conforming meshes, conformality is a property that greatly simplifies the method. It is also assumed to be a requirement in this thesis.

2.4 Local Adaptation of FEM meshes

The goal of adaptive computation is to optimize the computational resources used in a simulation which can be achieved by refining a mesh to increase its resolution on regions of high relative error in static problems or by refining and coarsening the mesh to follow physical anomalies in transient problems [95].

If a numerical method that has an accuracy of order $O(h^p)$, where h is a measure of the element size and p is the order of the polynomials used in the approximation, the convergence to the exact solution u can be obtained by decreasing h or by increasing p. In an optimal mesh, every element should have the same error. Thus, every element in which the local error condition is not satisfied is a candidate for refinement.

The adaptation of a mesh can be performed by changing the order of the polynomials used in the approximation (p-refinement), by modifying the structure of the mesh (h-refinement), or a combination of both (hp-refinement). p-refinement can be thought as increasing the amount of information associated with an element without changing the geometry of the mesh, where p is the polynomial order of the basis functions.

Chapter 3

PARED: An Overview

3.1 What is PARED?

PARED [21] is a system for the parallel adaptive solution of PDEs. This system is a continuation of earlier work on parallel meshes presented in [16, 18]. PARED supports the local refinement and coarsening of unstructured two- and three-dimensional meshes, and the dynamic repartitioning and load balancing of the work. Our design supports a dynamically changing environment. Elements and vertices (and associated equations and unknowns) migrate between processors to balance the workload. References to remote elements and vertices are updated as new elements or vertices are created, deleted or moved to a new processor. Although all the support code (more than 100,000 lines) is written in C++, our system relies on Fortran libraries such as BLAS [59, 30] and Lapack [5] that are optimized for each particular architecture for numerically intensive procedures.

PARED runs on serial machines and parallel computers. The serial version was developed in conjunction with the *Crystalline Meshes* project by Chatzi and Preparata at Brown University [25, 26]. On parallel machines, PARED runs on distributed memory computers in which processing nodes communicate by exchanging messages using MPI (Message Passing Interface)[37, 46, 87], which is the standard for message passing libraries. These machines consist of coarse-grained processing nodes connected through a high latency network. Each processing node cannot directly address a memory location in another node. Because each message has a high startup cost, efficient message passing algorithms must minimize the number of messages delivered. Thus, it is better to send a few large messages rather than many small ones. This is a very important constraint and has a significant impact on the design of message passing algorithms.

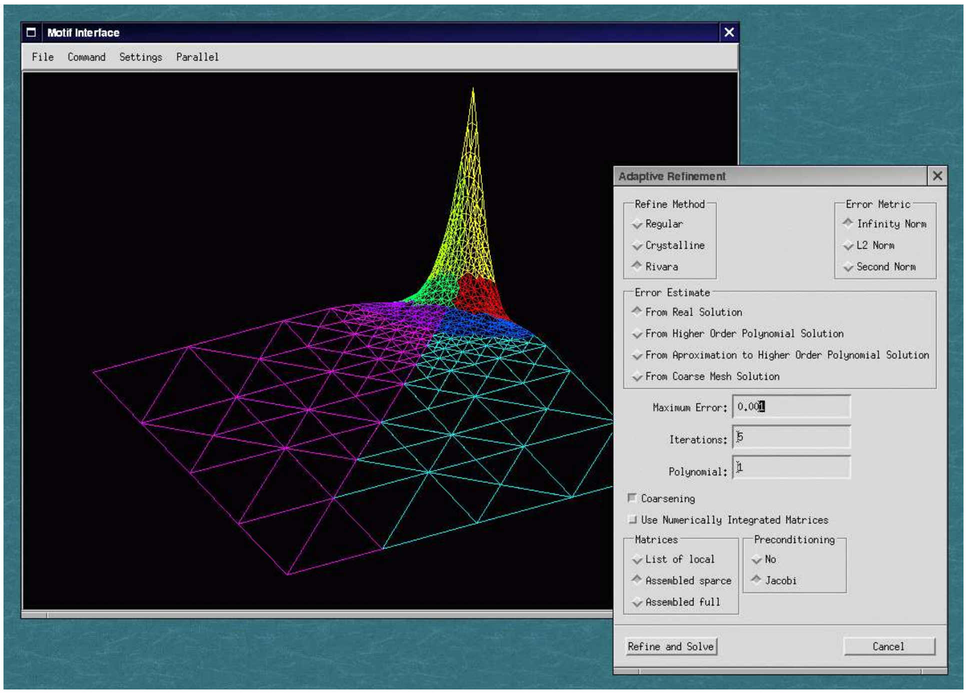


Figure 3.1: A snapshot of PARED's graphical user interface.

We have evaluated our system in two different parallel architectures: an IBM SP and a network of workstations (NOW). Each processing node of the SP contains four PowerPC 604e processors running at 333 Mhz in SMP mode with 1GB of common main memory. Processing nodes are connected through the SP switch, which is similar to an inverted butterfly, and provides relatively fast communication. The version of MPI used in this machine is the standard IBM MPI. The NOW consists of a variable number Sun Ultra-1 workstations, each having 128MB of memory and connected via a 100Mbps Ethernet network. For communication, the NOW uses MPICH (version 1.1.2), a freely available implementation of MPI from Argonne National Lab.

PARED has two modes of operation: interactive or batch. The interactive mode allows the user to visualize the changes in the mesh that results from its adaptation, partitioning and migration. The user controls the system through a GUI, as suggested in Figure 3.1, via a distinguished processor called the *coordinator*, P_C, which collects information from all the other processors. PARED uses OpenGL [39] to permit the user to view 3D meshes from different angles. Through the *coordinator*, the user can also give instructions to all processors such as specifying when and how to adapt the mesh or which strategy to use when repartitioning the mesh. The batch mode is used for production runs.

PARED uses remote references and smart pointers, two ideas commonly found in object oriented programming, to provide a simple replication mechanism that is tightly integrated with our mesh data structures. In adaptive computation, the structure of the mesh evolves during the computation as elements and vertices are created, destroyed or assigned to different processors. The use of remote references and smart pointers have greatly simplified the creation of dynamic parallel meshes.

When implemented in C++, a remote reference is just an object that consists of a processor number and memory address. A processor can use a remote reference to invoke methods on objects located in a remote processor. Method invocations and arguments destined for remote processors are marshaled into a few messages that contain memory addresses of the remote objects. In the destination processor(s), each address is converted to a pointer to an object of the corresponding type through which the method is invoked. Because the different processors are inherently trusted and MPI guarantees reliable communication, PARED does not incur the overhead traditionally associated with distributed object systems.

Smart pointers are used so that proxy objects can be destroyed when there are no more references to them. For example, in PARED vertices are associated with multiple elements. When the reference count of a vertex proxy reaches zero, the proxy is no longer attached to an element located in the processor and can be destroyed. If a vertex proxy is located in an internal boundary between processors, then some processor might have a remote reference to it. In that case, before a proxy is destroyed, it informs the copies in other processors to delete their references to it. This procedure insures that the shared vertex can then be safely destroyed without leaving dangerous dangling pointers referring to it in other processors.

Finally, PARED uses streamed non-blocking communication to hide the complexity and overhead of message passing where each object marshals and unmarshals itself onto a stream. The refinement and migration algorithm have a communication pattern that is different from most scientific code such as a parallel matrix-vector product in which the same set of memory locations are repeatedly exchanged between processors. In the refinement and migration algorithms, it is also difficult for the destination processors to estimate the size of the receiving buffers and the messages can become very large in the migration algorithm if a lot of data movement is required. To overcome these problems, our system uses automatic buffering that divides very large messages into smaller ones.

3.2 FEM Mesh Representation in Pared

All the different subsystems of PARED use a common mesh representation. This framework is based on two basic concepts: refinement trees and remote references between the copies of shared vertices located in different processors. The refinement (or adaptation) trees maintain the refinement history of the mesh and simplify tasks such as coarsening of the mesh. The remote references between shared vertices are used to maintain a consistent global data structure as the mesh evolves throughout the computation.

3.2.1 Refinement Trees

To support the dynamic adaptation of meshes we designed a hierarchical data structure of nested meshes. We assume that the user supplies an initial coarse mesh M^0 called the 0-level mesh. Starting from M^0 the adaptation procedure constructs a family of nested meshes $M^0, M^1, \ldots, M^t$. Let $M^t(D^t, V^t)$ be the mesh at time step t where $D^t = \{\Omega_1, \ldots, \Omega_n\}$ is a set of elements that approximate the domain Ω of interest and $V^t = \{V_1, \ldots, V_m\}$ is the set of vertices in the mesh. Every element in D^t at time step t is an unrefined element.

Let M^{t+1} be the new adapted mesh that results from refining a set of elements $R \subseteq D^t$ and coarsening another set of elements $C \subseteq D^t$ such that $R \cap C = \emptyset$. For each refined element $\Omega_a \in R$ we define $Children(\Omega_a) = \{\Omega_{a_1}, \Omega_{a_2}, \ldots, \Omega_{a_d}\}$ to be the elements resulting from the refinement of Ω_a, and let $Parent(\Omega_{a_k}) = \Omega_a$ for $1 \leq k \leq d$. Although different refinement methods produce different numbers of children, d is assumed to be a small integer.

We also define the *Level* of an element Ω_a so that $Level(\Omega_a) = 0$ if Ω_a is in the initial mesh and $Level(\Omega_a) = Level(Parent(\Omega_a)) + 1$ otherwise. The elements Ω_{a_k} can also be refined, and become the parents of elements. In PARED, when an element is refined, it does not get destroyed. Instead, the refined element inserts itself into a tree. The refined mesh forms a forest of refinement trees, one per initial mesh element. Thus, for every element Ω_a in the initial mesh PARED maintains a refinement history tree τ_a where every element except a *leaf* element is the *parent* of two or more elements. The leaf elements of these trees form the most refined mesh M^t on which the numerical simulation is based at time t. These trees are used in many of our algorithms. For example, in PARED a mesh is coarsened by replacing all the children of a refined element by their parent. Thus, M^0 is the coarsest mesh that our system can manipulate. Our repartitioning algorithm also takes advantage of these trees. When an element is migrated to another processor all its descendants are migrated as well.

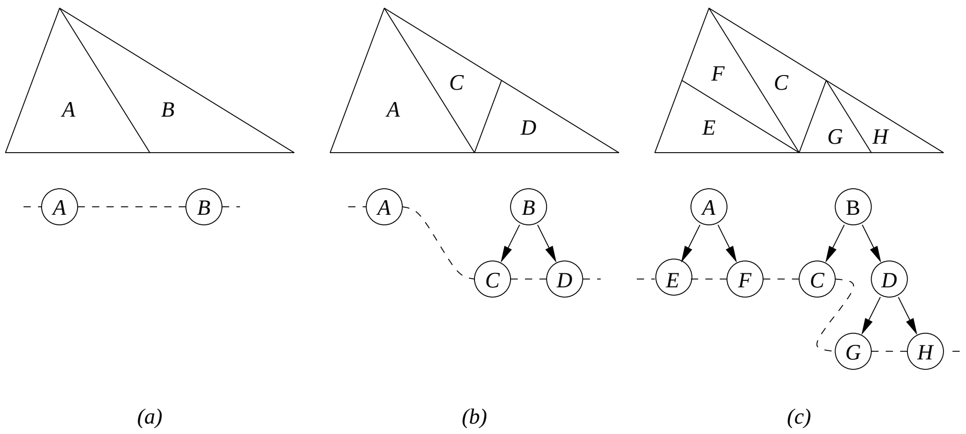


Figure 3.2: The refinement of the mesh in (a) using a nested refinement algorithm creates a forest of trees as shown in (b) and (c). The dotted lines identify the leaf triangles.

The multilevel representation of the mesh has the following properties:

- An element that has no parents has level 0 and belongs to the *coarse (initial) mesh* M^0. No coarsening is done above this level.
- An element with no children belongs to the *fine mesh* M^t. The numerical simulations are always based on the fine mesh, although coarser meshes are sometimes used (for example, in multigrid methods).
- An element could be at the same time in both the *coarse mesh* M^0 and the *fine mesh* M^t (for example before any refinement is done) or in any intermediate mesh.
- Only elements that are in the *fine mesh* M^t can be selected for refinement or coarsening. The hierarchy of elements is only modified at its leaves.
- As the elements are individually selected for refinement or coarsening the hierarchy can have different depths in different regions of the mesh.
- When an element Ω_a is refined it is replaced by its *children* in the new *fine mesh* M^{t+1}. To coarsen an element all its *children* must be selected for coarsening. In this case the *children* in the new *fine mesh* M^{t+1} are replaced by their *parent* and destroyed.

3.2.2 Implementing a Parallel Mesh With Remote References

One of the most common uses of remote references in PARED is to maintain the connectivity information between adjacent regions of the mesh that are located in different processors. PARED partitions the mesh *by elements*. Partitioning the mesh by elements has several advantages over partitioning it by nodes, as we will show in Chapter 5. In PARED every element is assigned to only one processor and mesh vertices are shared if they are adjacent to elements located in different processors. Proxies for vertices that are common to mesh elements on different processors are held in each of them.

Definition 1 Let V_p^i be the copy of vertex $V_p \in V^t$ located in processor i at time step t. We define $Ref(V_p^i) = \{(P_j, V_p^j) \mid V_p^j \text{ is a copy of } V_p \text{ in processor } j\}$.

This relation is symmetric so that if $(P_i, V_p^i) \in Ref(V_p^j)$ then $(P_j, V_p^j) \in Ref(V_p^i)$. If V_p is a node internal to a processor P_i, then $Ref(V_p^i) = \varnothing$. A node in an internal boundary can be shared by more than two processors. Hence if V_p is a shared node then $1 \leq |Ref(V_p^i)| \leq P-1$ where P is the number of processors. In general, we expect that $|Ref(V_p^i)| \ll P-1$. Mesh quality requirements guarantee that the elements do not contain sharp angles. Thus, the number of elements adjacent to any vertex is small, which results in few shared copies. Good partitions of the mesh also reduce the total number of references.

$Ref(V_p^i)$ is a dynamic structure that is implemented as a list. Its content varies through the simulation as vertices are moved between processors because it is possible that a new partition of the mesh converts an internal node into a shared node and vice versa or that it modifies the contents $Ref(V_p^i)$. The example in Figure 3.3 shows a mesh with 5 elements and 7 nodes. The nodes V_3 and V_4 are shared by two processors P_i, and P_j so $Ref(V_3^1) = \{(P_2, V_3^2)\}$ while $Ref(V_3^2) = \{(P_1, V_3^1)\}$.

There is no need to have more than one copy per node in each processor. Suppose that a processor i has two copies of the same node V_p^i and $V_p^{i'}$ so that $(P_i, V_p^{i'}) \in Ref(V_p^i)$. We can detect this condition because the reference points to a node in the same processor i. We then remove the copy $V_p^{i'}$ after updating all the references in other processors that point to $V_p^{i'}$ to point to V_p^i. For a similar reason we do not need or allow duplicate references in $Ref(V_p^i)$.

We also define $Adj(\Omega_a) = \{V_p | V_p \text{ is a vertex of } \Omega_a\}$ and $ElemAdj(V_p) = \{\Omega_a | V_p \text{ is a vertex of the element } \Omega_a\}$. The $Adj(\Omega_a)$ of an element Ω_a is the set formed by the vertices of Ω_a. In the case of triangular elements $|Adj(\Omega_a)| = 3$, and in the case of tetrahedral

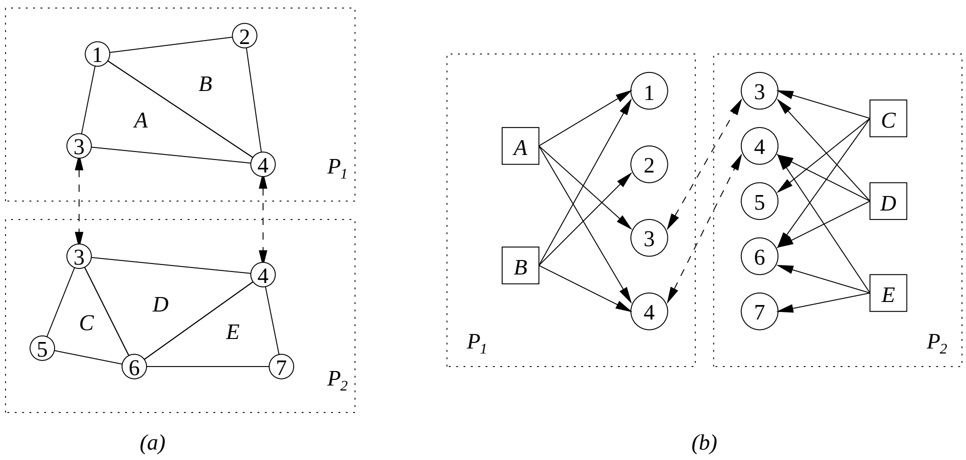


Figure 3.3: Mesh representation in a distributed memory machine using remote references.

elements $|Adj(\Omega_b)| = 4$. $ElemAdj(V_p)$ of a node V_p is the set formed by the elements containing V_p. In an unstructured mesh $|ElemAdj(V_p)|$ is not a constant. Although in theory we can construct meshes where $|ElemAdj(V_p)|$ can have arbitrary values, if the mesh is non-degenerate (the interior angles are not close to 0) we expect that $|ElemAdj(V_p)|$ be bounded above by a constant k. In a mesh partitioned by elements we can define $ElemAdj(V_p^i) = \{\Omega_a \mid \Omega_a \in ElemAdj(V_p) \text{ and } \Omega_a \text{ is located in processor } P_i\}$.

3.3 Using PARED to Solve PDEs

The flowchart shown in Figure 3.4 provides an overview of the adaptive process used by PARED. Although PARED supports several mesh types with different element shapes, in this thesis we concentrate on two- and three- dimensional meshes composed of triangles and tetrahedra.

To start a simulation PARED loads the initial mesh $M^0(S, V)$ into a distinguished processor called the *coordinator* P_C. As we mentioned in Section 3.2.2, PARED partition the mesh by elements. The coordinator creates a weighted dual graph $G(W, E)$ of the mesh, that is, a graph with one vertex $w_a \in V$ for every element $\Omega_a \in S$ and an edge $(w_a, w_b) \in E$ if two elements Ω_a and Ω_b in the initial mesh are adjacent. PARED computes a partition Π of the elements of the mesh by elements by first obtaining a partition of the vertices of G. In general, we assume that the graph G is small enough to be partitioned using a variety of serial graph partitioning algorithms on a single processor. Our system includes an interface

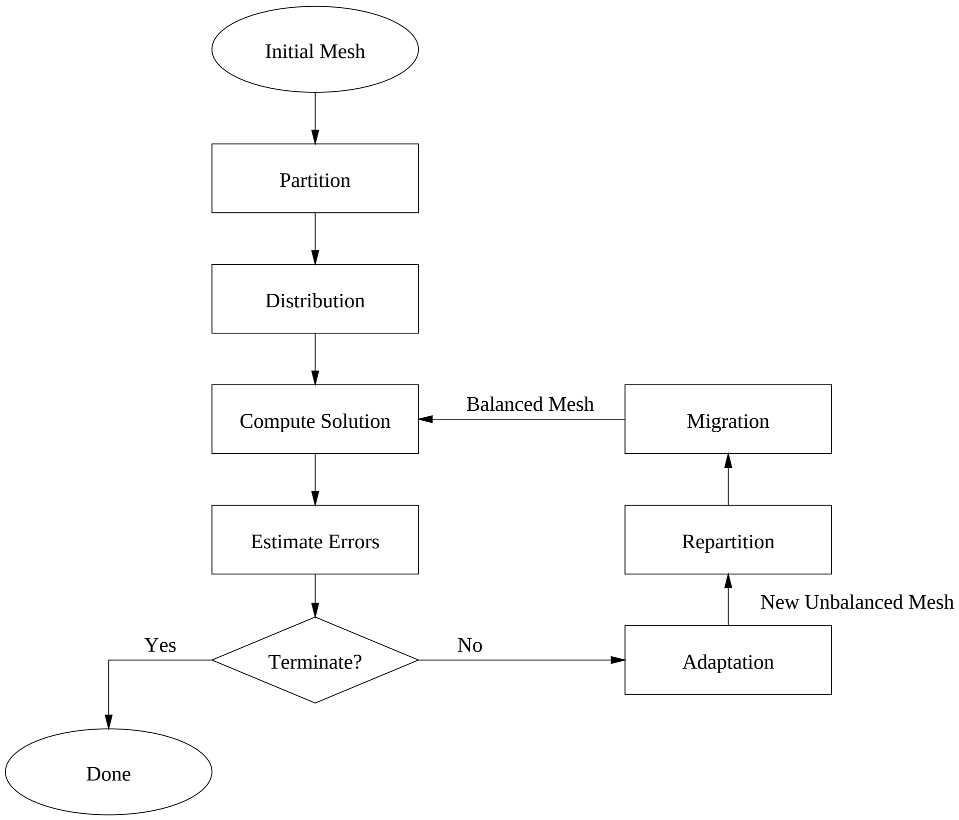


Figure 3.4: Outline of the different procedures executed by PARED to adaptively compute the solution of PDEs.

with the Chaco [48] library. In Chapter 5 we describe a new algorithm for partitioning adapted meshes.

The coordinator then distributes the mesh according to Π. In this phase, each processor initializes the references between their shared copies of vertices located in the internal boundaries between partitions. P_C maintains the dual graph during the simulation. G is a coarse approximation of a refined mesh M^t. Each vertex $w_a \in W$ has a corresponding element Ω_a in M^0. The weight of w_a is equal to the number of leaf elements in the refinement tree described in Section 3.2.1 rooted by Ω_a. Similarly, edge weights correspond to the communication required between two adjacent elements and are equal to the number of shared vertices, edges or faces between the leaf elements of two adjacent refinement trees. This weighted graph G is later used to repartition the mesh.

In the adaptive phase, a mesh is modified typically by first solving a linear system of equations $Ax = f$ associated with the current mesh and computing an error function that identifies elements that need to be refined and coarsened. PARED provides several iterative [6, 76] and direct solvers. In the FEM the matrix A is usually sparse and, on triangular or tetrahedral meshes, its non-zero structure is not regular. Our methods of choice are CG [49, 58, 84] on symmetric problems and GMRES [93] on non-symmetric ones, with and without preconditioning. Our solvers support an abstract representation of the system of equations so we can easily experiment with different sparse matrix representations.

On static problems, the mesh is typically refined until a desired error is obtained. Dynamic problems usually execute for a fixed number of iterations. When these termination conditions are not achieved, the mesh is locally adapted and the work rebalanced between processors. PARED uses the parallel h refinement algorithm presented in Chapter 4, which is based on longest edge bisection of triangular and tetrahedral unstructured meshes.

After the adaptation phase, PARED determines if a workload imbalance exists due to variations in the number of elements on individual processors. If so, it invokes the procedure described in Chapter 5 to decide how to repartition mesh elements between processors. Each processor sends to P_C the changes on the weights associated with the elements in the initial mesh. The coordinator updates the weights of the graph G, which it then uses to compute a new partition of the adapted mesh. P_C then informs each processor of the coarse elements and their refinement trees that need to be migrated and their destination.

These elements and the corresponding vertices are migrated between the processors according to the procedure explained in Chapter 6. PARED is then ready to resume another round of equation solving, mesh adaptation, mesh repartitioning, and work migration.

Chapter 4

Local Adaptation of Unstructured Meshes

4.1 Rivara's Longest Edge Bisection in Two Dimensions

Many h-refinement techniques [8] have been proposed to serially refine triangular and tetrahedral meshes. One widely used method is the longest-edge bisection algorithm proposed by Rivara [71, 72]. In its simplest form this recursive procedure (see Figure 4.1) splits each triangle Ω_a from a selected set of triangles R by adding an edge between the midpoint V_s of its longest side to the opposite vertex, creating two new triangles with equal area. This refinement method (as well as several other adaptation schemes) *propagate* the refinement to neighboring mesh elements so that an element $\Omega_b \notin R$ might also be refined to maintain the conformality of the mesh, as illustrated in Figure 4.2. The refinement of the shaded element in (a) creates a non-conforming vertex (black) on its longest edge. Because of continuity requirements of the basis functions used in the approximation the shaded triangle in (b) must now be refined to maintain a conforming mesh. There are several variations of this algorithm but in its simplest form it recursively bisects the non-conforming triangle by its longest edge. This process creates a new non-conforming vertices shown in (c) and (d). (g) shows the final conforming mesh.

The refinement of an element may propagate throughout the mesh (in the previous example only one element was initially in R but required the refinement of 6 elements). Nevertheless, this procedure is guaranteed to terminate because the edges it bisects increase in length. Building on the work of Rosenberg and Stenger [74] on bisection of triangles,

Bisect(Ω_i)
let V_p, V_q and V_r be vertices of the triangle Ω_i
let (V_p, V_q) be the longest side of Ω_i and let V_s be the midpoint of (V_p, V_q)
bisect Ω_i by the edge (V_r, V_s), generating two new triangles Ω_{i_1} and Ω_{i_2}
while V_s is a non-conforming vertex **do**
find a non-conforming triangle Ω_j adjacent to the edge (V_p, V_q)
Bisect(Ω_j)
end while

Figure 4.1: Longest edge (Rivara) bisection algorithm on triangular mesh.

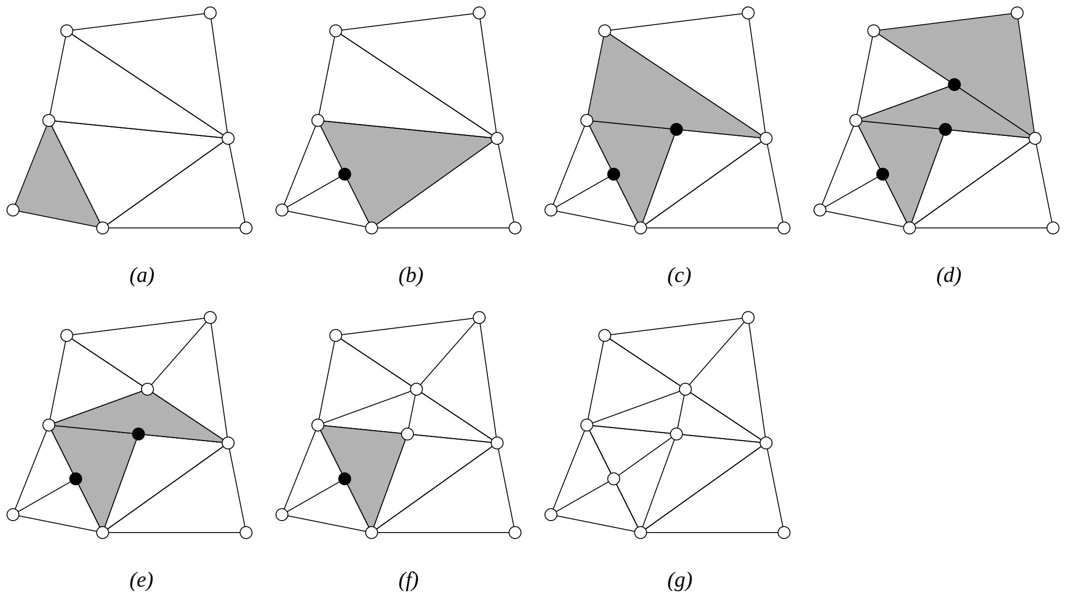


Figure 4.2: The refinement of the shaded triangle *(a)* propagates through the mesh creating three (black) non-conforming vertices *(b)-(f)*. Non-conforming elements adjacent to these vertices are recursively refined until all the mesh is conforming (g).

Rivara [71, 72] shows that this refinement procedure provably produces two dimensional meshes in which the smallest angle of the refined mesh is no less than half of the smallest angle of the original mesh. The resulting mesh is conforming and the transition between large and small elements is smooth.

4.2 Rivara's Longest Edge Bisection in Three Dimensions

The longest-edge bisection algorithm can be generalized to three dimensions [73] where a tetrahedron is bisected into two tetrahedra (Figure 4.3) by inserting a triangle between the

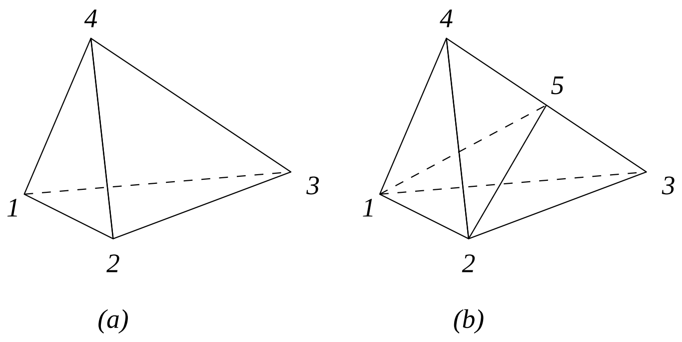


Figure 4.3: Refinement of a tetrahedron *(a)* by its longest edge $(3,4)$ into two tetrahedra (b). The refinement propagates to all the tetrahedra adjacent to the edge $(3,4)$.

midpoint of its longest edge and the two vertices not included in this edge. The refinement propagates to neighboring tetrahedra in a similar way. This procedure is also guaranteed to terminate, but unlike the two dimensional case, there is no known bound on the size of the smallest angle. This lower bound for the three dimensional case is unlikely to exist because the sum of the interior angles of a tetrahedron is not a constant and its longest edge is not necessarily opposite the largest angle [66]. Nevertheless, experiments conducted by Rivara [73] suggest that this method does not produce degenerate meshes in which the same angle is repeatedly divided.

In two dimensions there are several variations on the algorithm. For example a triangle can initially be bisected by the longest edge, but then its children are bisected by the non-conforming edge, even if it is that is not their longest edge [71]. In three dimensions, the bisection is always performed by the longest edge so that matching faces in neighboring tetrahedra are always bisected by the same common edge.

4.3 Parallel Refinement of Unstructured Meshes

In PARED the initial coarse mesh M^0 is assumed small enough so it can be stored in one processor and refined there using the serial refinement algorithms outlined in the previous sections. Nevertheless, as the number of elements and vertices increases in refined meshes, it is necessary to distribute the mesh between processors and perform its refinement in parallel.

Because the adaptation procedure is used to refine the mesh in regions of high relative error, when the mesh is distributed between processors it is likely that such regions are located in only one or few processors. The longest edge bisection algorithm and many other mesh refinement algorithms that propagate the refinement to guarantee conformality of the mesh are not local. The refinement of one particular triangle or tetrahedron Ω_a can propagate through the mesh and potentially cause changes in regions far removed from Ω_a. If neighboring elements are located in different processors, it is necessary to propagate this refinement across processor boundaries to maintain the conformality of the mesh.

In our parallel longest edge bisection algorithm each processor P_i iterates between a serial phase, in which there is no communication, and a parallel phase, in which each processor sends and receives messages from other processors. Let R be a global set of elements selected for refinement and let R_i be a subset of the elements of R assigned to processor P_i. In this case $R = \cup R_i$ and $R_i \cap R_j = \emptyset$ for $i \neq j$ because each element is assigned to only one processor.

In the serial phase, processor P_i refines the elements in R_i using the serial longest edge bisection algorithms outlined earlier. The refinement often creates shared vertices in the boundary between adjacent processors. P_i maintains a list for each of its neighboring processors P_j of the vertices shared with that processor that were created during the serial phase. To minimize the number of messages exchanged between P_i and P_j, P_i delays the propagation of refinement to P_j until P_i has refined all the elements in its processor.

The serial phase terminates when P_i has no more elements to refine. When this happens, P_i sends one message to every adjacent processor P_j on whose shared boundary it created one or more shared vertices. This message contains all the shared vertices that were created as a result of refining the elements in R_i. After sending these messages P_i listens for messages from other processors.

A processor P_i informs an adjacent processor P_j that some of its elements need to be refined by sending a message from P_i to P_j containing the non-conforming edges and the vertices to be inserted at their midpoint. Each edge is identified by its endpoints V_p and V_q and its remote references (see Figure 4.4 (a)). If V_p and V_q are shared vertices, then P_i has a remote reference to copies of V_p and V_q located in processor P_j. These references are included in the message, so that P_j can identify the non-conforming edge e and insert the new vertex V_s. A similar strategy is used when the edge is refined several times during the refinement phase, but in this case, the vertex V_s is not located at the midpoint of e.

Different processors can be in different phases during the refinement. For example, at any given time a processor can be refining some of its elements (serial phase) while neighboring processors have refined all their elements and are waiting for propagation messages (parallel

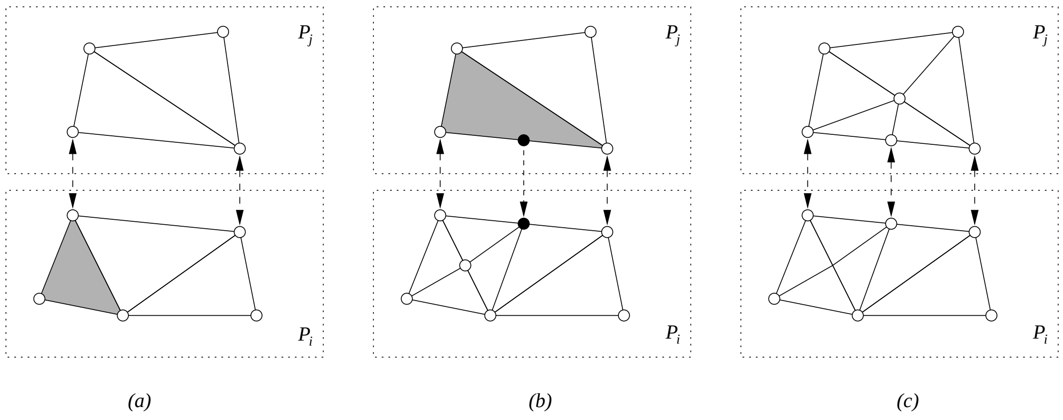


Figure 4.4: (a) In the parallel longest edge bisection algorithm some elements (shaded) are initially selected for refinement. (b) If the refinement creates a new (black) vertex on a processor boundary, the refinement propagates to neighbors. (c) Finally the references are updated accordingly.

phase) from adjacent processors. P_j waits until it has no elements to refine before receiving a message from P_i. For every nonconforming edge e included in a message to P_j, P_j creates its shared copy of the midpoint V_s (unless it already exists) and inserts the new non-conforming elements adjacent to V_s into a new set R_j' of elements to be refined. In this way the refinement propagates across the internal boundary between P_i and P_j to maintain the conformality of the global mesh. The remote references of the new vertices are updated accordingly. The copy of V_s in P_j must also have a remote reference to the copy of V_s in P_i. For this reason, when P_i propagates the refinement to P_j it also includes in the message a reference to its copies of shared vertices. These steps are illustrated in Figure 4.4. P_j then enters the serial phase again, where the elements in R_j' are refined.

The parallel algorithm is essentially the same for two- and three-dimensional meshes. In both cases triangles and tetrahedra are refined by their longest edge. The only difference is that in 2D, every triangle can have only one neighboring triangle over a shared edge so it needs to propagate the refinement to only one processor. Because a tetrahedron can have more than one neighboring tetrahedron over an edge and these tetrahedra can be located in different processors, the creation of a shared vertex in a 3D mesh might occur between one processor and several other processors.

4.3.1 The Challenge of Refining in Parallel

The description of the parallel refinement algorithm is not complete because refinement propagation across processor boundaries can create two synchronization problems. The first problem, *adaptation collision*, occurs when two (or more) processors decide to refine adjacent elements (one in each processor) during the serial phase, creating two (or more) vertex copies over a shared edge, one in each processor. It is important that all copies refer to the same logical vertex because in a numerical simulation each vertex must include the contribution of all the elements around it.

The second problem that arises, *termination detection*, is the determination that a refinement phase is complete. The serial refinement algorithm terminates when the processor has no more elements to refine. In the parallel version, a processor might have no elements to refine but it might still be waiting for refinement propagations from adjacent processors. Therefore, termination is a global decision that cannot be determined by an individual processor and requires a collaborative effort of all the processors involved in the refinement. We now describe these two problems and our solutions.

Adaptation collision

During the serial phase, every processor P_i refines a selected set R_i of its elements in parallel (these elements are shaded in the Figure 4.5). In the event that two or more processors P_i and P_j both refine an edge e that is shared between mesh elements located in different processors, they create a vertex over the shared edge before propagating the refinement to the others. In this case it is important that these processors do not consider these vertices to be distinct.

When a message arrives at P_j with the instruction to create a vertex V_r at the midpoint of a shared edge e, P_j can detect if the edge was already refined by using e to check the elements in P_j that contain it. If any of those elements was refined over e then P_j already has a copy of the vertex V_r. In that case, P_j does not create another copy of the same vertex but instead it adds to it a reference to the remote copy in P_i. As we mentioned earlier, this reference is included in the message from P_i to P_j. Otherwise, P_j creates the vertex V_r and then refines the non-conforming elements adjacent to it after which it adds a reference to that remote copy in P_i.

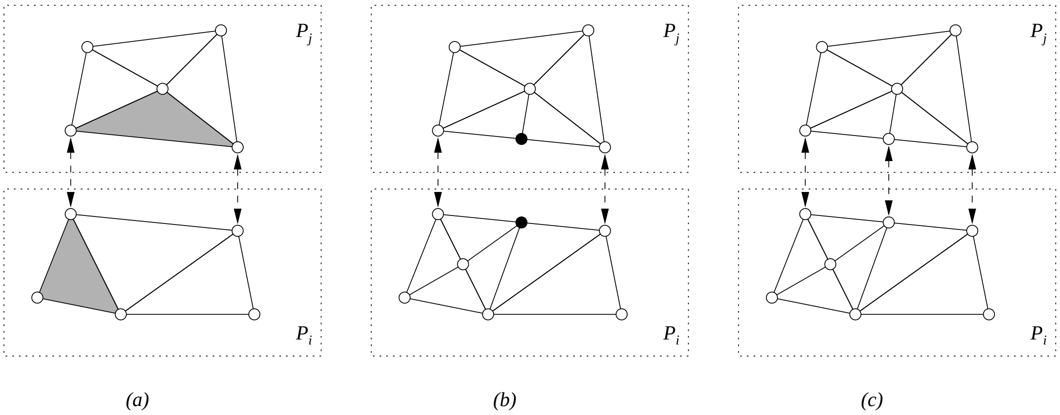


Figure 4.5: (a) Both processors select (shaded) mesh elements for refinement. The refinement propagates to a neighboring processor (b) resulting in more elements being refined (c). The refinement process is complete when each new shared vertex has a reference to its proxy on a neighboring processor.

Termination detection

The other problem mentioned earlier is the detection of the termination of refinement. Termination of refinement cannot be determined by individual processors because, although a processor P_i may have adapted all of its mesh elements in R_i, it cannot determine whether this condition holds for all other processors. For example, at any given time, no processor might have any more elements to refine. Nevertheless, the refinement cannot terminate because there might be some propagation messages in transit.

The algorithm for detecting the termination of parallel refinement is based on Dijkstra's general distributed termination algorithm [28, 11]. A global termination condition is reached when no element is selected for refinement. Hence if R is the set of all elements in the mesh currently marked for refinement, then the algorithm finishes when $R = \emptyset$.

The termination detection procedure uses message acknowledgments. For every propagation message that P_j receives, it maintains the identity of its source (P_i) and to which processors P_k it propagated refinements. Each propagation message is acknowledged. P_j acknowledges to P_i after it has refined all the non-conforming elements created by P_i's message and has also received acknowledgments from all the processors to which it propagated refinements.

A processor P_i can be in two states: an inactive state is one in which P_i has no elements to refine (it cannot send new propagation messages to other processors) but can receive messages. If P_i receives a propagation message from a neighboring processor, it moves from an inactive state to an active state, selects the elements for refinement as specified in the message and proceeds to refine them. Let R_i be the set of elements in P_i needing refinement. A processor P_i becomes inactive when:

- P_i has received an acknowledgment for every propagation message it has sent.
- P_i has acknowledged every propagation message it has received.
- $R_i = \emptyset$.

Using this definition, a processor P_i might have no more elements to refine ($R_i = \emptyset$) but it might still be in an active state waiting for acknowledgments from adjacent processors. When a processor becomes inactive, P_i sends an acknowledgment to the processors whose propagation message caused P_i to move from an inactive state to an active state.

We assume that the refinement is started by the coordinator processor, P_C (see Figure 4.6). At this stage, P_C is in the active state while all the processors are in the inactive state. P_C initiates the refinement by sending the appropriate messages to other processors. This message also specifies the adaptation criterion to use to select the elements R_i for refinement in P_i.

When a processor P_i receives a message from P_C, it changes to an active state, selects some elements for refinement either explicitly or by using the specified adaptation criterion, and then refines them using the serial bisection algorithm, keeping track of the vertices created over shared edges as described earlier. When it finishes refining its elements, P_i sends a message to each processor P_j on whose shared edges P_i created a shared vertex. P_i then listens for messages.

Only when P_i has refined all the elements specified by P_C and is not waiting for any acknowledgment message from other processors does it send an acknowledgment to P_C. Global termination is detected when the coordinator becomes inactive. When P_C receives an acknowledgment from every processor this implies that no processor is refining an element and that no processor is waiting for an acknowledgment. Hence it is safe to terminate the refinement. P_C then broadcasts this fact to all the other processors.

4.3.2 The Message Model for Refinement

The parallel refinement algorithm described above uses three different types of message:

for each processor P_i **do**
  P_C sends a *RefineMsg* to P_i specifying elements to refine and/or an adaptation criterion
end for
for each processor P_i **do**
  P_C waits for a *DoneMsg* from P_i
end for
for each processor P_i **do**
  P_C sends a *FinishMsg* to P_i to indicate the termination of the refinement
end for

Figure 4.6: The messaging algorithm executed by the coordinator P_C.

- A *RefineMsg* received by P_i indicates that some elements in P_i must be refined. At the beginning of the refinement phase, P_C sends a *RefineMsg* to all the processors. The processors select their elements to refine according to the specified refinement criterion. This message is also used when the refinement propagates between the processors. In this case, the message indicates which edges to refine and a reference to the new vertices.

- A *DoneMsg* is used to acknowledge each *RefineMsg* received by P_i from another processor P_j (including the coordinator).

- A *FinishMsg* from P_C to each processors signals the end of the refinement phase.

The steps executed by the coordinator P_C and by the other processors P_i are different. Figure 4.6 shows the algorithm executed by the coordinator. P_C initially broadcasts a *RefineMsg* to all the other processors to start the refinement phase which identifies the initial elements to refine. P_C then waits until it receives a *DoneMsg* from all the other processors. At this point, P_C broadcasts a *FinishMsg*.

Every other processor P_i is in a loop waiting for messages (Figure 4.7). P_i does not know which type of message it is going to receive next, so it has to be able to receive any of them. This algorithm is nondeterministic because at any given time P_i can receive a message from any of its neighbors or the coordinator. In MPI, messages received from the same source are received in order, but there is no guarantee about the order of messages from different sources. Therefore in different executions the sequence in which messages are received can be different.

If P_i receives a message whose type is a *RefineMsg* from the coordinator, then P_i selects a set of its elements for refinement and refines them using the serial bisection algorithm. If the *RefineMsg* is from another processor P_j, then for each edge specified in the message, P_i

while *true* **do**
  P_i waits for a message *msg* from other processors (including P_C)
  if *msg* type = *FinalizeMsg* **then**
    break
  else if *msg* type = *RefineMsg* **then**
    refine the specified elements using a serial refinement algorithm
    if refinement does not propagate to other processors **then**
      send a *DoneMsg* to the source processor of *msg*
    else
      send a *RefineMsg* to every processor to which the refinement propagates
    end if
  else if *msg* type = *DoneMsg* **then**
    let *msg* be a response to a previous *RefineMsg* from P_i to P_j. Assume that the refinement that propagated to P_j was caused by a message received by P_i from P_k
    if P_i is waiting for no more responses to *RefineMsg* **then**
      send a *DoneMsg* to P_k
    end if
  end if
end while

Figure 4.7: Message model for refinement by processor P_i.

creates or updates the reference to the corresponding vertex of that edge and then refines the adjacent non-conforming elements. If the refinement of the elements specified in the *RefineMsg* creates vertices over edges shared with adjacent processors, then P_i sends a corresponding *RefineMsg* to every such processor. Otherwise it sends a *DoneMsg* to the source of the *RefineMsg*.

As mentioned earlier, every *RefineMsg* must be acknowledged by a *DoneMsg* which is returned to the source of the *RefineMsg* when P_i has completed all the propagations caused by that message. This is true when

- P_i has refined all its local elements specified in the *RefineMsg* message, and
- P_i has also received a *DoneMsg* from every other processor to which each refinement has propagated.

P_i stops waiting for messages when it receives a *FinalizeMsg* from P_C. As suggested in Figure 4.8 (b), a processor might have returned a *DoneMsg* to the coordinator but might still have more elements to refine because it received a *RefineMsg* from an adjacent processor (P_2 in this case).

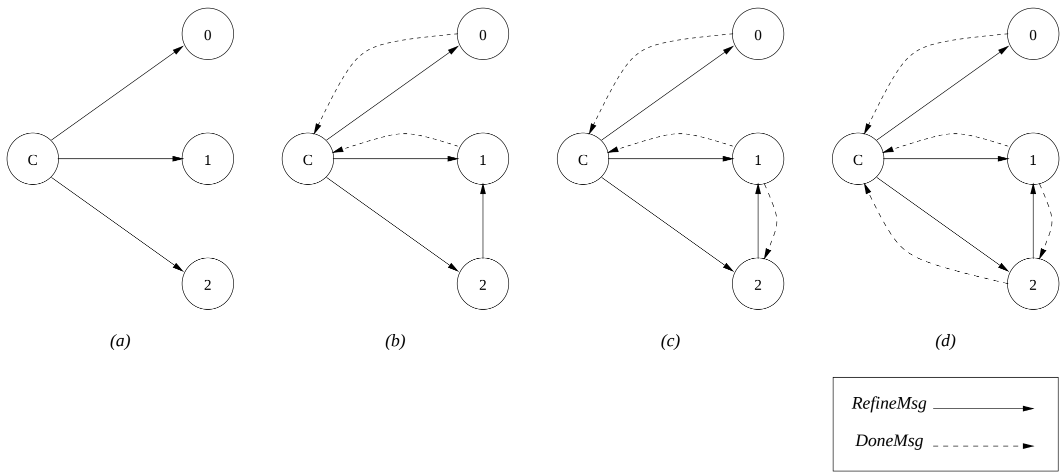


Figure 4.8: (a) P_C sends a *RefineMsg* to all processors. (b) Because P_0 and P_1 do not propagate the refinement to other processors they return a *DoneMsg* to P_C. P_2 propagates the refinement to P_1. (c) P_1 refines the nonconforming elements in P_1 without propagating the refinement to other processors and then sends a *DoneMsg* to P_2. (d) A *DoneMsg* is sent from P_2 to P_C.

When P_i propagates refinements to P_j, P_i does not block while waiting for an acknowledgment from P_j. Instead it waits in a loop accepting messages from other processors. A deadlock is very likely to occur if P_i were to block. This scheme, propagating the refinement and listening for messages, allows us to handle cases like the one shown in Figure 4.9 in which the refinement initiated in P_i propagates back to the same processor. In this example, the processors denoted P_j and P_k in Figure 4.7 are actually the same.

4.4 Properties of Parallel Refined Meshes

Our parallel refinement algorithm is guaranteed to terminate. In every serial phase the longest edge bisection algorithm is used. In this algorithm the refinement propagates towards progressively longer edges and will eventually reach the longest edge in each processor. Between processors the refinement also propagates towards longer edges. Global termination is detected by using the global termination detection procedure described in the previous section. The resulting mesh is conforming. Every time a new vertex is created over a shared edge, the refinement propagates to adjacent processors. Because every element is always bisected by its longest edge, for triangular meshes the results by Rosenberg and Stenger on the size of the minimum angle of two-dimensional meshes also hold.

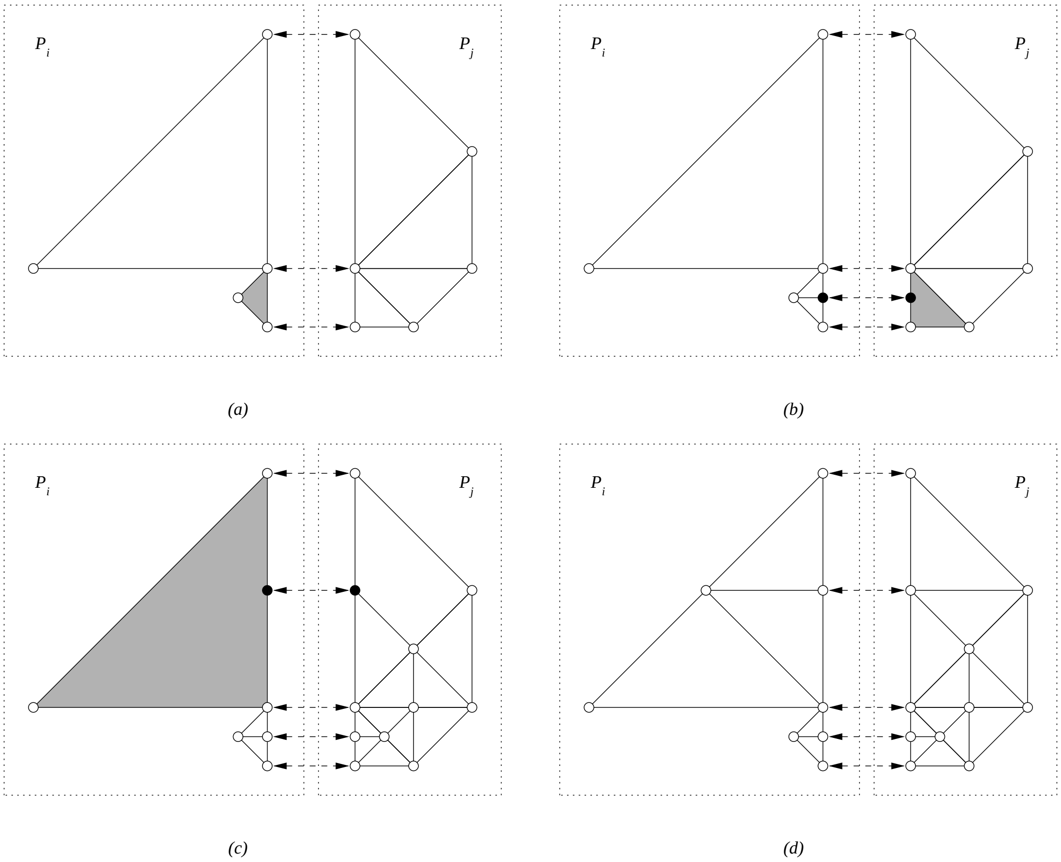


Figure 4.9: A cyclic propagation using the parallel longest edge bisection algorithm. (a) The refinement of an element (shaded) in P_i propagates to P_j in (b) and back to P_i in (c). When the procedure terminates the mesh is conforming as shown in (d).

It is not immediately obvious if the resulting meshes obtained by the serial and parallel longest edge bisection algorithms are the same or if different partitions of the mesh generate the same refined mesh. As we mentioned earlier, messages can arrive from different sources in different orders and elements may be selected for refinement in different sequences.

We now show that the meshes that result from refining a set of elements R from a given mesh M using the serial algorithms described in Sections 4.1 and 4.2 and the parallel algorithm presented in Section 4.3, respectively, are the same. In this proof we use the general longest-edge bisection (GLB) algorithm outlined in Figure 4.10 where the order in which elements are refined is not specified. In a parallel environment, this order depends on the partition of the mesh between processors. After showing that the resulting refined mesh is independent of the order in which the elements are refined using the serial GLB algorithm, we show that every possible distribution of elements between processors and every order of parallel refinement yields the same mesh as would be produced by the serial algorithm.

Theorem 1 The mesh that results from the refinement of a selected set of elements R of a given mesh M using the GLB algorithm is independent of the order in which the elements are refined.

> **Proof** An element Ω_a is refined using the GLB algorithm if it is in the initial set R or refinement propagates to it. An element $\Omega_a \notin R$ is refined if one of its neighbors creates a non-conforming vertex at the midpoint of one of its edges. The refinement of Ω_a by its longest edge divides the element into two nested sub-elements Ω_{a_1} and Ω_{a_2} called the children of Ω_a. These children are in turn refined by their longest edge if one of their edges is non-conforming. The refinement procedure creates a forest of trees of nested elements where the root of each tree is an element in the initial mesh M and the leaves are unrefined elements. For every element $\Omega_a \in M$, let τ_a be the refinement tree of nested elements rooted at Ω_a when the refinement procedure terminates.
>
> Using the GLB procedure elements can be selected for refinement in different orders, creating possible different refinement histories. To show that this cannot happen we assume the converse, namely, that two refinement histories H_1 and H_2 generate different refined meshes, and establish a contradiction. Thus, assume that there is an element $\Omega_a \in M$ such that the refinement trees τ_a^1 and τ_i^a, associated with the refinement histories H_1 and H_2 of Ω_a respectively, are

Let R be a set of elements to be refined
while there is an element $\Omega_i \in R$ **do**
  bisect Ω_s by its longest edge
  insert any non-conforming element Ω_b into R
end while

Figure 4.10: General longest-edge bisection (GLB) algorithm.

different. Because the root of τ_a^1 and τ_a^2 is the same in both refinement histories, there is a place where both trees first differ. That is, starting at the root, there is an element Ω_b that is common to both trees but for some reason, its children are different. Because Ω_b is always bisected by the longest edge, the children of Ω_b are different only when Ω_b is refined in one refinement history and it is not refined in the other. In other words, in only one of the histories does Ω_b have children.

Because Ω_b is refined in only one refinement history, then $\Omega_b \notin R$, the initial set of elements to refine. This implies that Ω_b must have been refined because one of its edges became non-conforming during one of the refinement histories. Let D_1 be the set of elements that are present in both refinement histories, but are refined in H_1 and not in H_2. We define D_2 in a similar way.

For each refinement history, every time an element is refined, it is assigned an increasing number. Select an element Ω_a from either D_1 or D_2 that has the lowest number. Assume that we choose Ω_a from D_1 so that Ω_a is refined in H_1 but not in H_2. In H_1, Ω_a is refined because a neighboring element Ω_b created a non-conforming vertex at the midpoint of their shared edge e. Therefore Ω_b is refined in H_1 but not in H_2 because otherwise it would cause Ω_a to be refined in both sequences. This implies that Ω_b is also in D_1 and has a lower refinement number than Ω_a contradicting the definition of Ω_a. It follows that the refinement histories are the same. $\Box$

Consider now the parallel refinement algorithm described in Section 4.3. Although it refines elements on individual processors serially, these processors refine their elements in parallel. As shown below, the resultant mesh is the same as would be produced in the serial case.

Corollary 1 Given a mesh M and a set of elements R to refine, for every partition of the elements between processors, the parallel GLB algorithm generates the same refined mesh as does the serial GLB algorithm.

> **Proof** For every partition of the elements and every order of refinement within processors that is consistent with the GLB algorithm, there is a linear order that can be established of element refinements. From Theorem 1 it follows that the refined mesh associated with every linear ordering is the same. □

Clearly the result of Theorem 1 holds for any nested refinement algorithm in which the refinement of elements occurs in an order that is independent of which side or face is non-conforming.

4.5 Mesh Coarsening

Because in PARED refined elements are not destroyed in the refinement tree, the mesh can be coarsened by replacing all the children of an element by their parent. If a parent element Ω_a is selected for coarsening, it is important that all the elements Ω_b that are adjacent to the longest edge of Ω_a are also selected for coarsening. If neighbors are located in different processors then only a simple message exchange is necessary. This algorithm generates conforming meshes: a vertex is removed only if all the elements that contain that vertex are all coarsened. It does not propagate like the refinement algorithm and it is much simpler to implement in parallel.

4.6 Parallel Refinement of Unstructured Meshes: Experimental Results

In the previous section we have shown that the meshes obtained using the parallel refinement algorithm described in this paper are the same as those obtained using the widely used serial longest-edge bisection algorithm. Therefore the quality of the resulting meshes using any of the measures mentioned in Chapter 2.3 is the same as that obtained using serial refinement.

To evaluate the performance of our parallel algorithm and to show that refinement does not incur any significant overhead we performed a series of tests using a network of four to thirty-two Sun Ultra-1 workstations. Using simple meshes with elements of regular shape we demonstrate that the performance of the parallel refinement algorithm strongly depends on the quality of the partition of the mesh between processors. If many adjacent elements are located in different processors (for example, when elements are randomly assigned to processors) the cost of communication is very high. However, for partitions of the mesh with relatively short boundaries between subdomains there is a reasonable communication overhead.

We also show the performance of the refinement algorithm on irregular unstructured meshes is similar to that observed on the simpler regular ones defined below. In Chapter 9, we examine the performance of our refinement algorithm when locally adapting unstructured meshes and compare it with the other phases of the adaptation procedure.

4.6.1 Global Refinement of Regular Meshes

The performance of our parallel refinement algorithm is dependent on the quality of the partition of the mesh. Starting from the regular two- and three-dimensional meshes defined in the unit square and cube, as shown in Figure 4.11, we performed successive global parallel refinements of all the elements of the mesh. In our two-dimensional example we start with a mesh that contains 256 elements and 145 vertices and after 15 successive refinements of each element we obtain a mesh with 4,194,304 elements and 2,099,201 vertices. The initial three-dimensional mesh contains 1,536 elements and 426 vertices. After 12 refinements its number of elements and vertices grows to 3,145,728 and 536,769, respectively. The total number of elements and vertices in each fine mesh M^t in each refinement phase is shown in Table 4.1. The global refinement procedure on these regular meshes does not *overrefine* (where some element not initially selected for refinement also get refined to obtain a conforming mesh) as we will in future examples: at each level, the selected elements are refined once and the size if the mesh is doubled.

After a mesh M^t is refined, the resulting fine mesh M^{t+1} is repartitioned between processors. PARED allows the user to choose from a large variety of partitioning algorithms. One of these is Multilevel-KL, a graph partitioning algorithm offered as part of Chaco [48]. It generates high quality partitions of meshes but at the cost of an overhead that is prohibitive for large meshes. Multilevel-KL was used in the experiments described below to provide good partitions for the initial assignment of elements to processors. Because Multilevel-KL is a serial algorithm, an entire graph must be moved to one processor to partition it. For very large refined meshes, the time required by Multilevel-KL is very large (on the order of 20 minutes in some cases) and required the use of a Sun server with 2 GB of memory. This is actually the limiting factor on the size of the meshes that we can handle.

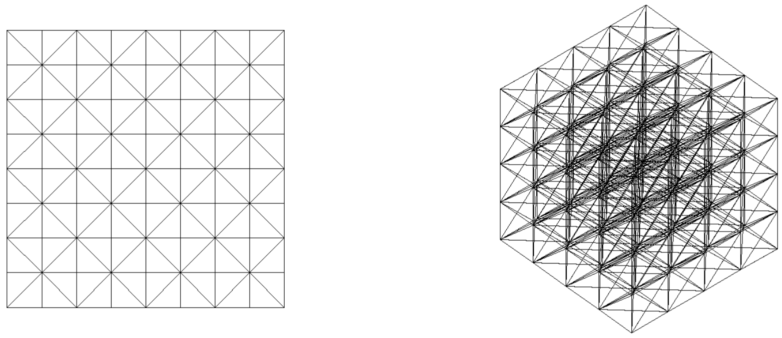


Figure 4.11: Initial two- and three-dimensional regular meshes.

In this section we give examples that show that we cannot study the refinement algorithm independently of the partition of the mesh. In these examples, every processor executes a similar amount of work to refine the mesh. A close to optimal partition is obtained by using the Multilevel-KL partition mentioned above. On the other hand, a balanced random distribution of the mesh requires the same amount of serial work in each processor (every element is refined once) but, because every new vertex is likely to be shared between two or more processors, it requires communication between them. A snapshot of the refined regular meshes partitioned randomly and by Multilevel-KL from a random partition are shown in Figure 4.12.

We used three different measures to evaluate the performance of the parallel refinement algorithm. For every processor, we count the number of new edge refinements sent to other processors during each of the successive global refinements of the mesh. This is a measure of the amount of communication occurring in the parallel refinement algorithm. The maximum number of edge refinements sent by any processor to all its neighbors in each level using both partitioning strategies are shown in Table 4.2 for the regular two-dimensional mesh and in Table 4.3 for the three-dimensional mesh. The amount of communication during refinement required for a random distribution of the mesh is significantly larger than for the mesh partitioned using Multilevel-KL. Also, the number of new shared edges using Multilevel-KL does not necessarily increase with each level of refinement even on a regular mesh because it depends on the actual partition of the mesh between processors. Therefore, it is difficult to predict the amount of communication required by the refinement algorithm even in these simple examples.

These differences in the amount of communication also have a great impact on the refinement time as we increase the size of the mesh and the number of processors. For the

	2D Regular Mesh		3D Regular Mesh	
Level	**Elements**	**Vertices**	**Elements**	**Vertices**
1	256	145	1536	429
2	512	289	3072	729
3	1024	545	6144	1241
4	2048	1089	12288	2969
5	4096	2113	24576	4913
6	8192	4225	49152	9009
7	16384	8321	98304	22065
8	32768	16641	196608	35937
9	65536	33025	393216	68705
10	131072	66049	786432	170001
11	262144	131585	1572864	274625
12	524288	263169	3145728	536769
13	1048576	525313		
14	2097152	1050625		
15	4194304	2099201		

Table 4.1: Number of elements and vertices of meshes obtained by the repeated refinement of regular two- and three-dimensional meshes.

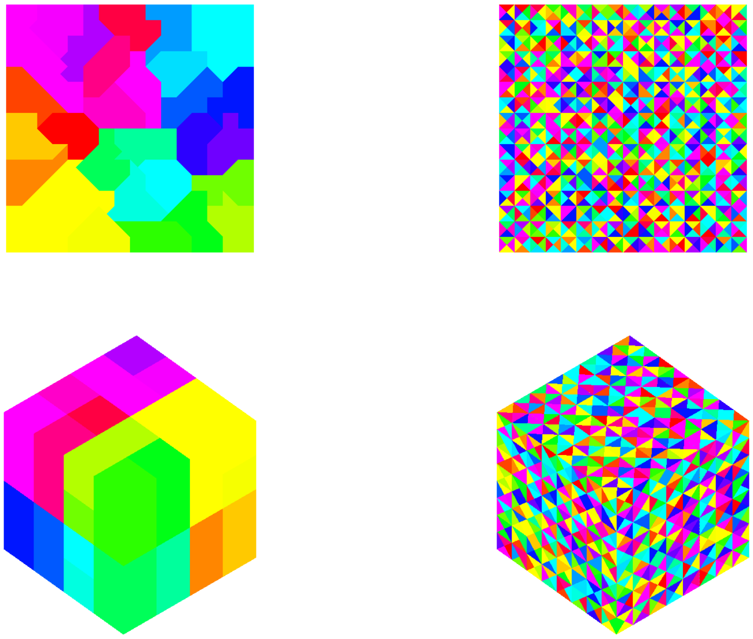


Figure 4.12: Regular two- and three-dimensional regular meshes partitioned between 32 processors using a partition provided by Multilevel-KL (left) and a random partition (right).

	4 Processors		8 Processors		16 Processors		32 Processors	
Level	**M-KL**	**Rand**	**M-KL**	**Rand**	**M-KL**	**Rand**	**M-KL**	**Rand**
1	6	64	6	39	6	20	4	12
2	4	120	6	90	5	43	6	26
3	0	237	0	170	3	89	3	55
4	13	504	11	320	9	167	9	84
5	4	1024	8	677	9	348	5	151
6	19	2075	24	1326	21	615	21	281
7	11	4129	10	2740	25	1233	13	504
8	43	8095	33	5314	32	2421	28	1003
9	25	16626	24	10800	30	4840	24	1964
10	66	32786	82	21307	70	9789	48	3910
11	48	66010	55	42844	83	19399	76	7899
12	99		158	85425	111	38291	100	15525
13			120		145	76806	89	30952
14					241		206	61452
15							207	

Table 4.2: Largest number of edge refinements sent by a processor to other processors in each refinement level obtained by performing successive refinements on a regular 2D mesh partitioned randomly and with the Multilevel-KL algorithm.

	4 Processors		8 Processors		16 Processors		32 Processors	
Level	**M-KL**	**Rand**	**M-KL**	**Rand**	**M-KL**	**Rand**	**M-KL**	**Rand**
1	23	422	21	559	21	521	15	302
2	41	644	30	890	32	958	16	606
3	0	1273	0	1952	0	2044	0	1569
4	78	3230	80	4265	51	3800	24	2482
5	90	4731	87	7393	66	7800	46	5669
6	7	10199	16	15755	18	16997	4	12617
7	260	25778	229	34332	343	32578	191	20808
8	425		323	57350	302	65188	203	49028
9	141		125		98	139756	101	104314
10			1278		986		886	173428

Table 4.3: Largest number of edge refinements sent by a processor to other processors in each refinement level obtained by performing successive refinements on a regular 3D mesh partitioned randomly and with the Multilevel-KL algorithm.

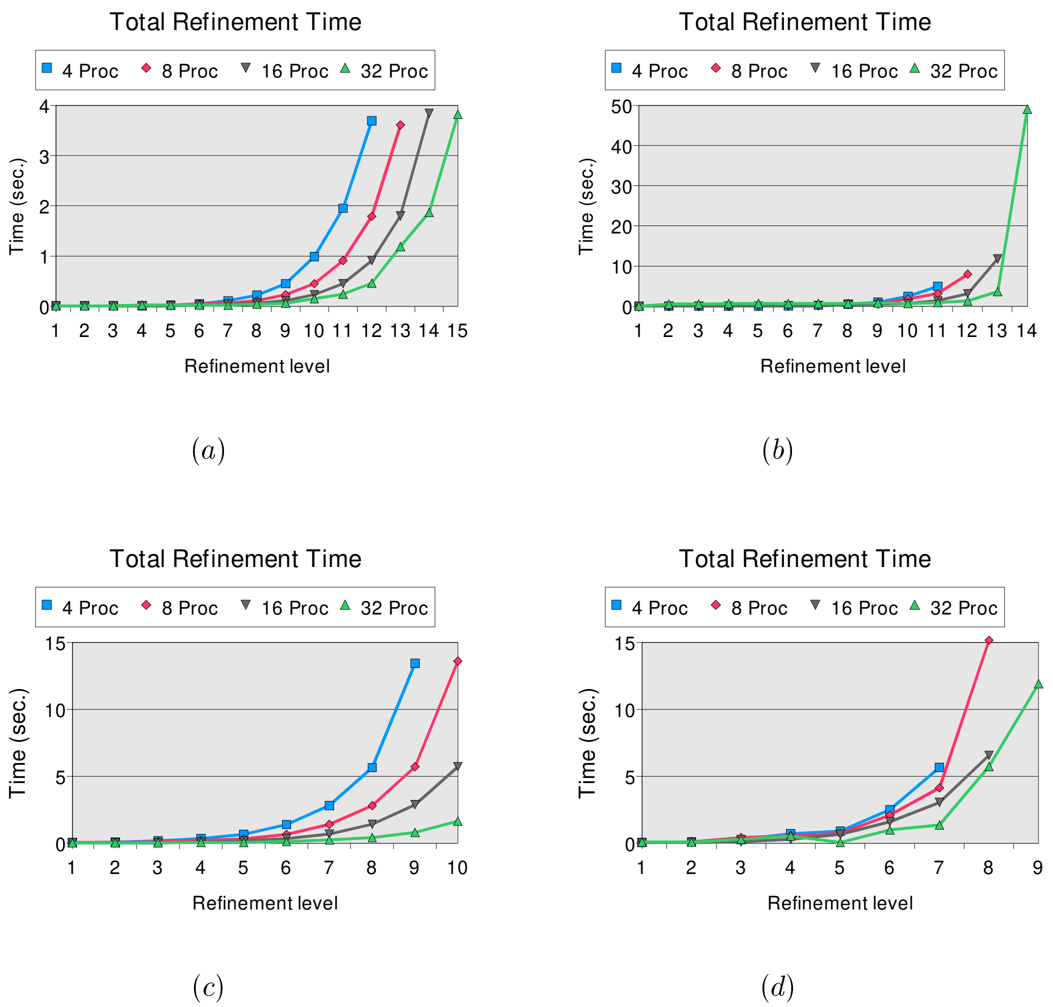
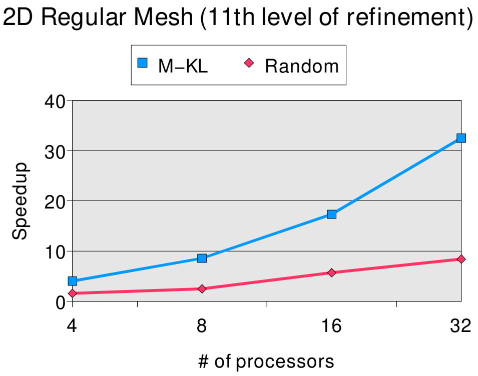
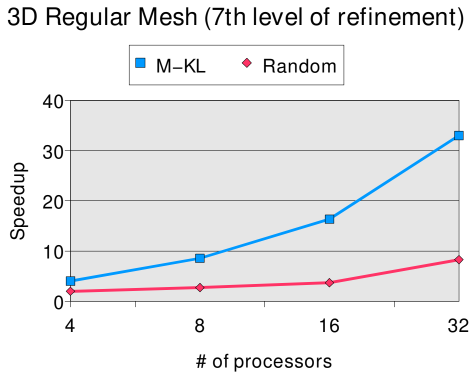


Figure 4.13: Refinement time for the global successive refinements of a regular two dimensional mesh with a (a) Multilevel-KL partition and (b) random partition. (c) and (d) show the same times for the refinement of a regular three-dimensional mesh.

regular two-dimensional mesh, the total time for each parallel refinement phase of Multilevel-KL and random derived partitions is shown in Figures 4.13 (a) and (b) respectively. For the three- dimensional mesh, these times are shown in Figures 4.13 (c) and (d). These figures illustrate that the refinement time for randomly distributed meshes is much larger than that for the same meshes partitioned using Multilevel-KL. It is also possible to process larger meshes with Multilevel-KL because it requires fewer shared copies of vertices and less memory to store them.



(*a*)



(*b*)

Figure 4.14: Relative speedups for the refinement of a regular mesh using Multilevel-KL and random partition for a (*a*) two-dimensional regular mesh of 262,144 elements and (*b*) three-dimensional regular mesh of 98,304 elements.

The speedup in the time to refine regular 2D and 3D meshes is shown in Figure 4.14 for random and Multilevel-KL-derived partitions. We show only the time to globally refine all elements and propagate changes across processor boundaries. The time to repartition a random mesh with Multilevel-KL and to migrate the mesh are not included. In both the 2D and 3D cases the speedup that is obtained with the well partitioned mesh is much higher than for the randomly partitioned mesh and almost linear. Also, we can report that the refinement time is dominated by the time for local refinement of meshes on individual processors, not the cost of interprocessor communication. The communication cost overhead is an important factor in the total cost of the refinement algorithm as the meshes grow in size for random partitions. This is due to the large number of new shared vertices created in randomized partitions.

4.6.2 Global Refinement of Irregular Meshes

The repeated bisection of all the elements in the regular 2D and 3D meshes considered above never creates a mesh whose smallest angle is less than 45 degrees in the the case of two-dimensional meshes and a smallest solid angle less than 15 degrees in the case of three-dimensional meshes. Also, it never refines an element more than once to maintain the conformality of the mesh.

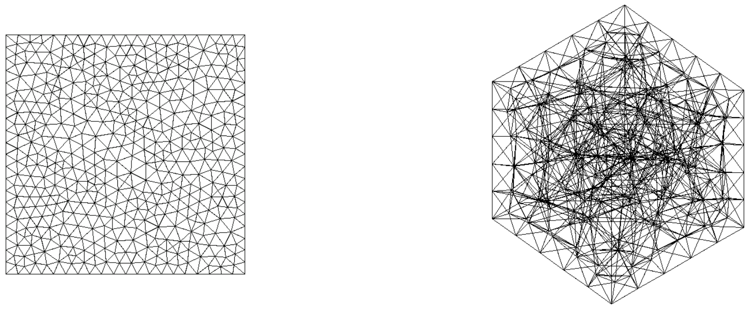


Figure 4.15: Initial two- and three-dimensional irregular meshes on a square and a cube.

	2D Irregular Mesh		3D Irregular Mesh	
Level	**Elements**	**Vertices**	**Elements**	**Vertices**
1	2239	1171	3179	728
2	5094	2629	10166	2177
3	11110	5677	31935	6472
4	23749	12040	101051	19554
5	49915	25214	320424	60762
6	103585	52125	1007549	187599
7	212831	106941	3177713	585128
8	434119	217726		
9	880745	441436		
10	1779869	891269		
11	3586501	1795390		

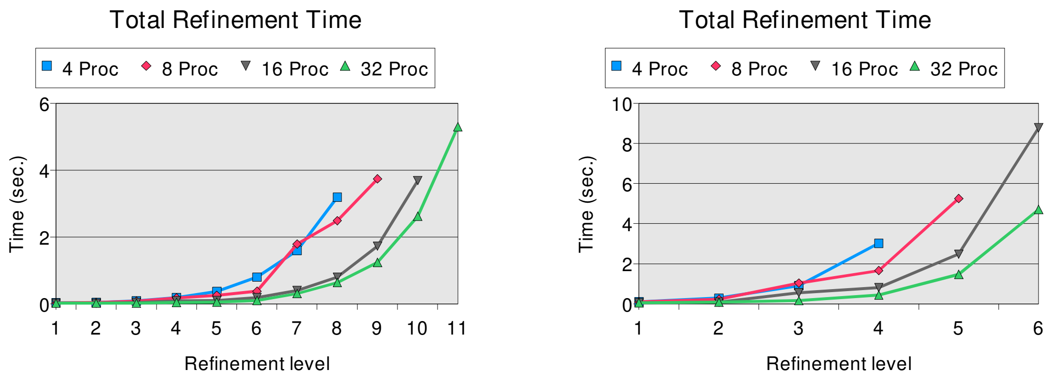
Table 4.4: Number of elements and vertices that result from successive refinement of irregular two- and three-dimensional meshes.

Typical unstructured finite-element meshes are similar to those shown in Figure 4.15. Successive refinement of these meshes more than doubles the size of the mesh, as shown in Table 4.4. That is, an element might be refined more than once during the refinement phase to maintain a conforming mesh, which is not the case in the previous regular examples.

Nevertheless, the performance of the algorithm using these irregular meshes is similar to our previous results. Table 4.5 shows that the maximum number of new edge refinements sent by any processor to another in each refinement is relatively small. The total refinement time in the different steps of the refinement are comparable to those in the regular meshes, as shown in Figure 4.16 for meshes of similar sizes.

Level	2D Irregular Mesh				3D Irregular Mesh			
	4 P	8 P	16 P	32 P	4 P	8 P	16 P	32 P
1	21	22	17	15	51	52	51	48
2	14	22	16	14	146	109	92	79
3	19	31	17	23	213	162	187	153
4	26	29	31	20	433	383	336	307
5	31	36	36	33		811	744	613
6	37	47	35	41			1613	1221
7	87	56	65	62				
8	73	131	83	74				
9		129	143	101				
10			190	143				
11				206				

Table 4.5: Largest number of new edge refinements sent by a processor to other processors in each refinement phase by performing successive refinements on an irregular 2D and 3D meshes (Multilevel-KL partitions).



(*a*) (*b*)

Figure 4.16: Refinement time for the global successive refinements of a irregular (*a*) two-dimensional mesh and (*b*) three-dimensional irregular mesh when Multilevel-KL is used.

Chapter 5

Mesh Partitioning and Repartitioning

5.1 Introduction

In this chapter we describe *Parallel Nested Repartitioning* (PNR), a new partitioning algorithm sketched in [18], that has its roots in multilevel partitioning algorithms [10, 47]. Our method quickly produces low cost, high quality partitions that minimize the amount of data that needs to be moved to rebalance a workload after a mesh adaptation. It has a very natural parallel implementation that allows us to repartition adapted meshes of arbitrary size. We demonstrate the effectiveness of PNR both analytically and empirically.

Starting with a coarse initial mesh, PARED locally adapts the mesh using an h-refinement algorithm until an error criterion is met. Attached to each coarse element is a refinement history tree whose leaves are the most refined elements into which the coarse element has been refined. PARED partitions the mesh by elements. A partition is obtained by PNR from the dual graph G of the coarse mesh.

Although G has much less information than is available in the adapted mesh M, we demonstrate through experiment and analysis that partitioning M using G gives balanced partitions with cut sizes comparable to those provided by standard partitioning algorithms [10, 47]. Unfortunately, as we show, small changes in M can produce very different partitions of it using either our initial version of PNR or standard partitioning algorithms. This *mesh migration problem* has also been addressed by others [12, 15, 78]. Biswas [12] permutes the subsets produced by a standard algorithm to minimize data movement. We show that this heuristic can still require that half the elements be migrated. Walshaw *et al* [15] and Schloegel, Karypis and Kumar [78] determine the number of elements that must move between processors to rebalance them using a technique of Hu and Blake [50] and then try to keep the cut size small by migrating elements on the boundaries between processors. Their heuristics require several iterations in which the same regions of the mesh are repeatedly migrated.

Our approach to mesh migration uses a modified version of PNR, which we show through experiment and analysis very quickly produces balanced partitions with low cut size and low migration cost. Our approach is to use a multilevel partitioning algorithm to partition G without repartitioning the coarsest graph but using a variation of the Kernighan-Lin heuristic as contracted graphs are expanded.

In this chapter we show the power of PNR through an experiment with PARED in which we track a disturbance through space over time. We show that PNR migrates a very small number of elements while maintaining a partition quality comparable to that produced by RSB [10].

5.2 Partitioning Finite Element Meshes

Parallel implementations of the FEM allocate a portion of the mesh $M(D,V)$ to each processor, where D is a set of elements and V is a set of vertices. For efficiency reasons, the partition of M should assign a similar amount of work to each processor while at the same time minimizing the communication needed between processors.

Two standard methods are used to partition a mesh M between p processors. The first removes a subset of its edges leaving p connected components of vertices which are then mapped to the processors. In this *partitioning by vertices* of the mesh, each vertex is assigned to a unique processor. PARED uses a second method. Each element is assigned to a unique processor and mesh vertices are shared if they are adjacent to elements assigned to different processors. This results in the partition $\Pi = \{\pi_1, \dots, \pi_p\}$ of the mesh *by elements*, where π_i is the set of elements assigned to processor P_i. Communication is then performed across the edges (in two-dimensional problems) or faces (in three-dimensional problems) that separate two elements of the mesh [79].

The most complex procedure in the FEM is the solution of linear systems of equations $Ax = y$ which, on large problems, can contain millions of unknowns. Sparse matrix-vector products, inner products and vector operations are the basic operations of most iterative solvers such as Conjugate Gradient and GMRES. When these operations are executed on a parallel computer, the most communication intensive procedure is the multiplication of a sparse matrix by a vector. In the FEM, the matrix A is the result of an assembly process of the local square matrix $L(\Omega_a)$ obtained from every element Ω_a. The size of $L(\Omega_a)$ depends on the order of the polynomial of the basis functions used in the approximation. The global matrix A is equal to $\sum_{\Omega_a \in D} TL(\Omega_a)$ where T is a projection operator that maps the local indices in the matrix $L(\Omega_a)$ to the global indices in A.

In a parallel environment the matrix A and vectors x and y are also partitioned between processors. If row k of A corresponds to a node that lies on a boundary between two or more processors, P_i and P_j, then the entries in A_i, the partition of matrix A associated with processor P_i, contains the contributions of the elements $\Omega_a \in \pi_i$ and the entries in A_j incorporates the contributions of the elements $\Omega_b \in \pi_j$. The matrix A_i is *partially assembled* since it only considers the contributions of the elements assigned to P_i. The fully assembled matrix is $A = \sum_{1 \leq i \leq p} A_i$. In a mesh partitioned by elements a parallel matrix-vector product requires communication across the common mesh edges and faces of elements assigned to different processors. The matrix-vector product $Ax = y$ is performed in two phases. In the first phase each processor computes $A_i x = y_i$. The resulting vectors y_i are partially assembled. In the second phase we communicate the individual vectors y_i to compute $y = \sum_{1 \leq i \leq p} y_i$ globally. In this second phase it is only necessary to exchange the entries in y_i between nodes located on the internal boundaries between subdomains.

A partition Π of a mesh $M(D, V)$ by elements is obtained from the dual graph $G(W, E)$ of the mesh where W is a set of graph vertices and E is a set of graph vertices. Each element $\Omega_a \in D$ has a corresponding graph vertex $w_a \in W$. Two elements $\Omega_a, \Omega_b \in D$ are *adjacent* if they share a common edge (in 2D) or face (in 3D). For every pair of adjacent elements $\Omega_a, \Omega_b \in D$ there is a corresponding edge (w_a, w_b) in E.

A partition $\Pi(G)$ of a mesh M with dual graph G is $\epsilon-$*balanced* if for every processor P_i, $|\pi_i| \leq (|\Pi|/p)(1+\epsilon)$, where $|\pi_i|$ is the number of elements assigned to P_i and $|\Pi| = |D|$ is the total number of elements in the mesh and ϵ is small. On a mesh partitioned by elements, the communication cost is a function of the size of the boundary of the subdomains and, on machines with a high latency network, on the number of adjacent subdomains. A common measure of the quality of a partition of the mesh is the number of shared nodes which is equal to the number of entries of the vector y that are exchanged during the second phase of a parallel matrix-vector product when using linear approximations. This measure is usually approximated in the dual graph G by the number of edges whose endpoints are in different subsets (the *cut*). If w_a and w_b are vertices in G that correspond to elements Ω_a and Ω_b in the mesh respectively, then

$$C_{cut}(\Pi(G)) = |\{(w_a, w_b) \mid (w_a, w_b) \in E, w_a \in \pi_i, w_b \in \pi_j, i \neq j\}|.$$

Because there is a one to one relation between a partition $\Pi(M)$ of the elements of a mesh M and a partition of the vertices $\Pi(G)$ of its dual graph G, we do not make a distinction between $\Pi(M)$ and $\Pi(G)$.

5.2.1 Review of Graph Partitioning Methods

The problem of partitioning a graph into p subgraphs of approximately equal size while minimizing the number of edges joining vertices in different subgraphs is known as the *p-way graph partitioning problem.*

Definition 2 Given an undirected graph $G(W, E)$ the p-way graph partitioning problem is to find a partition Π of the vertices W into subsets $\pi_1, \pi_2, \ldots, \pi_p$ such that $W = \bigcup \pi_i$, $\pi_i \bigcap \pi_k = \emptyset$ for $i \neq j$, $|\pi_i| \leq (|\Pi|/p)(1+\epsilon)$ and Π minimizes the number of edges in E that are incident on vertices located in different subsets.

This problem is NP-hard even in the simple case of bisecting a graph between two processors [41]. As a result many heuristics have been proposed for it.

One of the most successful heuristics for partitioning unstructured FEM meshes is Recursive Spectral Bisection (RSB) [68]. RSB is based on the computation of the eigenvector $\mathbf{u_2}$ (called the *Fiedler* vector[34, 35]) associated with the second smallest eigenvalue, λ_2, of the Laplacian matrix $L(G) = D - A$, where D is a diagonal matrix of vertex degrees and A is the adjacency matrix of the graph. RSB first partitions the graph into two subgraphs according to the entries of the corresponding vertices in the Fiedler vector. The vertices associated with the largest half of the entries in $\mathbf{u_2}$ are placed in one subset; the remaining vertices are placed in the second subset. These two subgraphs are in turn recursively bisected.

Local heuristics, such as the Kernighan-Lin algorithm (KL) [57], complement spectral methods by further improving the quality of a partition. Roughly speaking, KL is a local search algorithm that swaps vertices between two sets if that reduces the number of edges crossing the partition. The *gain* associated with a pair of vertices is the net reduction in the number of edges crossing the partition when they are swapped. While a swap with positive gain clearly reduces the the partition cut, KL also considers negative gains to escape local minima. When all vertices have been swapped and frozen, it backs up to the last swap that had the net largest gain.

RSB produces high quality partitions but, because of the high cost of computing eigenvectors of a large matrix, it is usually restricted to relatively small graphs. For large graphs, multilevel methods such as Multilevel-KL [48] provide a better tradeoff between quality and speed. Multilevel methods usually consist of three phases:

- *Graph contraction phase.* Starting from a initial graph G_0 (which in our case is the dual graph G) a sequence of increasingly coarser graphs $G_0, G_1, \ldots G_K$ is constructed until the number of vertices in G_K is less than some specified constant. In Multilevel-KL, the coarsening of the mesh is implemented using an edge contraction operation. To construct G_k from G_{k-1} pairs of unmatched adjacent vertices in G_{k-1} are matched and each pair is collapsed into one supervertex in G_k, whose weight is equal to the sum of the weights of the original vertices. The coarser graph G_k also inherits the unmatched vertices in G_{k-1}. G_k also preserves the edges and edge weights. If two matched vertices in G_{k-1} are adjacent to a common vertex, the weight of the corresponding edge in G_k is equal set to the sum of the weights of the original edges in G_{k-1}. In this way, the resulting reduced graph G_k maintains the global structure of G_{k-1}. Multilevel-KL randomly selects the vertices to match, but other multilevel schemes select highly connected vertices [55] or construct coarser graphs from a maximal independent set [10].

- *Coarse graph partitioning phase.* The coarse graph G_K is partitioned using any of a number of different heuristics. Because the number of vertices in G_K is usually much smaller than in G_0, spectral partitioners are commonly used. Multilevel-KL uses RSB to partition G_K.

- *Projection and improvement phase.* The partition found for G_i is projected to G_{i-1} for $1 \leq i \leq K$. In most methods, a partition of G_k naturally corresponds to a partition of G_{k-1}. For example, if the coarser graph is constructed by contracting edges, the matched vertices in G_{k-1} are assigned to the same subset as their super vertex. Other methods, such as ML [10], rather than projecting partitions transfer the Fiedler vector between graphs.

  After the projection local heuristics are used to improve the quality of the partition. Multilevel-KL and [55] use the optimized version of KL based on the work of Fiduccia and Matthesyses [33]. Their linear cost heuristic, rather than swapping vertices between two partitions, performs a sequence of individual vertex moves that minimizes the cost of a partition cut. The traditional KL heuristic is extended to multiway partitions as follows: the $gain(w_a, j)$ associated with any vertex w_a in G_k assigned to π_i is the reduction in the partition cut obtained by moving w_a from π_i to π_j. This heuristic selects the moves that produce the largest gains and, like KL, freezes moved vertices so they are not repeatedly moved between subsets; it also considers movements with negative gains with the expectation that they might result later in an overall improvement in the cut size.

In many physical problems in which the adaptive process is used to adjust the resolution of the mesh as the simulation evolves, the number of elements in the refined mesh M^t at time t is much larger than the number of elements in the initial mesh M^0. Thus, although it is possible to use a serial graph partitioning algorithm to partition and distribute M^0, it is not always feasible to use the same serial partitioning algorithm to rebalance the work of large refined meshes.

Unfortunately, many of the best serial graph partitioning heuristics do not easily accommodate parallel implementation. In [9] Barnard and Simon present a parallel implementation of their spectral algorithm. In their method each processor bisects the subgraphs that result from the recursive step of the bisection (for a small number of processor its speedup is very limited).

The Kernighan-Lin heuristic used in the projection phase of many multilevel schemes is P-complete [77] and does not parallelize well. Some approaches [77, 56] overcome this problem by moving or swapping clusters of vertices rather than individual vertices. Nevertheless, this parallel process is communication intensive, it is difficult to obtain good performance on loosely-coupled machines where it is not efficient on relatively small graphs.

Geometric graph partitioning methods[63, 64] rely on coordinate information, which in the case of finite element meshes, is usually readily available. For example, Recursive Coordinate Bisection (RCB) [85] recursively bisects a graph by first determining the coordinate of its longest dimension, then sorting the vertices according to this coordinate and finally dividing the vertices into two subsets with values below and above the median. Geometric heuristics are scalable but it is shown in [85] that they produce worse partitions than spectral methods.

5.3 The Repartitioning Problem

Traditional parallel FEM systems partition the mesh in a preprocessing step. The mesh is then mapped to processors and the simulation starts. This static approach to mesh partitioning is not sufficient for methods that dynamically modify the mesh as is the case in adaptive schemes. Also, in a parallel environment performance is severely impacted if processors do not execute nearly equal amounts of work. Consequently, it is necessary to dynamically repartition an adapted mesh as the simulation progresses.

The mesh repartitioning problem [94, 15, 29, 78, 12] is not as widely studied as the standard graph partitioning problem. In addition to the traditional goals of balanced partitions and minimum edge cut, the repartitioning of a graph must satisfy a new set of requirements that arise from its dynamic nature. Load balancing is embedded in the adaptive process. Therefore, the graph repartitioning must have a low cost relative to the solution time so it does not outweigh its possible benefits. Second, repartitioning of the mesh must be performed in parallel. It is inefficient to move the complete mesh to one processor to repartition it. Finally, the algorithm should use the current assignment of the mesh, so that it moves the smallest number of mesh elements to restore the workload balance.

Definition 3 Let $G(W, E)$ be an undirected graph and let its vertices W be partitioned into p subsets $\Pi = \{\pi_1, \ldots, \pi_p\}$ such that $W = \bigcup \pi_i$ and $\pi_i \cap \pi_j = \emptyset$ for $i \neq j$. Let $\epsilon > 0$ be small. Then Π is unbalanced if for some i, $1 \leq i \leq p$, $|\pi_i| > (|\Pi|/p)(1 + \epsilon)$. Given an unbalanced partition Π, the p-way graph repartitioning problem is to find a balanced partition $\widehat{\Pi}$ that minimizes the following function:

$$C_{\text{repartition}}(\widehat{\Pi}, \Pi, \alpha) = C_{\text{cut}}(\widehat{\Pi}) + \alpha C_{\text{migrate}}(\Pi, \widehat{\Pi}).$$

Here $C_{\text{cut}}(\widehat{\Pi})$ is the size of the cut associated with $\widehat{\Pi}$, $\alpha > 0$ is a constant that measures of the relative migration and solution times, and $C_{\text{migrate}}(\Pi, \widehat{\Pi})$, defined below, is the number of vertices that must migrate between partitions to restore balance, namely, to insure that $|\widehat{\pi}_i| \leq (|\widehat{\Pi}|/p)(1 + \epsilon)$.

$$C_{\text{migrate}}(\Pi, \widehat{\Pi}) = |\{w_k \mid w_k \in \pi_i, w_k \notin \widehat{\pi}_i\}|$$

The parameter α is used to penalize partitions that would only provide a marginal improvement in $C_{cut}(\widehat{\Pi})$ but require significant movement of data between processors.

The graph partitioning problem is a special case of the graph repartitioning problem in which $\alpha = 0$. Thus, graph repartitioning is also NP-hard. Otherwise we can obtain an optimal partition of any arbitrary graph $G(W, E)$ through a sequence of graph repartitions as follows:

- Start with a graph with W vertices and no edges and assign the vertices to subsets such that each subset π_i has $|\Pi|/p$ vertices.
- Add edges to the graph one by one and repartition the resulting graphs after each addition.

This procedure requires $|E|$ repartitions and computes an optimal partition of G. If each step could be done in polynomial time, G could also be partitioned in polynomial time. Thus, this apparently simpler problem is actually equally difficult.

Because the repartitioning problem is NP-hard, heuristics are needed for it. A natural one to use is the Kernighan-Lin algorithm with a gain function that reflects changes in the cost $C_{\text{repartition}}(\widehat{\Pi}, \Pi, \alpha)$ defined above. This idea is the basis for the heuristic introduced in Section 5.8.

5.4 The Parallel Nested Repartitioning Method (PNR)

PARED uses an alternative procedure for repartitioning adapted meshes that was originally outlined in [16, 17]. This algorithm operates on the graph G described above associated with the initial coarse mesh $M^0(D, V)$ and incorporates the idea that the refined mesh M^t at time t was obtained as a sequence of nested refinements from an initial coarse mesh M^0. Rather than computing directly a partition of M^t, PNR computes a partition of M^0 using an associated weighted dual graph G (which is described below) and then projects the partition to M^t. Thus, in PNR every element is moved at most once in each repartition of the mesh.

PNR has aspects that are common to and different from multilevel partitioning algorithms. Although both partition a coarse graph, the coarse graph is not constructed in PNR; it is given initially.

In order to contract the refined mesh M^t into the small graph G associated with $M^0(D^0, V^0)$ while preserving its global structure we define the function $ElemWeight(\Omega_a)$ on an element $\Omega_a \in D^0$ to be the total number of leaf elements in the refinement tree τ_a associates with the element Ω_a. We also define the function $EdgeWeight(\Omega_a, \Omega_b)$ on two adjacent elements $\Omega_a, \Omega_b \in D^0$ to be the total number of common mesh edges (in two-dimensional meshes) or common mesh faces (in three-dimensional meshes) between the leaf elements of the two refinement trees τ_a and τ_b of Ω_a and Ω_b, respectively.

To start the simulation, PARED loads the initial mesh M^0 into the coordinator processor P_C from which it creates a dual graph $G(W, E)$, as described in Section 5.2. In PARED G is a weighted graph in which initially every edge and vertex has unit weight. For every vertex w_a in the dual graph let $weight(w_a) = ElemWeight(\Omega_a)$ where element $\Omega_a \in D$ corresponds to w_a. Similarly, each edge $(w_a, w_b) \in E$ is assigned a $weight(w_a, w_b) = EdgeWeight(\Omega_a, \Omega_b)$. Dual graph vertex and edge weights represent computational intensity and communication cost respectively. We also define $weight(\pi_i)$ of the subset Π to be the sum of the weights of all the vertices in the subset and $weight(\Pi)$ to be the sum of the weights of all the subsets $\pi_i \in \Pi$.

PNR invokes a serial graph partitioning algorithm to compute an initial partition Π^0 of G. The mesh M^0 is then distributed between the processors according to this assignment. The processor P_C maintains a copy of the dual graph G that is later used to obtain new partitions of adapted meshes M^t.

5.4.1 Repartitioning the Adapted Mesh

In this section we describe the procedure used by PNR to dynamically repartition adapted meshes. Assume that M^{t-1} has been distributed between p processors and that the weights of the edges and vertices in the graph $G(W, E)$ located in the coordinator are consistent with M^{t-1}.

The local adaptation of M^{t-1} into a new mesh M^t using a nested adaptation algorithm creates a set of new leaf elements in a region $\widetilde{R}$ of the mesh defined by a set of elements of the original mesh $M^0(D^0, V^0)$ such that $\widetilde{R} \subseteq D^0$. This region includes not only the areas initially selected for refinement but also the regions that get refined to maintain the conformality of the mesh. The adaptation procedure can also coarsen a region $\widetilde{C} \subseteq D^0$ by replacing some elements of M^{t-1} by their parents. In both cases, the adaptation of the mesh usually results in imbalances of the work allocated to processors. Thus, if Π^{t-1} is the distribution of the mesh M^{t-1} and Π^t is an unbalanced distribution of the refined mesh M^t then it is necessary to obtain a new partition $\widehat{\Pi}^t$ of the current mesh that rebalances the work.

The PNR algorithm allows for a very natural parallel implementation. New values

Let $M^0(D^0, V^0)$ be an initial mesh.
Let M^t be mesh that results from t adaptations of M^0.
Let $\widetilde{R} \subseteq D^0$ and $\widetilde{C} \subseteq D^0$ be the refined and coarsened regions respectively at time t.
In parallel, compute $ElemWeight(\Omega_a)$ and $EdgeWeight(\Omega_a, \Omega_b)$ for $\Omega_a \in \widetilde{R} \cup \widetilde{C}$ and $\Omega_b \in D$.
Each processor sends its new weights to P_C.
P_C updates the graph G and computes a partition $\widehat{\Pi}^t = \{\widehat{\pi}_1^t, \ldots, \widehat{\pi}_p^t\}$.
for each processor P_i **do**
  for each vertex $w_a \in \pi_i^{t-1}, w_a \in \widehat{\pi}_j^t, i \neq j$ **do**
    P_C informs P_i that the element Ω_a and its refinement tree τ_a must move to P_j.
    P_i executes the move.
  end for
end for

Figure 5.1: Outline of the Parallel Nested Repartitioning Algorithm.

for *ElemWeight* and *EdgeWeight* are computed in parallel using only local information. Changes in these weights are sent to the common processor P_C which, after updating G, repartitions the graph into p subsets. At this point, all other processors wait until P_C sends back a message informing them of which elements to migrate. Finally, the migration is performed by moving fully refined subtrees τ_a. It is assumed that if P_i sends Ω_a to another processor P_j it also sends all its descendants. The pseudocode for the PNR algorithm is shown in Figure 5.1.

As explained earlier, the PNR algorithm can be divided into three phases:

- In parallel each processor P_i computes the new $ElemWeight(\Omega_a)$ for $\Omega_a \in \widetilde{R}_i \cup \widetilde{C}_i$, where $\widetilde{R}_i$ and $\widetilde{C}_i$ are the subsets of the refinement region $\widetilde{R}$ and coarsening region $\widetilde{C}$ located in processor P_i. In the same way it computes the *EdgeWeight* between two adjacent elements (Ω_a, Ω_b) where Ω_a or Ω_b are in $\widetilde{R}_i \cup \widetilde{C}_i$. P_i then sends these weights to P_C.

- Once P_C receives a message from each processor P_i it partitions the reduced graph G using a serial partitioning algorithm that allows for weights on vertices and edges. Because G is assumed to be relatively small, we can use algorithms that would be considered too expensive to apply to the refined mesh.

- Finally we resume the parallel phase. P_C sends a message to each processor P_i telling them which elements to migrate.

The first phase requires no communication to compute the new weights. Each processor

P_i can obtain the number of leaf children of an element Ω_a because the full refinement tree τ_a is located in P_i. Even if the elements Ω_a and Ω_b are located in different processors, each processor has a full copy of the shared vertices and can compute $EdgeWeight(\Omega_a, \Omega_b)$. Therefore in this phase, the communication cost depends on the size of the region $\widetilde{R} \cup \widetilde{C}$ because only the changes in the weights are sent to P_C. The communication cost of the last phase is proportional to the difference between the current partition Π^t and the balanced partition $\widehat{\Pi}^t$.

Our method does not require that the complete mesh M^t be in one processor in order to compute a partition. It is sufficient that the coarse initial mesh be small enough to fit into one processor; the refined mesh can be of arbitrary size. In PARED the dual graph G is also a dynamic graph that, although initially created from the mesh M^0, can evolve over time. The local refinement of the mesh can create vertices in G that have very high weight relative to other vertices which might lead to the impossibility of creating balanced partitions. PARED can detect this condition (i.e. $ElemWeight(\Omega_a) > \delta|\Pi|$) and allow the graph G to be expanded by replacing some coarse vertices by subgraphs. These dynamic graphs also allow PARED to handle problems such as the study of fractures in materials that require the modification of the structure of a mesh.

Many of the heuristics designed for graph partitioning can also be used to repartition the updated graph G. Unfortunately, when these heuristics are applied to slightly different problems they can generate very different results. For example, standard graph partitioning algorithms such as Multilevel-KL or RSB usually compute a new distribution of the adapted mesh that is very different from the current one and require a large movement of elements and vertices between processors. In Section 5.6 we discuss the sensitivity of heuristics to the repartition of G.

5.5 Quality of the Partitions Obtained from PNR

Because there are many more ways to partition the adapted mesh M^t than to partition M^0, a good partition of G does not necessarily imply a good partition of M^t. Recall that M^t is a finer graph than M^0. For this reason, there is more freedom to partition M^t than to partition M^0. However, the experiments reported in this section comparing partitions of M^t against those obtained by partitioning G and then projecting these partitions to M^t show that the resulting partitions obtained by both methods are of similar quality.

To test the effectiveness of PNR we compared the partitions it provides with those produced by Chaco's Multilevel-KL on adaptively refined unstructured two- and three-dimensional meshes. Our goal was to compute a solution to Laplace's equation $\Delta u = 0$ defined in $\Omega^2 = (-1, 1)^2$ with the following Dirichlet boundary conditions:

$$g(x, y) = \cos(2\pi(x - y))\frac{\sinh(2\pi(x + y + 2))}{\sinh(8\pi)}.$$

The analytical solution of this problem is known to be $u(x, y) = g(x, y)$ at every point of the domain Ω. This solution is smooth but changes rapidly close the corner (1,1). A similar problem has been defined in three dimensions.

Our initial two-dimensional mesh has 12,498 triangles while the three-dimensional initial mesh has 9,540 tetrahedra of similar size. The adaptation of the mesh according the L_∞ norm (maximum error between real solution and computed solution) creates a large number of elements in the region of rapid change in the solution, as shown in Figure 5.2. The total number of elements and vertices in each refined mesh is shown in Table 5.1. After each refinement, we compute a new partition of the adapted mesh using both Multilevel-KL and PNR with $\alpha = 0.1$. Chaco's Multilevel-KL always partitions the mesh using a dual graph G obtained from the refined mesh M^t. This graph has one vertex for every element in M^t. On the other hand PNR partitions the mesh using only the weighted graph G constructed from M^0.

In this problem, the local refinement of M^0 is not uniform. In regions of high relative error, the corresponding vertices of G have much larger weights in adapted meshes than those associated with unrefined elements. These problems, which result in a high variation of vertex weights in the graph G, are one of the most difficult cases to handle by a scheme like PNR. Recall that the weights of the vertices of G correspond to the number of elements in the fine mesh. After 8 levels of recursive refinement, the graph vertices that are close to the region of high activity have a weight larger than 256 while the vertices on smooth regions have unitary weight. As the weights of the vertices in G close to the corner approach the average subset weight $|\Pi|/p$, it becomes increasingly difficult to obtain balanced partitions with small internal boundaries. In the larger meshes, the resulting balance is within 3% of the average weight of a subset. Beyond this point it is necessary to modify the graph G to include more vertices in this region and allow for more freedom to compute partitions as explained in the previous section to obtain reasonably good and balanced partitions. PARED provides methods for modifying the structure of G, but these methods were not used in these tests.

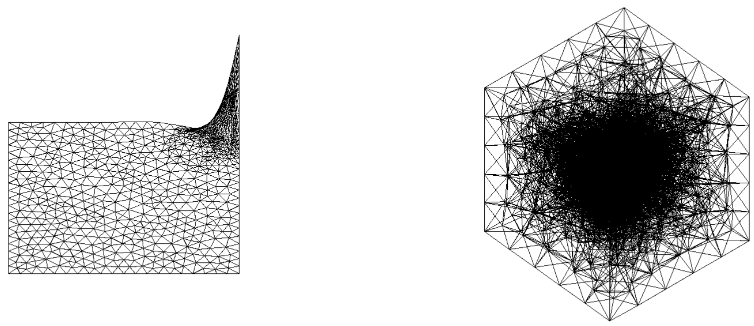


Figure 5.2: Irregular two- and three-dimensional regular meshes adaptively refined to solve Laplace's equation of a problem that exhibits high physical activity in one of its corners.

	2D Mesh		**3D Mesh**	
Level	**Elem**	**Vert**	**Elem**	**Vert**
0	12498	6394	9540	2013
1	14425	7384	10066	2118
2	22715	11532	12148	2525
3	30698	15544	17691	3580
4	43262	21858	30833	6051
5	56055	28279	70185	13262
6	76981	38776		
7	101737	51176		
8	135371	68031		

Table 5.1: Mesh sizes of the locally adapted 2D and 3D meshes at each refinement level.

The experiments reported in Table 5.2 demonstrate that very high quality partitions can be produced using PARED's refinement history trees, thereby greatly parallelizing the mesh partitioning problem. This data shows the number of shared vertices between processors in the partitions produced by Multilevel-KL and PNR for the two- and three-dimensional meshes described above. Although PNR partitions relatively small weighted graphs, it often produces partitions with smaller boundaries between subsets than those generated by Multilevel-KL.

5.5.1 Competitive Analysis of PNR

In this section we show that PNR can provide 2D mesh partitions that are competitive, that is, whose cut sizes are within a small constant factor of the best partitioning algorithm at the expense of a small additive change in the balance of mesh elements between processors.

2D Mesh												
	Multilevel-KL						**PNR**					
L	**4**	**8**	**16**	**32**	**64**	**128**	**4**	**8**	**16**	**32**	**64**	**128**
0	179	333	525	792	1141	1614	157	297	465	739	1043	1523
1	202	335	534	801	1167	1702	197	343	521	773	1164	1633
2	263	445	674	1023	1500	2118	245	437	675	996	1458	2076
3	270	473	775	1194	1748	2456	305	471	745	1120	1609	2316
4	350	571	895	1400	2080	2906	363	571	932	1352	1995	2809
5	388	642	1061	1595	2324	3341	350	624	980	1495	2179	3134
6	448	749	1202	1829	2706	3945	444	733	1175	1775	2620	3699
7	493	830	1357	2111	3112	4503	563	808	1351	2048	2971	4315
8	554	950	1547	2337	3544	5151	539	994	1557	2360	3595	5152

3D Mesh												
	Multilevel-KL						**PNR**					
L	**4**	**8**	**16**	**32**	**64**	**128**	**4**	**8**	**16**	**32**	**64**	**128**
0	334	489	674	935	1174	1437	372	536	737	931	1193	1458
1	321	478	729	975	1230	1495	382	517	682	979	1226	1483
2	366	559	785	1046	1350	1667	364	572	819	1088	1406	1695
3	398	681	979	1349	1717	2120	406	698	975	1302	1716	2038
4	631	1020	1453	1893	2441	3024	618	999	1481	1935	2410	2761
5	1243	1742	2561	3380	4374	5446	1377	1895	2551	3374	4306	5225

Table 5.2: Comparison of the quality of the partitions produced by Multilevel-KL and PNR. The tables show the number of shared vertices obtained by partitioning a sequence of locally adapted meshes with Multilevel-KL and PNR into 4 to 128 subsets.

The 3D case is unresolved.

PNR is a procedure that partitions the leaves of the mesh $M^t(D,V)$ at time t among p processors by partitioning the vertices of a weighted graph G and then replacing each vertex w_a of G by its corresponding refinement history tree τ_a whose leaves are leaves of $M^t(D,V)$. Vertex w_a has a weight equal to the number of leaves of τ_a; the weight of an edge between vertices w_a and w_b of G is equal to the number of edges between leaf elements in τ_a and τ_b.

A p-processor partition has an ϵ*-balance*, $\epsilon > 0$, if the number of mesh elements assigned to each processor is less than or equal to $(|G|/p)\,(1+\epsilon)$ where $|G|$ is the weight of the vertices of G. A balance is good if ϵ is small because the maximum number of mesh elements assigned to any processor cannot be less than $(|G|/p)$.

Let G^t denote the dual graph of the most refined mesh defined by the leaves of $M^t(D,V)$. It has one vertex for every mesh element and one edge for every pair of mesh elements that share an edge or face. Each vertex and edge have unit-weight.

Our goal is to show that from a good partition Π^t of G^t, that is, one that has an ϵ-balance for small ϵ and a small cut size C, we can construct a nearly equally good partition Π^0 of G, that is, one that when projected to a partition of Π^0 of $M^t(D,V)$ has an ϵ'-balance and cut size C' such that ϵ' and C' are at most constant multiples of ϵ and C, respectively. From this we conclude that PNR, which projects a partition of G to one of $M^t(D,V)$, is capable of producing partitions of $M^t(D,V)$ that are nearly as good as those obtained without requiring that cuts follow the boundaries of coarse mesh elements.

We derive this result by converting a partition Π^t to a partition Π^0. We move the boundaries between processors so that they pass along the periphery of the initial coarse mesh elements, edges in the case of triangles and faces in the case of tetrahedra. Because the boundaries can be moved in ways that affect the balance and cut size, it is important to move the boundaries carefully.

Note that a cut defined by Π^t can pass through a coarse mesh element Ω_k once or more than once. If the latter case holds, we order the segments of the cuts passing through each element so that the first cut divides Ω_k into two pieces and subsequent segments divide one of the two pieces. This reduces the problem to the first case, namely, when the cut divides Ω_k into two pieces.

We now consider the first case, when a cut passes through a coarse mesh element Ω_k once. Let Ω_k be the coarse mesh element that overlaps processors p_i and p_j. The cut

Let $B_{i,j}$ be the set of coarse mesh elements split between p_i and p_j.
Let α_i be the number of fine mesh elements assigned to p_i by Π^t.
Let β_i be the current number of fine mesh elements in p_i.
Let γ be the largest number of fine mesh elements in any coarse mesh element.
for $i = 1$ **to** $p - 1$ **do**
  for $j = i + 1$ **to** p **do**
    for Ω_k the next coarse element in $B_{i,j}$ **do**
      Unless $\beta_i - \alpha_i > \gamma$ or $\beta_j - \alpha_j > \gamma$, move
      the boundary splitting Ω_k to Ω_k's shorter periphery.
      Otherwise, move the boundary to Ω_k's longer periphery.
    end for
  end for
end for

Figure 5.3: Algorithm to convert a partition Π^t of G^t with balance B to a partition Π^0 of G with balance $B + \gamma$.

divides Ω_k into two sets. When the boundary is moved to the periphery of Ω_k, both sets are assigned either to p_i or p_j. Either move may change the size of the cut and/or the balance.

The code shown in Algorithm 5.3 describes a greedy algorithm that we use to create the partition Π^0 from the partition Π^t. It moves the boundary between p_i and p_j passing through coarse mesh element Ω_k to the shorter periphery of Ω_k unless this increases or decreases the number of fine mesh elements assigned to one of the processors to differ by more than γ from the number assigned by Π^t. Here γ is the maximum number of fine mesh elements in the most refined coarse mesh element. We shall show that under reasonable assumptions this condition insures that the cut size does not increase by more than a constant factor. We now discuss these assumptions.

Assumptions

When a mesh is refined, coarse elements are generally not all refined uniformly. A mesh element is *refined uniformly to depth d* if it satisfies the following conditions: a) if $d = 0$, the mesh element is unrefined, and b) otherwise, the two elements produced by a refinement are themselves recursively refined uniformly to depth $d - 1$.

Note that it is not a given that coarse elements can be uniformly refined to an arbitrary depth. This follows because when elements are refined, we require that the refined elements be conformal. Thus, it may be necessary to refine an element more than once to make it conform to a neighboring element that has been refined previously.

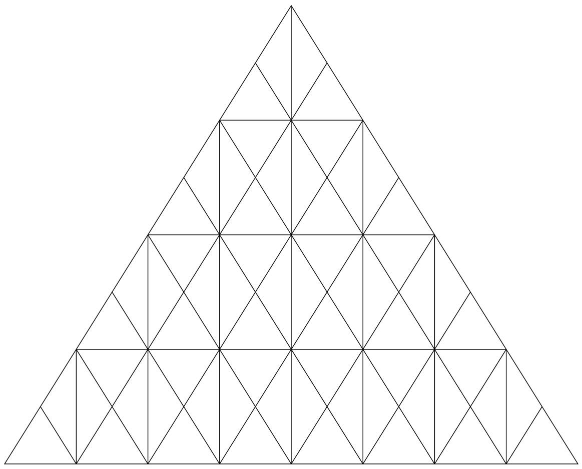


Figure 5.4: An 6-level uniform refinement of a triangle.

Our experience with 2D triangular meshes is that they are frequently refined uniformly. This is not as common with 3D tetrahedral meshes.

Since γ is the maximum number of fine mesh elements in the most refined coarse mesh element, if such elements are refined uniformly to depth d, $\gamma = 2^d$.

Analysis of the Greedy Algorithm

We now introduce several definitions that will be useful in our analysis of the cut size and balance of the partition Π^0 produced from Π^t by Algorithm 5.3.

Definition 4 Let Ω_k be a depth-d uniformly refined coarse mesh element and let c be length of a cut through Ω_k. The *cut expansion function* of Ω_k, $cut\_exp(d,c)$, is the maximum value for the ratio of the length of the shorter periphery of Ω_k defined by a cut of Ω_k to c, the size of the cut. The *weight function* of Ω_k, $weight(d,c)$, is the maximum number of fine mesh elements that can be contained between a cut through Ω_k of size c and the shorter periphery of Ω_k.

Lemma 1 In 2D let d be even. Then, $cut\_exp(d,c) = 3$ and, if $c \leq 2^{d/2-1}$, then $weight(d,c) = 2c^2$.

Proof Consider the 2D case in which each mesh element is a triangle Ω_k. Ω_k is refined into two triangles by the first level of refinement. The edge of Ω_k that is split by its first refinement is called its *base*. The two other edges are called *slopes*. Call the vertex at which the two slopes meet the *apex*. (See Figure 5.4 where the base is the bottom edge of the triangle).

Consider a cut that enters Ω_k by one boundary edge and exits by another. If the cut has size $c \leq 2^{d/2-1}$, the places where a such cut enters and exits are constrained. A cut either enters one slope and exits from the other or it enters one slope and exits from the base without passing through the triangle defined by the vertical through the apex and the other slope. The latter condition holds because the length of such a path would necessarily exceed $2^{d/2-1}$ in length.

Suppose the cut enters one slope and exits the other. Let it enter at the ath vertex from the apex ($a = 0$ at the apex) on one slope and exit from the bth vertex on the other. The shorter periphery defined by the cut has length $a+b$. The shortest cut connecting the entry and exit vertices is the maximum of a and b. Thus, the ratio of the shorter periphery, $a+b$, to the cut c (which is at least $\max(a,b)$) is at most 2. When the cut enters one slope and exits from the base, the cut that maximizes this ratio is a "vertical" cut , a vertical edge in Figure 5.4. This cut defines a periphery in which the base and slope portions have sizes c and $2c$, respectively. Thus, $cut\_exp(d,c) = 3$.

We now obtain the weight function. Given a cut of size c we maximize the number of triangles contained between the cut and the boundary of Ω_k defined by the shorter periphery. Obviously this cut is a shortest cut between two boundary edges of Ω_k. Since a cut of size at most $2^{d/2-1}$ is assumed, either a) the cut enters from one slope and exits from the other, or b) it enters from one slope and exits from the base. By inspection it is easy to see the larger number of elements is contained in the region defined by a vertical and that this region contains at most $2c^2$ elements. Thus, $weight(d,c) = 2c^2$. $\Box$

We now analyze the performance of our greedy Algorithm.

Theorem 2 Let the partition Π^t of $M^t(D,V)$ have have an ϵ-balance and cut size C. Under the assumption that each coarse mesh element is refined uniformly to depth d, Algorithm 5.3 produces from Π^t a partition Π^0 of $M^t(D,V)$ with cut size at most $9C$ that respects the boundaries of elements in $M^0(D,V)$ and for which each processor has at most $(|G|/p)(1+\epsilon)+(p-1)d^2$ mesh elements.

Proof Let's follow the steps taken by Algorithm 5.3. (Note that $\gamma = 2^d$.) Let q be such that the cut through the first q coarse mesh elements are moved to their shorter peripheries but on the $(q+1)$st element the cut is moved to the longer periphery. It follows that the number of elements that change sides as a result of moving the first q elements exceeds γ.

Let the lengths of those portions of the original cut defined by Π^t that pass through the first $q+1$ mesh elements processed by Algorithm 5.3 be c_i, $1 \le i \le q+1$. Let the lengths of the peripheries to which these cuts are moved be b_i, $1 \le i \le q+1$. Because the first q cuts are moved to the shorter peripheries, it follows from Lemma 1 that $\sum_{i=1}^{q} b_i \le 3\sum_{i=1}^{q} c_i$. The $(q+1)$st cut is moved to the longer periphery and is replaced by a cut of length $b_{q+1} \le 3\left(2^{d/2}\right)$, the length of the boundary of the most highly refined coarse mesh element.

We now bound the expansion in the cut length that occurs during the first $q+1$ steps of Algorithm 5.3. We consider two cases, namely, a) when one or more of the first q cuts is long, namely, when $c_{i_0} > 2^{d/2-1}$ for some $1 \le i_0 \le q$, or b) when each of the first q cuts is small, namely, when $c_i \le 2^{d/2-1}$ for all $1 \le i \le q$.

Since $b_{q+1} \le 3\left(2^{d/2}\right)$, in the first case $2^{d/2-1} < c_{i_0}$ from which it follows that $b_{q+1} < 6c_{i_0}$. In this case we have the following bound on the expansion in the cut length:

$$\sum_{i=1}^{q+1} b_i \le \left(\sum_{i=1}^{q} b_i\right) + b_q \le 3\sum_{i=1}^{q} c_i + 6c_{i_0} \le 9\sum_{i=1}^{q+1} c_i$$

In the second case, namely, when $c_i \le 2^{d/2-1}$ for all $1 \le i \le q$, we use Lemma 1 to bound the number of elements that change processors. Since at most $2(c_i)^2$ elements change processors when moving the cut through the ith element, it follows that at most $2\sum_{i=1}^{q}(c_i)^2$ elements change processors during the first q steps. Since this number must be at least $\gamma = 2^d$, we have the following relationship.

$$2^d \le 2\sum_{i=1}^{q}(c_i)^2 \le 2\left(\sum_{i=1}^{q} c_i\right)^2$$

It follows that

$$b_{q+1} \le 3\left(2^{d/2}\right) \le 3\sqrt{2}\sum_{i=1}^{q} c_i$$

and

$$\sum_{i=1}^{q+1} b_i \le 3(1+\sqrt{2})\sum_{i=1}^{q+1} c_i$$

Thus, the cut size expands by at worst a factor of 9 on the first $q+1$ boundary movements. Repeating the same argument on the remaining elements, we have the desired bound on the cut size.

Since the number of elements assigned to any processor changes by at most γ elements when moving elements between any two processors, it changes by at most $(p-1)\gamma$ when moving elements between all p processors. $\square$

We see that in 2D if $(p-1)d^2$ is at most $(|G|/p)\epsilon$, the partition Π^0 derived from the ϵ-balanced partition Π^t with cut size C is a 2ϵ-balanced partition with a cut size at most $9C$.

5.6 The High Migration Cost of RSB and Multilevel-KL

RSB and Multilevel-KL are very effective methods for partitioning unstructured meshes. Nevertheless, when applied to the repartitioning of adapted meshes, their resulting partitions require a large movement of data between processors which is usually proportional to the size of the mesh, as we now demonstrate. Tables 5.3 and 5.4 show the results of repartitioning a series of unstructured two- and three-dimensional meshes using the RSB algorithm, respectively. Starting from unstructured meshes with elements of similar size we perform successive global refinements (each coarse mesh elements is refined the same number of times) to obtain a family of meshes of increasing size. These meshes contain between 5094 and 103585 triangles in the 2D example and between 3179 and 101051 tetrahedra in the 3D case.

Each mesh M^{t-1} is then partitioned between 4 to 64 processors using RSB bisection. The quality of the resulting balanced partitions Π^{t-1} measured by number of shared vertices is shown in the third column of the tables. We then locally adapted each mesh M^{t-1} in one of the corners using the Laplace problem introduced in Section 5.5 which generates in a new refined mesh M^t. This adaptation procedure creates very few elements (less than 380) relative to the size of the mesh. Because of its localized nature all the new elements are created in only one or two processors. Let Π^{t-1} be the balanced partition of the mesh M^{t-1} before its local adaptation and let Π^t be the unbalanced distribution of the mesh after its refinement.

We then compute a new partition $\widehat{\Pi}^t$ using the same RSB algorithm used to partition M^{t-1}. Even when the meshes M^{t-1} and M^t are very similar and only differ in one of their corners the migration cost between the current unbalanced distribution Π^t and the target partition $\widehat{\Pi}^t$, $C_{migrate}(\Pi^t, \widehat{\Pi}^t)$, can be almost equal to the number of elements of the mesh, as shown in the sixth column of Tables 5.3 and 5.4. Although spectral methods are invariant with respect to node numbering or coordinate transformations, almost all the elements of the mesh need to be moved to a new processor. The results obtained from Multilevel-KL are similar.

We have also explored a method [12] for reducing the migration cost of these algorithms

Proc	M^{t-1} (before ref) Elem	$C_{cut}(\Pi^{t-1})$	M^t (after ref) Elem	$C_{cut}(\widehat{\Pi}^t)$	$C_{migrate}(\Pi^t, \widehat{\Pi}^t)$	$C_{migrate}(\Pi^t, \widetilde{\Pi}^t)$
4	5094	99	5269	95	2627	2627
8		168		159	3341	831
16		273		274	4458	1551
32		421		421	5046	2270
64		615		629	5129	2354
4	11110	137	11411	152	9192	2010
8		249		250	9696	3383
16		405		410	10444	4747
32		633		647	11061	5684
64		926		960	11230	5284
4	23749	311	23902	291	16477	14519
8		488		480	19182	13117
16		700		670	22620	11104
32		1000		980	23441	11374
64		1463		1425	23530	11711
4	49915	331	50072	410	35601	23152
8		569		680	49190	18507
16		920		977	49264	22147
32		1408		1431	49776	21972
64		2067		2159	50050	23639
4	103585	788	103786	863	38433	38433
8		1121		1193	77099	43272
16		1690		1728	93892	51125
32		2380		2403	99397	50264
64		3297		3310	102277	50278

Table 5.3: Migration cost resulting from repartitioning a series of two-dimensional unstructured meshes of increasing size using the RSB algorithm. M^{t-1} is the mesh before refinement and distributed according a balanced partition Π^{t-1} obtained from RSB. M^t is the refined mesh. $\widehat{\Pi}^t$ is a new balanced partition of M^t also produced by the RSB algorithm and $\widetilde{\Pi}^t$ is a permutation of $\widehat{\Pi}^t$ that minimizes data movement.

Proc	M^{t-1} (before ref) Elem	$C_{cut}(\Pi^{t-1})$	M^t (after ref) Elem	$C_{cut}(\widehat{\Pi}^t)$	$C_{migrate}(\Pi^t, \widehat{\Pi}^t)$	$C_{migrate}(\Pi^t, \widetilde{\Pi}^t)$
4	3179	157	3427	156	3129	1324
8		221		223	3134	1809
16		301		313	3263	1595
32		387		401	3316	1688
64		470		488	3327	1783
4	10166	312	10543	346	9390	6242
8		456		468	10089	4878
16		633		663	10387	5199
32		838		869	10500	5821
64		1075		1114	10530	5418
4	31935	651	32314	672	26365	16114
8		946		1006	29867	15230
16		1352		1385	31619	18055
32		1828		1838	32145	17357
64		2373		2381	32196	17725
4	101051	1282	101381	1426	38148	38148
8		1847		2074	69142	33930
16		2860		2987	77465	46124
32		4020		4121	87328	41955
64		5330		5402	94344	49232

Table 5.4: Migration cost resulting from repartitioning a series of three-dimensional unstructured meshes of increasing size using the RSB algorithm. M^{t-1} is the mesh before refinement and distributed according a balanced partition Π^{t-1} obtained from RSB. M^t is the refined mesh. $\widehat{\Pi}^t$ is a new balanced partition of M^t also produced by the RSB algorithm and $\widetilde{\Pi}^t$ is a permutation of $\widehat{\Pi}^t$ that minimizes data movement.

by permuting the partition $\widehat{\Pi}^t$ generated by RSB to minimize $C_{migrate}(\Pi^t, \widetilde{\Pi}^t)$. The task of finding a permutation $\widetilde{\Pi}^t$ of $\widehat{\Pi}^t$ is equivalent to computing the maximum bipartite graph matching problem, where the weight of an edge (i, j) of the graph corresponds to the weights of the elements located in processor i and assigned to subset $\widehat{\pi}_j^t \in \widehat{\Pi}^t$.

In these examples we use a simple incremental algorithm to approximate the optimal permutation. We first generate a $p \times p$ matrix B where row i corresponds to processor P_i and column j corresponds to the subset $\widehat{\pi}_j^t \in \widehat{\Pi}^t$. The entry $B_{i,j}$ is equal the sum of the weights of the elements currently in processor P_i that are assigned to $\widehat{\pi}_j^t$. This is the number of elements that have to be migrated from processor P_i to processor P_j according to the newly computed partition $\widehat{\Pi}^t$. We then search for the maximum value in this matrix. Assume this is the $B_{i,j}$ entry. This value corresponds to the maximum movement of data between a processor P_i to another processor P_j. We then assign the subset $\widehat{\pi}_j^t$ to P_i, remove row i and column j from B and repeat this processor until all the subsets in $\widehat{\Pi}^t$ have been assigned to processors.

The value of $C_{migrate}(\Pi^t, \widetilde{\Pi}^t)$ is shown in the last column of Tables 5.3 and 5.4. The total migration cost for $\widetilde{\Pi}^t$ is approximately half of the migration cost for $\widehat{\Pi}^t$ but it is still proportional to the size of the mesh.

5.7 Bounding the Migration Cost

In the previous section we have shown that standard graph partitioning algorithms, when applied to the repartitioning problem, often require significant movement of data between processors which creates serious contention for an interprocessor network. In this section we derive a lower bound for the migration cost under certain reasonable assumptions. In the next section we present a new repartitioning heuristic. The migration cost for this heuristic is close to the lower bound derived in this section.

Assume that Π^{t-1} is a balanced distribution of a mesh between p processors and that m new elements are created in the refinement phase in only one processor, say P_o, resulting in an unbalanced partition Π^t. Also assume that the goal is to minimize the cost of migration subject to the requirement that the distribution of elements between processors is balanced. That is, we do not impose the additional requirement that the number of nodes that are shared by elements located in different processors is minimized.

Under the above conditions P_o only needs to send $(p-1)(m/p)$ of its elements to other processors and every processor should have m/p more elements after the migration is complete. This could be done by moving $p-1$ sets of m/p elements from P_o to every other processor with the migration cost shown below.

$$C_{migrate}(\Pi^t, \widehat{\Pi}^t) = (p-1)\frac{m}{p} = m(1-\frac{1}{p})$$

Unfortunately, this reallocation of elements generates disconnected sets of elements in every processor. A more realistic bound can be obtained by restricting the migration of elements to be between adjacent processors. Let H^t be the processor connectivity graph at time t using the current distribution Π^t. H^t has one vertex for every processor and an undirected edge between two processors that have adjacent elements. To obtain a balanced distribution $\widehat{\Pi}^t$ processor P_o must send m/p elements to every other processor P_j, but only along the edges of H^t. This procedure effectively shifts the boundaries of the mesh and reduces the probability of creating disconnected subsets of elements in each processor.

Let the minimum distance between processors P_i and P_j in H^t be $d_{i,j}$. The movement of m/p elements from P_o to P_j has a migration cost of $d_{o,j}(m/p)$ if only edges of H^t are used giving the following total migration cost:

$$C_{migrate}(\Pi^t, \widehat{\Pi}^t) = \sum_{j \neq o} d_{o,j}\left(\frac{m}{p}\right).$$

For example, if processors in the graph H^t form a two-dimensional $\sqrt{p} \times \sqrt{p}$ mesh and P_o is located in one of its corners, $d_{o,j} \leq 2(\sqrt{p}-1)$ and the total migration cost required to rebalance the mesh after creating m new elements in P_i is

$$C_{migrate}(\Pi^t, \widehat{\Pi}^t) \leq 2(\sqrt{p}-1)(p-1)\frac{m}{p} \leq 2\sqrt{p}\,m.$$

Under these assumptions, the total migration cost $C_{migrate}(\Pi^t, \widehat{\Pi}^t)$ only depends on the number of processors p and the number of new elements m and is independent of the mesh size. For example, when $p = 64$ the bound is $16m$. The heuristic described in the following section gives a better migration cost than this bound primarily because the minimum distances $d_{0,j}$ are smaller than predicted by a square grid.

5.8 Minimizing the Migration Cost

In this section we introduce a new heuristic that we have developed to repartition the weighted graph G located in the coordinator that greatly reduces the number of mesh elements that need to move in order to repartition a good unbalanced partition.

The critical step in PNR is the selection of the graph partitioning algorithm to partition and repartition the dual weighted graph G in the coordinator. PARED can use a variety of algorithms to partition G, including Chaco's implementation of RSB and Multilevel-KL. Although traditional graph partitions can rebalance the work and generate partitions with small cuts, they do not consider migration cost. As we have shown in Section 5.6, RSB and Multilevel-KL when applied to G usually require the movement of half of the elements of the adapted mesh, even on examples where there is relatively little adaptation. We have also observed that these methods do not perform well when there is great variation of vertex weights and usually produce partitions with disconnected sets of elements assigned to the same processor.

At the beginning of a simulation the coordinator P_C partitions the initial mesh M^0 using our implementation of the Multilevel-KL [47] algorithm. As described in Section 5.2.1 this algorithm creates a sequence of smaller graphs $G_0, \ldots, G_K$ by collapsing adjacent vertices, where G_0 is the dual graph G of the initial mesh M^0. The coarsest graph G_K is then partitioned using Chaco's RSB [48]. This partition is then projected to the finer graphs. As in Multilevel-KL, our implementation uses a variation of the KL heuristic between projections to improve the partition quality. This heuristic selects one by one the movements of vertices between subsets that result in the largest reduction in the partition cut. Moved vertices are not examined again in the same iteration and negative movements are also considered but, rather than evaluating every possible vertex like traditional KL heuristics, it terminates if no improvements have been observed for some time (around 100 movements). Finally, we restore the best partition obtained in this process by reversing the subsequent vertex movements.

Assuming that there is a small change between meshes M^{t-1} and M^t, (for example, in a time dependent problem that slowly moves a highly refined region) the current unbalanced distribution Π^t can be used as the starting point for a local search heuristics to compute a new partition $\widehat{\Pi}^t$. The standard Kernighan-Lin heuristic is designed to work between balanced partitions. When it swaps two vertices assigned to different subsets, the relative work assigned to each processor is maintained. To insure that processors have similar amounts of work, the standard Multilevel-KL only considers movements from subsets that have at least the average number of vertices to subsets that have less than the average. Therefore, both heuristics require that the initial distribution be balanced. None of these heuristics would move or swap vertices that increases the cut size but improves the balance between processors.

As explained in Section 5.2.1, to repartition the graph G with PNR, P_C first receives the new weights from every processor P_i and uses these weights to update G_0 and the coarser graphs $G_1, \ldots, G_K$. This is a recursive procedure. If the adaptation of the mesh modifies the weight of a vertex w_a or an edge (w_b, w_c) in G_0, then we propagate these changes to the corresponding parents of w_a, w_b and w_c in G_1. These changes propagate to the coarser meshes. This procedure does not modify the connectivity of the sequence of graphs.

Our new heuristics that prevents small changes in a mesh from producing a large migration cost consists of Multilevel-KL in which we do not use RSB to repartition the coarsest graph G_K. The resulting algorithm is outlined in Figure 5.5. The selection of vertices and evaluation of partitions is not only based on the cut size. Instead, we use the cost function $C_{repartition}$ defined below between Π^t, an unbalanced distribution of the mesh, and $\widehat{\Pi}^t$, the goal partition where $\alpha, \beta > 0$ are constants.

$$C_{repartition}(\Pi^t, \widehat{\Pi}^t, \alpha, \beta) = C_{cut}(\widehat{\Pi}^t) + \alpha C_{migrate}(\Pi^t, \widehat{\Pi}^t) + \beta C_{balance}(\widehat{\Pi}^t). \qquad (5.1)$$

Here

$$C_{balance}(\widehat{\Pi}^t) = \sum_{i}^{p} \left(weight(\widehat{\pi}_i^t) - \frac{weight(\widehat{\Pi}^t)}{p} \right)^2$$

where $\widehat{\pi}^t$ is the ith subset in the partition $\widehat{\Pi}^t$. The first term in $C_{repartition}$, namely C_{cut}, selects for partitions with small boundaries between subdomains. The second term minimizes data movement. The last term balances the subsets. Here α and β are parameters that represent the relative importance of these three values. We have obtained the best results when β is allowed to vary between the graphs, allowing slightly unbalanced partitions in G_K while enforcing balanced partitions in G_0. In this way, the task of rebalancing the subsets does not rely only on the coarser graphs.

We now describe our local search procedure. Let Π_k^t be a partition of the vertices of G_k at time t, where G_k a graph in the sequence $G_0, \ldots, G_K$. The $gain(w_a, i, j, \alpha, \beta)$ associated with moving vertex w_a in subset π_i to subset π_j, where $\pi_i, \pi_j \in \Pi_k^t$, is the net reduction in the function $C_{repartition}$.

$$gain(w_a, i, j, \alpha, \beta) = (\sum_{w_b \in \pi_j} weight(w_a, w_b) - \sum_{w_c \in \pi_i} weight(w_a, w_c))$$
$$+ \alpha(mweight(w_a, j) - mweight(w_a, i))$$
$$+ 2\beta(weight(w_a)(weight(\pi_i) - weight(\pi_j)) - weight^2(w_a))$$

The *migration weight* $mweight(w_a, j)$ is equal to the number of elements Ω_d in the fine mesh M^t of the tree rooted at w_a that are located in processor P_j at time t and $weight(\pi_i)$ is the weight of the vertices in subset π_i. Before starting the local improvement at each level, we compute the *cut gain* $\sum_{w_b \in \pi_j} weight(w_a, w_b) - \sum_{w_c \in \pi_i} weight(w_a, w_c)$ for every vertex w_a with respect to an adjacent subset j, the *migration gain* $(mweight(w_a, j) - mweight(w_a, i))$ and the *balance gain* $(weight(w_a)(weight(\pi_i) - weight(\pi_j)) - weight^2(w_a))$. Since we only allow movements of vertices that are adjacent to other subsets, only these vertices need to be considered. A movement modifies the cut gain of adjacent vertices but the migration gain is constant. On the other hand, the balance gain depends on pairs of subsets.

As in Multilevel-KL, we maintain a square table with an entry for each pair of subsets consisting of priority queues based on gains. That is, the possible movements between a pair of subsets is sorted by potential gain. To implement the local heuristic, we select the vertex movement with largest gain from this table, move the vertex between subsets and update the entries in the table corresponding to adjacent vertices. The moved vertex is marked so it is not inserted in the table again. This process iterates through the heads of the P^2 priority queues. In FEM meshes the number of adjacent vertices is relatively small. In dual graphs of 2D triangular meshes and 3D tetrahedral meshes each graph vertex has at most three or four neighbors respectively. The time to remove the maximum gain from a priority queue and updating the neighbors is bounded by $O(\log n)$, where n is the number of boundary elements in a subdomain π_i. A vertex move between π_i and π_j modifies the difference $weight(\pi_i) - weight(\pi_j)$. Rebuilding these priority queues requires $O(n)$ steps.

We performed the tests described in Section 5.6 using our new PNR algorithm to partition and repartition the coarse mesh. These results are shown in Tables 5.5 and 5.6 for the same two- and three-dimensional meshes respectively. In these tests we used $\alpha = 0.1$ and $\beta = 0.8$, obtaining partitions with $\epsilon < 0.01$. The quality of the partitions measured by the number of shared vertices generated by PNR and RSB (shown in Tables 5.3 and 5.4) is very similar. On the other hand, the total migration cost $C_{migrate}(\Pi^t, \widehat{\Pi}^t)$ is much smaller than those measured previously and does not significantly increase with mesh size. These results are close to the estimates derived in Section 5.7. On the other hand, if the number

for every processor P_i **do**
  P_C receives the new weights $weight(w_a)$ and $weight(w_a, w_b)$ and updates the graph G_0
end for
for $k = 1$ to K **do**
  P_C updates the weights of G_k from G_{k-1}
end for
for $k = K$ down to 1 **do**
  P_C improves the current partition
  P_C projects the partition from G_k to G_{k-1}
end for

Figure 5.5: Outline of PNR's procedure to repartition the dual graph G in the coordinator P_C.

of new elements created in the refinement phase is large and proportional to the size of the mesh, any of the previous heuristics is likely to provide similar results.

5.9 A Transient Problem

In the previous section we have shown that on locally adapted meshes PNR produces partitions with a cut size similar to the ones produces by standard graph partitioning methods but requires a much smaller data movement. In this section we use our method to follow a disturbance across a domain. We show that the repeated use of our heuristic maintains the quality of the partitions while retaining its small migration cost. In Section 5.6 we have shown that RSB and Multilevel-KL can require migrating about half of the mesh elements between partitions. While this does not imply that every repartition of the mesh with these methods results in such high migration costs, it very frequently does, as our experiments reported below show.

To study these issues we solve Poisson's equation $\Delta u = f$ over the domain $\Omega^2 = (-1, 1)^2$ where the solution $u(x, y, t)$ is the known function

$$u(x, y, t) = \frac{1}{1 + 100(x + t)^2 + 100(y + t)^2}$$

and compare the partitions produced by RSB and PNR. We solved this problem for 100 time steps in which t varies from -0.5 to 0.5. This function is smooth with a peak of 1 at the coordinates $x = y = -t$ and close to zero almost everywhere else. Thus, as t varies from -0.5 to 0.5, the peak moves along a diagonal from $(0.5, 0.5)$ to $(-0.5, -0.5)$. The computed solution $\hat{u}$ at the initial and final iterations are shown in Figure 5.6(a) and (b), respectively.

Proc	M^{t-1} (before ref)		M^{t} (after ref)		
	Elem	$C_{cut}(\Pi^{t-1})$	**Elem**	$C_{cut}(\widehat{\Pi}^{t})$	$C_{migrate}(\Pi^{t}, \widehat{\Pi}^{t})$
4	5094	89	5269	91	132
8		154		162	280
16		261		290	430
32		394		442	483
64		591		642	681
4	11110	151	11411	151	226
8		260		262	489
16		400		415	773
32		601		659	967
64		866		935	1146
4	23749	197	23902	199	115
8		347		352	245
16		564		578	332
32		883		932	415
64		1302		1351	512
4	49915	291	50072	289	156
8		547		549	251
16		885		899	373
32		1346		1368	531
64		1995		2038	581
4	103585	426	103786	429	151
8		802		789	321
16		1314		1319	469
32		1970		1971	623
64		2982		3042	731

Table 5.5: Migration cost resulting from repartitioning a series of two-dimensional unstructured meshes of increasing size using the PNR algorithm. M^{t-1} is the mesh before refinement and distributed according a balanced partition Π^{t-1} obtained from PNR. M^t is the refined mesh. $\widehat{\Pi}^t$ is a new balanced partition of M^t also produced by the PNR algorithm and $\widetilde{\Pi}^t$ is a permutation of $\widehat{\Pi}^t$.

Proc	M^{t-1} (before ref) Elem	$C_{cut}(\Pi^{t-1})$	M^t (after ref) Elem	$C_{cut}(\widehat{\Pi}^t)$	$C_{migrate}(\Pi^t, \widehat{\Pi}^t)$
4	3179	132	3427	135	178
8		184		195	303
16		278		303	394
32		362		404	492
64		448		486	538
4	10166	313	10543	329	298
8		409		428	437
16		628		639	662
32		806		851	856
64		1042		1105	877
4	31935	574	32314	568	311
8		867		869	456
16		1332		1317	852
32		1760		1802	940
64		2344		2360	1060
4	101051	1249	101381	1244	282
8		1797		1819	406
16		3042		3044	672
32		4315		4279	904
64		5424		5391	1271

Table 5.6: Migration cost resulting from repartitioning a series of three-dimensional unstructured meshes of increasing size using the PNR algorithm. M^{t-1} is the mesh before refinement and distributed according a balanced partition Π^{t-1} obtained from PNR. M^t is the refined mesh. $\widehat{\Pi}^t$ is a new balanced partition of M^t also produced by the PNR algorithm and $\widetilde{\Pi}^t$ is a permutation of $\widehat{\Pi}^t$.

Our initial mesh is the same 2D unstructured mesh used in Section 5.5 with 12,498 triangles of nearly similar size. When the L_2 and L_∞ norms are used, the number of mesh elements needed to maintain a fixed error varies with t. Since this risks confusing the amount of data that must migrate and the comparative advantage of our repartitioning technique over RSB and Multilevel-KL, we refined the mesh to maintain about a constant number (45,000 to 50,000) of mesh elements in the adapted meshes. As t increases: a) we compute a new solution, $\hat{u}$, using the old mesh; b) to each mesh element is assigned a value equal to the largest computed solution at its vertices; c) an element is marked for one two or three refinements depending on whether the new value is in the range 0 to 0.02, 0.02. to 0.04, or 0.04 to 0.06; d) if by the previous value the element was marked for a different number of refinements, the element is either coarsened or refined by the appropriate number of steps. The initial and final adapted meshes are shown in Figure 5.6(c) and (d), where it is possible to identify the three concentric circles given by the increasing refinement value $\hat{u}$. In each of the 100 time steps, after the solution and adaptation of the mesh we also compute a new partition $\widehat{\Pi}^t$ using RSB and PNR.

Figure 5.7 shows the number of shared vertices of each partition of $\widehat{\Pi}^t$ obtained by RSB and PNR for 4, 8, 16 and 32 processors. In PNR we used the parameters $\alpha = 0.1$ and $\beta = 0.8$ in Equation 5.1. The resulting partitions have less than 0.01 imbalance between subsets. Even though PNR is a local heuristic, in these examples the cut size of the partitions that it produces does not deteriorate over time and is similar to the ones produced by RSB, a very successful graph partitioning method.

Figure 5.8 shows the number of elements migrated by the three methods, a) RSB, b) RSB after computing a permutation $\widetilde{\Pi}^t$ of $\widehat{\Pi}^t$ that reduces migration, as explained in Section 5.6, and c) PNR. RSB usually migrates between 50% and 100% of the total number of elements between repartitions. Although not reported here, the results for Multilevel-KL are similar. The total movement significantly decreases with the permutation $\widetilde{\Pi}^t$ of the subsets to processors but we still observe peaks of more than 46% of the total elements and an average movement of 21% for 32 processors. This method is characterized by sharp peaks, where some repartitions resulted in small migrations while others require a significant movement of data.

The total migration cost resulting from PNR is small compared with the other methods (on the average it is between 1.2% of the elements for 4 processors and 5.5% for 32 processors) and is smooth. PNR resulted in only two peaks with more than 10% data movement between iterations. The total data movement produced by PNR over all iterations

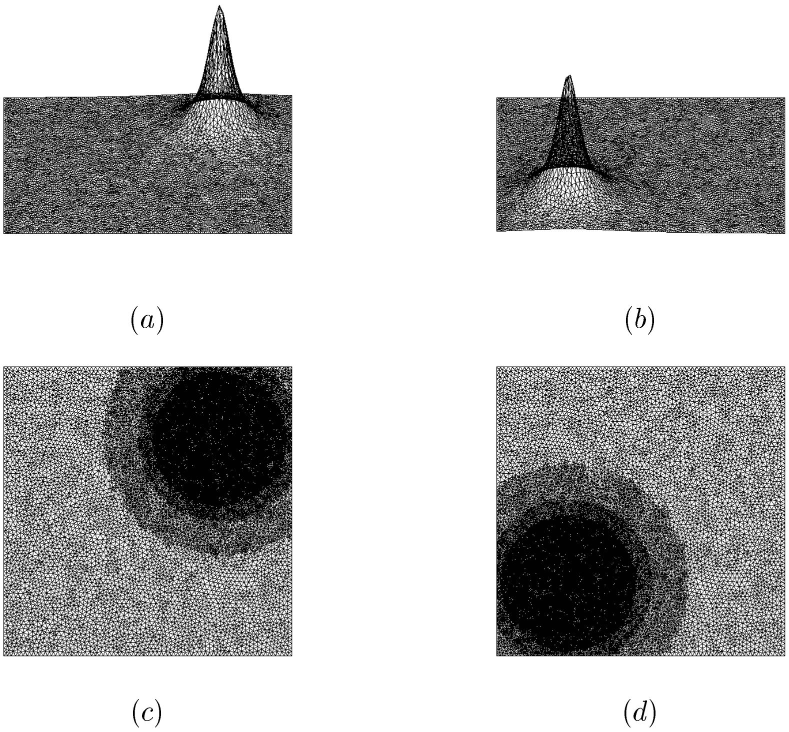


Figure 5.6: (a) and (b) show the computed solution u with $t = -0.5$ and $t = 0.5$. (c) and (d) illustrate the adapted mesh at these two different time steps.

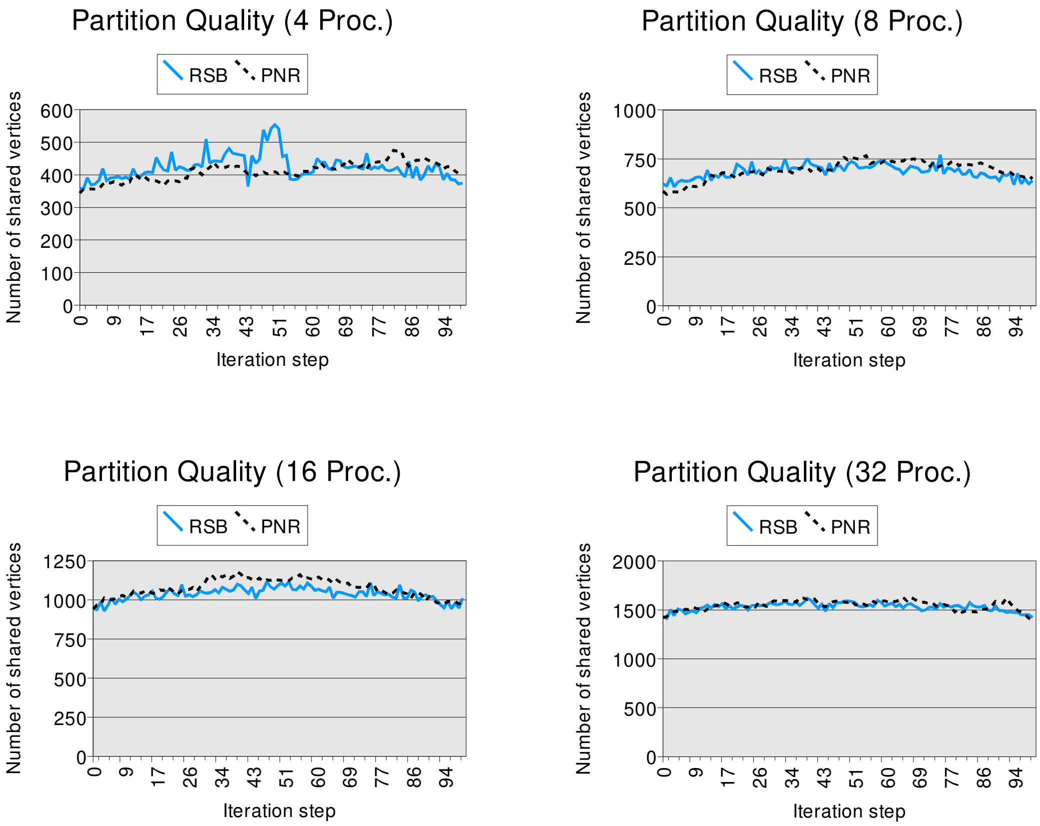


Figure 5.7: Quality of the partitions measured by the number of shared vertices produced by RSB and PNR for 4, 8, 16 and 32 processors for each of the 100 time steps between $t = -0.5$ to $t = 0.5$.

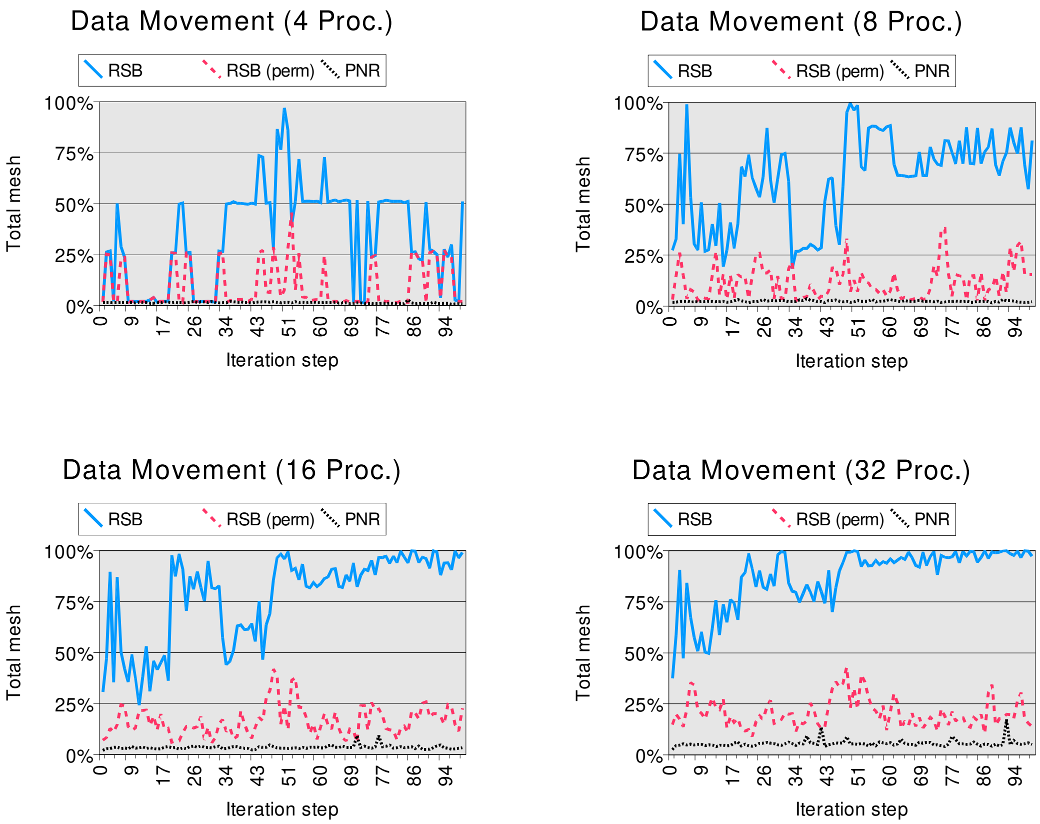


Figure 5.8: Elements moved between time steps of partitions produced by RSB, permuted RSB and PNR for 4, 8, 16 and 32 processors.

was between 12% and 27% of the total number of elements moved by the permuted RSB heuristic.

Chapter 6

Mesh Migration

Although we demonstrated in the previous chapter how to compute a new partition $\widehat{\Pi}^t$ that balances the work, at this stage of the computation the mesh is still distributed according to an unbalanced distribution Π^t of the mesh. In this chapter we present a procedure to migrate elements and nodes between the processors of a distributed memory machine. We use the same notation introduced in the previous chapters: Π^t is the current (probably unbalanced) distribution of the mesh between processors, $\pi_i^t \in \Pi^t$ is the set of elements assigned to processor P_i according to the partition Π^t, V_p^i is the copy of vertex V_p located in P_i, (P_j, V_p^j) is a reference to the copy of vertex V_p located in processor P_j, $Ref(V_p^i)$ is the set references $\{(P_j, V_p^j)\}$ such that V_p^i has a reference to its copy V_p^j in processor P_j, $Adj(\Omega_a) = \{V_{p_1}, \dots, V_{p_n}\}$ is the set of vertices contained in element Ω_a and $ElemAdj(V_p^i) = \{\Omega_1, \dots, \Omega_m\}$ is the set of elements adjacent to V_p in P_i.

6.1 Introduction

Let $\widehat{\Pi}^t = \{\widehat{\pi}_1^t, \dots, \widehat{\pi}_p^t\}$ be a goal distribution of the mesh. If $\widehat{\Pi}^t \neq \Pi^t$, then there is at least one element Ω_a such that $\Omega_a \in \pi_i^t \in \Pi^t$ and $\Omega_a \in \widehat{\pi}_j^t \in \widehat{\Pi}^t$ where $i \neq j$. Remember that we assume that the mesh is partitioned by elements so that the π_i and π_j partitions are disjoint, $\Omega_a \notin \pi_j^t$ and $\Omega_a \notin \widehat{\pi}_i^t$. To adjust the mesh to the $\widehat{\Pi}^t$ partition we need to move Ω_a from P_i to P_j.

Assume that we need to move an element Ω_a from P_i to P_j, that is $\Omega_a \in \pi_i^t$ and $\Omega_a \in \widehat{\pi}_j^t$ and that $Adj(\Omega_a) = \{V_{p_1}, \dots, V_{p_n}\}$ is the set of vertices of Ω_a. We first send a message from P_i to P_j. If V_p is a vertex of Ω_a that is shared between P_i and P_j we only include a reference in the message. Otherwise, P_j creates the new copy V_p^j and it initializes $Ref(V_p^j) =$

$Ref(V_p^i) \cup (P_i, V_p^i)$, so that the new copy has references to the original copy of V_p and all its other copies in other processors. Using V_p^j processor P_j creates the element Ω_a.

We then update the references to the new vertex copies. If V_p^j is a new copy of vertex created as a result of a message from P_i, then P_j sends a message to each processor P_k. P_i finally deletes its element Ω_a. It can happen that Ω_a was the only element that pointed to V_p in P_i. In this case we wish to remove the unnecessary copy of V_p^i located in P_i. Processor P_i sends a message to each processor P_j such $(P_j, V_p^j) \in Ref(V_p^i)$. Once all the other processors remove their references to V_p^i we can delete the vertex. This guarantees that other processors do not maintain references to nonexistent vertices.

Our algorithm considers several cases that assume that either P_j has a local copy of the vertices or they are included in the message to P_j:

- For each vertex $V_p \in Adj(\Omega_a)$, if $(P_j, V_p^j) \notin Ref(V_p^i)$ (so V_p is not a shared node between P_i and P_j at time t) then we need to send V_p from P_i to P_j along with element Ω_a. P_j uses this information to create its copy of vertex V_p^j which it then uses to initialize the element Ω_a in P_j.

- Otherwise, if $(P_j, V_p^j) \in Ref(V_p^i)$ (so V_p is a shared node between P_i and P_j at time t and P_j has a local copy V_p^j) then we should not create the V_p^j vertex copy again. When P_i sends the element Ω_a to P_j, it also includes the reference (P_j, V_p^j) instead of the complete vertex V_p. Then P_j can use V_p^j to create Ω_a. This condition has an important implication: processor P_j cannot delete its copy of V_p^j until it has received all its new elements, even if processor P_j has already sent the only element Ω_b that points to V_p^j to another processor P_k because some other processor P_i might expect P_j to have a copy of V_p.

- If processor P_i sends two or more elements Ω_a and Ω_b to P_j and there is a common vertex $V_p \in Adj(\Omega_a) \cap Adj(\Omega_b)$ (so V_p is a vertex of both Ω_a and Ω_b) then only one copy V_p^j should be created in P_j and both elements Ω_a and Ω_b should refer to it in the destination processor.

- If two processors P_i and P_k send the elements Ω_a and Ω_b to the same processor P_j where elements Ω_a and Ω_b are adjacent elements and there is a shared vertex $V_p \in Adj(\Omega_a) \cap Adj(\Omega_b)$, i.e. $(P_k, V_p^k) \in Ref(V_p^i)$ and $(P_i, V_p^i) \in Ref(V_p^k)$, then P_j should detect that V_p^i and V_p^k are actually two copies of the same vertex. In this case P_j should create only one copy V_p^j for both elements Ω_a and Ω_b.

- Finally, if Ω_a and Ω_b are adjacent elements that share a common vertex V_p and processor P_i sends Ω_a to another processor P_j and P_k sends Ω_b to P_l then we should insure that $(P_l, V_p^l) \in \mathit{Ref}(V_p^k)$ and $(P_k, V_p^k) \in \mathit{Ref}(V_p^l)$ (so the two copies V_p^k and V_p^l refer to each other).

The migration algorithm uses three different kinds of messages:

- A *move* message $\mathit{MoveMsg}(i,j)$ sent from a source processor P_i to a destination processor P_j is used to migrate one or more elements Ω_a and their associated vertices from P_i to P_j. We assume that this message has two other fields. The first field $\mathit{MoveMsg}(i,j).\mathit{vertices}$ is a set such that $\mathit{MoveMsg}(i,j).\mathit{vertices}(\Omega_a)$ contains the vertices of element Ω_a. For each vertex $V_p \in \mathit{Adj}(\Omega_a)$, if P_j has a local copy of V_p^j then only the reference (P_j, V_p^j) is included in the message. Otherwise we send the complete vertex V_p, including its coordinates, boundary location, etc. The second field, $\mathit{MoveMsg}(i,j).\mathit{ref}$ is also a set that has one entry for every vertex V_p not located in P_j. $\mathit{MoveMsg}(i,j).\mathit{ref}(V_p)$, contains the references of V_p to its copies in other processors.
- An *add reference* message $\mathit{AddRefMsg}(i,j) = \{(V_p^i, V_p^j)\}$ sent from a source processor P_i to a destination processor P_j is used to add a reference to vertex V_p^j in processor P_j. In this case the source processor must have a reference to the remote copy. When P_j receives this message, it sets $\mathit{Ref}(V_p^j) = \mathit{Ref}(V_p^j) \cup (P_i, V_p^i)$ on processor P_j.
- A *delete reference* message $\mathit{DelRefMsg}(i,j) = \{(V_p^i, V_p^j)\}$ sent from a source processor P_i to a destination processor P_j is used to remove a reference from the vertex V_p^j in P_j so $\mathit{Ref}(V_p^j) = \mathit{Ref}(V_p^j) - (P_i, V_p^i)$. Again, the source processor must have a reference to the remote vertex copy.

6.2 Overview of the Migration Algorithm

Our migration algorithm can be divided into five phases that are executed by each processor in sequence:

- *Element and vertex transfer*: involves sending the selected elements and vertices from P_i to P_j such that $\pi_i^t \cap \widehat{\pi}_j^t \neq \emptyset$.
- *Remote reference update*: updates the references to the new vertex copies.

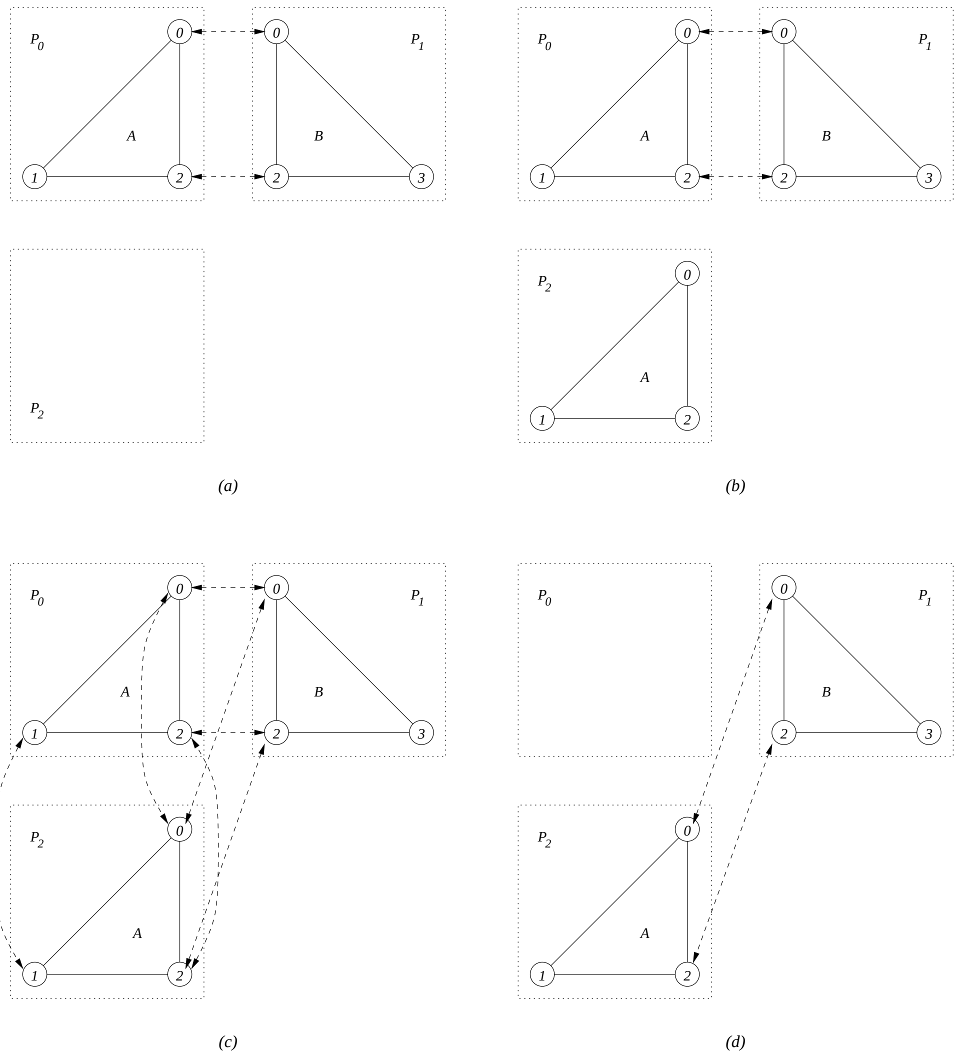


Figure 6.1: A simple migration example. (a) shows the initial mesh. The goal is to move Ω_a from P_0 to P_2. (b) We first copy Ω_a to P_2 and (c) update the references. (d) We finally delete the element Ω_a in P_0.

- *Element deletion*: removes the element $\Omega_a \in \pi_i^t$, $\Omega_a \notin \widehat{\pi}_i^t$ in each processor P_i. Some vertices are no longer used and are also removed.
- *Removal of remote references*: before deleting a shared vertex we insure that there are no remote references to it.
- *Vertex deletion*: removes the unnecessary vertices from the source processors.

A simple illustration of this algorithm is shown in Figure 6.1. In this example we show a mesh with two elements partitioned between three processors P_0, P_1 and P_2. Our goal is to move the element Ω_a form P_0 to P_2. We initially send a $MoveMsg(0,2) = \{\Omega_a\}$ from processor P_0 to processor P_2 that includes V_0, V_1 and V_2. We initialize $Ref(V_0^2) = \{(P_1, V_0^1), (P_0, V_0^0)\}$. We also initialize $Ref(V_1^2)$ and $Ref(V_2^2)$. P_2 then sends an $AddRefMsg$ to P_0 and P_1 to update the references to the new copies of V_0 and V_2 located in P_2. P_0 then deletes its element Ω_a. Since it can also remove V_0^0, V_1^0 and V_2^0, P_0 also sends a $DelRefMsg$ to the two other processors.

There is one more condition that we need to enforce which is illustrated in Figure 6.2. If two different processors P_i and P_j send a shared vertex V_p to two other processors P_k and P_l, $k \neq l$, we must guarantee that the two new copies of V_p also refer to each other. Figure 6.2 (c) is not an acceptable mesh because we lost some connectivity information. This information is not lost in the final mesh shown in Figure 6.2 (d)

6.3 The Mesh Migration Procedure used in PARED

We will explain the algorithm in more detail using the example in Figure 6.3. There we show a mesh composed of 8 elements $(\Omega_a, \ldots, \Omega_h)$ and 9 nodes $(V_0, \ldots, V_8)$ partitioned between 4 processors (P_0, P_1, P_3, P_3). In the top half Figure 6.3 we show the initial partition Π^t and in its bottom half we show the target partition $\widehat{\Pi}^t$. Our goal is to move the elements from the initial distribution of the mesh Π^t to the destination partition $\widehat{\Pi}^t$. This can be done by executing the commands:

- P_0: move Ω_a to P_3.
- P_1: move Ω_d to P_2.
- P_2: move Ω_e to P_3 and Ω_f to P_0.
- P_3: move Ω_g to P_1 and Ω_h to P_2.

The example used in this section shows a complicated migration sequence where there is no intention of improving the quality of the partition. In fact, the target distribution has a larger number of shared nodes and references than the original one. The initial representation of the mesh showing the references between multiple copies of the same vertex is illustrated in Figure 6.4.

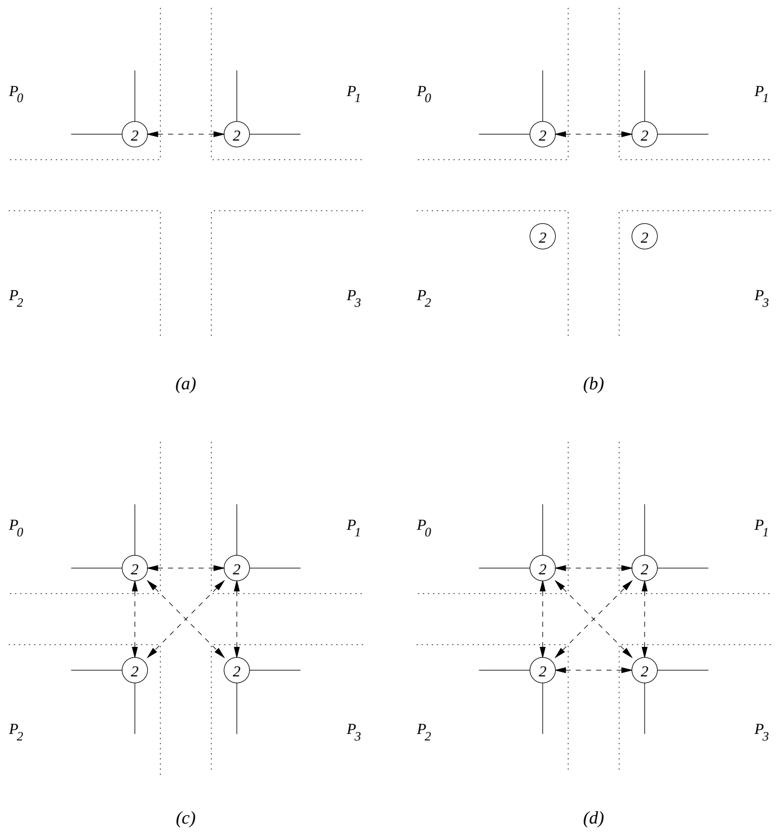
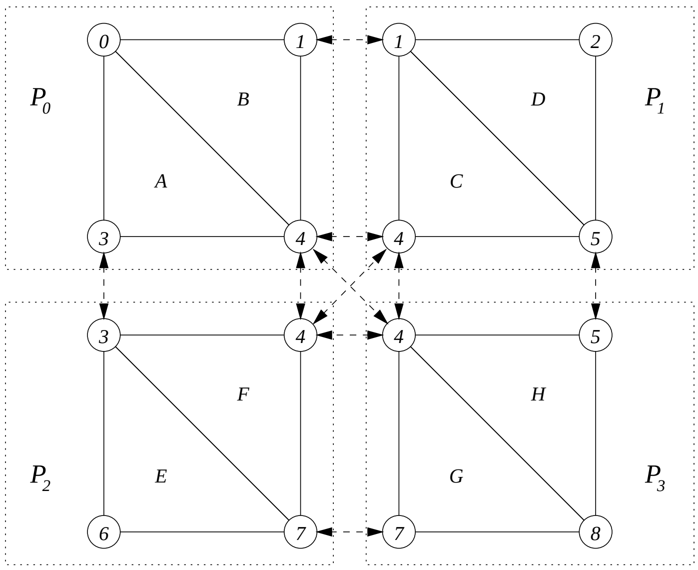
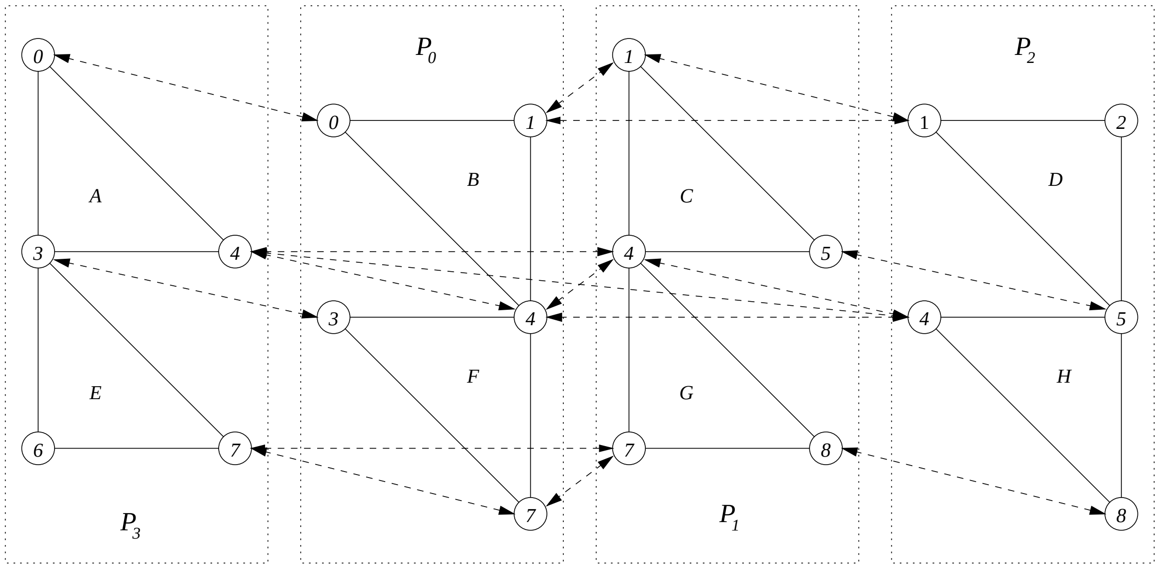


Figure 6.2: The mesh shown in (a) has shared vertex V_2 between P_0 and P_1. (b) P_0 sends the vertex to P_2 while P_1 sends the element to P_3. (c) shows an incorrect mesh because there are no references between the copies of V_2 located in P_2 and P_3. (d) shows the correct final mesh.



(a)



(b)

Figure 6.3: Migration of elements from an initial partition Π^t (a) to a target partition Π^{t+1} (b) showing the multiple copies and remote references.

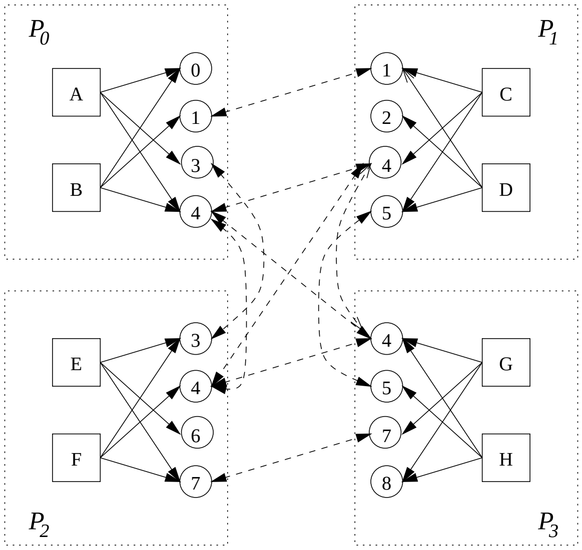


Figure 6.4: Initial distribution of the mesh between four processors for example in Figure 6.3 (a).

6.3.1 Element and Vertex Transfer Phase

In the first phase of the migration procedure we send the elements and vertices to the destination processors. If there is an element Ω_a located in P_i and $\Omega_a \in \widehat{\Pi}_j^t$ then we send a $MoveMsg(i,j) = \{\Omega_a\}$ message from P_i to P_j. If an element Ω_a refers to a vertex V_p^i of which P_j has no local copy then P_i must also include the vertex in the message. Determining if P_j has a local copy of V_p is easy: we only need to look at the references to remote copies of V_p^i (is $(P_j, V_p^j) \in Ref(V_p^i)$?) in the sending processor P_i. If we find that P_j has a local copy V_p^j then we use that copy to create the element Ω_a in P_j. When we send a vertex in the message we also include all the references to other copies. This way the receiving processor can create its local copy and then send a message to the other processors to update their references to it. Also when we are sending multiple elements to a processor we need to be careful to include only one copy of the vertices. The description of the first part of this phase is shown in Figure 6.5. Note that this phase, as well as the remaining ones in this chapter, is independent of the mesh type and can be used for two- and three-dimensional meshes. The initial messages sent by each processor for the previous example are:

- P_0: move Ω_a to P_3 by sending the message $MoveMsg(0,3) = \{\Omega_a\}$. Include in the message the nodes V_0 and V_3 and a reference to V_4^3. In P_3 use these two nodes and the existing copy of V_4 to create the element Ω_a.

for each element Ω_a such that $\Omega_a \in \pi_i^t, \Omega_a \in \widehat{\pi}_j^t, i \neq j$ **do**
  insert Ω_a into $MoveMsg(i,j)$
  for each node $V_p \in Adj(\Omega_a)$ **do**
    if $(P_j, V_p^j) \in Ref(V_p^i)$ **then**
      insert (P_j, V_p^j) into $MoveMsg(i,j).vertices(\Omega_a)$
    else
      insert V_p into $MoveMsg(i,j).vertices(\Omega_a)$
      insert (P_i, V_p^i) into $MoveMsg(i,j).ref(V_p)$
      for each reference $(P_k, V_p^k) \in Ref(V_p^i)$ **do**
        insert (P_k, V_p^k) into $MoveMsg(i,j).ref(V_p)$
      end for
    end if
  end for
end for
for each processor P_j **do**
  if $i \neq j$ and $MoveMsg(i,j) \neq \emptyset$ **then**
    send $MoveMsg(i,j)$
  end if
end for

Figure 6.5: Element and vertex transfer phase: procedure executed by processor P_i to send a selected set of elements and vertices to a destination processor P_j.

- P_1: send Ω_d to P_2 by sending the message $MoveMsg(1,2) = \{\Omega_d\}$ with the nodes V_1, V_2 and V_5.
- P_2: send Ω_e to P_3 (with V_3 and V_6 and a reference to V_7^3) and Ω_f to P_0 (with V_7 and reference to V_3^0 and V_4^0) using the $MoveMsg(0,3) = \{\Omega_e\}$ and $MoveMsg(2,0) = \{\Omega_f\}$ respectively.
- P_3: send Ω_g to P_1 (with V_7 and V_8 and references to V_1^4 and V_7^2) and send Ω_h to P_2 (with V_5 and V_8 with references to V_4^2 and V_5^1) using the $MoveMsg(1,1) = \{\Omega_g\}$ and $MoveMsg(3,2) = \{\Omega_h\}$ respectively.

Once a processor P_j receives a message $MoveMsg(i,j)$ it first creates the new vertices as specified in the message and then constructs the elements. If V_p is a vertex of an element Ω_a such that $\Omega_a \in MoveMsg(i,j)$ and $V_p \in MoveMsg(i,j).vertices(\Omega_a)$ such that V_p is not located in P_j then a new copy V_p^j is created in P_j and $Ref(V_p^j)$ is initialized to $MoveMsg(i,j).ref(V_p)$ (remember that this also includes a reference to the sending processor copy of V_p in the source processor P_i). At this point the new copy contains a reference to the original vertex but not vice versa. It is responsibility of P_j to inform the other processors

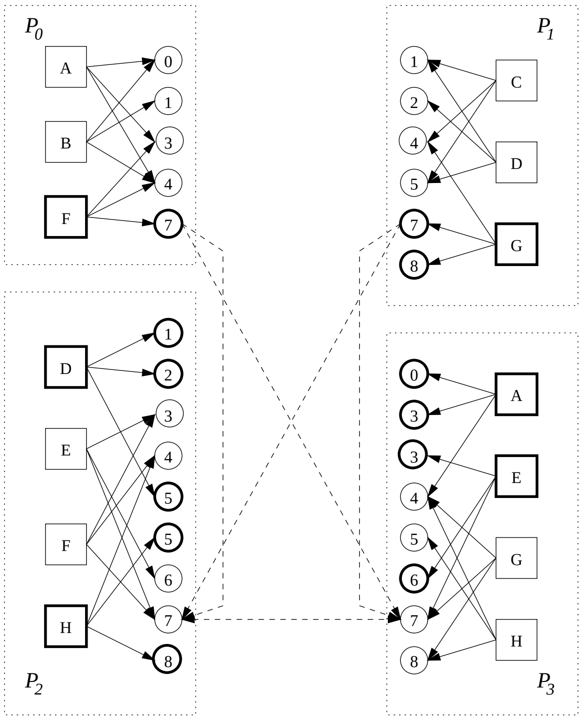


Figure 6.6: Element and vertex transfer phase: state of the mesh after creating the new copies in the destination processors. We highlight the new elements vertices created in this phase in each processor. We only show the references between the multiple copies of vertex V_7 to reduce the complexity of this figure.

of the newly created copy.

Continuing with the example, when P_2 sends Ω_f to P_0 it should also include a copy of V_7. Before P_0 constructs the element Ω_f it should first create the vertex V_7^0. Using that vertex and the local copies V_3^0 and V_4^0 processor P_o can then create the element Ω_f. When P_1 receives the element Ω_g it also creates a copy of V_7 but the copies in P_0 and P_1 of that vertex know nothing about each other at this moment.

There is another problem: P_2 receives Ω_d from P_1 and Ω_h from P_3 and both messages include the vertex V_5 (a similar problem happens in P_3 with V_3). We will explain later how to handle these situations. Figure 6.6 shows the mesh at this stage that highlights the new elements and vertices in each processor. For simplicity, we only include the references between the multiple copies of V_7. Figure 6.7 presents an outline of this receiving phase.

for each message $MoveMsg(i, j)$ sent from other processor P_i to P_j **do**
  receive $MoveMsg(i, j)$
  for each element $\Omega_a \in MoveMsg(i, j)$ **do**
    for each node $V_p \in MoveMsg(i, j).vertices(\Omega_a)$ **do**
      if V_p does not exist in P_j **then**
        create the vertex V_p and initialize $Ref(V_p^j) = MoveMsg(i, j).ref(V_p)$
      end if
    end for
    create the element Ω_a
  end for
end for

Figure 6.7: Element and vertex transfer phase: procedure executed by each processor P_j to receive a transfer message from processor P_i and to create the elements and vertices indicated in the message.

for each new node V_p^i **do**
  for each reference $(P_j, V_p^j) \in Ref(V_p^i)$ **do**
    insert (V_p^i, V_p^j) into $AddRefMsg(i, j)$
  end for
end for
for each processor P_j **do**
  if $i \neq j$ and $AddRefMsg(i, j) \neq \emptyset$ **then**
    send $AddRefMsg(i, j)$
  end if
end for
for each message $AddRefMsg(k, i)$ sent from other processor P_k to P_i **do**
  receive $AddRefMsg(k, i)$
  for each reference $(V_p^k, V_p^i) \in AddRefMsg(k, i)$ **do**
    insert (P_k, V_p^k) into $Ref(V_p^i)$
  end for
end for

Figure 6.8: Remote reference update phase: outline of the algorithm executed by each processor P_i to update the references of new shared vertices.

6.3.2 Remote Reference Update Phase

In the next phase we update the references to the new nodes. Assume that V_p^j is a node created in P_j in the previous phase as the result of a $MoveMsg(i, j)$. P_j needs to inform P_i and all the other processors P_k that have a copy of V_p about the location of the new copy V_p^j in memory so they can create a reference to it. Using $Ref(V_p^j)$, P_j sends a $AddRefMsg(j, k)$ for each reference $(P_k, V_p^k) \in Ref(V_p^j)$. This procedure is outlined in Figure 6.8.

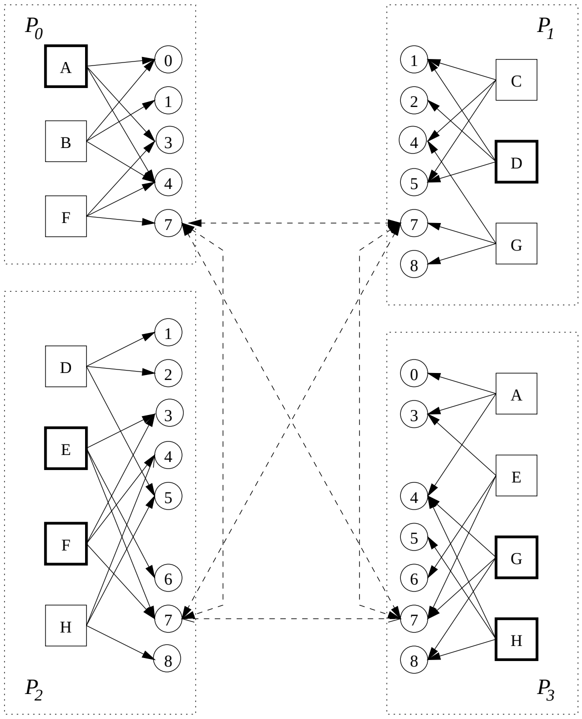


Figure 6.9: Remote reference update phase: state of the mesh at the end of this phase. The elements that will be deleted in each processor in the next phase are highlighted

In the previous example P_0 sends a message to P_2 and P_3 to update their references to V_7^0. P_1 also sends a message to P_2 and P_3 so they can update the reference of V_0^1. P_2 knows that there is a copy of V_0 in processor P_0 because it included that copy in the $MoveMsg(2,0)$ of the previous phase When P_2 receives a message from P_1, P_2 detects that there is more than one new copy of V_7 so it informs P_0 to update the reference from V_7^0 to V_7^1. The same thing happens in P_3.

At this stage we also detect that there are two copies of V_5 in P_2 and two copies of V_3 in P_3 because they try to update references to vertices located in the same processor. One of the copies is destroyed and the corresponding elements are updated accordingly. The state of the mesh at the end of this phase is shown in Figure 6.9.

6.3.3 Element Deletion, Removal of Remote Vertices, and Vertex Deletion

We now remove the elements $\Omega_a \in \pi_i^t \cap \widehat{\pi}_j^t, i \neq j$ from the source processors P_i. The state of the mesh after this step is shown in Figure 6.10. Moreover, if there is some vertex V_p such

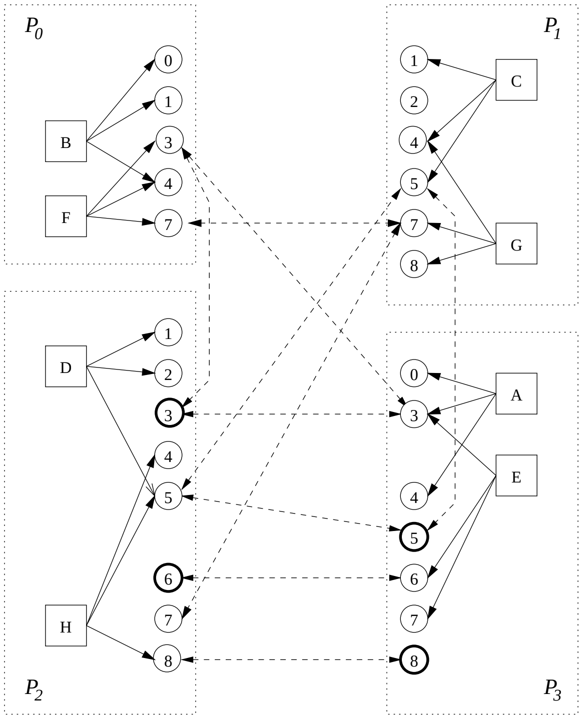


Figure 6.10: Element Deletion: state of the mesh after deleting the elements from the source processors but before removing the unnecessary vertices. We only include the references for vertices that are going to be removed in the next phase

that $ElemAdj(V_p^i) = \emptyset$ we delete the vertex to free memory. For every vertex copy V_p^i such that $ElemAdj(V_p^i) = \emptyset$ processor P_i sends a *DeleteMsg* message to every processor P_j such that $(P_j, V_p^i) \in Ref(V_p^i)$. When P_j receives this message, it removes the reference (P_i, V_p^i) from $Ref(V_p^j)$. P_i can now delete its copy V_p^i. The deletion is now safe because V_p^i is not contained by any element in P_i and no other processors have references to it. Figure 6.11 shows the procedure for this three last phases.

In our example the destruction of Ω_e and Ω_f in processor P_2 causes the deletion of V_3^2, V_6^2 and V_7^2. Note that the node V_4^2 is referenced by the new element Ω_h so it is not deleted.

Finally we destroy the nodes when the number of references to them is zero. The representation of the mesh at the end of the migration is shown in Figure 6.12.

for each element Ω_a in P_i that has been moved to P_j, $i \neq j$ **do**
  delete Ω_a
  for each vertex $V_p^i \in Adj(\Omega_a)$ **do**
    remove Ω_a from $ElemAdj(V_p^i)$
    if $ElemAdj(V_p^i) = \varnothing$ **then**
      for each reference $(P_j, V_p^j) \in Ref(V_p^i)$ **do**
        insert (V_p^i, V_p^j) into $DelRefMsg(i,j)$
      end for
    end if
  end for
end for
for each processor P_j **do**
  if $DelRefMsg(i,j) \neq \varnothing$ **then**
    send $DelRefMsg(i,j)$
  end if
end for
for each message *DelRefMsg(k, i)* sent from other processor P_k to P_i **do**
  receive $DelRefMsg(k,i)$
  for each reference (V_p^k, V_p^i) in the message $DelRefMsg(k,i)$ **do**
    remove (P_k, V_p^k) from $Ref(V_p^i)$
  end for
end for
delete the vertices V_p from P_i such that $ElemAdj(V_p^i) = \emptyset$

Figure 6.11: Procedure executed by each processor P_i in the three phases of the migration algorithm.

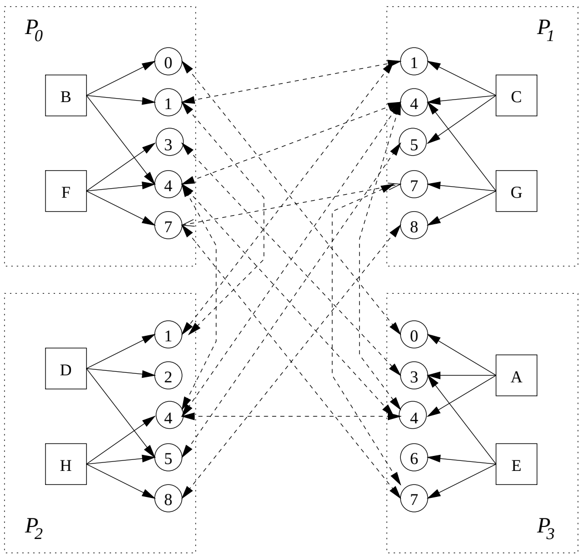


Figure 6.12: A migration example: internal representation of the mesh at the end of the migration phase.

Chapter 7

The Engineering of Pared

7.1 Introduction

In this chapter we present a description of the main classes that constitute Pared and their corresponding relations. Pared is a system that adaptively computes solutions of PDEs for a variety of mesh types and differential equations. In Chapter 3 we present a short overview of Pared and explained how to use our system to solve PDEs.

Pared can execute on serial and parallel computers. The serial version, which is based in earlier work [17, 18] and was implemented in conjunction with Chatzi, was also used for the Crystalline meshes project [25] by Chatzi and Preparata. In parallel computers, the adaptive solution involves several phases of solving systems of linear equations, error estimation, mesh refinement and coarsening, computing new distributions of the mesh and migrating mesh data structures and associated equations and unknowns between processors.

One of the most powerful ideas in Pared is the notion of *dynamic meshes*. Pared allows the easy creation and destruction of mesh objects in each individual processor while, at the same time, it maintains a consistent global mesh distributed across multiple processors. The mesh classes used to create our meshes are described in Section 7.2. Sections 7.3 and 7.5 present the classes that we have have designed to simplify the parallel refinement and migration of the mesh. The partitioning of the mesh uses several graph classes, which is the subject of Section 7.4. Pared also provides a graphical user interface to allow the user to interact with the system. These classes, which also control the serial and parallel execution of the system, are presented in Section 7.6. Pared provides several methods for computing solutions to linear systems of equations as explained in Section 7.7. Finally, in Section 7.9

we explain how to extend our system to solve PDEs other than the predefined ones.

The figures in this chapter show Unified Modeling Language (UML) [13, 75] class diagrams and were obtained directly from the source code using Rational's Rose software modeling environment. UML is a standard language for object oriented design. In UML a class is displayed as a rectangle, which includes its name and may include attributes and methods. Relations between classes are represented with lines. Parent and children classes (or "is-a" relationships) are joined with a solid line with an arrowhead pointing to the parent class. Aggregations (or "has-a" relationships) model "whole/part" relationships (e.g. a mesh has many elements) and are represented with lines with an open diamond at the container ("whole") end. Compositions are special kinds of aggregations with deeper semantics: an object (the "whole") is responsible for managing the parts. Parts can only exist inside the whole and are created and destroyed by it. Compositions are represented in UML with filled diamonds.

Many classes in PARED use reference counting [62], a commonly used concept in object-oriented systems that provides a simple mechanism for handling dynamic objects and managing memory. Objects that use reference counting are created from the heap and accessed through special pointers called smart pointers. When a new smart pointer to an object is created it increments the reference count of the pointed object. When the smart pointer is destroyed, the reference counting is decremented. In the case that this count is zero the object can be safely removed from memory because is not reachable by any smart pointer. The only restriction is that the programmer must not create standard C pointers to that object. In PARED classes that use reference counting extend the common `Obj` class. Smart pointers to a particular object of type `T` are created from the templated class `Ptr<T>`.

7.2 Mesh Data Structures

The basic data structures in PARED are the mesh, element and vertex classes. A mesh can be seen as a complex container for elements and vertices. Elements and vertices are stored in the mesh in two *intrusive* lists [88] which are implemented using templates. One important property of intrusive lists is that both inserting and removing objects from the list can be done in constant time. Every class that uses intrusive lists in PARED extends a common `IItem` class that includes pointers to the next and previous element in the list. Therefore, every object can be inserted in at most one intrusive list at the same time. In PARED when a new element or vertex is created it automatically inserts itself into the corresponding list of elements and vertices in the current mesh. Similarly, when an element or vertex is destroyed because it is no longer referenced, it automatically removes itself from the corresponding list.

PARED is an adaptive system in which elements and vertices are continuously created and deleted. In Chapter 3 we explained that these adaptive meshes are represented as a forest of trees. In our system, every element has a pointer to its parent element and a list of references to its children elements, which is empty in the case of unrefined elements. Every mesh also contains a root element whose children are the elements in the initial mesh.

The intrusive list of vertices in the mesh contains all the vertices in the mesh. On the other hand, the intrusive of elements only contains the elements in the most refined mesh. This mesh is used in the simulation. When an element is refined, the refined element is removed from the list of elements and its children are inserted into it. The new vertices created in the refinement of the element are always inserted in the vertex list. When an element is coarsened, the children are removed from the list and replaced by their parent. Any vertex that is no longer referenced by an element as a result of mesh coarsening automatically removes itself from the vertex list before it gets destroyed.

7.2.1 Mesh Classes

The mesh classes consist of the abstract class `FEMesh` which is the parent of two classes `FEMeshCommon` and `FEMeshStub`. These relationships are represented by the lines with an arrow towards the parent class in Figure 7.1.

In the serial version of the system the current mesh is represented as an instance of the concrete classes `FEMesh2D` and `FEMesh3D` that extend the `FEMeshCommon` class and implement two- and three-dimensional meshes respectively. A fraction of the mesh is also a mesh. Therefore in the parallel system the distributed mesh is represented as a set of `FEMesh2D` and `FEMesh3D` objects, one in each processor. These objects in each processor contain only the elements and vertices currently assigned to the processor.

`FEMeshCommon` contains most of the actual methods that operate on a mesh. `FEMeshStub` is a class that is used in the parallel version of our system and that shares the same interface as `FEMeshCommon` so the methods can be invoked on objects of the same class in a similar way. In a parallel environment, the coordinator processor creates an instance of the `FEMeshStub` class that receives commands from the console (Section 7.6). `FEMeshStub` does not implement these methods directly but it broadcasts the commands it receives to the

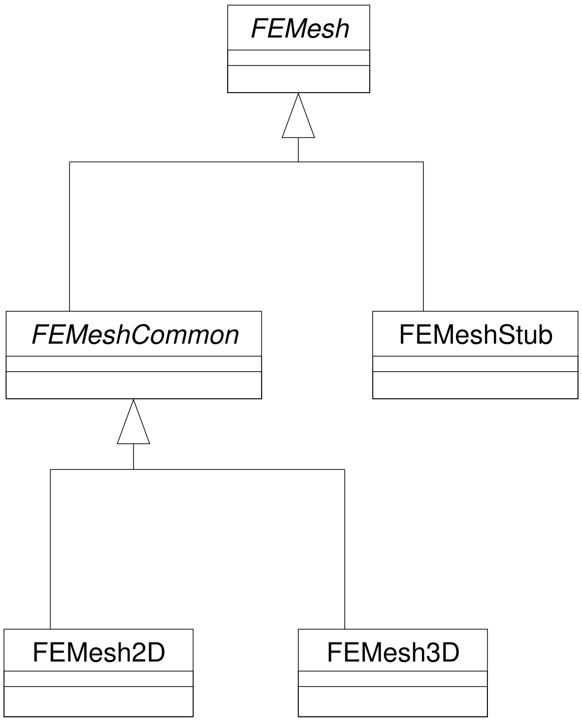


Figure 7.1: `FEMesh` classes. `FEMeshCommon` is an actual mesh or a portion of the mesh located in a processor. The concrete classes are `FEMesh2D` for 2D meshes and `FEMesh3D` for 3D ones. To simplify the use of parallel meshes we created the class `FEMeshStub` that implements the same interface (or public methods) as `FEMeshCommon`.

processors. All the processors receive these commands through another console class which invokes the corresponding methods in their instances of `FEMesh2D` or `FEMesh3D`.

7.2.2 Element Classes

PARED supports a variety of mesh types that can consist of different element shapes. For this reason, we provide a hierarchy of element classes that is rooted by the class `RealElement` and extended by the classes `RealElement2D` and `RealElement3D` for 2D and 3D meshes respectively as shown in Figure 7.2. We further extend this classes to create elements of type `Triangle` or `Quadrilateral` and `Tetrahedron` or `Hexahedron`. To support Crystalline meshes [25] we also allow elements of type `Pentahedron` and `Heptaheron`, which are not shown in Figure 7.2. In PARED it is not required that all the elements are of the same type and we can mix two- or three-dimensional shapes in the same mesh.

Every real element contains a vector of smart pointers to the vertices that define the

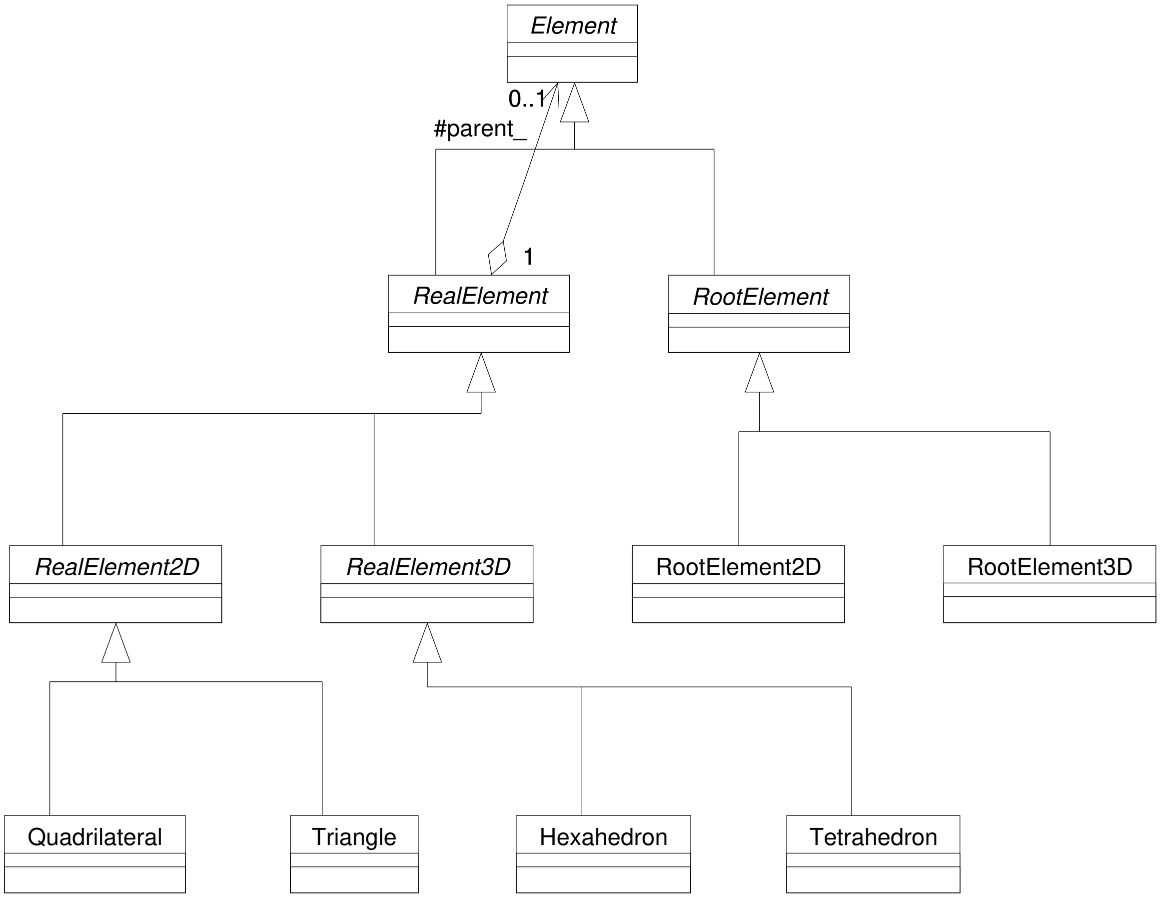


Figure 7.2: The `Element` class hierarchy supports two- and three-dimensional elements. The `RootElement` classes store references to the elements in the initial mesh.

element. The number of entries in this vector depends on the element type. To support the structure of refinement trees, every element maintains a pointer to its parent (the element that was refined to create this element) and a list of smart pointers to its children (the nested elements into which the element is refined). The pointer to the parent is used to simplify the coarsening phase. By using a list, we allow for arbitrary refinement strategies that have different numbers of children.

The reference count in the elements is from the parents to the children. The root element has a list of smart pointers to the elements of the initial mesh while the elements have a list of smart pointers to their children. On the other hand, the pointer from a child to its parent is a simple C pointer. One of the difficulties of using smart pointers in C++ is the necessity of avoiding circular structures. For example, an object that points to another object which in turn point to the first object. These structures can have very complicated shapes and be unreachable but they are not deleted because their count never becomes zero. These errors are avoided by having only one direction for smart pointers.

Besides the elements that are part of the mesh, PARED supports another type of element that we call `RootElement`. This class is also extended into `RootElement2D` and `RootElement3D` for 2D and 3D meshes respectively. Every mesh contains only one root element. The parent of an element of the initial mesh (which is not obtained as a result of refinement) is the root element and the children of the root are the elements in the initial mesh. We created the superclass `Element` that is the parent class of both `RealElement` and `RootElement`. The pointer to the parent element is of type `Element` so that the elements of the initial mesh and the elements created as a result of refinement can be handled in the same way.

In PARED, to delete a whole refinement tree (for example, after migrating a subtree to another processor), we only need to remove the parent element from the children list of the root element. This process automatically deletes all the children of that element as their reference count becomes zero. In turn most vertices in a refinement tree will be destroyed because they are not longer referenced by any element. Also, to delete a mesh, we only need to delete the root element which in turn will delete the mesh, level by level, and the vertices.

7.2.3 Vertex Classes

A vertex is defined by its coordinates in two and three dimensions. In PARED there is a common class `Vertex` that is specialized into the classes `Vertex2D` and `Vertex3D` as shown

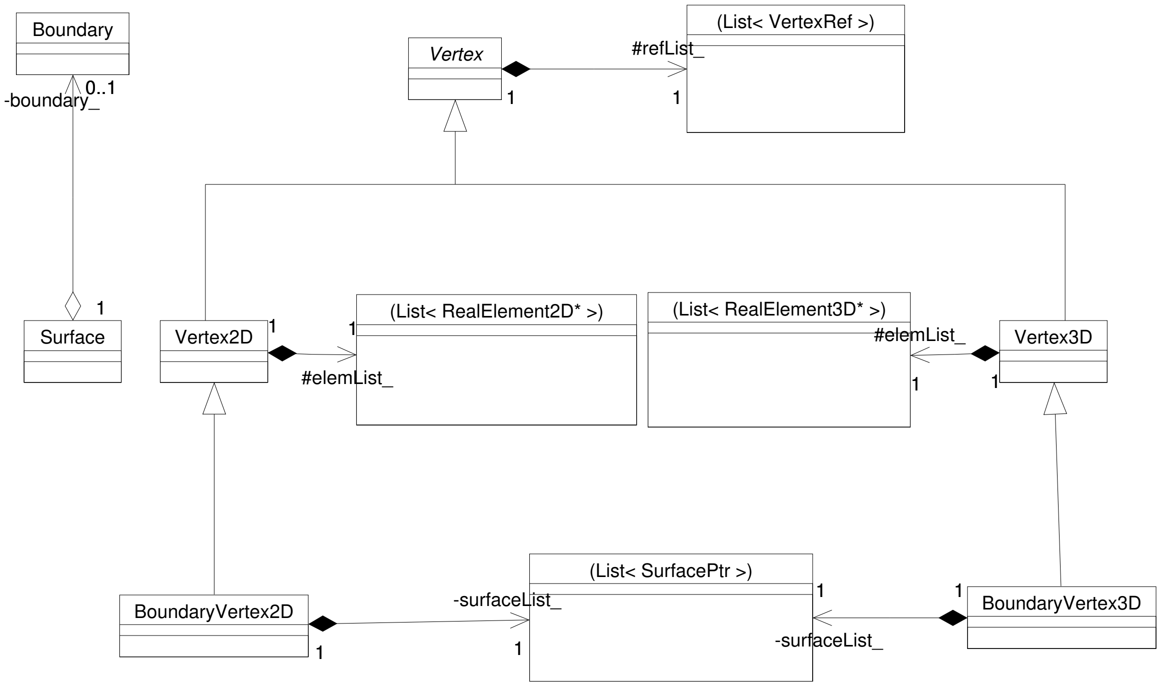


Figure 7.3: The `Vertex` class hierarchy. PARED defines classes for two and three dimensional vertices (`Vertex2D` and `Vertex3D`). Vertices in the boundary of the domain (of type `BoundaryVertex2D` and `BoundaryVertex3D`) contain references to the `Boundary` objects.

in Figure 7.3.

In many cases it is necessary to find the elements that are adjacent to a particular vertex. For that reason every vertex maintains a list of pointers to the elements that contain it. Again, to avoid the problem of circular smart pointers, the list of elements in the vertices is of standard C pointer of type `RealElement2D` and `RealElement3D`. When an element is created or destroyed it inserts or removes itself into the lists of its associated vertices.

One particular type of vertex is the boundary vertex: a vertex located on the boundary of a domain. In PARED a mesh boundary contains several `Surface`s. A set of surfaces with common boundary conditions form a `Boundary` object. PARED provides two classes, `BoundaryVertex2D` and `BoundaryVertex3D`, that extend the classes `Vertex2D` and `Vertex3D` respectively.

PARED does not explicitly store edges and faces. Every edge is defined by references to two vertices. Faces are defined by three or more vertices. Nevertheless, the vertices, edges and faces of every element are ordered and it possible to create temporary instances of the classes `Edge` or `Face` as needed.

7.2.4 Parallel Meshes

In PARED the meshes used in the serial and parallel version of the system are very similar. We use a partition of the mesh between processors by elements, where elements are located in only on processor and vertices are shared if they are are adjacent to elements located in different processors. Every one of these processors contains a copy of the shared vertex. During the simulation vertices that are not shared can become shared as adjacent elements are moved between processors. Similarly, shared vertices can become non-shared.

To maintain the consistency of the global mesh it is important to identify the multiple copies of the same vertex located in different processors. In the systems of equations used to solve the partial differential equations, every unknown corresponds to a node that might be located in a boundary between processors. In this case, it is important that the node includes the contributions of all the elements around that node.

In PARED each vertex has a list of `VertexRef` objects associated with it that is empty in the case of non-shared vertices. `VertexRef` is a remote reference, consisting of a processor number and a memory address in the remote address of each a copy of the associated vertex. Using this address, a processor can send a message to another processor to invoke a method on the remote copy. In the destination processor, the reference is converted to a pointer of type `Vertex2D` or `Vertex3D` through which the method is invoked.

7.3 Classes Controlling the Refinement of Meshes

The procedure for refining serial and parallel meshes using the longest-edge bisection method [19, 20] is very similar. In both cases we iterate through a list of references to selected triangular or tetrahedral elements, invoking the `refineRivara` method in each of them. In this section we outline the additional classes used for the parallel refinement of meshes using the longest-edge bisection algorithm.

The parallel refinement procedure is controlled by an object of the class `RefineMgr` (Figure 7.4). This class is a *singleton* class [40] (at most one instance of the class can exist in each process at any time) that handles all the synchronization steps explained in Chapter 4. Every processor, including the coordinator, creates an instance of `RefineMgr` by executing its static method `start` before refining any element.

In parallel meshes the refinement of triangle or tetrahedron can create a new vertex on an internal boundary between processors. As we have explained in Chapter 4 we delay the

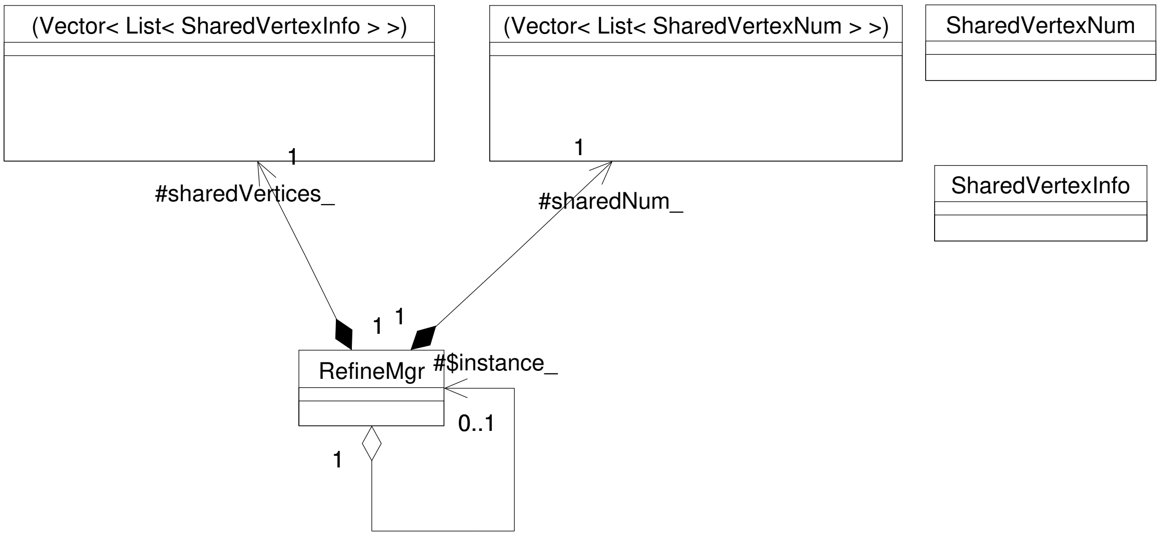


Figure 7.4: Mesh refinement classes. `RefineMgr` uses the classes `SharedVertexInfo` and `SharedVertexNum` to manage the parallel refinement of meshes.

propagation of refinement across processor boundaries. When processor P_i creates a new vertex in a boundary between one or more processors P_j it creates one or more instances of the auxiliary class `SharedVertexInfo` (one for each adjacent processor) which it inserts in `RefineMgr`. Therefore, `RefineMgr` is also a container for all the newly created shared vertices and in the current implementation it contains a vector of lists of `SharedVertexInfo`. This class includes all the information required to create the shared vertex in the adjacent processor P_j and to return to processor P_i the address of the new vertex.

After each processor has completed the refinement of the initial elements it calls the `wait` method of the `RefineMgr` class. In this method every processor sends the `SharedVertexInfo` objects in the `RefineMgr` object created during the refinement phase to the adjacent processors and then listens for messages. If it receives a new refinement message, it creates the vertices as specified in the message and proceeds to refine the unrefined adjacent elements if necessary using their `refineRivara` method. In the case the element is already refined by the edge including in the message, it just updates the remote references to the new vertex.

The `RefineMgr` object also keeps track of the acknowledgment messages. Every refine message is acknowledged as explained in Chapter 4. The coordinator waits until it receives an acknowledgment from every processor involved in the refinement and it then signals the termination of the refinement phase by broadcasting a message. When a processor other than the coordinator receives a message from the coordinator, it stops listening from messages and exits the `wait` method.

7.4 Partitioning of Meshes

In PARED the graph classes have the responsibility of computing partitions of the mesh. The partitions of the mesh using PNR are obtained from a sequence of K coarser graphs $G_0, \ldots G_K$. For that reason, the `Graph` in PARED is represented as a vector of K of `GraphLevel` objects, where each level corresponds to one of the K graphs. Each `GraphLevel` contains a vector of instances of `GraphPartition`. The `GraphPartition` class contains an intrusive list of `GraphVertex`. Adjacent `GraphVertex` in each graph are joined by variable number of `GraphEdge` objects. Therefore, the `GraphPartition` p entry of the the `GraphLevel` l object of the `Graph` class contains all the graph vertices of the graph G_l assigned to subset p. To compute new partitions `GraphVertex` objects move between `GraphPartition` objects in the same graph level. A graph diagram of these classes are shown in Figure 7.5.

The first `GraphLevel` of the `Graph` object contains one `GraphVertex` for every element in the mesh and graph vertices are joined by a `GraphEdge` if the corresponding elements are adjacent. `GraphElement` and `GraphVertex` include an integer weight that represents the computation and communication cost respectively. From this first fine graph we construct the sequence of coarser graphs using an edge contraction operation until the number of vertices in the coarser graph is less than some specified constant. The edge contraction operation maintains the relation between the vertices in two adjacent levels so a partition of a coarser graph can easily be projected to the next level. To partition a graph we move or swap graph vertices between instances `GraphPartition` objects of a given level.

Finally a `GraphAssignment` object represents a current assignment of graph vertices (or their corresponding mesh elements) to processors. This class is used to compute the migration cost and to tell the processors which elements to migrate.

7.5 Classes Controlling Mesh Migration

As in mesh refinement, PARED provides a class called `MigrateMgr` (Figure 7.6) that handles the synchronization required for the migration of meshes between processors. `MigrateMgr` is also a singleton class and contains methods for the different steps outlined in Chapter 6.

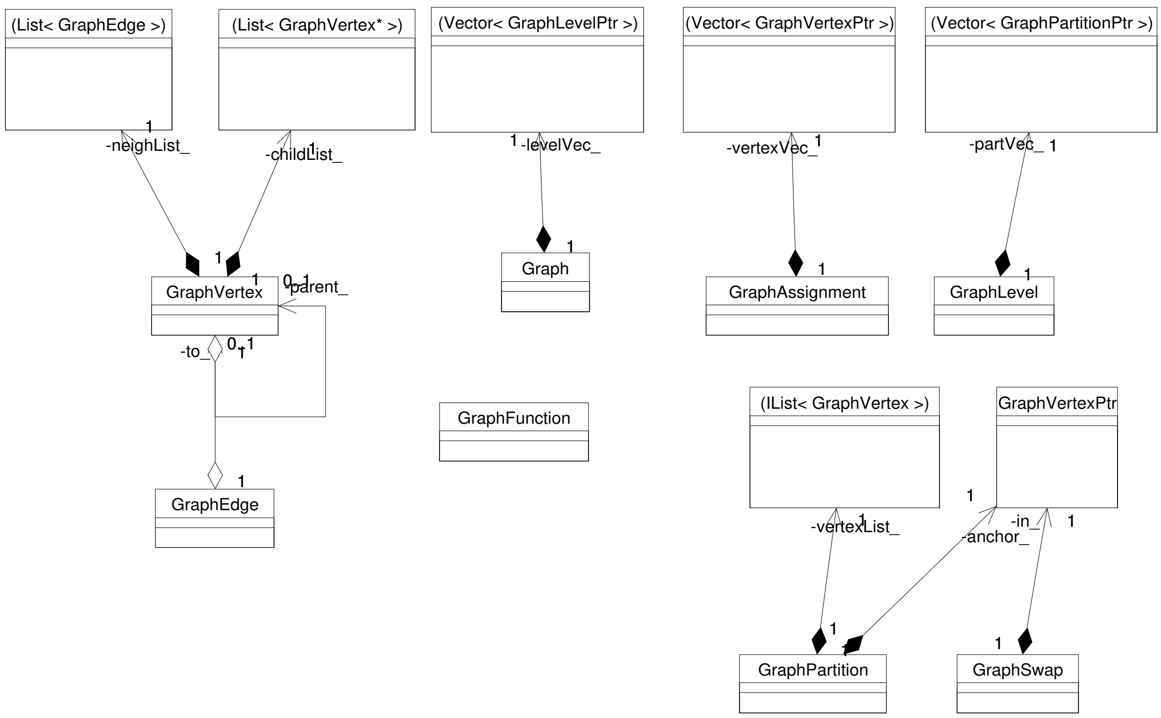


Figure 7.5: Class diagram of the classes used to compute partitions of the mesh.

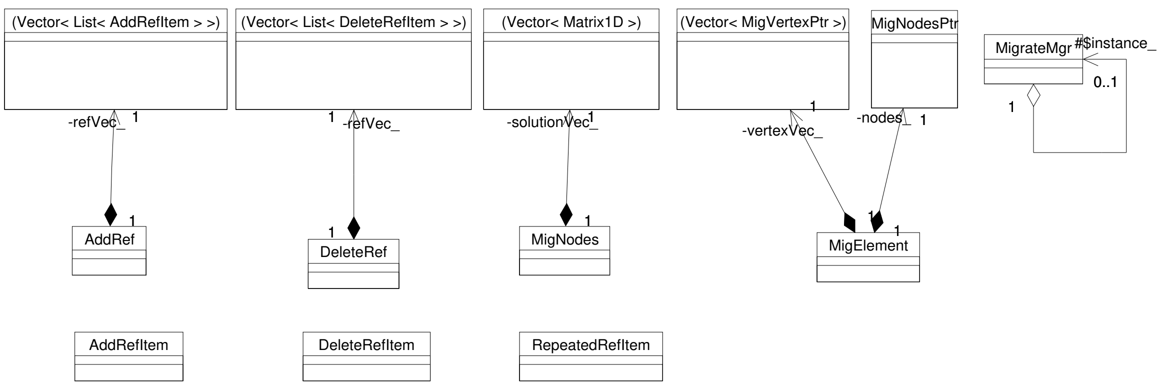


Figure 7.6: Mesh migration classes. The manager class `MigrateMgr` uses a variety of auxiliary classes such as `AddRef` and `DeleteRef`.

MigrateMgr uses several additional classes. The AddRefItem class represents that a remote copy of the vertex should add a reference to a new copy of that vertex. Similarly DeleteRefItem is used to remove references in remote processors so vertices can safely be deleted. Several objects of the classes AddRefItem and DeleteRefItem destined for the same processor are grouped into lists by the classes AddRef and DeleteRef of which there is one list per processor. Finally, the class RepeatedRefItem stores references to repeated vertices, for example, when the same vertex is received from two or more processors. In that case all but one of the copies are destroyed to eliminate unnecessary storage and avoid errors and the references and adjacent elements are updated to maintain only one copy of the same vertex in each processor.

The migration procedure starts by creating the MigrateMgr object and providing a vector of p lists (one for every processor) of references to RealElement. These lists contain a reference to the elements in the initial mesh whose refinement trees must be migrated to the other processors. The lists are then extended to include all the elements in the subtree. If an element is in the list, we insert its children into the list, but with the condition that the parent appears in the list before its children. Therefore, in the destination processor, we can first construct the parent before its children which then is in turn used to create the children.

The different methods of MigrateMgr that are required to migrate the mesh are invoked from the migrateAux method of the class FEMeshCommon. This method first calls the methods sendElements and receiveElements of the MigrateMgr class to send and receive elements (including the refinement trees), vertices, nodes and temporary solutions. These methods use two additional classes called MigElement and MigVertex that roughly correspond to mesh elements and vertices but are easier to serialize in a message.

After receiving and constructing the mesh objects we obtain an object of the class AddRef that contains the references of the newly created vertices. We then send and receive these references by calling the sendReferences and receiveReferences of the AddRef and call the updateReferencesToShared of the MigrateMgr to update the older copies of the vertices with the new references that we just received. We then find the repeated vertices in the same processor. A vertex is repeated if it contains a reference to another vertex located in the same processor, but before destroying it we instruct other processors to delete its references to it by using the class DeleteRef.

We now delete the elements that were sent to other processors. This in turn automatically deletes vertices than are no longer referenced by any element in the processor because

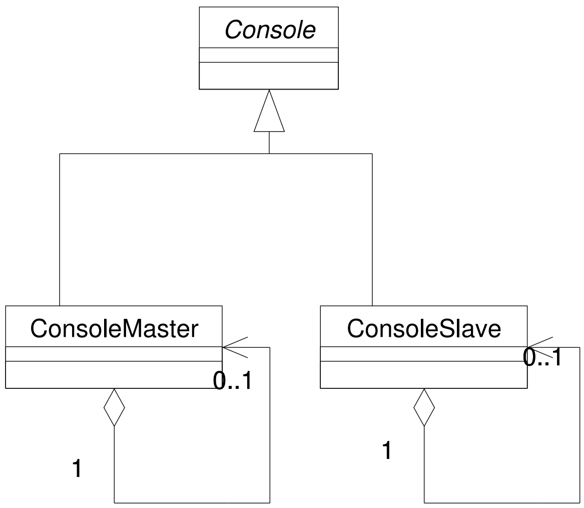


Figure 7.7: Users interact with Pared through `Console` objects. The `ConsoleMaster` class is used in serial systems. In parallel mode a `ConsoleMaster` is located in the coordinator processor, while the remaining processors contain a `ConsoleSlave` object.

their reference count becomes zero. Some of these vertices might have copies in other processors so before deleting them we create an object of the class `DeleteRef` to remove the references to them from other processors.

7.6 Console Classes

In Pared, a `Console` allows the user to communicate with the system. The console receives the commands invoked by the user and delivers these instructions to the mesh classes where they are executed.

`Console` is also a singleton class. In serial systems, Pared creates at startup an object of the class `ConsoleMaster` that subclasses `Console` (Figure 7.7). The parent class contains a pointer of type `FEMesh` to the active mesh which is an instance of `FEMesh2D` or `FEMesh3D`.

The `ConsoleMaster` object receives all the instructions from the user such as how to refine the mesh or compute a solution. These instructions can come from our Motif graphical user interface or through a test file that automates repeated commands. The `ConsoleMaster` object is also used to provide information to the user. To render the mesh, the mesh classes do not call OpenGL [39] commands directly. Instead they render the mesh through the console.

On parallel systems, only the coordinator processor (or processor number 0) creates an instance of `ConsoleMaster`. All the other processors creates an instance of `ConsoleSlave` that also extends the class `Console` and contain a pointer to the `FEMesh` object located in the processor. The `ConsoleMaster` object in the coordinator receives the commands for the user, but rather than invoking methods on `FEMesh2D` or `FEMesh3D` objects directly, it invokes a method on an object of type `FEMeshStub`. This class also inherits from the class `FEMesh` and shares the same interface with `FEMesh2D` and `FEMesh3D`. Nevertheless, rather than executing the mesh methods directly, `FEMeshStub` marhalls the methods and parameters into messages that it then broadcasts to the other processors.

All the remaining processors wait until they receive a message from the coordinator in a method of the `ConsoleSlave` object. They then decode the message and parameters in the `ConsoleSlave` object and invoke the method in their corresponding `FEMesh2D` or `FEmesh3D` object.

To render the mesh in parallel, the process is inverted. Each processor renders its region of the mesh by calling methods on their `ConsoleSlave` object. These objects capture these instructions that they marshal and send to the coordinator. The `ConsoleMaster` object in the coordinator receives these messages and invokes the respective OpenGL commands to render the mesh.

7.7 Representing Systems of Equations

PARED provides a hierarchy of classes for creating vectors and arrays of double precision values. This hierarchy is rooted by the abstract class `AbsMatrix` which is the parent of the classes `AbsMatrix1D`, `AbsMatrix2D` and `AbsMatrix3D` for 1, 2 and 3 dimensional arrays respectively as shown in Figure 7.8.

`Matrix1D` is a concrete class that extends the class `AbsMatrix1D` and stores one dimensional vectors. `Matrix2D` is used to store full two-dimensional matrices. In finite element analysis most of the matrices are sparse and full matrices can only be used in very small problems. In many problems, the matrix is also symmetric.

There are several sparse matrix representations available in PARED. `SmallMatrix2D` is a very common representation in finite element systems in which a full matrix is stored as a list of small local matrices, one for each element in the mesh. With every local matrix we associate an index vector that maps the entries in the local matrix to their rows and columns in the full matrix.

`SparseMatrix2D` uses a matrix storage that is optimized for multiplying the matrix by

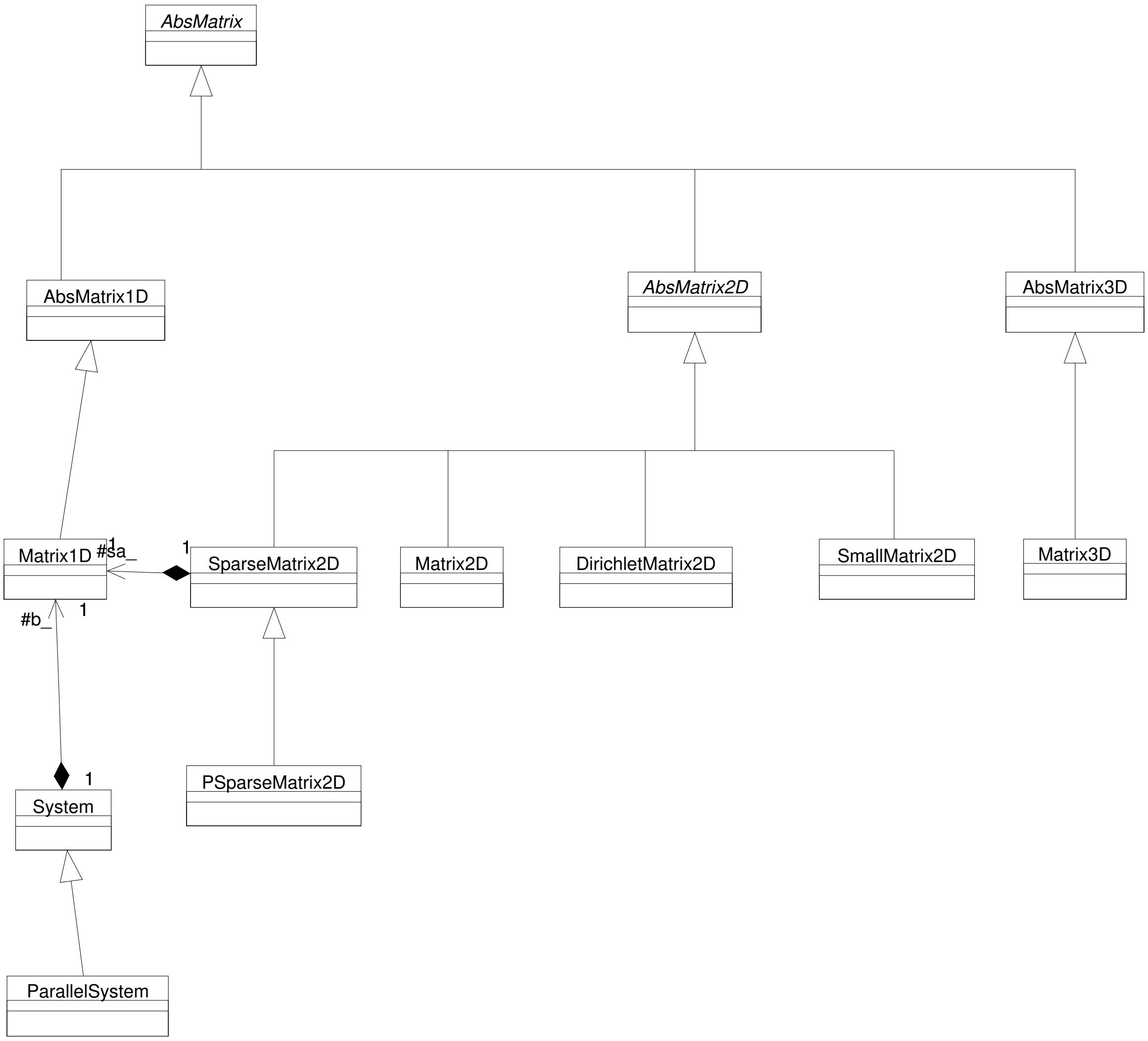


Figure 7.8: Matrix class diagram. PARED supports one- two- and three-dimensional matrices using a variety of storage methods.

.

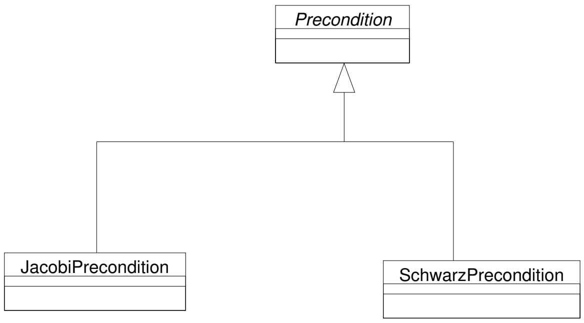


Figure 7.9: The system class combines `AbsMatrix2D`, `AbsMatrix1D` and `Precondition` and provides methods for solving linear systems.

a vector. This sparse matrix is known as Row Indexed Sparse Storage [70, 31, 32]. The nonzero entries in the matrix are stored row by row in a vector of values. The corresponding columns are stored in a vector of indices. The diagonal entries in the matrix are commonly used in many linear solvers and for that reason they are assigned to the first entries in the vector of values. `PSparseMatrix2D` is a permuted `SparseMatrix2D` that is used in direct solvers.

We are experimenting with a storage for sparse matrices that we call `DirichletMatrix2D`. In this representation, a matrix is stored as a vector of `SparseMatrix2D` objects. The first matrix of the vector contains the internal nodes. The remaining matrices correspond to the boundaries of the domain. On problems with multiple dimensions and multiple matrices with different boundary conditions we can use the same entries and reduce the amount of storage required.

To solve systems of equations PARED uses direct and iterative solvers such as Conjugate Gradient [58], GMRES [93] and their preconditioned versions. These solvers are implemented as methods of the class `System`. The `System` class contains a pointer to a matrix (of type `AbsMatrix2D`) and a right hand side vector (of type `AbsMatrix1D`) and its solvers take as a parameter a solution vector (also of type `AbsMatrix1D`). Our two dimensional matrices provide the `mult` method that multiplies the matrix by a vector.

Preconditioned solvers (Figure 7.9) require the use a concrete class that extends the abstract class `Precondition`, such as `JacobiPrecondition` or `SchwarzPrecondition` (both additive and multiplicative). These classes provide the abstract method `apply` that applies the preconditioner in the solution of the system of equations.

The procedure for parallel solvers is very similar. In this case, we construct an instance of the class ParallelSystem that extends the class System. The only difference in the iterative solvers is the way the matrix-vector products and inner-products are computed. In the parallel solver, after calling the mult of the sparse matrix, it is necessary to accumulate the results that correspond to unknowns located on processor boundaries (see Section 5). Parallel inner-products also compute a global sum of all the processor inner-products.

7.8 Higher Order Polynomials

One of the extensions that we added to our original system is the ability to compute solutions of PDEs using higher-order basis functions. In our original system, each mesh vertex had an associated set of basis functions, one for each degree of freedom.

We now describe the classes used to support higher order polynomials (see Figure 7.10). A Node is associated with a basis function. A mesh element contains a variable number of nodes depending on the degree of the approximation, which are of type Node2D or Node3D for 2D or 3D meshes respectively. Nodes that are located in a mesh vertex, edge or face include the contributions of all the elements around it, and roughly correspond to an equation and an unknown in a system of linear equations.

It is not necessary to actually create the nodes. It is sufficient to be able to determine the corresponding node indices so that the local element matrices can be correctly assembled in a global matrix and the corresponding node coordinates to create the local element matrices. For that reason, we created the NodeIndex hierarchy. This parent class is specialized into NodeIndex2D and NodeIndex3D. Finally, every element shape (triangles, quadrilaterals, tetrahedra, hexahedra) provides a class for each polynomial that can be used in the simulation. For example, the class NodeIndexTetrahedron_TET10 defines quadratic basis functions in a tetrahedron. Each tetrahedron using this basis function contains 10 nodes, that are located in the four vertices and at the midpoint of the six edges of the tetrahedron. Similarly, we can define new node index classes to support a variety of basis functions in each element.

7.9 Specifying Problems in PARED

The Problem class provides a flexible mechanism for customizing PARED. By extending the Problem hierarchy the user can define and solve her own partial differential equations besides the ones already predefined in the system. In this section explain the steps that

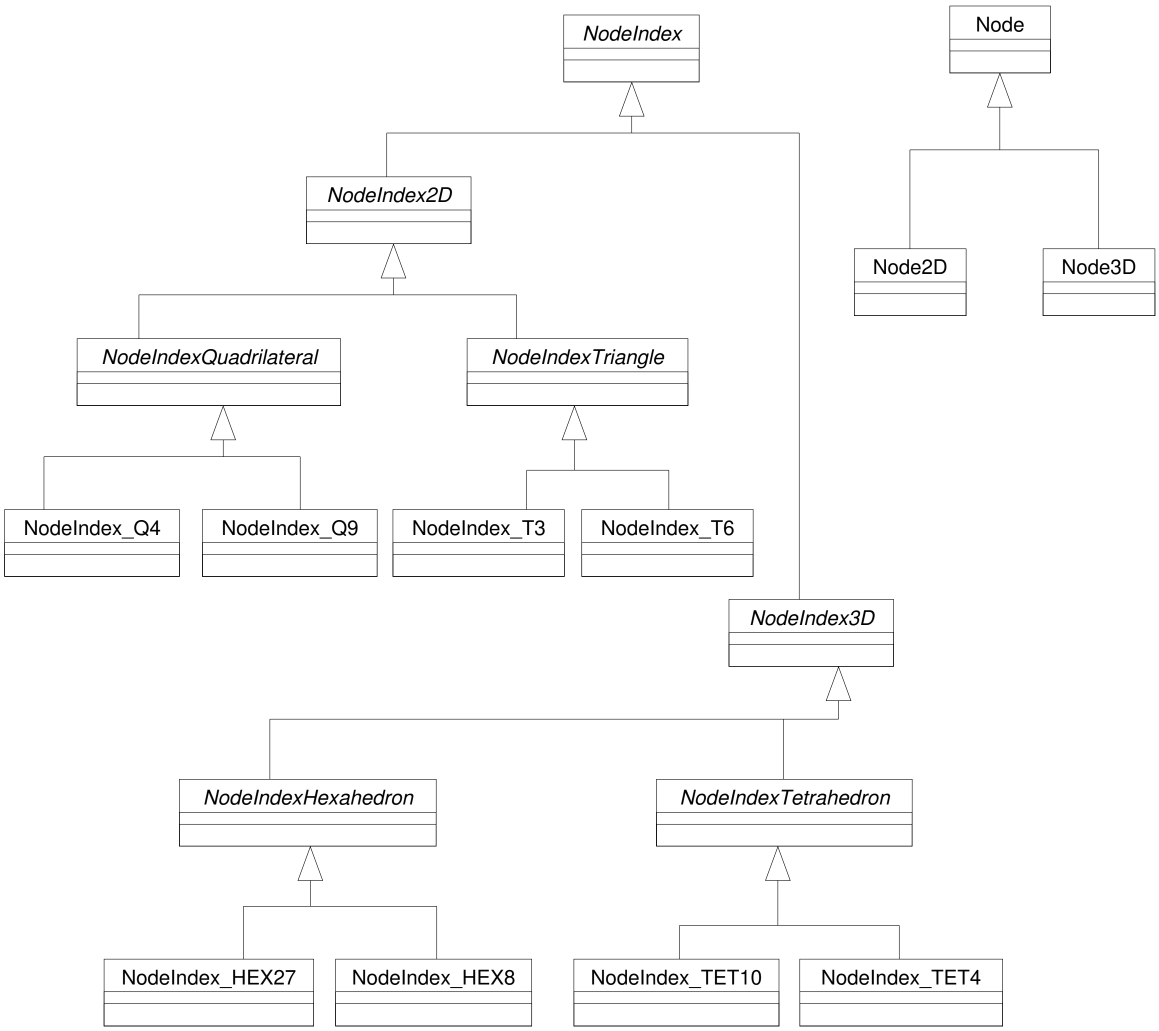


Figure 7.10: Diagram of the classes used to define higher order approximations.

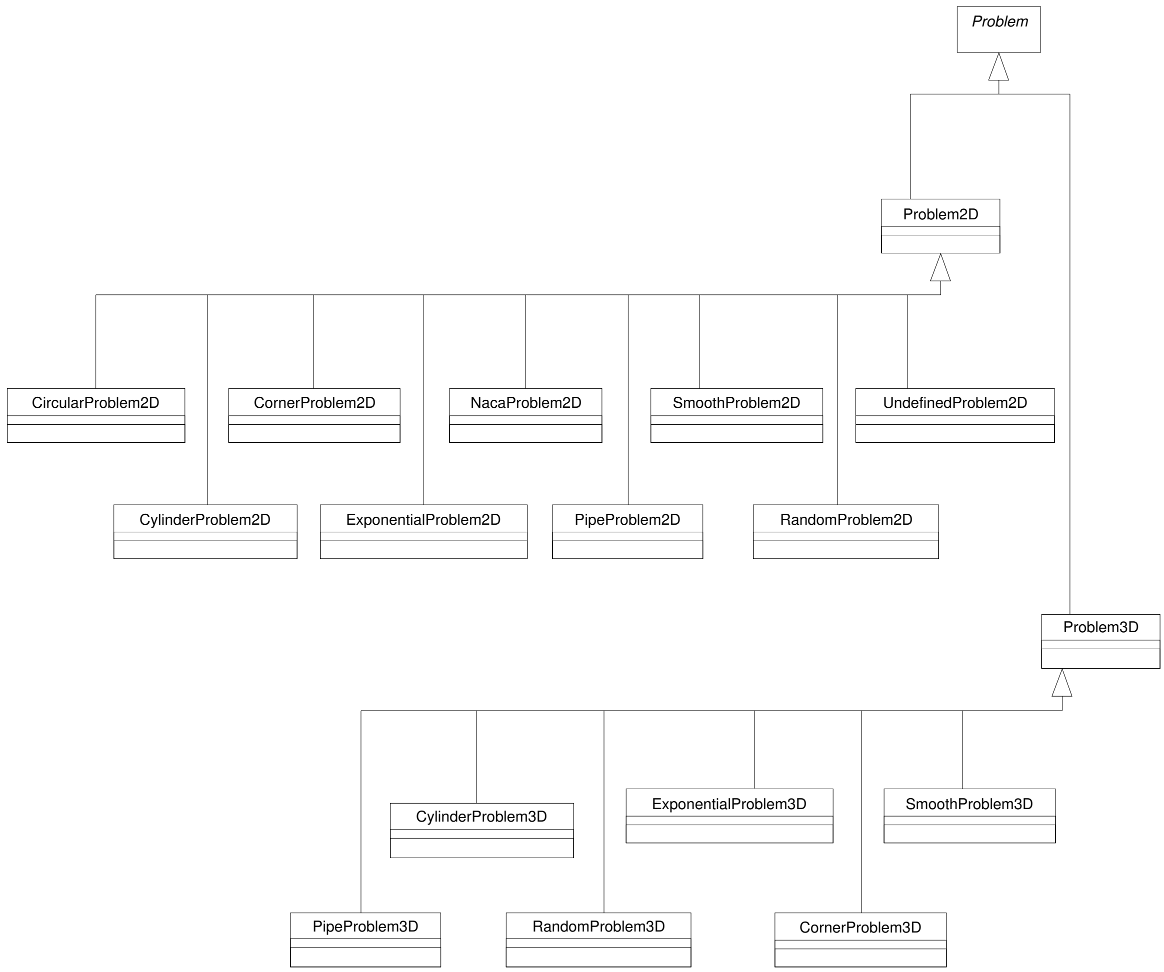


Figure 7.11: The `Problem` class hierarchy defines the equation to solve.

are necessary for defining new differential equations in PARED and we discuss possible improvements to this process.

7.9.1 Defining New Problems in PARED

The `Problem` class (Figure 7.11) and its two subclasses `Problem2D` and `Problem3D` encapsulate the concepts that define a partial differential equation, such as its right hand side, number and type of boundary conditions, initial values and temporary or computed solutions.

To define a new differential equation the user must provide a new class that extends either the `Problem2D` or `Problem3D` classes and that provide a definition to five of the abstract methods declared in `Problem`. For example:

```
class RandomProblem3D : public Problem3D {
private:

public:
  RandomProblem3D() : Problem3D(RANDOMPROBLEM3D, 2, 1, 1) {}

  virtual double exactSolution(int degree, double x, double y, double
  z = 0);
  virtual double rhs(int degree, double x, double y, double z = 0);
  virtual double dirichletValue(int degree, int num, double x, double
  y, double z = 0);
  virtual double neumannValue(int num, int degree = 0);
  virtual boundary_t boundaryType(int num, int degree = 0);
};
```

which is defined in the files `Problem.H` and `Problem.C`. In this class, `degree` refers to the degree of the unknowns (e.g. `u`, `v`, `w` or `p`) and `num` to a boundary number. The method `exactSolution` returns the value of the functions at any point of the domain (if known) and the `rhs` method returns the right hand side function. Our system supports two different types of boundaries (of type `boundary_t`): `NEUMANN` and `DIRICHLET`. Each surface in the boundary of the domain has a boundary number and the boundary vertices in the mesh must specify their corresponding surfaces. The parameters in the constructor of the `Problem` class specify the problem type (of type `problem_t`; i.e. `RANDOMPROBLEM3D`), the number of different boundaries, the number of solutions and the number of temporary vectors (including the solutions). This last parameter is used to migrate the temporary vectors between processors along with the solutions in the parallel solution of the partial differential equations.

To select a particular differential equation to solve the user must invoke the method `ConsoleMaster::setProblem`, which initializes a data member of the `Console` class.

7.9.2 Predefined Differential Equations

PARED provides several predefined partial differential equations:

- `CornerProblem2D` and `CornerProblem3D` defines the equation $-\Delta u = 0$ with Dirichlet boundary conditions

$$g(x,y) = \cos(2\pi(x-y))\frac{\sinh(2\pi(x+y+2))}{\sinh(8\pi)}$$

  in $\partial\Omega$. For the 2D problem this function is defined in the domain $\Omega = (-1,1)^2$. In the 3D case the domain is the cube $\Omega = (-1,1)^3$. The solution $u = g$ is known. These solutions are smooth with a high activity peak close to the corner $(1,1)$ and $(1,1,1)$.

- `RandomProblem2D` and `RandomProblem3D` defines the equations $-\Delta u = g$ with $g(x,y) = \sin(1000.0\cos(1000.0xy))$ and $g(x,y,z) = \sin(1000.0\cos(1000.0xyz))$ in the unit square and cube with Dirichlet boundary conditions $u = 0$ in $\partial\Omega$.

- `SmoothProblem2D` and `SmoothProblem3D` are the two equations $-\Delta u = g$ with $g(x,y) = 2\pi^2\sin(\pi x)\sin(\pi y)$ and $g(x,y,z) = 3\pi^2\sin(\pi x)\sin(\pi y)\sin(\pi z)$ in the unit square and cube.

- `PipeProblem2D` and `PipeProblem3D` solves the Navier-Stokes equation for incompressible flow in a 2D and 3D pipe respectively.

- `CylinderProblem2D` and `CylinderProblem3D` also solve the Navier-Stokes equation for incompressible flow, but in this case, the domain is interrupted by a circle (in 2D) or a sphere (in 3D).

- `ExponentialProblem2D` and `ExponentialProblem3D` define the equation $-\Delta u = g$ for

$$g(x,y,t) = \frac{1}{1+100(x+t)^2+100(y+t)^2}$$

  and

$$g(x,y,z,t) = \frac{1}{1+100(x+t)^2+100(y+t)^2+100(z+t)^2}$$

  in the square $(1,-1)^2$ and cube $(1,-1)^3$ with Dirichlet boundary conditions $u = g$ in $\partial\Omega$.

7.9.3 `Problem` and PARED

Besides the facility to extend it to solve new differential equations, the class `Problem` also stores the computed solutions and a user defined number of temporary vectors used during the solution. The number of these vectors is defined by the user and their size corresponds to the number of unknowns. In the parallel version of PARED, when the system migrates regions of the mesh between processors, along with the elements and vertices it also migrates the entries in the vectors of the `Problem` object that correspond to these regions. To store and retrieve solutions `Problem` provides methods such as `getComputedSolution`, `storeSolution` and `invalidateSolution`.

We assume that there is always one active problem which is defined in the `Console` class. `Problem` also contains a static method `getProblem` that returns an instance of the concrete classes that extend `Problem`.

7.9.4 Future Improvements to the `Problem` class

Ideally it should not be necessary to modify the code to solve new differential equations. Instead the user would provide a data file with a problem description. The main difficulty with this approach is to define a file format and to provide a parser that would accept arithmetic expressions.

The predefined problems defined above solve Poisson and Navier-Stokes equations. PARED also supports other differential equations such the Helmholtz equation $u + \Delta u = f$. Besides the Navier-Stokes problems, we have not experimented on solving systems of partial differential equations. These aspects need to be resolved to create a complete and flexible parallel simulation system.

Chapter 8

The Communications Library of Pared

8.1 Motivation

The communication patterns in many subsystems of Pared are different from most scientific code in which the same vectors of uniform type are repeatedly exchanged between processors. In Pared the mesh refinement, mesh repartition and mesh migration phases involve the communication of complex objects usually referred through pointers. Although there is limited support for user-defined data types in communication libraries such as MPI [37], which are called *derived* data types, this approach does not extend well for objects of variable size and there is no support for class inheritance. In our system objects can have a variable size, are usually referenced by virtual pointers and are scattered around memory. MPI has significant performance penalties if the data is discontinuous or not separated by a constant stride.

For these reasons we have developed a new library on top of the MPI library. All the code in Pared communicates through this library so it is easy to replace lower level function calls by another communications library such as PVM [42]. Our library was designed to simulate a standard C++ stream interface [89, 51]. In fact, our original implementation extended the standard C++ iostream class hierarchy but this approach was later dropped for performance reasons. From the point of view of an individual processor sending an object to another processor using our library is equivalent to writing that object to a file while receiving an object is similar to reading the object from a file. The library synchronizes the communication between processors. If a destination processor wants to receive an object from a stream, it waits until the object is inserted by the source processor into the stream. A processor can also block if there is no available space in the sending processor. In this case, objects need to be removed from the stream before new objects can be inserted.

The design of our library goes beyond the goal of providing C++ bindings to the C function calls of MPI (which has been addressed in latest versions of the standard [38]) or providing a set of classes that are wrappers to the MPI API such as OOMPI [61]. Instead, our goal is to provide a mechanism to communicate a large and unspecified number of relatively small objects without significantly affecting the raw MPI performance and without requiring that a message be delivered for every simple object or remote method invocation.

At the lower level our library uses the standard MPI interface to exchange data. We have investigated several variations that use the blocking and non-blocking primitives of MPI. Although we have designed our library to simplify the exchange the complex and irregular data, we pay no performance penalty to communicate vectors of objects of similar type. In this chapter we describe the different classes that compose our library and their relations. We then compare the performance of our library against IBM's implementation of MPI, LAM [24] from the Ohio State University and MPICH from Argonne National Laboratories and Mississippi State University [45] on an IBM SP and a network of SUN workstations.

8.2 New Classes in Our Communications Library

The basic components of our library are communicators and ports, buffers (where the objects are temporarily marshaled and unmarshaled into messages) and streams that use ports and buffers to exchange objects. In this section we provide an overview of these concepts, and describe the corresponding C++ classes. An UML diagram of the classes that form our library and their relations is shown in Figure 8.1.

8.2.1 Communicators and Ports

The simplest class in our library is the `MPICommunicator` class that is a wrapper around the communicator concept of MPI. In MPI a communicator is a group of processes with an associated context. This *opaque* object (it can only be accessed through handles) is used in all global and point-to-point MPI communication primitives. Each communicator has a `size` which is the number of processes that form the group and every process in the group has a unique `rank` which is its position in the group. By default MPI provides a

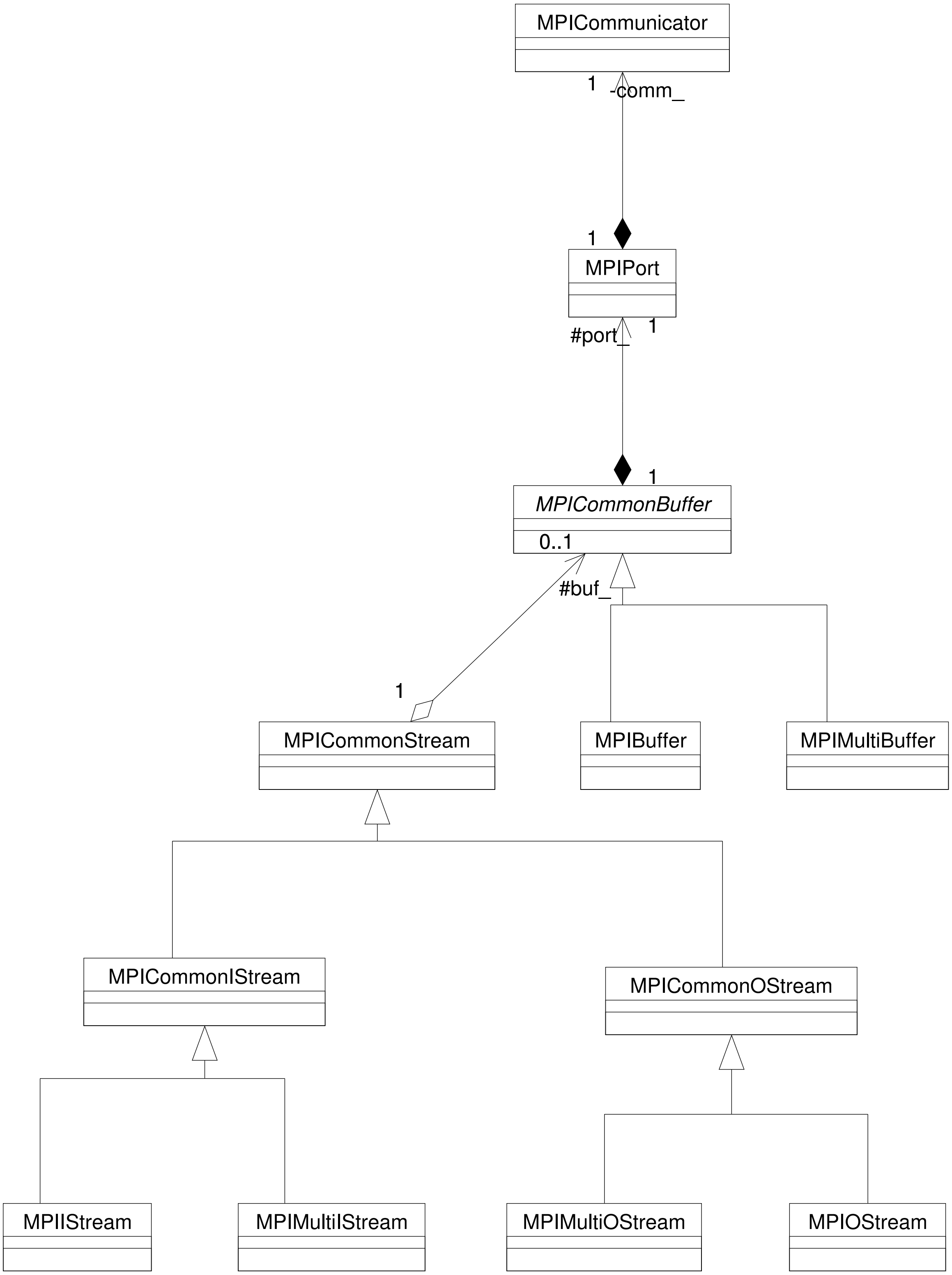


Figure 8.1: Class diagram of PARED's communications library.

`MPI_COMM_WORLD` which contains all the processes that started the computation.

The `MPICommunicator` class supports global communications primitives for computing global maximums, minimums and sums of elementary data types by calling `MPI_Allreduce`. The `MPICommunicator` class can also be used to create barriers in the execution of the code by invoking its method `barrier`.

MPI provides four different methods to exchange messages, where the *blocking* (`MPI_Send` and `MPI_Recv`) and *non-blocking* (`MPI_Isend` and `MPI_Irecv`) ones are the two most widely used. After a blocking send or receive returns from its function call, the buffer used in the call is ready to be reused (in the case of the send) or the buffer contains the message (in a receive).

For the blocking primitives the MPI standard states the following:

> It is up to MPI to decide whether outgoing messages will be buffered. MPI may buffer outgoing messages. In such case, the send call may complete before a matching receive is invoked. On the other hand, buffer space may be unavailable, or MPI may choose not to buffer outgoing messages, for performance reasons. In this case, the send call will not complete until a matching receive has been posted, and the data has been moved to the receiver.

This statement implies that even very simple codes can deadlock and that some codes successfully terminate in some machine architectures while it might deadlock in others. This situation became apparent when we moved our code from the network of workstations. Although our system was running fine using MPICH and LAM in the network of workstations, it required the reordering of some message exchanges in the SP to solve large problems.

To allow the overlap of communication and computation, or to allow overlap of multiple messages our library also uses the non-blocking primitives. A non-blocking send and receive only indicate the beginning of communication and it is necessary to test its completion with a `wait` before the buffers are modified.

An `MPIPort` is a pair of `MPICommunicator` and `rank` (or process number within the communicator) to and from where messages are sent or received. `MPIPort` can also include an optional `tag` so messages can be marked. Therefore the receiving processor can specify not only the source but also the type of message to receive. `MPIPort` provides methods for sending and receiving simple primitive data types and for sending and receiving buffers of a sequence of bytes of a specified length through the port. These methods in turn call the standard blocking and non-blocking MPI function calls.

8.2.2 Buffers

The abstract class `MPICommonBuffer` behaves like the buffer class used in the C++ iostream class hierarchy. In PARED a buffer has a length, a data region from which to insert and remove data and a port. The same buffers can be used for sending and receiving messages. This current mode is identified by a flag. Like the iostream buffers, our class has three pointers:

- `sptr` indicates the beginning of the buffer.
- `eptr` indicates the end of the buffer or the end of the read part.
- `pptr` indicates where to put the next character or from where it can be obtained.

It is possible to insert objects in the output buffers with the method `put` that takes as a parameter the size of the object and returns a pointer to the location where the object should be copied. In the case that the buffer is full, the `put` method calls the virtual method `send` which in turn calls an MPI send function to send the buffer to the processor indicated in the port associated with the buffer. Input buffers use a similar process. The `get` method of the `MPICommonBuffer` returns a memory location of a specified size and also calls the pure virtual method `recv` to receive data using MPI function calls.

The implementation of `send` and `recv` is in the classes `MPIBuffer` and `MPIMultiBuffer`. `MPIBuffer` contains a buffer of characters of a specific size and only uses blocking communication. On the other hand, the class `MPIMultiBuffer` accepts a vector of character arrays as buffers and accepts both blocking and non-blocking primitives. The most important feature of class `MPIMultiBuffer` is that it is possible to insert and remove objects from one of its buffers while the class is sending and receiving messages on the other ones.

8.2.3 Stream Classes

The library is completed with the stream classes. The abstract class `MPICommonStream` contains a pointer to a `MPICommonBuffer`. Output streams inherit from `MPICommonOStream` while input streams extend the class `MPICommonIStream` which are both subclasses of `MPICommonBuffer`. These two classes contain methods for sending and receiving vectors of objects.

Finally, the concrete stream classes create buffers of the appropriate type. The classes `MPIOstream` and `MPIIStream` use blocking `MPIBuffer` while the classes `MPIMultiOStream` and `MPIMultiIStream` use the more complex `MPIMultiBuffer`. It is possible to interchange the stream types. For example, our system accepts sending a message with `MPIOstream` while receiving it in another processor using a `MPIMultiIStream`. The only restriction is that they use the same buffer length so the MPI messages at the lower level do not get corrupted.

We also provide `operator<<` and `operator>>` procedures that use `MPICommonOStream` and `MPICommonIStream` and take as a parameter primitive data types.

8.3 Using Our Library to Exchange Messages

Figure 8.2 shows a sample use of the library that sends a vector of doubles from processor 0 to processor 1:

- line 1: the file `MPI.H` contains all our class declarations.
- lines 2-3: we define the vector and buffer sizes.
- line 7: `MPICommunicator::init` is a static method that initializes the library.
- line 10: this example uses the default communicator that includes the two processors.
- line 11: a barrier synchronizes the processors.
- line 13: `rank` returns the processor number. Processor 0 executes the true part of the `if`, while processor 1 executes the `else`.
- line 15: processor 0 creates a port to processor 1.
- line 17: using the port, processor 0 creates an output stream that uses one buffer with blocking communication.
- lines 18-20: processor 0 sends the vector to processor 1 by invoking the stream `operator<<` in each of the elements of the vector. The vector size is 8192 bytes while the buffer size is 256 bytes. Therefore, sending the vector requires 32 MPI message sends inside the library.
- line 21: a `flush` sends any remaining data in the stream to the receiving processor.
- line 23: processor 1 creates a port to processor 0.

```
#include "MPI.H"
const long vecsize = 1024;
const long bufsize = 256;

int main (int argc, char **argv)
{
   MPICommunicator::init(&argc, &argv);
   long i;
   double *myvector = new double[vecsize];
   MPICommunicator comm;
   comm.barrier();

   if (comm.rank() == 0) {
      fillvector(myvector, vecsize);
      MPIPort port(1);
      // sending the vector
      MPIOStream ostr(port, bufsize);
      for (i = 0; i < vecsize; i++) {
        ostr << myvector[i];
      }
      ostr.flush();
   } else {
      MPIPort port(0);
      // receiving the vector
      MPIMultiIStream istr(port, 0, bufsize, 0, 3);
      for (i = 0; i < vecsize; i++) {
        istr >> myvector[i];
      }
   }
   MPICommunicator::finalize();
   delete [] myvector;
   return 0;
}
```

Figure 8.2: A simple example that communicates a vector of doubles between two processors using our library.

- line 25: using the port, processor 1 creates an input stream that uses three buffers with non-blocking communication.
- line 26-28: processor 1 receives the vector using the stream.
- line 30: we terminate MPI with the static method `MPICommunicator::finalize`.

The loops in lines 18-20 and 26-28 are inefficient because they require a function call to `operator<<` and `operator>>` for every element in the vector. To improve the performance we can replace these loops by the lines `ostr.vecwrite(myvector, vecsize);` and `istr.vecread(myvector, vecsize);` respectively to send and receive vectors of doubles or other data types.

As it can be observed in the previous example our library does not require the programmer to directly manage the allocation and deallocation of buffers, one of the most difficult and error prone tasks in message passing codes. Also, we do not require that the receiving processor knows in advance the size of the vector. For example, we can modify the program to send the size of the vector before sending its contents. The receiving processor can use this size to allocate a vector of the appropriate size. We can also modify the buffer size to allow optimal performance.

The process for communicating user defined objects is similar to the one used to read and write objects in C++. For every class that we want to communicate we define the functions code `operator<<` and `operator>>`. This procedure is shown in Figure 8.3.

We can now use the `operator<<` and `operator>>` on input and output streams as in the previous example to exchange objects of type `MPITestClass`. A similar process can be used to exchange more complex data types such as lists, trees or graphs.

8.4 Performance Analysis

8.4.1 Introduction

In this section we evaluate the performance of our library in two different architectures: a network of SUN workstations and an IBM-SP.

The network of workstations (NOW) is located in the Center for Information and Technology at Brown University and consists of approximately 200 Sun Ultra-10 workstations that are connected through a 100 Mb/sec. Fast Ethernet network. Each workstation contains a 440 Mhz. UltraSPARC processor and 256 MB of local memory and runs Sun's Solaris

```
class MPITestClass {
   friend MPICommonOStream &operator<<(MPICommonOStream& os,
                                       const MPITestClass& t);
   friend MPICommonIStream &operator>>(MPICommonIStream& is,
                                       MPITestClass& t);
private:
   long index_;
   char buf_[256];
   long n_;
   char ch_;
   double f_;
public:
   MPITestClass() {...}
};

MPICommonOStream &operator<<(MPICommonOStream& os, const MPITestClass& t)
{
  os << t.index_ << t.buf_ << t.n_ << t.ch_ << t.f_;
  return os;
}

MPICommonIStream &operator>>(MPICommonIStream& is, MPITestClass& t)
{
  is >> t.index_ >> t.buf_ >> t.n_ >> t.ch_ >> t.f_;
  return is;
}
```

Figure 8.3: To communicate user defined objects the user must provide an implementation to the input and output stream operators.

2.7 operating system. The latency required to send a message between two machines depends on the relative distance between these machines. On workstations connected to the same subnetwork we measured the latency to be 158 μsecs. while in machines connected to different subnetworks the latency is 260 μsecs.

We tested the library in the NOW using two MPI implementations. MPICH [45] (version 1.1.2) is a version of MPI developed by the Argonne National Laboratories and Mississippi State University. This implementation executes on a wide variety of architectures and in our network of workstations uses a lower level library called P4 [14]. Another widely used MPI library is LAM [24] that was originally developed by the the Ohio State University and it is now maintained by the University of Notre Dame.

The IBM RS/6000 SP contains 24 Model F50 (Silver) nodes and is located at the Technology Center for Advanced Scientific Computation and Visualization at Brown University. Each computing node consists of four 332 Mhz. 604e Power-PC processors that share 1 GB. of main memory and running IBM AIX 4.3 operating system. The nodes are connected through an SP switch with an MX adapter that has a hardware limit of 150 MB/sec. Sending a message using IBM's implementation of MPI between two nodes has a latency of 30 μsecs.

All the MPI implementations are optimized to exchange vectors of contiguous data. In the tests discussed in the following subsections we exchanged vectors of 1 to 256K double precision floating point numbers between two or more processors. We use this simple data type to compare the performance that we achieve using our library against the optimal performance that can be obtained in the system.

8.4.2 Point-to-Point Communication

Figure 8.4 shows the bandwidth in Mb/sec. by sending vectors of sizes ranging between 1 to 256K doubles from a source processor to a destination processor in the NOW (using MPICH and LAM) and the IBM SP. In the NOW, both processors are located in the same subnetwork while in the IBM SP the processors are located in different nodes so actual network communication (rather than shared memory) takes place.

The *Block* data series corresponds to the MPI blocking function calls (`MPI_Send` and `MPI_Recv`) while the *No Block* series use the non-blocking primitives of MPI (`MPI_ISend` and `MPI_IRecv`). Although the non-blocking calls allow to overlap computation and communication, in these tests there is no computation and both methods achieve similar bandwidth.

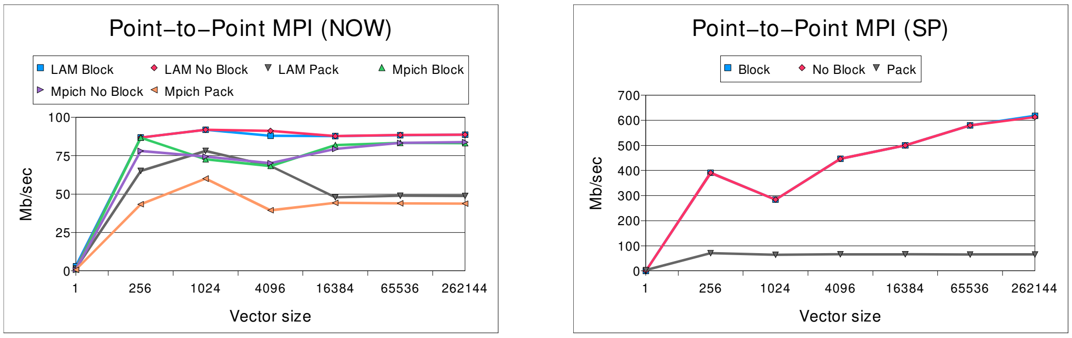


Figure 8.4: Point-to-Point MPI Communication. Bandwidth obtained by sending vectors of 1 to 256K doubles using MPI's blocking, non-blocking and packing function calls in a NOW and IBM SP.

In the NOW, the bandwidth increases rapidly with vector size but it stabilizes in vectors of 256 doubles at around 88 Mb/sec. using LAM and around 83 Mb/sec. with MPICH. In the IBM SP the performance also increases with element size but there is a slight decrease in vectors of 1024 doubles because the SP uses two methods for exchanging messages. Small messages are immediately delivered while long messages are sent only when the receiving processor is ready to receive.

The final data series show the result of *packing* the vectors using MPI. As we have explained earlier, in PARED the data to send to other processors is almost never contiguous. Therefore it is necessary to either send many messages of very small size (with the corresponding communication cost that results from very high message latencies) or combine the data into one contiguous message. This process in MPI is called *packing* and, in these examples, involves looping through all the elements in the vector calling the MPI functions `MPI_Pack` and `MPI_Unpack` for every element in the array in the sending and receiving processor respectively. This process creates a buffer that can then be sent and received using the same blocking and non-blocking function calls described before. Looping through all the elements of the vector and calling the `MPI_Pack` and `MPI_Unpack` functions is a very expensive process. As we can see in these figures, the bandwidth drops to less than 50 Mb/sec. in the NOW and to 66 Mb/sec. in the SP.

Figure 8.5 shows the result of using our `MPIOStream` and `MPIIStream` classes with buffer sizes of 4K, 16K, and 64K in the NOW (with LAM and MPICH) and in the IBM SP. These classes use blocking MPI function calls at the lower level to exchange messages. The top two figures show the bandwidth that we achieve by looping through all the elements in the vector and calling the corresponding `operator<<` and `operator>>` for each double in the array. The total number of messages depends on the vector and buffer size, but for large vectors (256K doubles), the total memory requirements are much smaller (4K to 64K bytes) than the one required in the previous MPI examples (256K*8 bytes). In the NOW, again we rapidly reach the communication limit determined by the speed of the Ethernet network. In this example, LAM also has an advantage over MPICH. Using LAM all the buffer sizes result in similar bandwidth. This is not the case with MPI where the buffers of size for 4K have the best performance while buffers of 16K bytes have the worst. In the SP, the bandwidth is between 175 Mb/sec. (for 16K byte buffers) to 215-240 Mb/sec. (for 4K byte buffers). This is between 30% to 50% the raw MPI performance in the SP observed in the previous figure but is 3 to 4 times larger than the one that we would have obtained if we have used MPI to pack the messages.

The two bottom figures show the performance obtained by calling the `vecwrite` and `vecread` methods of the stream classes, rather than calling stream input and output operators for each element of the vector. The buffer size also varies between 4K bytes and 64K bytes, so every vector read and write is also decomposed into several MPI messages. In the NOW there is no significant advantage by calling the vector read and writes because the whole communication is limited by the network speed and not by processor speed. On the other hand, in the SP the vector operators double the bandwidth.

Figure 8.6 shows the results for `MPIMultiOStream` and `MPIMultiIStream`, where each stream contains 2 buffers of 4K 16K and 64K bytes and uses non-blocking MPI primitives. The results in these figures are very similar to the ones shown in Figure 8.5. Nevertheless, the `MPIMultiOStream` and `MPIMultiIStream` allow the overlap of computation and communication.

8.4.3 Ping-Pong Communication

Another common test of point-to-point communication between two processors is the Ping-Pong test where a source processor first sends and then receives a vector of doubles, while the other processor first receives the vector and then sends it back. These results are shown in Figures 8.7, 8.8 and 8.9 operators for the raw MPI function calls, stream and multistream methods respectively.

As expected, these tests confirm the results presented in the the previous section. The bandwidth in all cases increases with vector size, although more slowly than in the previous

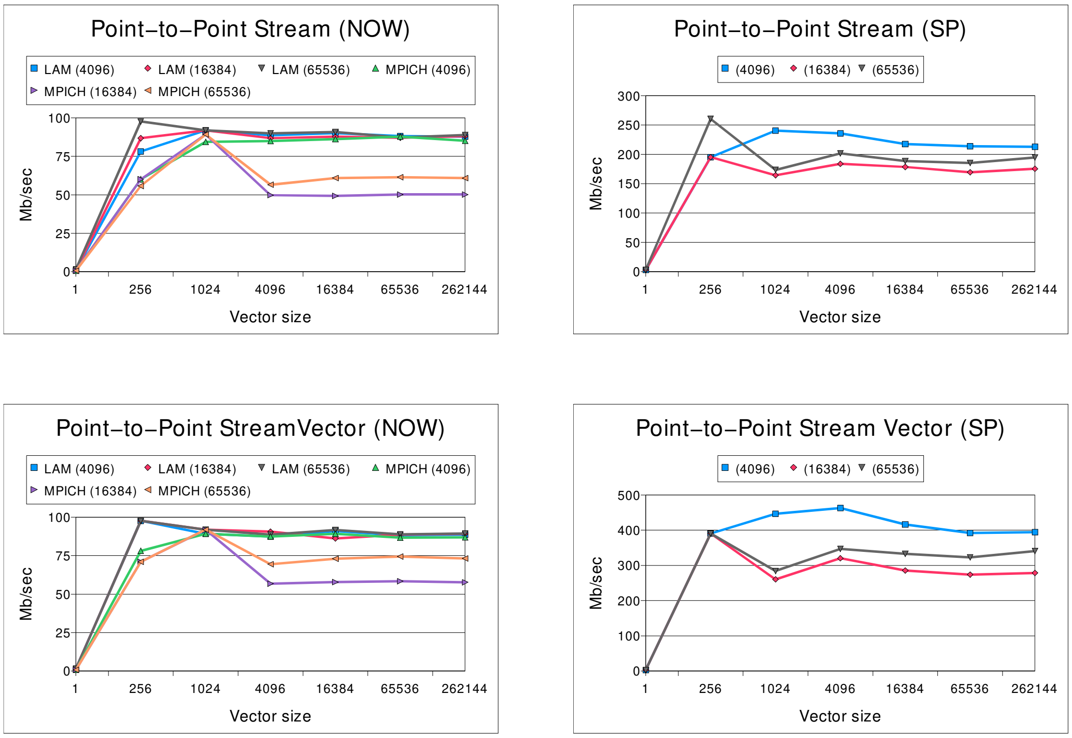


Figure 8.5: Point-to-Point Stream Communication. Bandwidth obtained by sending vectors of 1 to 256K doubles using the `MPIOStream` and `MPIIStream` with buffers of 4K, 16K and 64K bytes in a NOW and IBM SP.

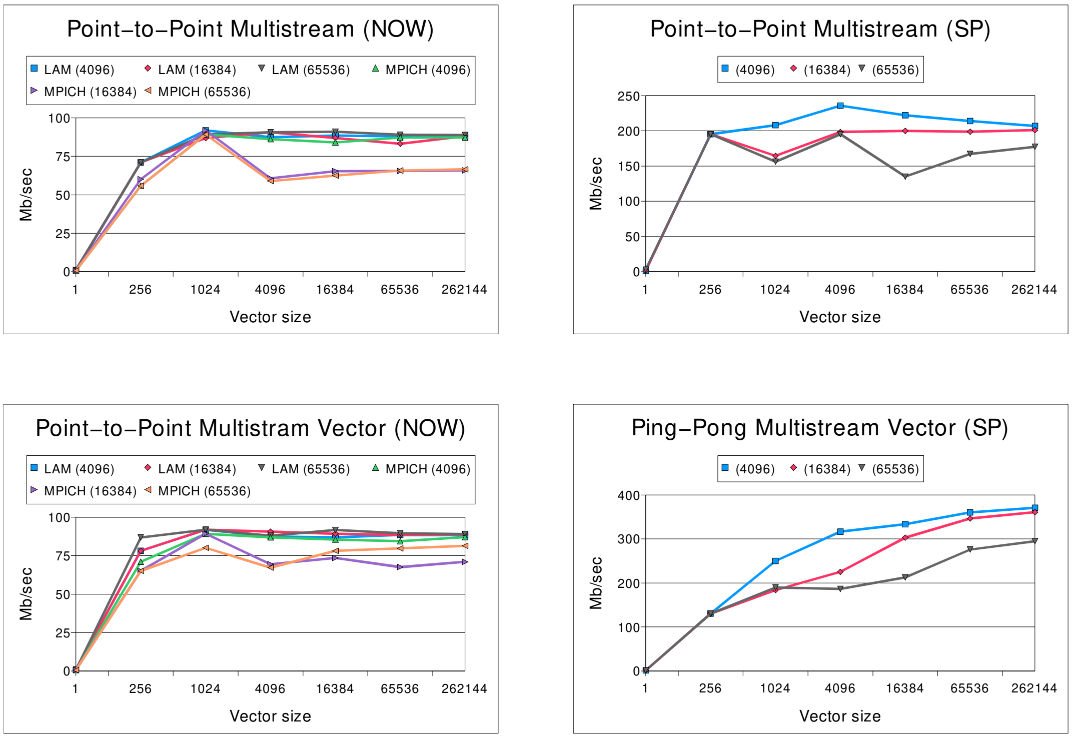


Figure 8.6: Point-to-Point MultiStream Communication. Bandwidth obtained by sending vectors of 1 to 256K doubles with the classes `MPIMultiOStream` and `MPIMultiIStream` with buffers of 4K, 16K and 64K bytes in a NOW and IBM SP.

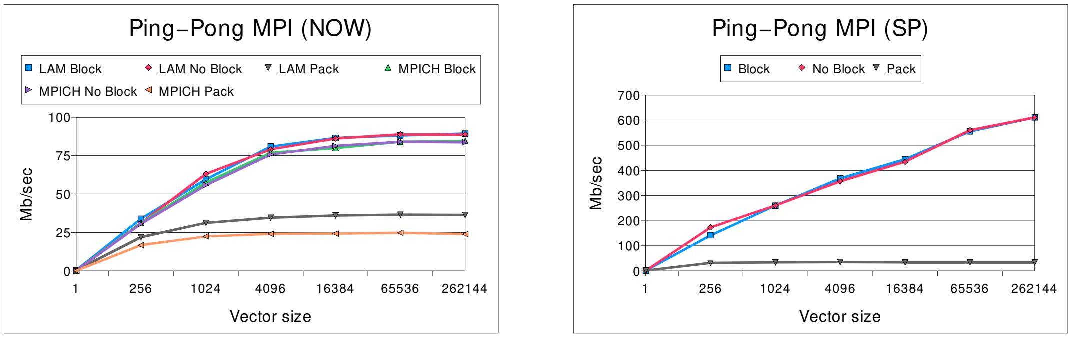


Figure 8.7: Ping-Pong MPI Communication. Bandwidth obtained by sending vectors of 1 to 256K doubles using MPI's blocking, non-blocking and packing function calls in a NOW and IBM SP.

examples, and requires vectors of 16K doubles to achieve maximum bandwidth in the NOW and reach the network limit. The bandwidth using MPI packing is slightly lower: 36 Mb/sec. in the NOW and 23 Mb/sec. in the SP.

Although in the point-to-point examples we synchronize the processors in a barrier before beginning each test, the ping-pong test explicitly synchronizes the two processors at every exchange and smoothes some of the variations that we have observed before. In these tests, the advantage of using smaller buffers of approximately 4K bytes in the SP is clearer for both stream and multistream communication.

8.4.4 All-to-All Communication

The most demanding communication pattern in parallel FFTs and parallel Spectral Navier-Stokes solvers is the all-to-all communication where each processor sends a vector to the remaining processors and also receives a vector from each other processor. This communication pattern easily swamps a network, specially on the NOW where all the workstations share a common wire.

MPI supplies function calls for all-to-all (`MPI_Alltoall`) communication that require a very large amount of memory. In the tests shown in this section, we evaluate a complete exchange using the point-to-point communications using the following procedure: each processor iterates through i phases, from 1 to $P-1$ where P is the total number of processors. In each phase i, each processor p exchanges a vector with processor p `XOR` i. In this way, we avoid potential deadlocks in the IBM SP and each processor does not maintain many

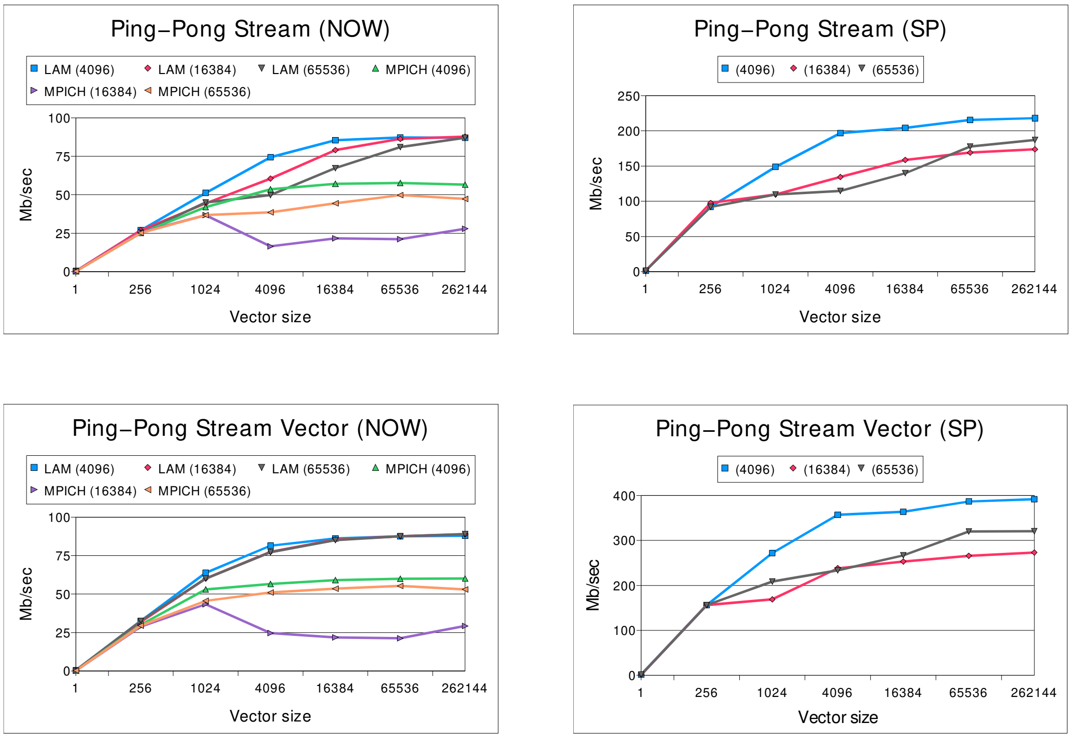


Figure 8.8: Ping-Pong Stream Communication. Bandwidth obtained by sending vectors of 1 to 256K doubles using the `MPIOStream` and `MPIIStream` with buffers of 4K, 16K and 64K bytes in a NOW and IBM SP.

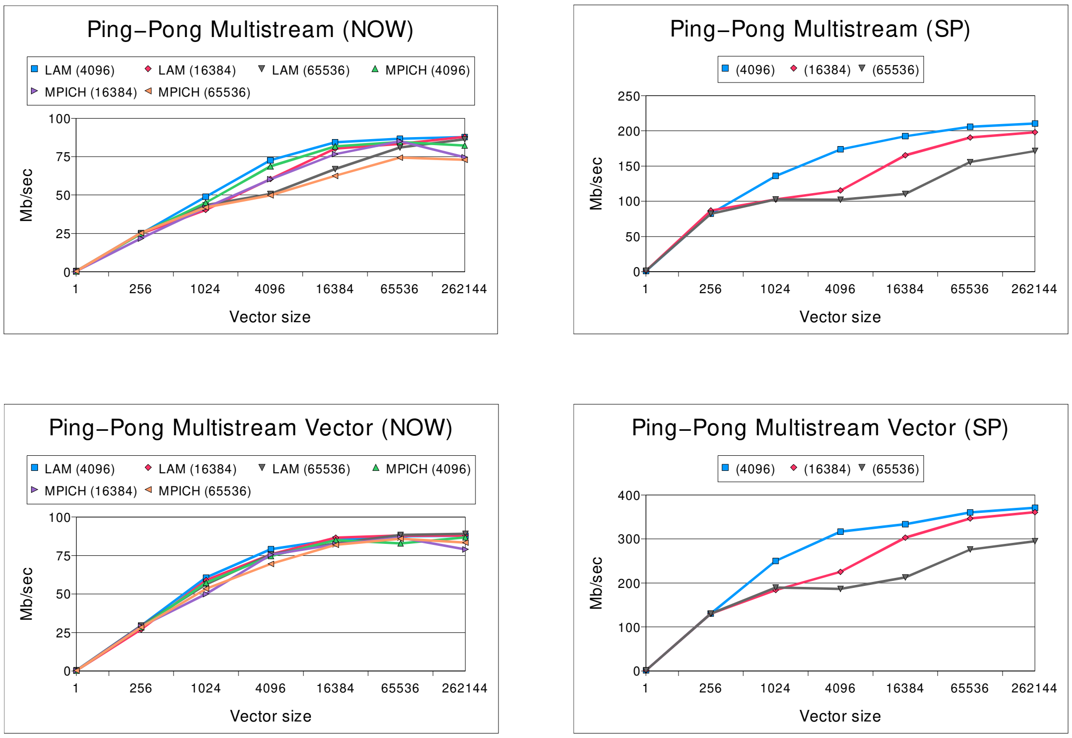


Figure 8.9: Ping-Pong MultiStream Communication. Bandwidth obtained by sending vectors of 1 to 256K doubles with the `MPIMultiOStream` and `MPIMultiIStream` classes with buffers of 4K, 16K and 64K bytes in a NOW and IBM SP.

pending receives as would be the case in other all-to-all methods.

The results for this test are shown in Figure 8.10 for NOW (4 to 16 processors) and the IBM SP (4 to 32 processors) for the MPI libraries and for our stream libraries. In the NOW, the blocking MPI has a per processor bandwidth of a little bit more than 40 Mb/sec. while the packing primitives have a maximum bandwidth of less than 20 Mb/sec, but there is no significant performance penalty for increasing to a relatively small number of a processors. Our stream libraries use LAM MPI in the NOW at a lower level and slowly increase until reaching 40 Mb/sec. for large vectors. In this case, there is a small penalty to pay for 16 processors. Trying to run this test for 32 or more processors in the NOW was extremely difficult and we usually obtain a less than 10Mb/sec.

The blocking MPI bandwidth in the SP slowly decreases as we increase the number of processors, from 307 Mb/sec. for 4 processors to 122 Mb/sec. for 32 processors in the case of vectors of 256K doubles. The packing MPI calls have a bandwidth of 17 Mb/sec. Our libraries loop through all the elements in the vector calling the respective `operator<<` and `operator>>` and achieve slightly more than 100 Mb/sec. for 4 processors and 87 Mb/sec. for 32 processors. In this case, there is a slight advantage on using the multistream version of our streams.

8.4.5 Ring Communication

Fortunately, PARED requires all-to-all communication only when the mesh is randomly distributed between the processors because in this case each processor is adjacent to every other processor.

A more common situation is that processors exchange messages with a relatively small number of adjacent processors but with several processor sending messages at the same time. We evaluate this situation using a ring test where every processor p first sends a vector to processor $p+1$ and then receives a vector from $p-1$. The first and last processors also exchange messages to complete the ring. The results for this test for the NOW and the SP are shown in Figure 8.11 and are comparable to the ones in the previous section. The number of messages exchanged in each phase for the all-to-all communication and for the ring communication is the same $(p-1)$. The only difference is that the ring test executes only two phases while the all-to-all test requires $p-1$ communication phases.

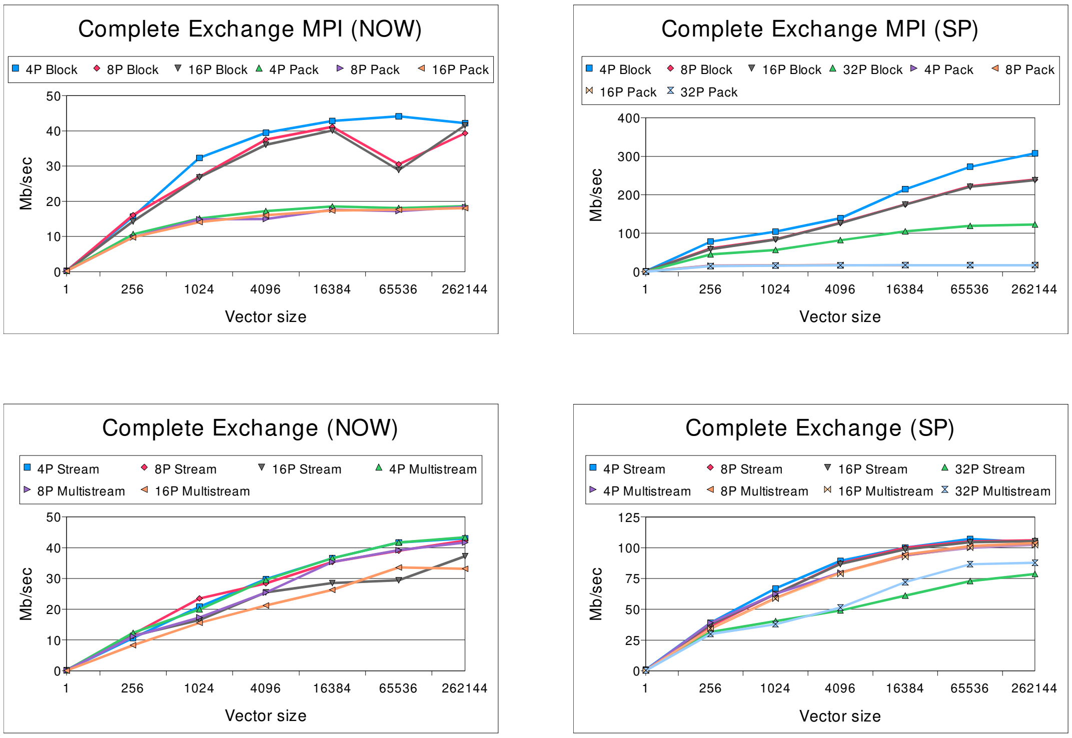


Figure 8.10: All-to-All communication. Bandwidth obtained by exchanging between all the processors in a NOW and IBM SP for 4 to 32 processors sending and receiving of 1 to 256K doubles to and from all the other processors in a NOW and IBM SP.

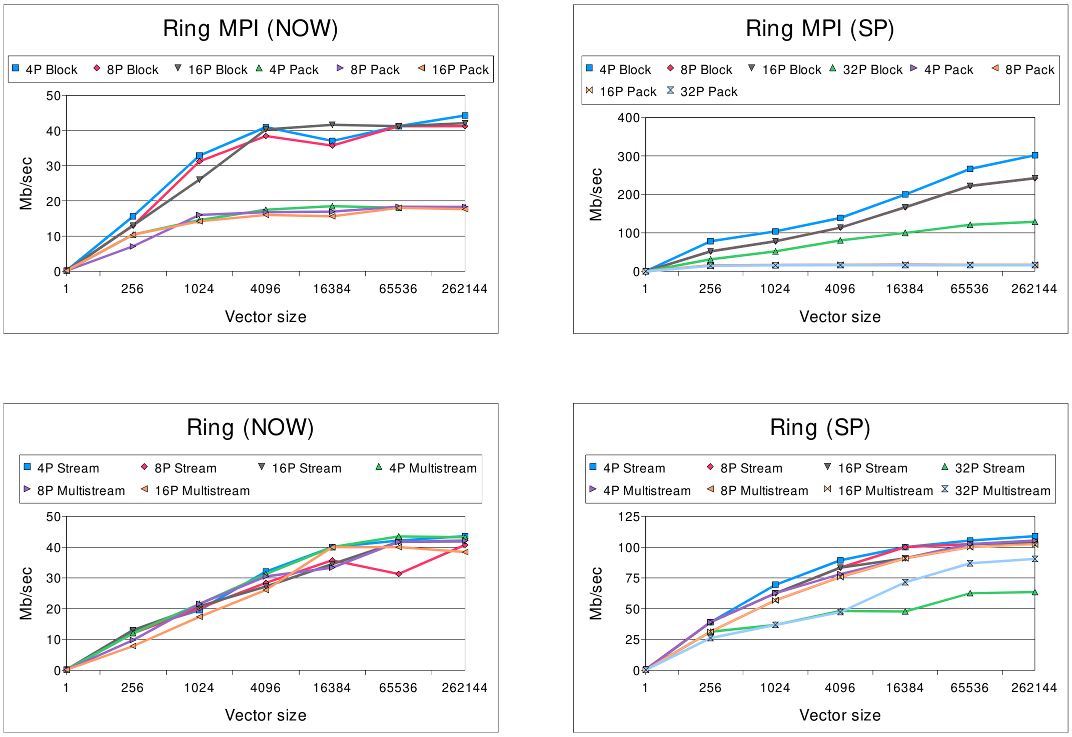


Figure 8.11: Ring communication. Bandwidth obtained by first sending and receiving of 1 to 256K doubles to the next and previous processor respectively in a NOW and IBM SP.

Chapter 9

Experimental Results

9.1 Putting it All Together

In this chapter we study two different problems using PARED. The first experiment shows the adaptive solution of a Poisson equation using two- and three-dimensional meshes. This test demonstrates that our system can rapidly generate and manipulate very large locally adapted meshes that contain more than 6,000,000 elements. In this static problem, the total time is dominated by the migration time because there is a relatively large change of the mesh in each adaptation phase. This example emphasizes the importance of reducing the migration cost.

Later we use PARED to simulate an unsteady flow past a cylinder. This is a very important scientific problem and involves the solution of nonlinear Navier-Stokes partial differential equations. In this transient example, the mesh is refined and coarsened to follow the vortices that appear behind the cylinder.

Both examples in this chapter adapt the mesh according to local error estimates using the longest edge bisection algorithm described in Chapter 4, repartition the mesh using the PNR algorithm (Chapter 5), and migrate the mesh to rebalance the work load (Chapter 6).

9.2 A Laplace Example

A good static test for our framework is the two-dimensional problem defined by Laplace's equation $\Delta u = 0$ in the square $\Omega = (-1, 1)^2$ with the following Dirichlet boundary condition:

$$g(x, y) = \cos(2\pi(x - y)) \frac{\sinh(2\pi(x + y + 2))}{\sinh(8\pi)}$$

This is the same problem used to compare repartitioning algorithms in Chapter 5.5. As we mentioned earlier, the analytical solution to this problem is known to be $u(x, y) = g(x, y)$ at every point of the domain Ω. This solution is smooth but changes rapidly close to the corner $(1, 1)$.

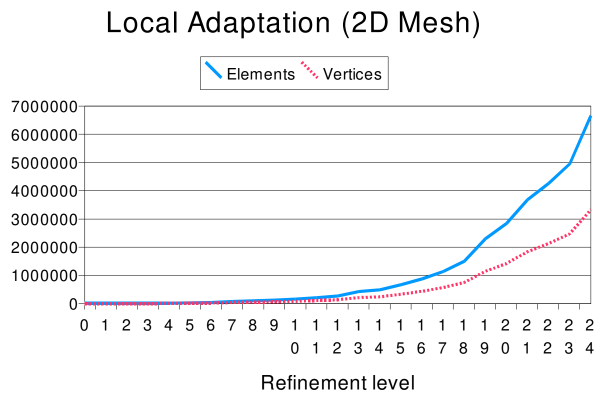
To solve this problem adaptively we generated an unstructured initial mesh with 6394 vertices and 12498 triangles of similar size using our own Delaunay mesh generator. We defined a similar problem in 3D, starting from an unstructured mesh that contains 2013 vertices and 9540 tetrahedra. We first loaded these initial meshes into the coordinator. This processor computed an initial partition of the mesh using the PNR heuristic (which is outlined in Chapter 5) and distributed the meshes between p processors, $4 \leq p \leq 64$, according to this partition.

We then performed a sequence of local adaptations of the mesh. In each of these t mesh adaptation levels or phases, we first computed a solution using the current mesh. Based on error estimates, we selected a new set of elements that we refined by their longest edge, as explained in Chapter 4. In this example of a static problem, only refinement was used. Finally, in each level the mesh is partitioned and migrated using PNR.

Because the analytical solution of these problems is known, it was possible to select the elements to refine using the L_∞ norm between the computed solution $\hat{u}^t$ at level t and the real solution u. At every level t we defined the error $e_j = |u_j - \hat{u}_j^t|$ of every node j where the maximum error $e_{max} = \max(e_j)$ is the maximum error of all the nodes. We then refined all the elements that contained a node k such that $e_k \geq \alpha e_{max}$ with $\alpha = 0.5$.

The FEM solutions are valid at every point of the domain. Therefore, the computed solution u^{t-1} can be used as a good initial guess to compute a new solution u^t. At every time step these tests required the solution a linear system of equations $Ku^t = f$. Rather than solving for u^t at every time step, we defined $u^t = u^{t-1} + \Delta u$ and then we solved $K\Delta u = f - Ku^{t-1}$ using a Conjugate Gradient solver with a Jacobi preconditioner.

Figure 9.1 shows the number of elements and vertices in M^t as a function of successive local adaptations for the 2D and 3D problems. After 24 refinement levels the 2D mesh contained 6,658,350 elements and 3,329,802 vertices and after 36 refinement levels the 3D mesh contained 6,303,346 elements and 1,164,782 vertices. These meshes are 532 and 660 times larger than the initial meshes respectively. The ratio of the area of the largest triangle to



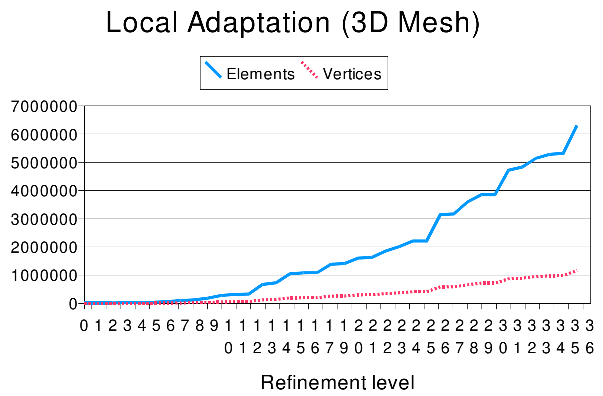


Figure 9.1: Number of elements and vertices that result from successive local refinements of irregular two- and three-dimensional meshes.

the smallest triangle in the original 2D mesh was 6.88. This same ratio was 22,028,735.25 in the most refined 2D mesh. For 3D meshes, the ratio of the volume of the largest tetrahedron to the smallest one was 13.04 and 17,930,689.01 for the initial and final meshes respectively. The adaptation of the mesh reduces the maximum error (L_∞ norm) from 0.001922 in the initial 2D mesh to 4.710^{-6} in the finer meshes. On the 3D problem, the error is reduced from 0.0756 in the initial mesh to 0.000529 in the locally refined meshes.

9.2.1 Experiments on an IBM SP

We compare the times spent by PARED in each of its four phases of partition, migration, solution and refinement on the 2D and 3D versions of Laplace's equation described above on an IBM SP [3, 82] parallel computer containing four to 64 processors. These times for each adaptation levels are presented in Figures 9.2 and 9.3 for the 2D and 3D problems, respectively.

Computing a solution for Δu rather than for $\hat{u}^t$ at every time step significantly reduced the solution time. In earlier tests [21], obtaining a solution for u^t required more than 2,300 CG iterations for large 2D meshes. A solution for Δu required fewer than 160 CG iterations to reduce the residual by 10^-6 in the 2D examples. The only drawback of this strategy is that the previous solutions u^{t-1} must be migrated between processors along the elements and vertices. The solution Δu on large 3D meshes converged in as little as 20 CG iterations compared to the 100 iterations reported in our earlier work.

In these problems with small regions of high gradients most of the refined elements are located on one or a few processors. Most of the refinement time is spent refining elements that are local to a processor; there is very little communication overhead. Therefore, these results confirm the ones obtained in Chapter 4.6. The refinement time does not necessarily increase with increasing refinement levels. Also, it is slightly more expensive to refine 3D meshes than 2D ones for meshes of similar sizes. For example, to create 655,436 new triangles at level 22 required 4.88 sec. on 64 processors. To create 294,899 new tetrahedra at level 17 requires 5.45 sec. on 64 processors.

The repartitioning time remains almost constant in both problems as the number of elements increases because a partition is computed from the small weighted graph G obtained from the initial mesh. The partition time slowly increases with the number of processors and varies between 0.45 sec. and 1.99 sec.

As explained in Chapter 6 the use of standard partitioning algorithms to repartition G generally causes half of the elements to move to a new processor. In that chapter we have also derived a lower bound for problems that perform very localized refinement in one of the corners of the domain. This lower bound depended in the number of new elements created and on the number of processors. With only 4 processors, the migration cost on the partitions derived with PNR is small (in 2D meshes) or similar (in 3D meshes) to the solution time but, as expected, it becomes increasingly important for a large number of processors. These synthetic problems are not a good test for the performance of PNR

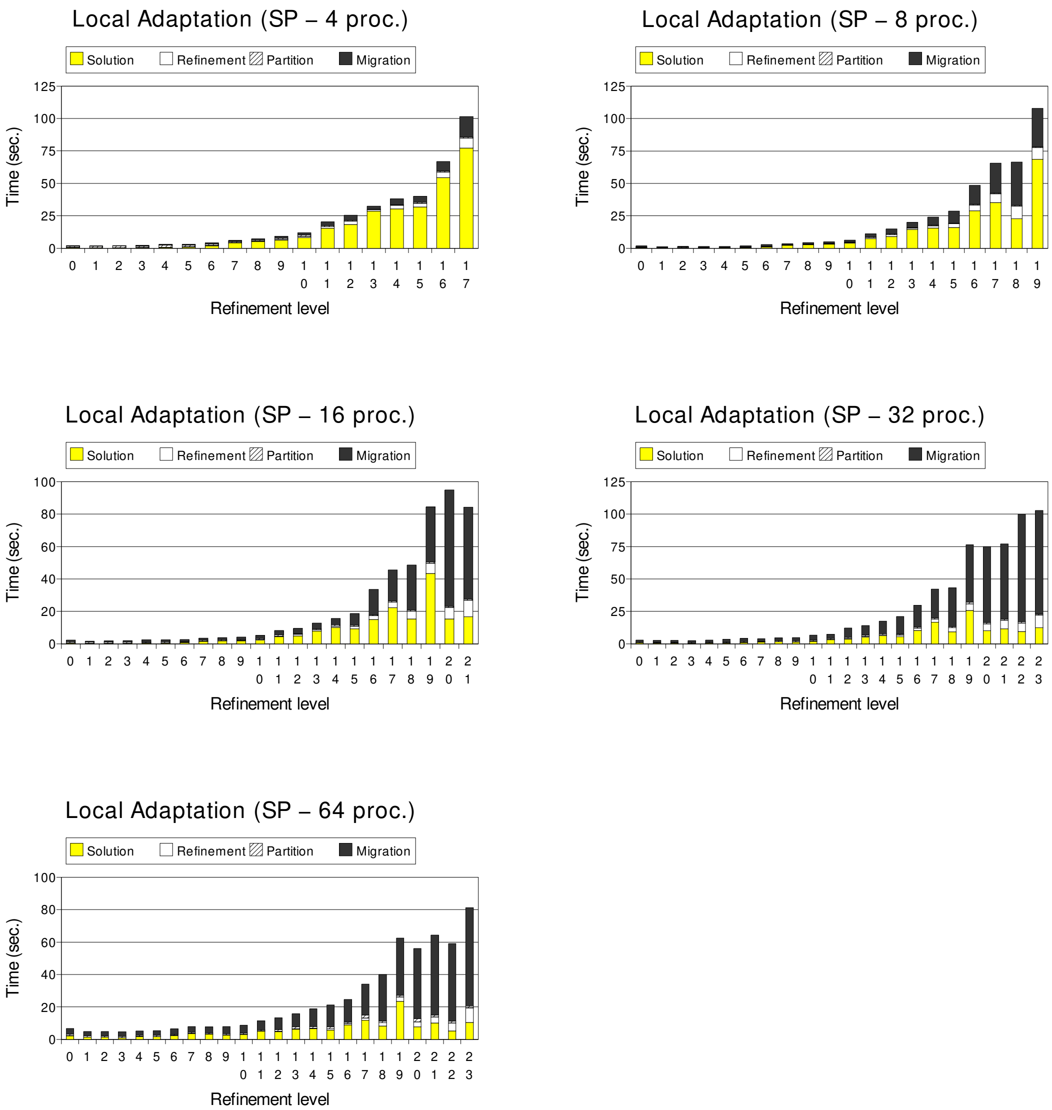


Figure 9.2: Times for each refinement phases of the locally adapted two-dimensional problem on 4, 8, 16, 32 and 64 processors of an IBM SP parallel computer.

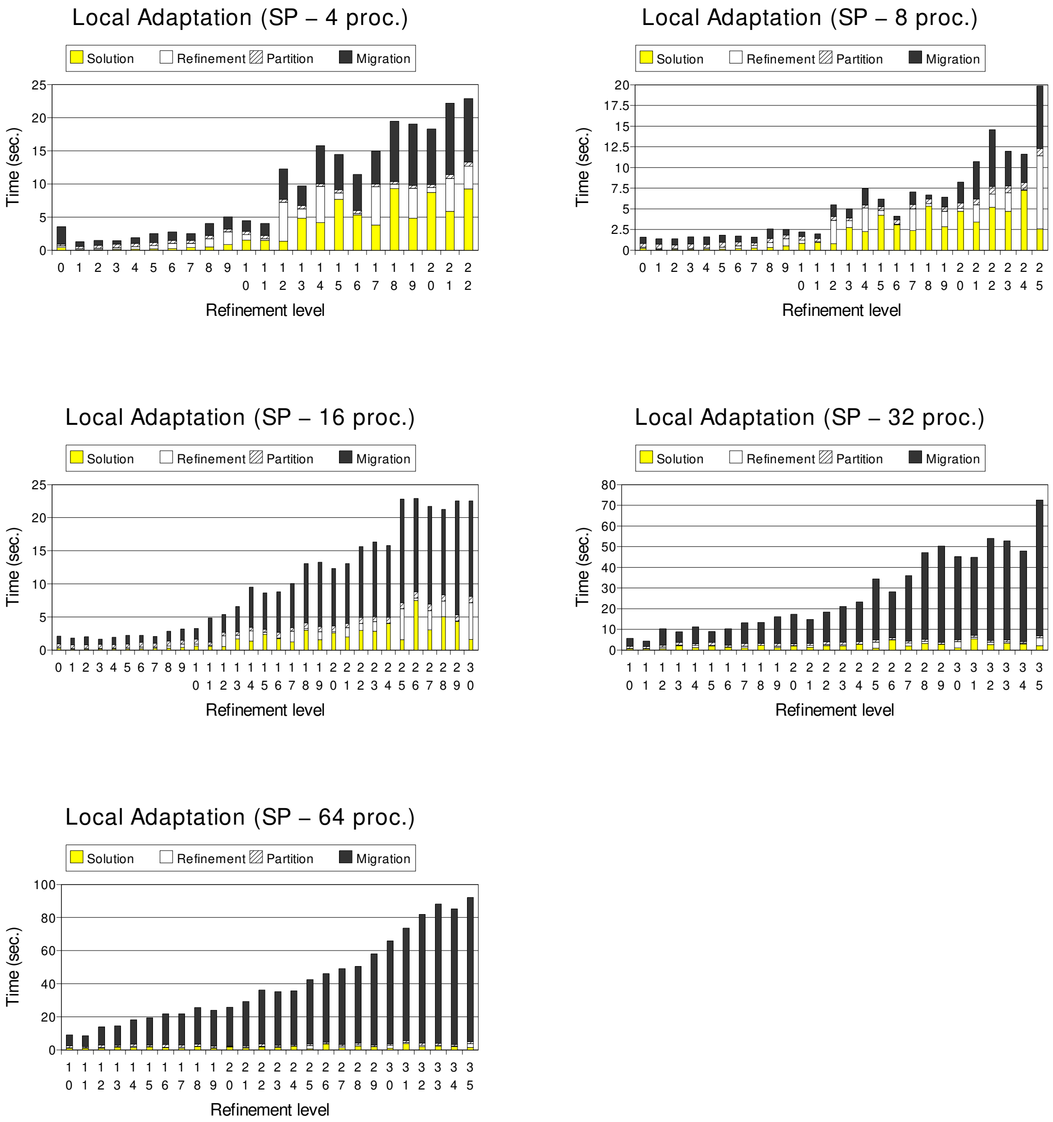


Figure 9.3: Times for each refinement phases of the locally adapted three-dimensional problem on 4, 8, 16, 32 and 64 processors of an IBM SP parallel computer.

because our goal was to generate large locally adapted meshes in a relatively short period of time and show that our system can manipulate these meshes. In these tests at every time step we create a large number of new elements relative to the size of the mesh and we migrate approximately 20 % of the total number of elements.

9.2.2 Experiments on Sun Workstations

We conducted the same experiments using the LAM [24] communications library on a network of four to 64 Sun Ultra-10 workstations, each having 256MB of memory and connected via a 100Mb/s Ethernet network. Although, the network of workstations (NOW) is not a controlled environment and does not have the benefit of a fast switch, both of which are characteristic of the SP, the performance achieved on our test problems is not very different from that obtained on the SP.

The NOW has a higher latency which mainly affects smaller messages, such as the global sums for the Conjugate Gradient. For that reason it is more difficult to obtain speedups in the solution time in the NOW than on the SP. Also, on the NOW there is a larger potential for network congestion because all the processors communicate through the Ethernet.

Figures 9.4 and 9.5 shows the solution and total times for the 64-processor Sun NOW. These results apply to the two- and three-dimensional problem described in Chapter 9.2 respectively. The relative solution time for small problems on the NOW is much larger than it is for large problems. This is due to the higher latency of the NOW. On the other hand, we have not observed a significant increase of the migration time on large meshes, which will indicate a network saturation.

9.3 A Navier-Stokes Problem

9.3.1 Problem Formulation

The motion of incompressible Newtonian fluid with constant density ρ and constant viscosity μ is governed by the Navier-Stokes equations

$$\begin{aligned} \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} &= -\frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u} + g \qquad \text{in}\,\Omega \\ \nabla \cdot \mathbf{u} &= 0 \qquad \text{in}\,\Omega \end{aligned} \tag{9.1}$$

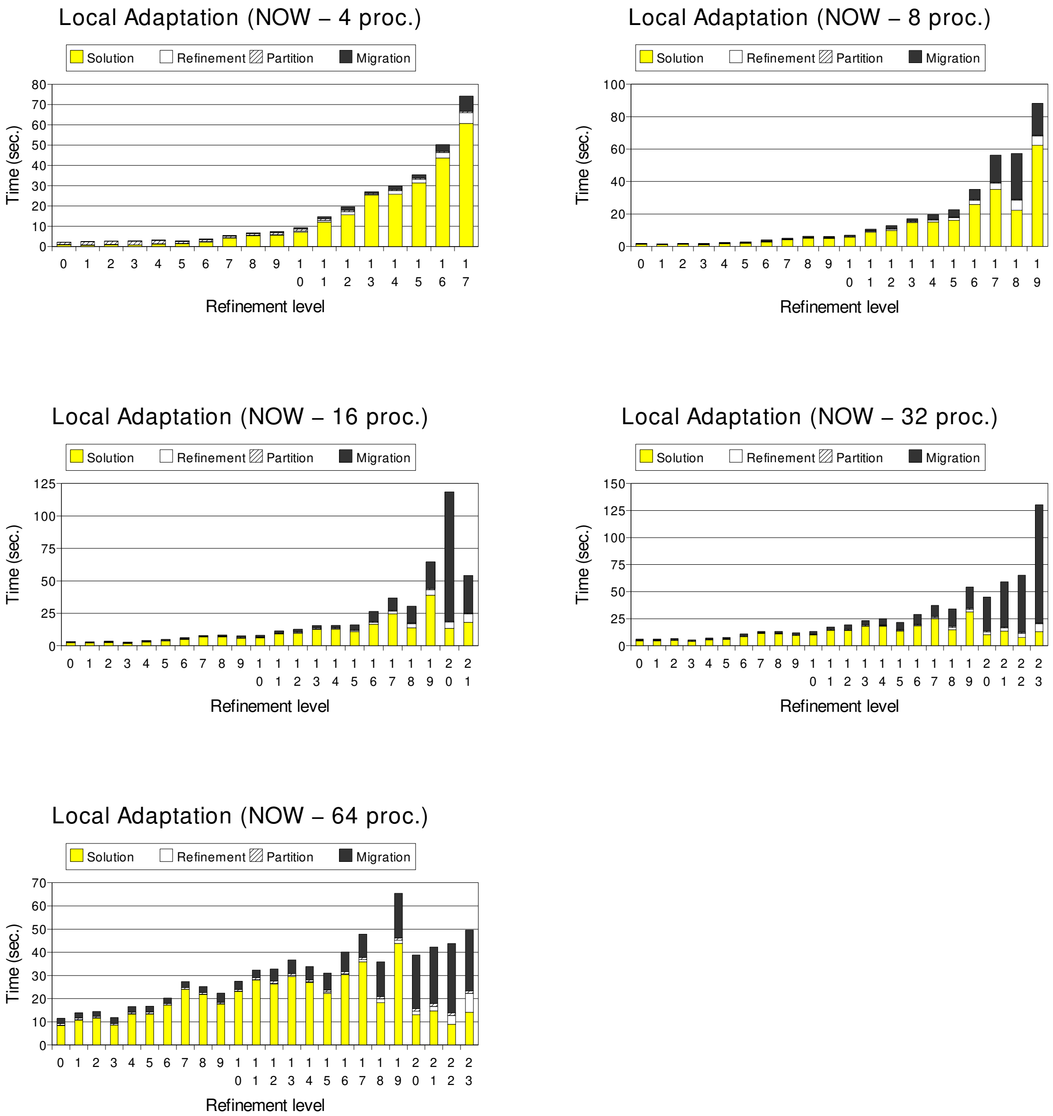


Figure 9.4: Times for each refinement phases of the locally adapted two-dimensional problem on 4, 8, 16, 32 and 64 processors of a network of workstations.

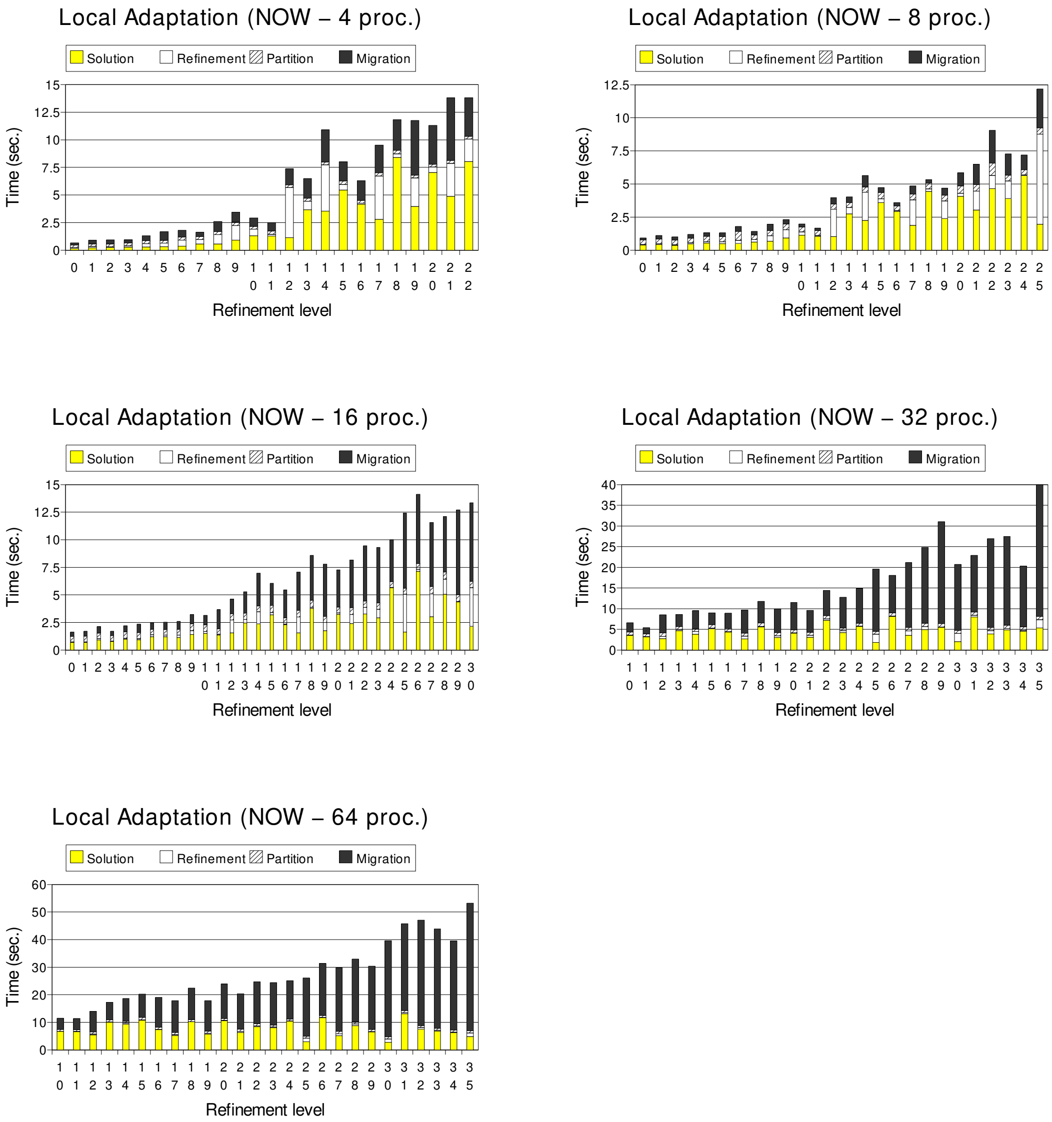


Figure 9.5: Times for each refinement phases of the locally adapted three-dimensional problem on 4, 8, 16, 32 and 64 processors of a network of workstations.

where $\mathbf{u} = (u, v, w)^T$ is the velocity vector, p is the pressure, and ν is the kinematic viscosity $\nu = \mu/\rho$. For example, at 15°C the air has a kinematic viscosity of $0.15\,\mathrm{cms}^{-1}$ while water has $\nu = 0.01\,\mathrm{cms}^{-1}$. $\nu\nabla^2\mathbf{u}$ is a viscous term where ∇^2 is the Laplace operator $\partial^2/\partial x^2 + \partial^2/\partial y^2 + \partial^2/\partial z^2$. The derivation of these equations from first principles is shown in [2, 96].

The Reynolds number

$$\mathrm{Re} = \frac{UL}{\nu}$$

is a parameter of the typical flow speed U and the characteristic scale of the flow L. Fluid flows with high and low Reynolds numbers show different behavior. Steady flows at high Reynolds numbers are unstable and become turbulent, a phenomenon that is not observed at low Reynolds numbers.

In the example shown in this section we study the formation of vortices past a cylinder of unit diameter at high Reynolds numbers. The fluid is initially at rest and is suddenly accelerated with speed U perpendicular to its axis. During a very short initial phase the flow is irrotational after which two eddies appear at the back of the cylinder. These eddies first grow in size and later became asymmetric. In later stages the flow becomes unsteady shedding vortices alternatively from the two sides of the cylinder in a phenomena that is known as the Von Karman vortex street [92].

9.3.2 Solution Strategy

The method used in this example is based on the formulation presented by Patera and Fischer in [36] which is based on the fractional step method of Orszag and Kells [65]. This method approximates the non-linear Navier-Stokes problem by iteratively computing solutions to simpler Helmholtz and Poisson equations.

The solution of the transient system given at Equation 9.1 is obtained by a semi-implicit scheme using a temporal discretization, where the solution at every time step t consists of three computational steps. Based on the previously computed solution $\mathbf{u}^{t-1}$, $\mathbf{u}^{t-2}$ and $\mathbf{u}^{t-3}$ we first compute $\hat{\mathbf{u}}$ from the convective term

$$\frac{\hat{\mathbf{u}} - \mathbf{u}^{t-1}}{\Delta t} = -C^t$$

where C^t is given by a 3rd order Adams-Bashforth discretization

$$C^t = \frac{23}{12}\mathbf{u}^{t-1} \cdot \nabla\mathbf{u}^{t-1} - \frac{16}{12}\mathbf{u}^{t-2} \cdot \nabla\mathbf{u}^{t-2} + \frac{5}{12}\mathbf{u}^{t-3} \cdot \nabla\mathbf{u}^{t-3}.$$

$\tilde{\mathbf{u}}^0 = 0,\ \tilde{p}^0 = 0$
for $t = 1$ to T **do**
  $M\hat{\mathbf{u}} = f$
  solve $K\Delta p = 1/(\Delta t)Q^T\hat{\mathbf{u}} - K\tilde{p}^{t-1}$
  $\tilde{p}^t = \tilde{p}^{t-1} + \Delta p$
  $M\hat{\hat{\mathbf{u}}} = g$
  solve $M\tilde{\mathbf{u}^{t+1}} - \Delta t/\mathrm{Re}K\tilde{\mathbf{u}^{t+1}} = M\hat{\hat{\mathbf{u}}}$
  if n is an adaptive step **then**
    compute a new solution $\tilde{\mathbf{u}}_2^t$ and $\tilde{p}_2^t$
    estimate errors and adapt the mesh
    obtain a new partition and rebalance the work
  end if
end for

Figure 9.6: Procedure for adaptively solving the incompressible Navier-Stokes equations.

We then compute the pressure p^tusing the newly computed $\hat{\mathbf{u}}$

$$\nabla^2 p^t = \frac{1}{\Delta t}\nabla \cdot \hat{\mathbf{u}}.$$

Finally, we use the viscous term to obtain $\mathbf{u}^t$

$$\frac{\mathbf{u}^t - \hat{\hat{\mathbf{u}}}}{\Delta t} = \frac{1}{\mathrm{Re}}\nabla^2\mathbf{u}^t$$

where

$$\frac{\hat{\hat{\mathbf{u}}} - \hat{\mathbf{u}}}{\Delta t} = -\nabla p^t.$$

The procedure for adaptively computing a solution to this transient problem is summarized in Figure 9.6. $\tilde{\mathbf{u}}^t = (\tilde{u}^t, \tilde{v}^t, \tilde{w}^t)$ is an approximation to the components of the velocity vector $\mathbf{u}^t$ and $\tilde{p}^t$ is an approximation to the pressure p^t at time t, M is the mass matrix and K is the stiffness matrix.

We first specify the initial values for $\tilde{\mathbf{u}}^0$ and $\tilde{p}^0$ and we then iterate through a specified number T of iterations. The first step of each iteration is to obtain $M\hat{\mathbf{u}} = f$, where $f = \Delta t C^t - \tilde{\mathbf{u}}^{t-1}$.

We can now obtain a solution for the pressure $\tilde{p}^t$. We define

$$\tilde{p}^t = \tilde{p}^{t-1} + \Delta p$$

and compute a solution to

$$K\Delta p = \frac{1}{\Delta t}Q^T\hat{\mathbf{u}} - K\tilde{p}^{t-1} \tag{9.2}$$

where $Q = (Q_x, Q_y, Q_z)$ is the asymmetric matrix obtained from the FEM discretization of the term $\nabla \cdot \hat{\mathbf{u}}$. We then use Δp to update $\tilde{p}^t$ which then use to compute $\hat{\hat{\mathbf{u}}}$.

Finally we solve the three independent systems of equations

$$H\tilde{\mathbf{u}}^t = M\hat{\hat{\mathbf{u}}} \tag{9.3}$$

where $H = M - (1/\mathrm{Re})\,K$ is the Helmholtz matrix.

The linear system in Equation 9.2 is the worse conditioned system in this discretization. Therefore, in this problem we compute its solution using an Overlapping Schwarz preconditioning method. On the other hand, the systems of equations in Equation 9.3 converge in very few iterations.

Every few time steps we adapt (refine and coarsen) the mesh. Unfortunately, the real solutions $\mathbf{u}^t$ and p^t are not known as in the example shown in Section 9.2. Nevertheless, we can approximate these functions by computing solutions using meshes with different resolutions [96] (e.g. by varying the orders of the polynomials or by using a coarser or finer mesh). We can then select the regions where the difference between the solutions is the largest.

Thus, if $\tilde{\mathbf{u}}_i^t$ is the approximation described above using basis functions of order i we can now compute a new solution $\tilde{\mathbf{u}}_j^t$, $i \neq j$, using polynomials of order j. Then we can refine every element where $||\tilde{\mathbf{u}}_j^t - \tilde{\mathbf{u}}_i^t|| > e_{\max}$ where $||e||$ an error norm and $e_{\max}$ is our desired maximum error in each element. Similarly, we can coarsen the elements where $||\tilde{\mathbf{u}}_j^t - \tilde{\mathbf{u}}_i^t|| < e_{\max}$.

In a parallel environment it is necessary to rebalance the work between the processors after adapting the mesh. As in the previous examples, we compute a new partition of the mesh which we then migrate along with the solutions $\tilde{\mathbf{u}}^t$, $\tilde{\mathbf{u}}^{t-1}$, $\tilde{\mathbf{u}}^{t-2}$ and $\tilde{p}^t$. We are now ready to begin a new iteration step.

9.3.3 Numerical Results

Figure 9.7 shows our initial 2D mesh consisting of 6026 triangles and 3144 vertices. Figures 9.8 to 9.14 show the adapted meshes at different time steps using $\mathrm{Re} = 250$ where the mesh is distributed between 8 processors. These figures show the formation of unsteady vortices that we have mentioned earlier.

We performed a complete simulation with linear basis functions of 30 seconds of real time in 32 processors of the IBM SP. We specified a time step $\Delta t = 0.002$ seconds and $\mathrm{Re} = 250$,

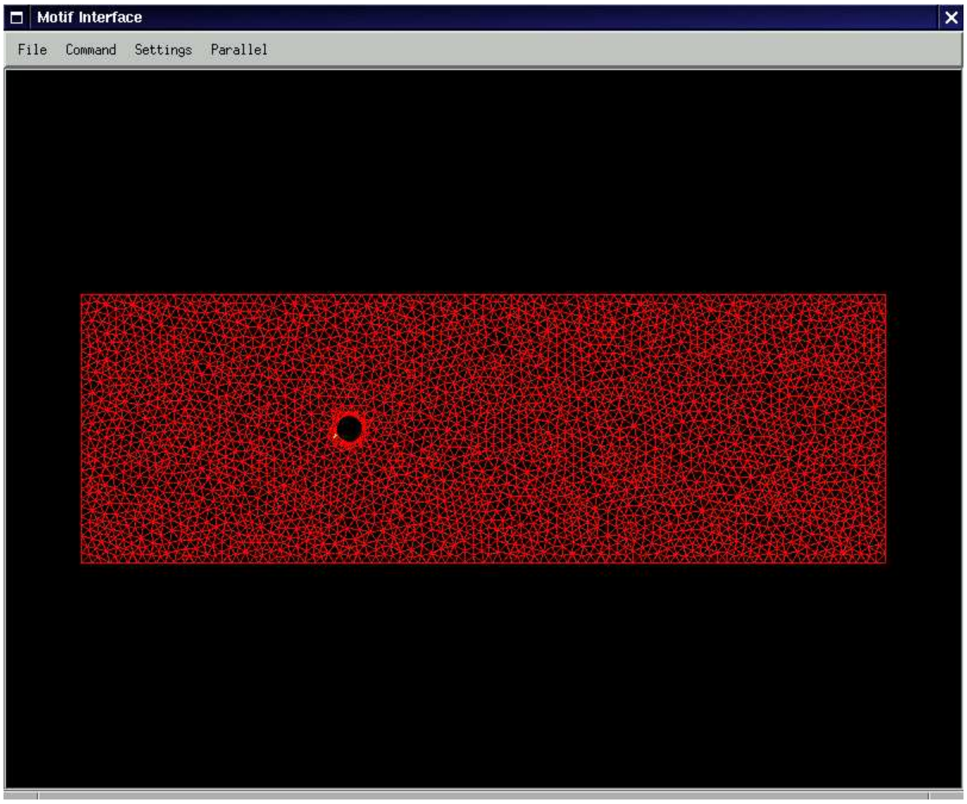


Figure 9.7: Initial 2D mesh used to study the flow past a cylinder.

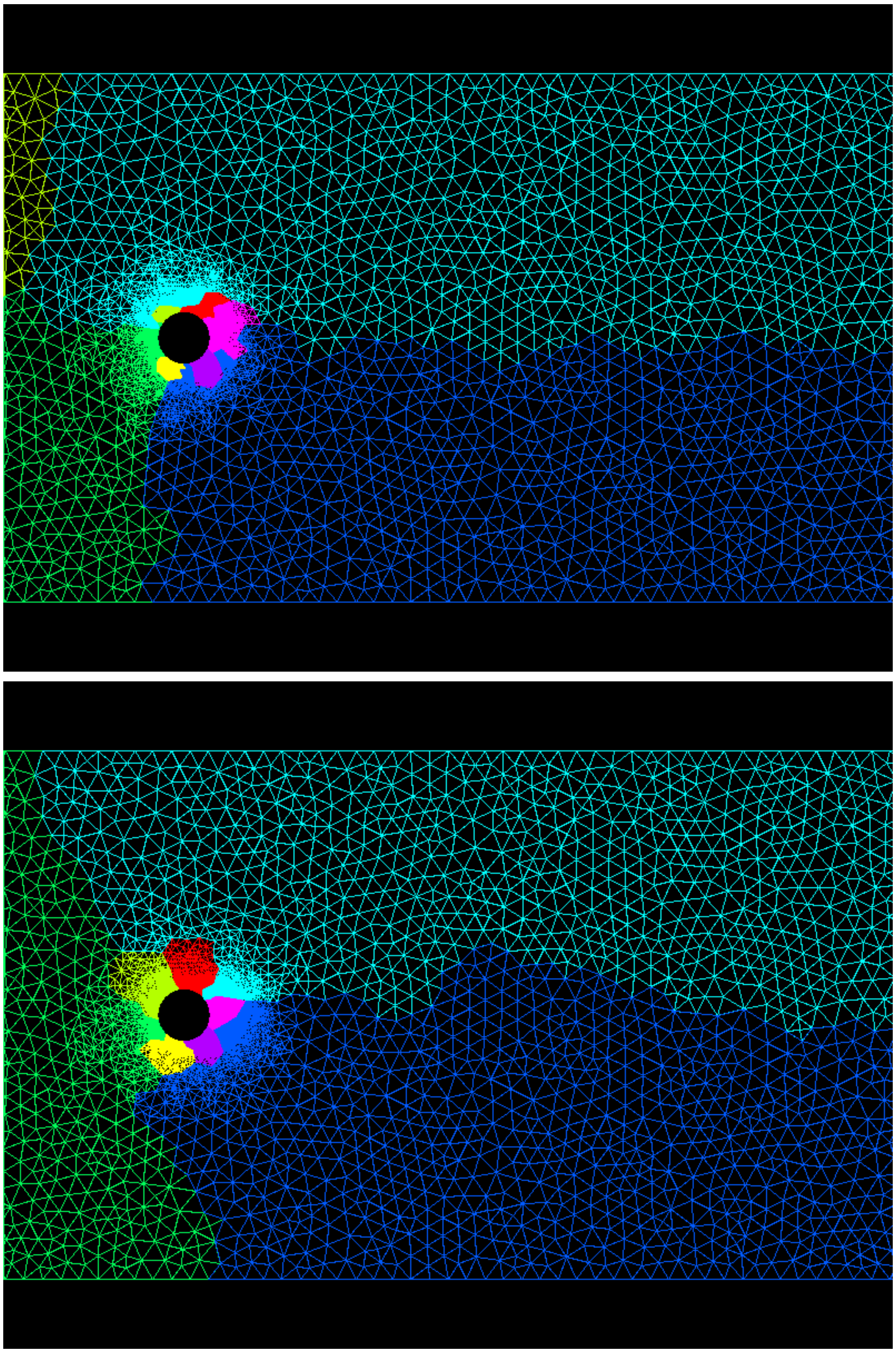


Figure 9.8: Refined mesh at time $t = 0.5$ secs. (top) and $t = 1.25$ secs. (bottom).

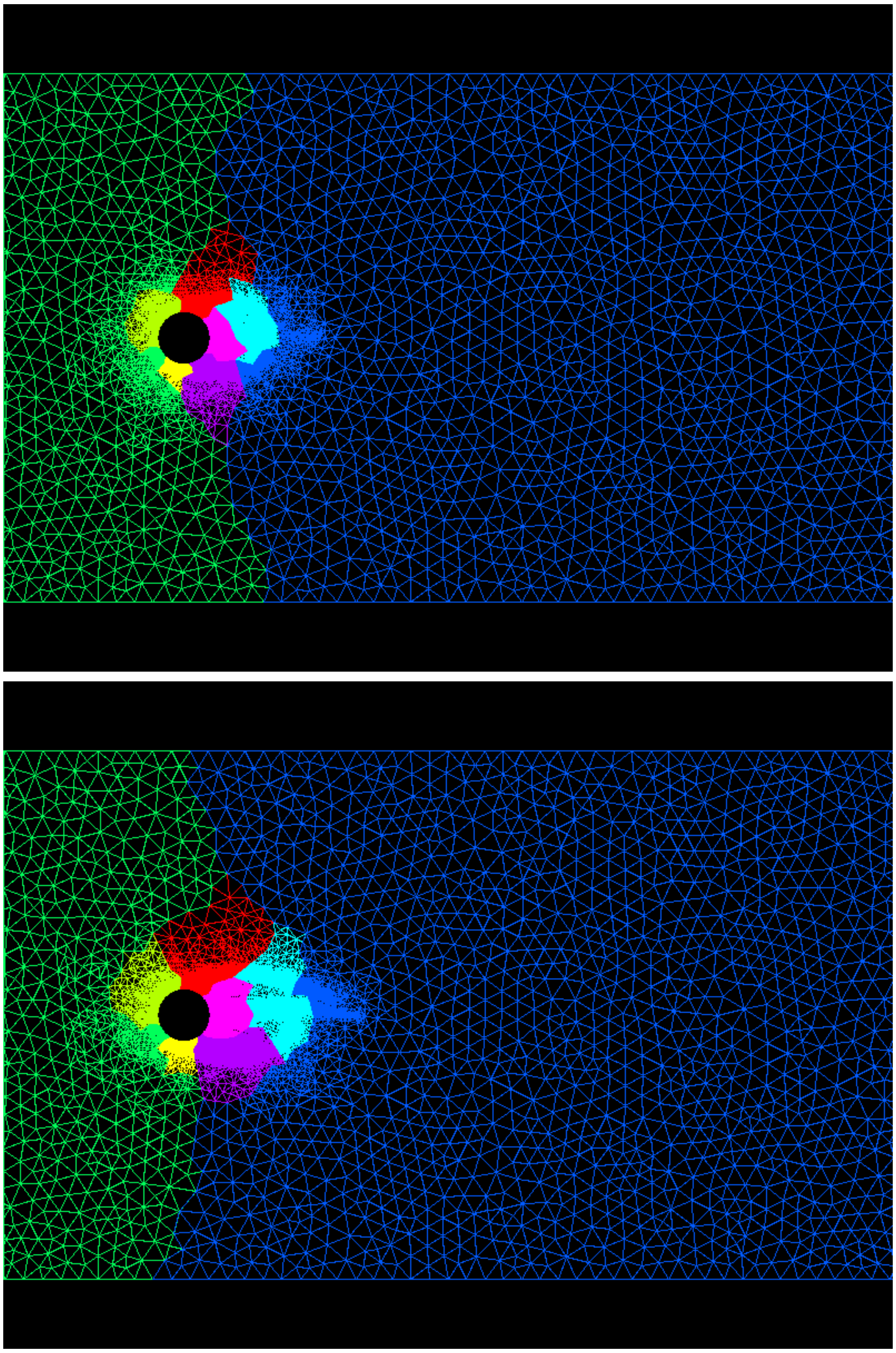


Figure 9.9: Refined mesh at time $t = 2.5$ secs. (top) and $t = 5$ secs. (bottom).

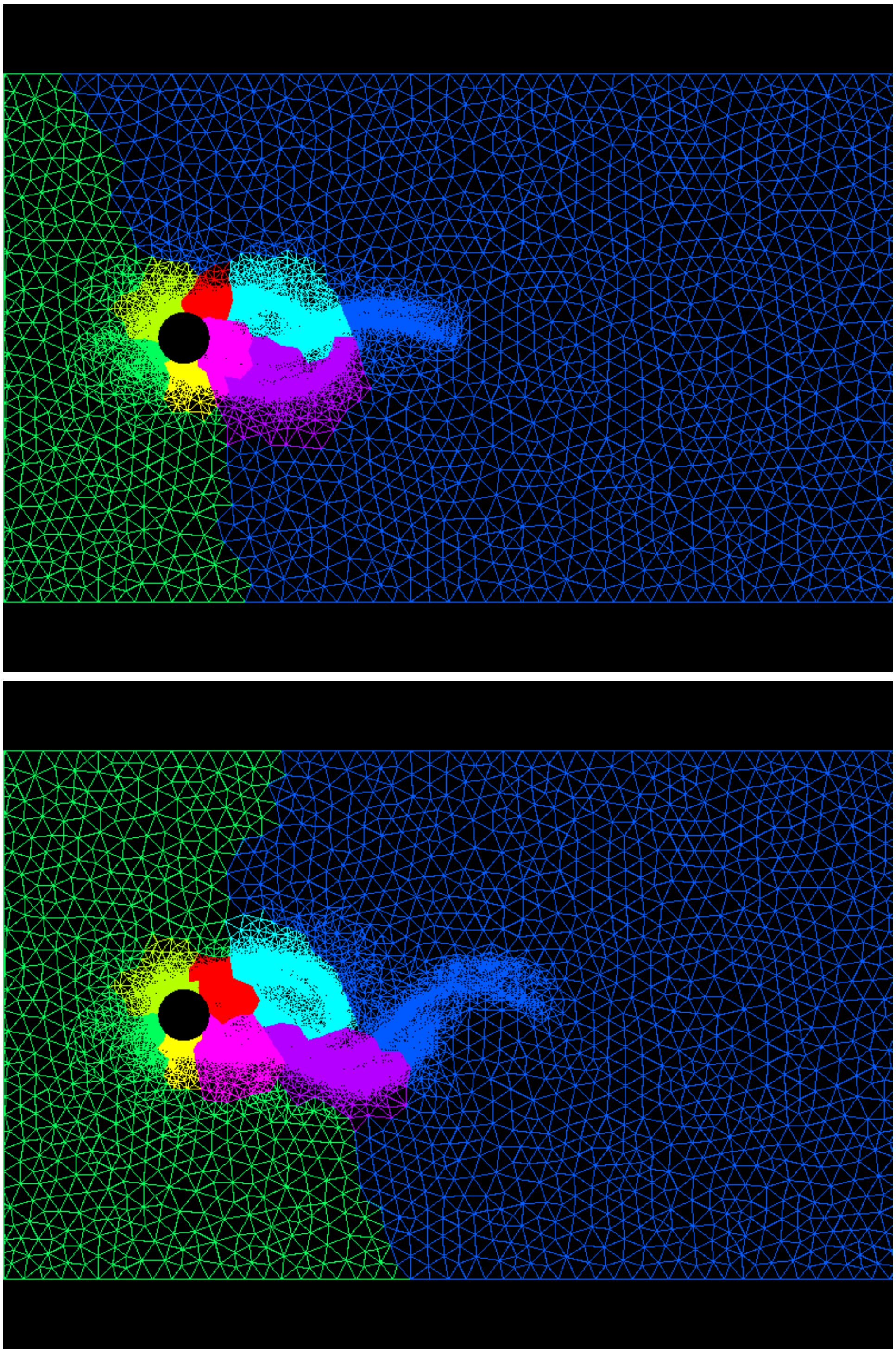


Figure 9.10: Refined mesh at time $t = 7.5$ secs. (top) and $t = 10$ secs. (bottom).

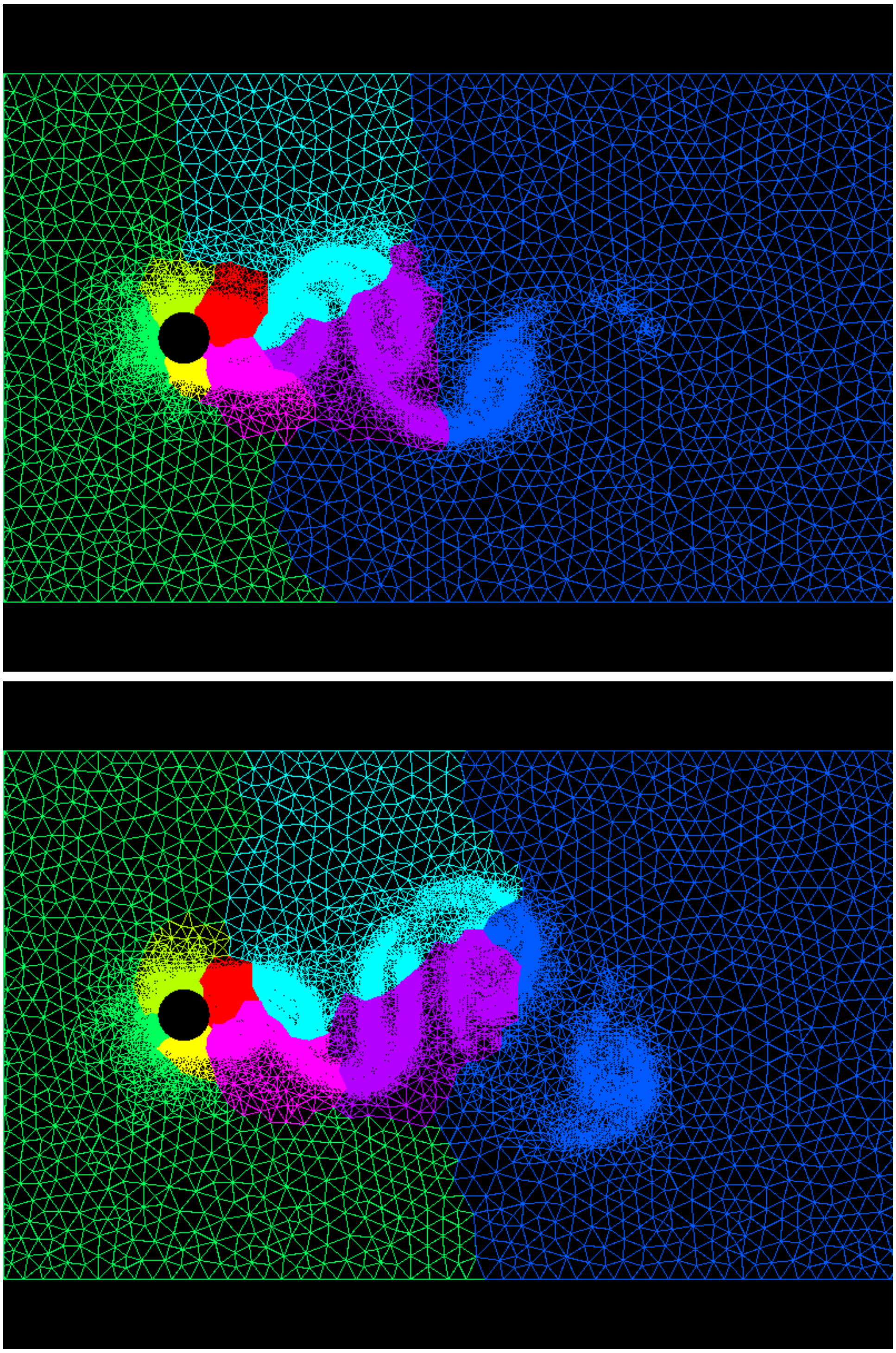


Figure 9.11: Refined mesh at time $t = 12.5$ secs. (top) and $t = 15$ secs. (bottom).

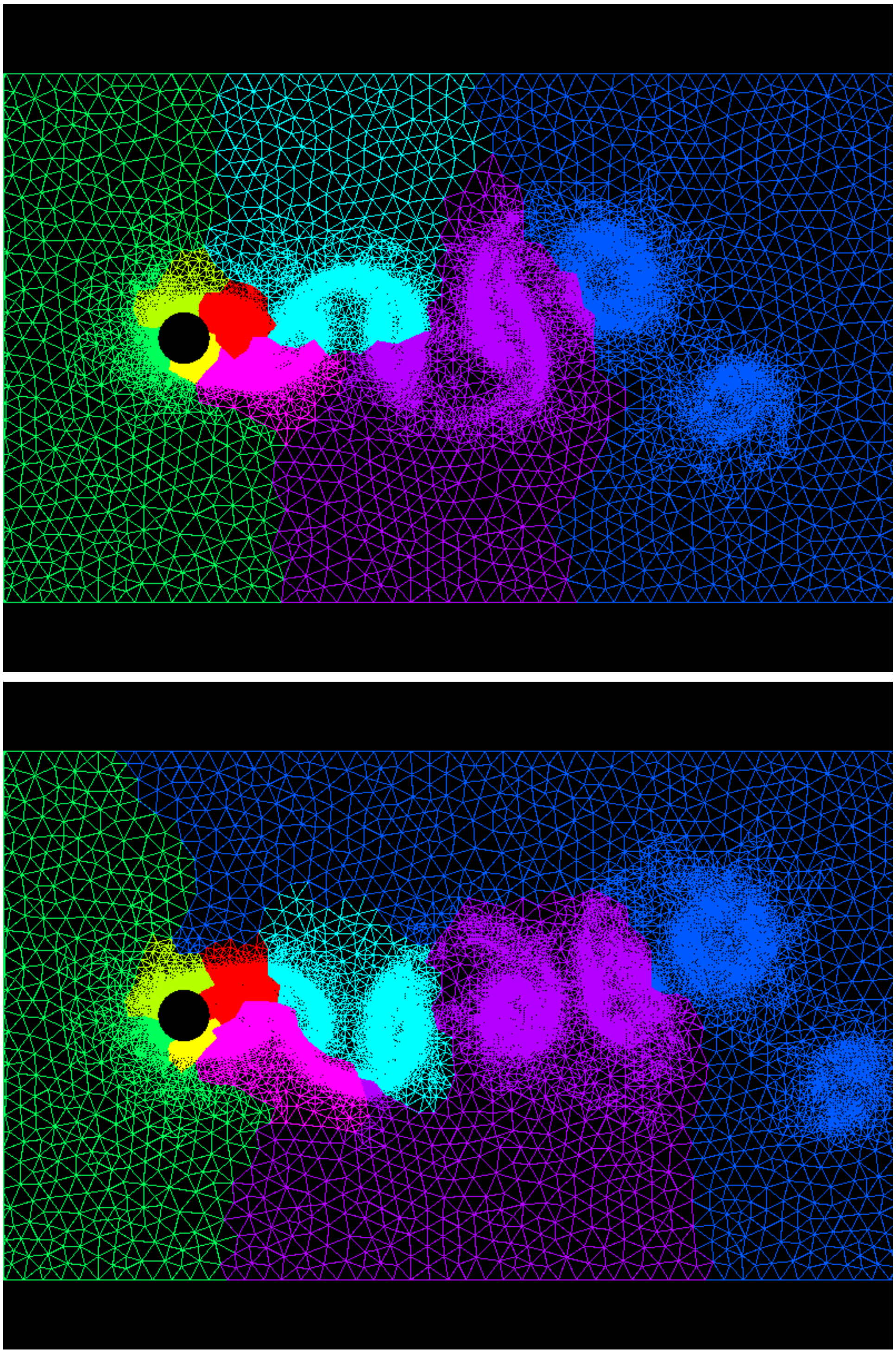


Figure 9.12: Refined mesh at time $t = 17.5$ secs. (top) and $t = 20$ secs. (bottom).

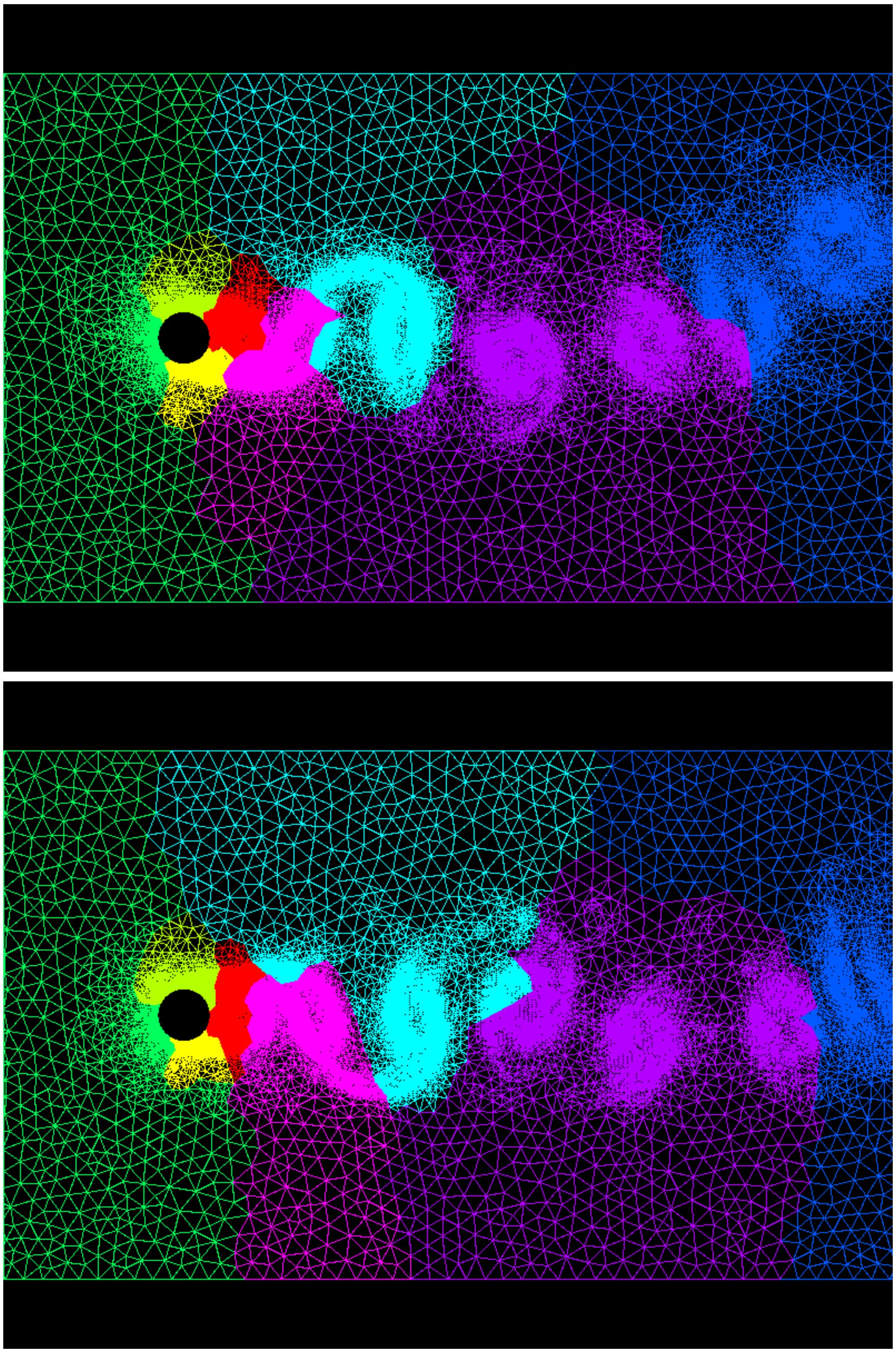


Figure 9.13: Refined mesh at time $t = 22.5$ secs. (top) and $t = 25$ secs. (bottom).

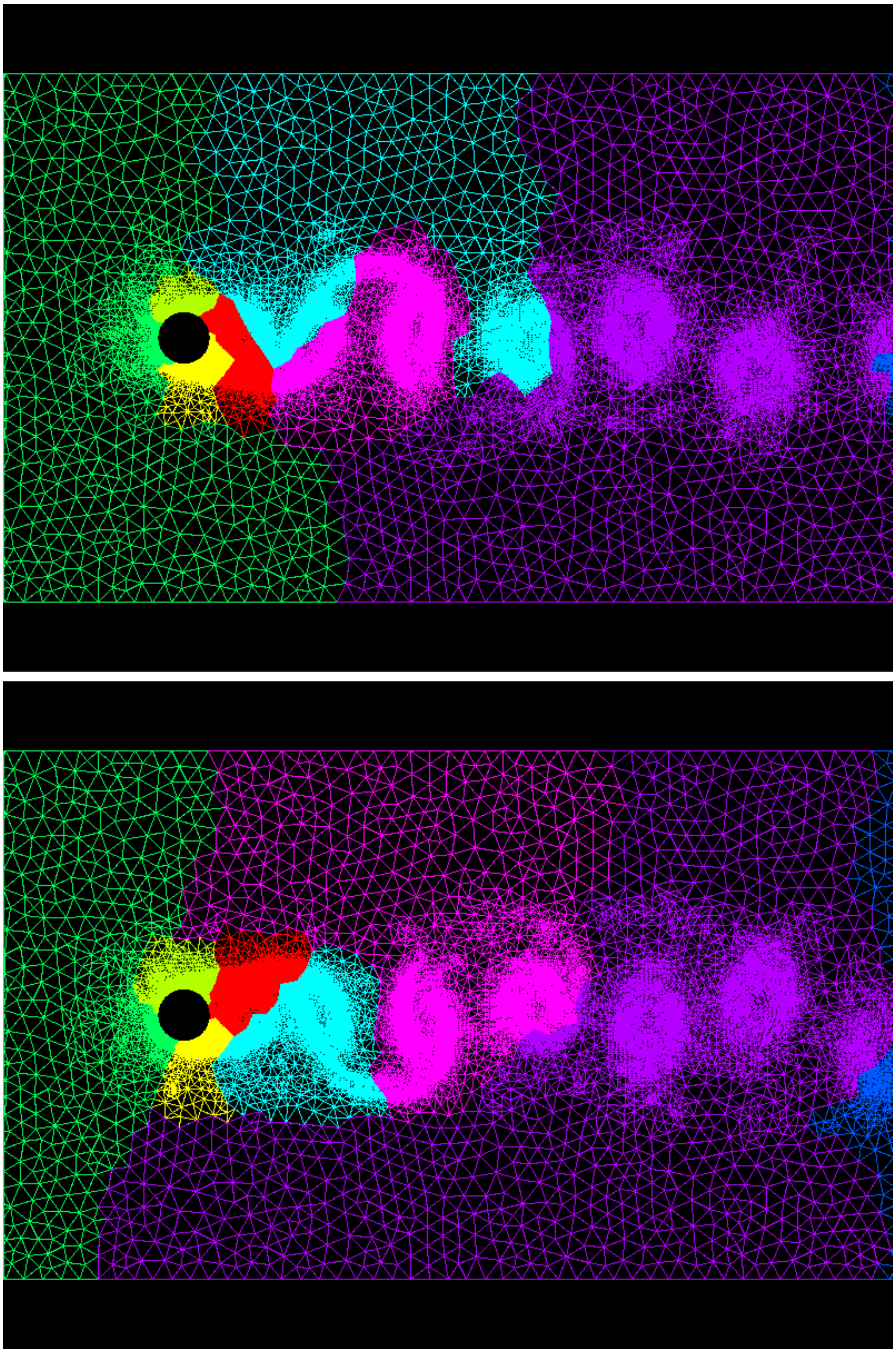


Figure 9.14: Refined mesh at time $t = 27.5$ secs. (top) and $t = 30$ secs. (bottom).

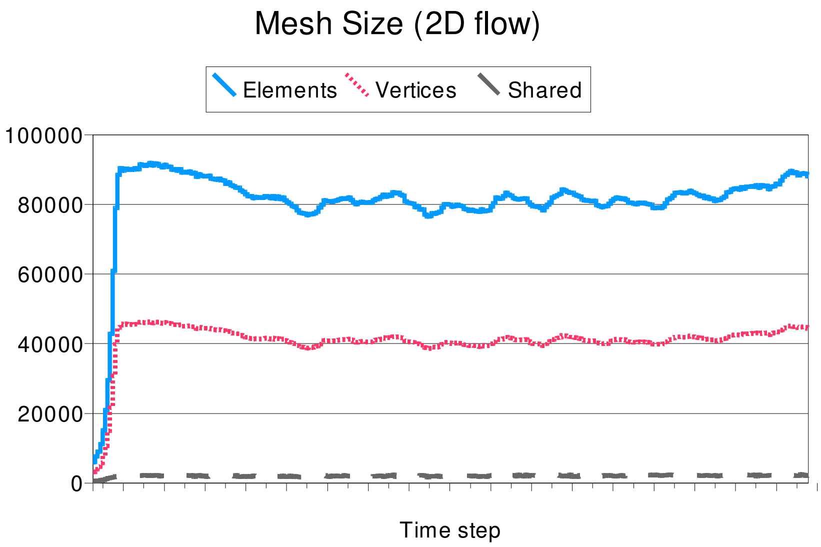


Figure 9.15: Number of elements, vertices and shared vertices for time step t, $0 \leq t \leq 15,000$, in the adapted meshes used to simulate a turbulent flow.

requiring a total of 15,000 iterations. Every 50 iterations we also compute a solution for u and v with quadratic basis functions. The solution of these systems of equations do not impose any significant overhead because the corresponding matrices are well conditioned.

We defined the errors $e_u = \tilde{u}_2^t - \tilde{u}_1^t$ and $e_v = \tilde{v}_2^t - \tilde{v}_1^t$ and the error norm

$$\begin{aligned} ||e_u|| &= Me_u + Ke_u, \\ ||e_v|| &= Me_v + Ke_v. \end{aligned}$$

After specifying the desired error $e_{\max} = 0.001$ we refined every element Ω_a by its longest edge if either $||e_u||_{\Omega_a} > e_{\max}$ or $||e_v||_{\Omega_a} > e_{\max}$. On the other hand, if both errors were less than $e_{\max}$ the element was selected for coarsening.

Figure 9.15 shows the number of elements, vertices, and shared vertices of the mesh as a function of the iteration step. Starting from our initial mesh shown in Figure 9.7 the adaptive procedure rapidly creates a refined mesh containing 81,047.05 elements and 40,859.25 vertices on average with a maximum of 91,885 elements and 46,375 vertices.

Figure 9.16 shows the fraction of the total time spent in each of the steps of the procedure presented in Figure 9.6. The largest fraction corresponds to the solution of Δp which is

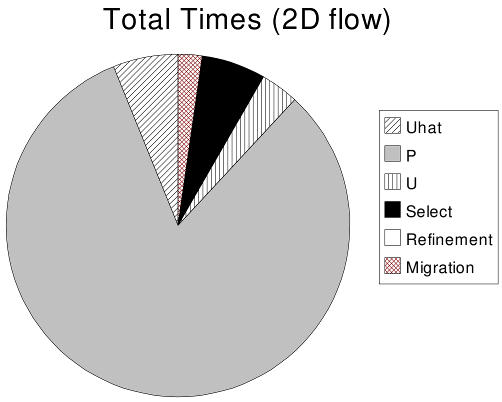


Figure 9.16: Fraction of the total time used to compute $\hat{\mathbf{u}}$, solve p, solve $\mathbf{u}$, select elements for refinement and coarsening using higher order polynomials, adapt the mesh and partition and migrate the mesh for all $15,000$ time steps of a turbulent flow simulation.

computed using the Conjugate Gradient method with a Schwartz precondition [86] and an overlap of 1. The solution of each subdomain (one per processor) is obtained with a LUP decomposition and we used reverse Cuthill-McKee [43] ordering of the matrices to minimize the filling. The solutions for Δp requires approximately 170 iterations on the largest meshes (as opposed to more than 1000 iterations if a Jacobi preconditioner was used). On average, the solution for Δp required 4.04 secs. at every time step with a maximum of 7.63 secs.

The computation for $\hat{\mathbf{u}}$ and the solution of $\tilde{\mathbf{u}}^t$ using linear basis functions are obtained in 0.3 secs. and 0.18 secs. on average respectively. The solution to each of the systems $Hu = M\hat{\hat{u}}$ and $Hv = M\hat{\hat{v}}$ using a Conjugate Gradient solver with a Jacobi preconditioner can be computed in parallel where the iterations of both solvers are interleaved. In this way we reduce by half the number of messages and global sums computed in this phase.

The solution $\tilde{\mathbf{u}}^t$ using quadratic basis functions (which also include the creation of the new systems of equations and the estimation of errors), the refinement and coarsening of the selected elements and the partitioning and migration of the mesh using PNR require an average of 14.74 secs., 0.1 secs. and 5.28 secs. every 50 time steps. Therefore, the procedures to select, adapt and rebalance the work only take a small fraction (around 8%) of the total time. This fraction can be modified by changing the frequency of the adaptation steps.

Adaptive meshes are near optimal for the solution of steady and transient flows. One important question we can ask is how large a static mesh must be in order to achieve the same results presented in this section as the vortices move through the domain. We can obtain a lower bound on the size of this mesh by storing the maximum number of leaf elements that each adaptation tree creates in the course of the simulation. This is a lower bound on the size of the static mesh because the tree might, for example, be refined in its left branch and later in its right branch and still maintain the same maximum weight. In this test, this lower bound is 384,639 elements which is 4.75 times larger than our average mesh.

We can also obtain an upper bound on the size of the static mesh by computing the maximum depth of each refinement tree and then use that number to estimate the number of elements. The upper bound for this problem is 1,509,768 elements with is 18.63 times larger than our average dynamic mesh.

Chapter 10

Future Work

We are happy to report that most of the original goals of Pared have been achieved. When we started this project our ideal scenario included the solution of transient problems such as the Navier-Stokes equation shown in Chapter 9. In these examples, the mesh is repeatedly refined and coarsened in parallel and the different pieces of the mesh move between address spaces as the system follows the regions with high errors. Therefore, we have shown that Pared can also be used to solve complex problems besides our commonly used "corner" example.

Nevertheless, more extensive testing and fine tuning of the system is needed. Our system is flexible and can be used to solve other problems besides the ones already defined. On the numerical side we can always take advantage of improved preconditioners and solvers and more accurate error estimation. For problems like Navier-Stokes we need to use better solvers. We have developed a parallel overlapping Schwarz preconditioner (which is used in the examples) for the Conjugate Gradient method, but we still do not obtain the performance that we expected. We have not implemented a coarse grid accelerator, which again is basically a problem of interpolating solutions (in particular, when using higher order approximations). We have also developed a parallel 3D Navier-Stokes solver but we have not extensively tested it.

There is also some work remaining in the refinement, repartitioning and migration procedures. The longest edge refinement bisection assumes that there is only one longest edge in each element, but what happens if more than two edges have the same longest length? In triangular meshes, the resulting meshes might be different but maintain all the desired mesh properties. In three dimensions, if two adjacent tetrahedra are bisected by different edges of a common face, the faces of their children will not match. This problem has never been discussed in the literature before and is particularly difficult to solve in parallel meshes. We have developed an adaptation procedure that breaks ties based on relative "age" of the edges and seems to guarantee surface conformality even on parallel meshes, but we are still not able to assure that this procedure is correct.

Our system already includes classes to represent complex domains and boundaries which are part of a mesh generator for unstructured meshes that we have implemented. The refinement procedure can be improved to use these classes so that it inserts new points in domain boundaries and not only at the midpoint of an element edge. Therefore the resulting refined meshes can better fit the boundaries than the original coarser meshes. The major difficulty of this approach is that we loose the theoretical guarantees regarding the smallest angle of the refined meshes. There are several special cases that we must also consider that are common in computational geometry (such as "does the new edge intersect another edge or boundary?")

The PNR heuristic uses several magic numbers (such as the α and β parameters that specify the move and unbalance costs) that provide an indication of the relative costs and benefits of small interprocessor boundaries, balanced subsets and low data movement. We do not have an easy way to estimate this parameters based on the current mesh and the machine architecture.

In our system it is easy to move pieces of the mesh between processors. When running in an undedicated network of workstations it should be possible to add all the idle workstations to our computation. When a user logs into the workstation, we should repartition the mesh or move the work to another idle processor. All the functionality to use a dynamic set of processors is already in our system but we have not extensively evaluated it.

Finally, it would also be highly desirable that the partitioning system and the communications library be developed into two separate libraries that can be used independently of PARED. Both subsystems were originally developed independently and have well defined interfaces with the rest of PARED.

This thesis extensively uses the concept of *dynamic distributed* data structures. For example, the mesh is a container whose elements are distributed across many address spaces. These distributions have very different costs and for performance reasons some of these elements might be replicated in several processors. At runtime new elements are dynamically created and destroyed requiring the computation of new distributions of the elements in the distributed container. The most interesting question from a computer science point of view is how to extend these ideas to other problem domains. Our system took advantage of the fact that we have a restricted domain and a large knowledge of what are good mappings of elements to processors. We would like to investigate is this knowledge can be automatically obtained from the system.

Bibliography

[1] Programming parallel computers without programming hosts.

[2] D. J. Acheson. *Elementary Fluid Dynamics.* Oxford University Press, 1990.

[3] T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M. Snir. SP2 system architecture. *IBM Systems Journal*, 38:414–446, 1999.

[4] J. E. Akin. *Finite Elements for Analysis and Design.* Academic Press, 1994.

[5] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *Lapack Users' Guide.* SIAM, Philadelphia, 2nd. edition, 1995.

[6] O. Axelsson. *Iterative Solution Methods.* Cambridge University Press, 1994.

[7] I. Babuska and A. Aziz. On the angle condition in the Finite Element Method. *Int. J. Numer. Meth. Eng.*, 12, 1978.

[8] R. E. Bank, A. H. Sherman, and A. Weiser. Refinement algorithms and data structures for regular local mesh refinement. *Scientific Computing*, 1983.

[9] S. T Barnard and H. Simon. A parallel implementation of multilevel recursive spectral bisection for application to adaptive unstructured meshes. In *Proceedings of the seventh SIAM conference on Parallel Processing for Scientific Computing*, 1995.

[10] S. T. Barnard and H. D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. In *Proceedings of the 6th SIAM conference on Parallel Processing for Scientific Computing*, pages 711–718, 1993.

[11] Dimitri P. Bertsekas and John N. Tsisiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice Hall, 1989.

[12] Rupak Biswas and Leonid Oliker. Load balancing unstructured adaptive grids for CFD. In *SIAM 7th Symposium on Parallel and Scientific Computation*, 1997.

[13] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[14] Ralph Butler and Ewing Lusk. Monitors, messages, and clusters: The p4 parallel programming system. *Parallel Computing*, 20:547–564, April 1994. (Also Argonne National Laboratory Mathematics and Computer Science Division preprint P362-0493).

[15] M. Cross C. Walshaw and M. G. Everett. Parallel dynamic graph partitioning for adaptive unstructured meshes. *Journal of Parallel Processing and Distributed Computing*, 47:102–108, 1997.

[16] José G. Castaños. The dynamic adaptation of parallel mesh-based computation. Master's thesis, Brown University, May 1996.

[17] José G. Castaños and John E. Savage. The dynamic adaptation of parallel mesh-based computation. Technical Report CS-96-31, Department of Computer Science, Brown University, October 1996.

[18] José G. Castaños and John E. Savage. The dynamic adaptation of parallel mesh-based computation. In *SIAM 7th Symposium on Parallel and Scientific Computation*, 1997.

[19] José G. Castaños and John E. Savage. Parallel refinement of unstructured meshes. Technical Report 99-10, Brown University, 1999.

[20] José G. Castaños and John E. Savage. Parallel refinement of unstructured meshes. In *IASTED International Conference on Parallel and Distributed Computing and Systems*, 1999.

[21] José G. Castaños and John E. Savage. Pared: a framework for the adaptive solution of PDEs. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, 1999.

[22] J. E. Castillo, editor. *Mathematical Aspects of Numerical Grid Generation*. SIAM, 1991.

[23] Michael A. Celia and William G. Gray. *Numerical methods for Differential Equations*. Prentice Hall, 1992.

[24] Ohio Supercomputer Center:. MPI primer: Developing with LAM, 1996.

[25] V. Chatzi and F. P. Preparata. Integer-coordinate crystalline meshes. In *Proceedings of the Swiss Conference on CAD/CAM*, 1999.

[26] V. Chatzi and F. P. Preparata. Introduction to integer-coordinate crystalline meshes. Technical Report CS-99-01, Brown University, 1999.

[27] D. E. Culler, R. M. Karp, D. Patterson, S. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Comm. ACM*, 39:78–85, 1996.

[28] E. W. Dijkstra and C. S. Sholten. Termination detection for diffusing computations. *Inf. Proc. Lett.*, 11, 1980.

[29] P. Diniz, S. Plimpton, B. Hendrickson, and R. Leland. Parallel algorithms for dynamically partitioning unstructured grids. In D. Bailey et al., editor, *Parallel Processing for Scientific Computing*, pages 615–620. SIAM, 1995.

[30] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14:1–17, 1988. get this.

[31] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman. Yale sparse matrix package. Technical Report 112, Department of Computer Science, Yale University, 1977.

[32] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman. Yale sparse matrix package. Technical Report 114, Department of Computer Science, Yale University, 1977.

[33] C. M. Fiduccia and R. M. Mattheyses. A linear time heuristic for improving network partitions. In *Proceedings of the 19th IEEE Design Automation Conference*, 1992.

[34] M. Fiedler. Algebraic connectivity of graphs. *Czechoslovak Mathematical Journal*, 23:298–305, 1973.

[35] M. Fiedler. A property of eigenvectors of non-negative symmetric matrices and its application to graph theory. *Czechoslovak Mathematical Journal*, 25:619–633, 1975.

[36] Paul F. Fischer and Anthony T. Patera. Parallel spectral element solutions of eddy-protmoter channel flow. Technical Report CRPC-90-13, Center for Research on Parallel Computation, California Institute of Technology, 1990.

[37] Message Passing Interface Forum:. MPI: A message passing interface standard, 1994.

[38] Message Passing Interface Forum:. MPI-2: Extensions to the message-passing interface, 1997.

[39] Chris Frazier, editor. *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.1.* Addison-Wesley, 1997.

[40] Erich Gamma, Richard Helm, Ralph Johnson, John Vissides, and Grady Booch. *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[41] M. Garey, D. Hohnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[42] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[43] A. George. Computer implementation of the finite element method. Technical Report STAN-CS-208, Stanford University, Department of Computer Science, 1971.

[44] P. L. George. *Automatic mesh generation: application to the finite element method.* Wiley and Sons, Ltd., 1991.

[45] W. Gropp and E. Lusk. *User's Guide for MPICH: A portable Implementation of MPI.* Argonne National Lab and Mississippi State University.

[46] W. Gropp, E. Lusk, and A. Skellum. *Using MPI: Portable parallel programming with the Message Passing Interface.* MIT Press, 1994.

[47] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. Technical Report SAND93-1301, Sandia National Laboratories, 1993.

[48] B. Hendrickson and R. Leland. The Chaco user's guide, version 2.0. Technical Report SAND94-2692, Sandia National Laboratories, 1995.

[49] M. R. Hestenes and E. L. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards*, 49:409–436, 1952.

[50] Y.F. Hu and R.J. Blake. An optimal dynamic load balancing algorithm. Technical Report Preprint DL-P-95-011, Daresbury Laboratory, Warrington, WA4 4AD, UK, 1995.

[51] Cameron Hughes, Thomas Hamilton, and Tracey Hughes. *Object Oriented I/O Using C++ Iostreams*. John Wiley and Sons, 1995.

[52] Claes Johnson. *Numerical solution of partial differential equations by the finite element method*. Cambridge University Press, 1995.

[53] Mark T. Jones and Paul E. Plassmann. Parallel algorithms for the adaptive refinement and partitioning of unstructured meshes. In *Proceedings of the Scalable High-Performance Computing Conference*, 1994.

[54] Mark T. Jones and Paul E. Plassmann. Parallel algorithms for the adaptive refinement. *SIAM J. on Scientific Computing*, 18:686–708, 1997.

[55] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report CORR 95-035, University of Minnesota, Dept. of Computer Science, 1995.

[56] G. Karypis and V. Kumar. Parallel multilevel graph partitioning. Technical Report CORR 95-036, University of Minnesota, Dept. of Computer Science, 1995.

[57] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 29:291–307, 1970.

[58] C. Lanczos. Solution of systems of linear equations by minimized iterations. *Journal of research of the National Bureau of Standards*, 49:33–53, 1952.

[59] C. L. Lawson, R. J. Hanson, D. Lincaid, and F. T. Krogh. Basic linear algebra subprograms for FORTRAN usage. *ACM Transactions on Mathematical Software*, 5:308–323, 1979. get this.

[60] A. Liu and B. Joe. Relationship between tetrahedron shape measures. *BIT*, 34, 1994.

[61] Anderes Lumsdaine, Jeffrey M. Squyres, and Brian C. McCandless. Object oriented MPI (OOMPI): A C++ class library for mpi version 1.0. In *Parallel Object-Oriented methods and Applications Conference , POOMA'96*, 1996.

[62] Scott Meyers. *Effective C++*. Addison-Wesley, 1992.

[63] G. L. Miller, S.H. Teng, W. Thurston, and S. A. Vavasis. Automatic mesh partitioning. In A. George, J. Gilbert, and J. Liu, editors, *Sparse Matrix Computations:Graph Theory Issues and Algorithms*, pages 57–84. New York, 1993.

[64] Gary L. Miller, Shang-Hua Teng, and Stephen A. Vavasis. A unified geometric approach to graph separators. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 538–547, 1991.

[65] S. A. Orszag and L. C. Kells. Transition to turbulence in plane poiseuille flow and plane couette flow. *Journal of Fluid Mechanics*, pages 159–205, 1996.

[66] C. Ozturan, H.L. deCougny, M. S. Shephard, and J. E. Flaherty. Parallel adaptive mesh refinement and redistribution on distributed memory computers. Technical Report TR93-26, Department of Computer Science, Rensselaer Polytechnic Institute, 1993.

[67] V.N. Parthasarathy, C.M. Graichen, and A.F. Hathaway. A comparison of tetrahedron quality measures. *Finite Elements in Analysis and Design*, 15:255–261, 1993.

[68] Alex Pothen, Horst D. Simon, and Kang-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal of Matrix Analysis*, 11(3):430–452, 1990.

[69] Kenneth G. Powell, Philip L. Roe, and James Quirk. Adaptive-mesh algorithms for computational fluid dynamics. In *Algorithmic Trends in Computational Fluid Dynamics*. Springer-Verlag, 1991.

[70] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C.* Cambridge University Press, 1992.

[71] Maria Cecilia Rivara. Algorithms for refining triangular grids suitable for adaptive and multigrid techniques. *International Journal for Numerical Methods in Engineering*, 20:745–756, 1984.

[72] Maria Cecilia Rivara. Selective refinement/derefinement algorithms for sequences of nested triangulations. *International Journal for Numerical Methods in Engineering*, 28:2889–2906, 1989.

[73] Maria Cecilia Rivara. A 3-D refinement algorithm suitable for adaptive and multi-grid techniques. *Communications in Applied Numerical Methods*, 8:281–290, 1992.

[74] I. G. Rosenberg and F. Stenger. A lower bound on the angle of triangles constructed by bisecting the longest side. *Mathematics of Computation*, 29(130):390–395, 1975.

[75] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[76] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, 1996.

[77] J. E. Savage and M. Wloka. Parallelism in graph partitioning. *Journal of Parallel and Distributed Computing*, 13:257–272, 1991.

[78] Kirk Schloegel, George Karypis, and Vipin Kumar. Multilevel diffusing schemes for repartitioning of adaptive meshes. *Journal of Parallel Processing and Distributed Computing*, 47:109–124, 1997.

[79] E. J. Schwabe, G. E. Blelloch, A. Feldmann, O. Ghattas, J. R. Gilbert, G. L. Miller, D. R. O'Hallaron, J. R. Shewchuck, and S. Teng. A separator-based framework for automated partitioning and mapping of parallel algorithms for numerical. In *Proc. 1992 DAGS Symposium*, 1992.

[80] H. R. Schwarz. *Finite Element Methods*. Academic Press, 1988.

[81] Granville Sewell. *The Numerical Solution of Ordinary and Partial Differential Equations*. Academic Press, 1988.

[82] Xianneng Shen and David Klepacki. Message passing on the RS 6000 SP. *AIXpert*, December 1997.

[83] M. S. Shephard, J. E. Flaherty, H. L. DeCougny, C. Ozturan, C. L. Bottasso, and M. Beall. Parallel automated adaptive procedures for unstructured meshes. In *Parallel Computing in CFD*. AGARD, 1995.

[84] Jonathan Richard Shewchuck. An introduction to the conjugate gradient method without the agonizing pain. Technical Report CMU-CS-94-125, School of Computer Science, Carnegie Mellon University, 1994.

[85] H. D. Simon. Partitioning of unstructured meshes for parallel processing. *Computing Systems Eng.*, 1991.

[86] Barry Smith, Peter Bjorstad, and William Gropp. *Domain Decomposition: parallel multilevel methods for elliptic partial differential equations*. Cambride University Press, 1996.

[87] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The complete reference.* MIT Press, 1996.

[88] Bjarne Stroustrup. *The C++ programming Language.* Addison-Wesley, 1991.

[89] Steve Teale. *C++ IOStreams Handbook.* Addison-Wesley, 1993.

[90] J. F. Thompson, editor. *Numerical Grid Generation.* North-Holland, 1982.

[91] J. F. Thompson, Z. U. A. Warsi, and C. W. Mastin. *Numerical Grid Generation: Foundations and Applications.* North-Holland, 1985.

[92] T. von Karman. *Aerodynamics: selected topics in the light of their historical development.* Cornell University Press, 1954.

[93] H. F. Walker. Implementation of the GMRES method using householder transformations. *SIAM Journal of Scientific Computing*, 9:152–163, 1988.

[94] R. D. Williams. DIME: A user's manual. Technical Report C3P 861, Caltech Concurrent Computation, 1990.

[95] Roy Williams. Adaptive parallel meshes with complex geometry. *Numerical Grid Generation in Computational Fluid Dynamics and Related Fields*, 1991.

[96] O. C. Zienkiewicz and R. L. Taylor. *The Finite Element Method.* McGraw-Hill, 4th edition, 1994.