

# MESH DATA MANAGEMENT COMPONENTS FOR PETASCALE ADAPTIVE UNSTRUCTURED MESH BASED SIMULATIONS

By

Ting Xie

A Thesis Submitted to the Graduate  
Faculty of Rensselaer Polytechnic Institute  
in Partial Fulfillment of the  
Requirements for the Degree of  
DOCTOR OF PHILOSOPHY  
Major Subject: COMPUTER SCIENCE

Approved by the  
Examining Committee:

---

Dr. Mark S. Shephard, Thesis Adviser

---

Dr. Christopher D. Carothers, Member

---

Dr. Elliot Anshelevich, Member

---

Dr. Kenneth E. Jansen, Member

---

Dr. Seegyoung Seol, Member

Rensselaer Polytechnic Institute  
Troy, New York

May 2012  
(For Graduation August 2012)

© Copyright 2012  
by  
Ting Xie  
All Rights Reserved

# CONTENTS

LIST OF TABLES . . . . .	vi
LIST OF FIGURES . . . . .	viii
ACKNOWLEDGMENT . . . . .	x
ABSTRACT . . . . .	xii
1. INTRODUCTION . . . . .	1
1.1 Background and Motivation . . . . .	1
1.2 Organization . . . . .	6
1.3 Nomenclature . . . . .	7
2. RELATED INVESTIGATIONS . . . . .	8
2.1 Data Models in Mesh-Based Simulation . . . . .	9
2.1.1 Geometric model . . . . .	9
2.1.2 Attributes . . . . .	10
2.1.3 Mesh . . . . .	11
2.1.4 Solution fields . . . . .	13
2.2 General Topology-Based Mesh Data Structure . . . . .	14
2.2.1 Topological entities and adjacencies . . . . .	14
2.2.2 Geometric classification . . . . .	16
2.2.3 Mesh representation options . . . . .	16
2.3 Distributed Mesh Data Structure . . . . .	19
2.3.1 Part and part boundary . . . . .	19
2.3.2 Partition model . . . . .	22
2.4 Mesh Partitioning . . . . .	23
2.5 Mesh Migration . . . . .	26
3. GENERIC COMPONENTS . . . . .	29
3.1 Generic Programming in Scientific Computing Software . . . . .	29
3.2 Simulation Data Model Analysis . . . . .	31
3.3 Set Component . . . . .	32
3.3.1 Design and implementation . . . . .	32

3.3.2	Application . . . . .	37
3.3.2.1	Mesh entity set for iMesh: . . . . .	37
3.4	Iterator Component . . . . .	38
3.4.1	Design and implementation . . . . .	38
3.4.2	Applications . . . . .	40
3.4.2.1	Mesh entity traversal in part by type and/or topology: . . . . .	41
3.4.2.2	Mesh entity traversal by reverse classification: . . . . .	43
3.4.2.3	Part boundary mesh entity traversal: . . . . .	45
3.4.2.4	Mesh entity traversal in set by type and/or topology: . . . . .	46
3.4.2.5	Geometric entity traversal by type: . . . . .	46
3.5	Tag Component . . . . .	48
3.5.1	Design and implementation . . . . .	49
3.5.2	Application . . . . .	51
3.6	Scaling Studies on Mesh Adaptation Using Generic Components . . . . .	55
4.	ENTITY SET . . . . .	58
4.1	Introduction to Set Theory . . . . .	58
4.2	Entity Set Definitions . . . . .	59
4.3	Entity Set Functionality . . . . .	61
4.4	Mesh Sets to Deal with Boundary Layer Meshes . . . . .	65
4.4.1	Partition objects . . . . .	67
4.4.2	Pre-processing for graph-based mesh partitioning . . . . .	69
4.4.3	Mesh migration algorithm . . . . .	73
4.5	Examples and Applications . . . . .	75
4.5.1	Mesh partition and migration with P-sets . . . . .	75
4.5.2	Support boundary layer mesh adaptation . . . . .	78
5.	MESH MATCHING . . . . .	80
5.1	Geometric Model with Periodic Boundary Conditions . . . . .	81
5.2	Matched Mesh Definition and Functionality . . . . .	82
5.3	Construct Matched Mesh from Mesh File . . . . .	84
5.4	Pre-processing for Graph-based Mesh Partitioning . . . . .	85
5.5	Mesh Migration Algorithm . . . . .	89
5.5.1	Exchange entities . . . . .	90
5.5.2	Remove unused entities . . . . .	92

5.6	Examples and Applications . . . . .	93
5.6.1	Mesh partition and migration . . . . .	95
5.6.2	Support matched mesh adaptation . . . . .	96
6.	MULTIPLE PARTS PER PROCESS . . . . .	98
6.1	Distributed Mesh Data Structure . . . . .	99
6.1.1	Part and part id . . . . .	99
6.1.2	Enhanced partition model . . . . .	101
6.1.3	Partition entities . . . . .	102
6.2	Global and Local Graph-based Partitioning . . . . .	104
6.2.1	Partition objects . . . . .	105
6.2.2	Collecting neighboring partition objects for global partitioning	106
6.2.3	Collecting neighboring partition objects for local partitioning .	106
6.2.4	Post-processing . . . . .	108
6.3	Mesh Migration Algorithm . . . . .	108
6.3.1	Entity migration algorithm . . . . .	108
6.3.2	Communications between part boundary entities . . . . .	111
6.3.3	Entire part migration algorithm . . . . .	116
6.4	Examples and Applications . . . . .	119
6.4.1	AAA model with $O(10^8)$ elements on 512 processors . . . . .	121
6.4.1.1	Graph-based global partitioning . . . . .	121
6.4.1.2	Graph-based local partitioning . . . . .	122
6.4.1.3	Graph-based local partitioning followed by ParMA .	124
6.4.1.4	Compare the three methods . . . . .	125
6.4.2	AAA model with $O(10^8)$ elements on 2,048 processors . . . . .	127
6.4.2.1	Graph-based global partitioning . . . . .	128
6.4.2.2	Graph-based local partitioning . . . . .	129
6.4.2.3	Hypergraph-based local partitioning . . . . .	131
6.4.2.4	Compare the three methods . . . . .	132
6.4.3	Mesh partitioning with predictive load balancing . . . . .	135
6.4.4	AAA model with billions of elements for large-scale adaptive simulations . . . . .	138
7.	CONCLUSION AND FUTURE WORK . . . . .	141
	REFERENCES . . . . .	143

## LIST OF TABLES

3.1	Scaling results of air-bubble mesh adaptation on ANL Intrepid . . . . .	57
4.1	Set functionality at mesh level . . . . .	62
4.2	Basic set functionality at set level . . . . .	63
4.3	Hierarchical set functionality at set level . . . . .	63
4.4	Set binary functionality at set level . . . . .	63
4.5	Set functionality at entity level . . . . .	65
4.6	Average execution time of boundary layer mesh repartitioning . . . . .	77
5.1	Average execution time of matched mesh partitioning . . . . .	96
6.1	Average execution time of 133M mesh repartitioning on 512 processors through global ParMetis . . . . .	122
6.2	Average number of regions on each part of 133M repartitioned mesh on 512 processors through global ParMetis . . . . .	122
6.3	Average execution time of 133M mesh repartitioning on 512 processors through local ParMetis . . . . .	123
6.4	Average number of regions on each part of 133M repartitioned mesh on 512 processors through local ParMetis . . . . .	124
6.5	Average execution time of 133M mesh repartitioning on 512 processors through local ParMetis followed by ParMA . . . . .	125
6.6	Average number of regions on each part of 133M repartitioned mesh on 512 processors through local ParMetis followed by ParMA . . . . .	125
6.7	Compare average execution time of three repartitioning methods on 512 processors . . . . .	126
6.8	Compare region imbalance of partitions of three repartitioning methods on 512 processors . . . . .	127
6.9	Average execution time and region imbalance of 2,048-part partition on 512 processors through global ParMetis . . . . .	128
6.10	Average execution time of 133M mesh repartitioning on 2,048 processors through global ParMetis . . . . .	129

6.11	Average number of regions on each part of 133M repartitioned mesh on 2,048 processors through global ParMetis . . . . .	129
6.12	Average execution time of 133M mesh repartitioning on 2,048 processors through local ParMetis . . . . .	130
6.13	Average number of regions on each part of 133M repartitioned mesh on 2,048 processors through local ParMetis . . . . .	130
6.14	Average execution time of 133M mesh repartitioning on 2,048 processors through local PHG . . . . .	132
6.15	Average number of regions on each part of 133M repartitioned mesh on 2,048 processors through local PHG . . . . .	133
6.16	Compare average execution time of three repartitioning methods on 2,048 processors . . . . .	133
6.17	Compare region imbalance of partitions of three repartitioning methods on 2,048 processors . . . . .	135
6.18	Average number of regions on each part of 8.57B repartitioned mesh on 16,384 processors through local PHG . . . . .	140

## LIST OF FIGURES

1.1	Example of boundary layer mesh adaptation . . . . .	2
1.2	Key steps in parallel adaptive mesh-based simulation . . . . .	5
2.1	Example geometry-based problem definition . . . . .	11
2.2	Representation of field defined over mesh . . . . .	13
2.3	Example of 3D mesh adjacencies . . . . .	15
2.4	Example of simple model and mesh showing geometric classification . .	17
2.5	Example of 3D mesh representations . . . . .	18
2.6	Distributed mesh on four processors with one part per processor . . . .	20
2.7	Distributed mesh and partition classification . . . . .	23
2.8	Example of 2D mesh and partition object diagram . . . . .	25
2.9	Example of 2D mesh migration . . . . .	28
3.1	Class diagram of generic set component . . . . .	35
3.2	Class diagram of mesh and mesh entity inherited from generic tag com- ponent's classes . . . . .	51
3.3	Segment of straight pipe model with air bubbles . . . . .	56
4.1	Example part of boundary layer mesh . . . . .	66
4.2	Distributed mesh with P-sets . . . . .	68
4.3	Distributed boundary layer mesh on four parts . . . . .	75
4.4	Whole body model composed of 78 arteries . . . . .	76
4.5	Example of boundary layer mesh adaptation on pipe model . . . . .	78
4.6	Two example parts of partitioned and adapted boundary layer mesh . .	78
4.7	Example of boundary layer mesh adaptation on heat exchanger model .	79
5.1	Two types of periodic boundary conditions . . . . .	81
5.2	Example of distributed 2D matched mesh . . . . .	82



5.3	Example of graph-based matched mesh partitioning . . . . .	85
5.4	Example of 2D matched mesh migration . . . . .	89
5.5	Example of two partitioned matched meshes on cube model on CCNI BG/L . . . . .	95
5.6	Example of matched mesh adaptation on sector model . . . . .	97
5.7	Example of matched mesh adaptation on pipe model . . . . .	97
6.1	Distributed mesh on two processes with two parts per process . . . . .	100
6.2	Partition model view of distributed mesh . . . . .	101
6.3	Example of global and local partitioning . . . . .	104
6.4	Example of 2D mesh migration with multiple parts per process (part 1)	112
6.5	Example of 2D mesh migration with multiple parts per process (part 2)	113
6.6	Geometry and mesh of AAA model . . . . .	119
6.7	Number of regions on each part of 133M mesh on 2,048 parts . . . . .	120
6.8	Region imbalance ratio on each part of 133M mesh on 65,536 parts . . .	131
6.9	Number of regions on each part of adapted 187M mesh on 1,024 parts with/without <i>PredLB</i> . . . . .	136
6.10	Number of regions on each part of adapted 187M mesh on 4,096 parts with/without <i>PredLB</i> . . . . .	136
6.11	Number of regions on each part of 1B mesh on 2,048 parts . . . . .	139

## ACKNOWLEDGMENT

The completion of this thesis owes much to the help of many people. It is my great pleasure to take this opportunity to express my sincere gratitude towards them. Without their guidance and support, this thesis would not be possible.

My first thanks go to my advisor Professor Mark S. Shephard to express my gratitude for his invaluable guidance towards this thesis. His constant encouragement and consistent patience throughout my Ph.D. studies lightened my tough way to accomplish this degree in the interdisciplinary area of computer science and scientific computing, which was entirely new to me before I came to SCOREC. I also want to thank him for his painstaking efforts of reviewing this thesis to greatly improve the overall quality.

I am deeply grateful to Professor Kenneth E. Jansen, who enriched me the knowledge on real applications of my research and provided me a chance to work with advanced supercomputers. His insights on large-scale simulations greatly contributed to this thesis.

I gratefully thank Professor Christopher D. Carothers and Professor Elliot Anshelevich for teaching me high performance computing and algorithm courses and for providing me with a solid foundation for my research. I would like to thank Dr. Seegyoung Seol for her invaluable technical advices. I have learned a lot from her about the advanced programming techniques and the importance of programming with concepts. I gratefully thank their time and efforts to serve on my doctoral committee and reviewing this thesis. Their valuable feedback helped me improve this thesis in many ways.

I would like to thank the great SCOREC people, Professor Onkar Sahni, Dr. Xiaojuan Luo, Dr. Andrew C. Bauer, Dr. Min Zhou, Alexander Ovcharenko, Kaiqiu Lu, Cameron Smith, Misbah Mubarak, and others, for their help and fruitful discussions. I gratefully thank all my friends, Zhen Wang and Yixiao Zhang, and others, for their sincere help and support.

Most of all, I would like to express my gratitude to my beloved parents, Yubin

Xie and Baogu Li, for their heartily support during my entire life. Their love helped me go through the difficult times. Moreover, I would like to thank my parents-in-law for always being supportive in my work and taking care of my daughter. Lastly, I give my special thanks to my husband Zhi Zhou and my daughter Yiting Zhou, who gave me emotional support, thoughtful encouragement, and persistent joy in the past years of my studies.

This work was supported by the US DOE Office of Science's SciDAC/ITAPS program. I greatly appreciate the financial support from this agency.

## ABSTRACT

The simulation of complex physics problems over general 3D geometries can be effectively done using unstructured meshes that in many cases will have millions or billions of elements that can be only solved on massively parallel computers. The objective of this thesis is to support specific new unstructured mesh functionalities required by large-scale adaptive simulations building on the Flexible and distributed Mesh DataBase (FMDB).

To extend FMDB's functionalities in a sustainable manner that FMDB can easily evolve to support future application requirements, this work introduces a set of generic programming components for sets, iterators and tags designed for use in adaptive simulation software tools.

FMDB is then extended to address three specific requirements from large scale adaptive simulations. First, the generic set component is extended to support mesh set functions for mesh entities in parallel and is applied to support boundary layer mesh adaptations. Second, to support specific applications where the mesh representation on specific unconnected geometric model entities must match, such as applications with periodic boundary conditions, the capability of mesh matching is developed. Third, since it is desirable to increase the number of processors in the simulation as the mesh size increases, the capability to have multiple parts per process is developed to define new mesh partitions with alternative partitioning strategies and migration algorithms, based on an enhanced partition model.

With the addition of these capabilities, FMDB has been able to support a large class of adaptive unstructured mesh simulations on petascale supercomputers, including IBM BlueGene (BG/P) and Cray system. Applications on meshes of billions of elements distributed over  $O(100,000)$ 's of processors demonstrate the effectiveness of the software components developed in this work.

# CHAPTER 1

## INTRODUCTION

### 1.1 Background and Motivation

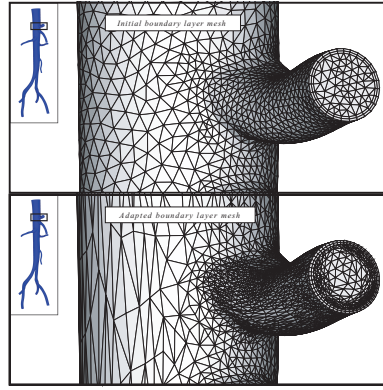
Unstructured mesh methods, like finite elements and finite volumes, are used to support effective analysis of complex physical behaviors modeled by partial differential equations (PDE) over general three-dimensional domains. The most reliable and potentially efficient application of unstructured mesh methods is through the application of adaptive methods where an error estimator, or indicator, is used to control where and how the mesh is to be modified. Adaptive unstructured mesh simulations on general 3D geometries with complex physics require underlying meshes with millions, or even billions, of entities. The simulation on such a large mesh can be only carried out on massively parallel computers that require the mesh be distributed over the computing nodes and/or cores of the parallel computer. The following issues must be considered in the mesh operations [1]:

- Setting-up and constructing distributed meshes efficiently.
- Understanding of how to represent unstructured meshes to meet the application requirements.
- Applying dynamic load balancing to regain load balance. As the mesh modifications change the number of mesh entities, the load balance of the mesh between the processors is also changed.

The consequence of the above factors is that all steps in a large-scale adaptive unstructured mesh simulation must be efficiently executed on a distributed mesh and the mesh must be effectively redistributed as required at various steps in the simulation.

Unstructured meshes are characterized by allowing any number of elements adjacent to a single node. One obvious advantage that unstructured meshes have over structured grids is to represent arbitrarily complex geometric domains. Combined with appropriate numerical methods, adaptive unstructured mesh methods

have demonstrated a number of extra advantages, such as improving the geometric approximation as the mesh is adapted [2], generating anisotropic meshes to align with the geometric and solution features [3]. Moreover, anisotropically adapted meshes can provide two or three orders of magnitude fewer elements than a more uniform mesh for the same level of accuracy. The RPI SCOREC group [2, 3, 4, 5, 6] has developed an effective anisotropic mesh adaptation procedure controlled by an anisotropic mesh metric field for realistic geometries to construct fully unstructured, anisotropic meshes. This mesh adaptation procedure has been extended to support viscous flow problems that require high-aspect ratio, orthogonal and graded layers of elements at no-slip walls resulting in boundary layer meshes (see Figure 1.1).



**Figure 1.1: Anisotropic adaptivity, preserving boundary layer mesh structure in an artery model [7].**

Practical adaptive analysis problems often involve large scale and complicated geometries such as patient-specific cardiovascular flow simulations [8], and place extreme demands on the mesh infrastructure, which is executed underneath providing all needed mesh-based operations and thus strongly influences the overall performance of the simulation. However, those who develop the individual analysis or adaptation code are typically not capable of developing all the required mesh data structures and operations to support their applications and want to have the required mesh infrastructure taken care of by meshing experts. To support various application requirements of representing and manipulating unstructured mesh data, there needs a substantial effort at the mesh infrastructure level.

One substantial effort to address the needs of unstructured meshes on massively parallel computers is the SciDAC Interoperable Technologies for Advanced Petascale Simulations (ITAPS) Center [9], now continuing as part of the FASTMath institute [10]. ITAPS includes three core data types: the geometric data, the mesh data, and the field data [9, 11, 12]. These core data types are associated with each other through data relation managers. The parallel ITAPS data model supports a distributed memory representation where the mesh is distributed over independent processors of the computer [13]. To support the needs of unstructured mesh applications, ITAPS has defined a set of common interfaces: Geometry (*iGeom*), Mesh (*iMesh/iMeshP*), Field (*iField*), Data Relation Manager (*iRel*), and *iBase* that contains the utilities and definitions used by other interfaces [9, 11, 12, 13]. These interfaces are interoperable allowing multiples tools to integrate into a single simulation [9]. Their efforts toward petascale simulations have demonstrated the scalability of their tools that use the ITAPS interfaces and Zoltan [14] dynamic load balancing services [13].

As an implementation of the ITAPS iMesh/iMeshP interface, the Flexible and distributed Mesh DataBase (FMDB) [15, 16] is designed to provide the set of functions needed to support the ITAPS mesh adapt services. FMDB focuses on dealing with adaptively changed mesh data and provides all the needed mesh-based operations, such as distributed mesh operations and dynamic mesh load balancing in parallel computations. On top of a general topology-based mesh representation and a partition model, FMDB supports a general distributed mesh representation. Starting from the original FMDB implementation, the objective of this thesis work is to address unstructured mesh representation and operations for specific application requirements, especially in a distributed environment on massively parallel computers, and thus to support large-scale adaptive unstructured mesh simulations on petascale supercomputers.

To support a broad range of functionalities and future application needs in a sustainable and extendable way in FMDB, this thesis work takes advantage of the basic software engineering developments and applies generic programming methods [17, 18, 19]. In contrast to the traditional programming paradigm in which data

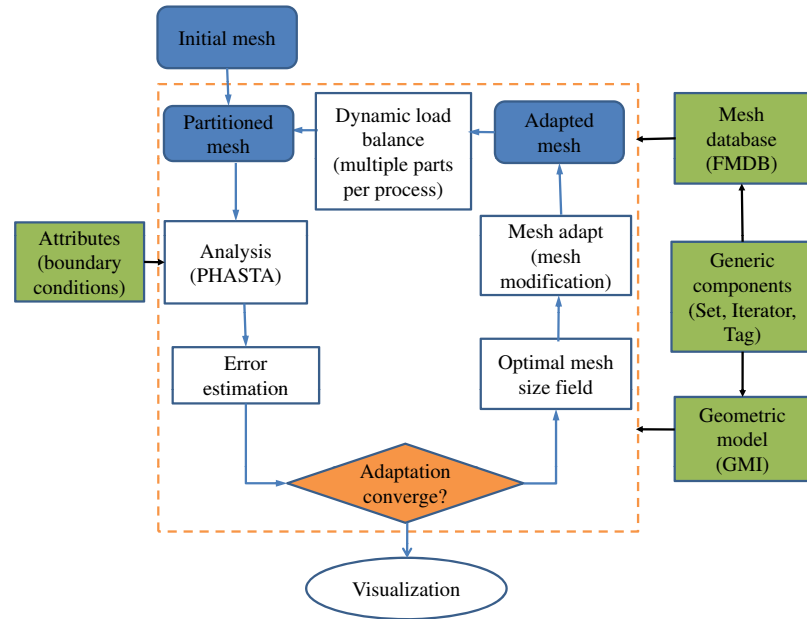
structures and algorithms are developed for specific data types and requirements, generic programming focuses on the development of efficient, reusable software components which can be reused for many similar implementations through appropriate abstractions for their common requirements. This thesis introduces an on-going effort for efficient, reusable generic software components and their applications for adaptive unstructured mesh-based simulations. In this work, the design and implementation of the three generic components - set, iterator, and tag - which are in use as utilities are presented. Their applications to satisfy various iterator, set and tagging needs are demonstrated within the geometric model and mesh data infrastructure.

Figure 1.2 demonstrates the main components in an adaptive simulation loop. An adaptive mesh-based analysis usually starts with an initial mesh generated from a mesh generation tool on a geometric model using *a priori* specification of the mesh size. The initial mesh is distributed across the processors of the parallel computer. Based on the resulting partitioned mesh, an analysis tool performs the computation and obtains a solution. This solution is then evaluated by an error estimator [3, 5] to specify the mesh size field [3] which quantifies the desired mesh resolution over the problem domain. Local mesh modification operations [3, 20, 21] are carried out to satisfy the specific mesh size field. The adapted mesh and computation solution are then transferred to the next step of the adaptive loop. This process iterates until it comes to an acceptable level of solution accuracy.

To move forward to petascale simulations, each software component in the framework should be scalable and efficient. Prior to this research the software components available could be carried out in serial as well as in parallel but with a modest number of processing cores. To support various unstructured mesh applications on  $O(100,000)$ 's of processing cores brings extra complexities. In this work, the mesh infrastructure, FMDB, is extended to address the specific requirements. More specifically, the extensions developed include:

- Extend the generic set component to fully support set functions for mesh entity sets. A specific requirement from some applications such as boundary layer mesh applications is the ability to maintain specific sets of mesh entities





**Figure 1.2: Key steps in parallel adaptive mesh-based simulation.**

together during parallel computations. While the overall set functions must support sets that may span multiple parts in a partition, the focus of this work is to support sets that are constrained to a single part for the efficiency issues. Two types of mesh sets are defined in parallel: P-set with the single part constraint, and NP-set without the single part constraint. Parallel mesh operations such as mesh partition and migration, and related algorithms are developed to support the set operations for P-sets. The boundary layer mesh adaptation in parallel clearly demonstrate the use of P-sets.

- Provide mesh matching to support specific classes of applications that have the requirement for the mesh representation on specific geometric model faces to be identical topologically and geometrically (transformed). Parallel mesh operations such as mesh partition and migration, and related algorithms are developed to deal with mesh matching.
- In the adaptive cycle of large-scale simulations, as the mesh size increases through mesh adaptation, a mesh redistribution is desired on an adapted mesh

to create new partitions on a larger number of parts so the subsequent steps can be executed on a larger number of processors. In this work, multiple parts per process partitioning has been developed and implemented such that it can be easily applied with two alternative partitioning strategies and mesh migration algorithms to define various new mesh partitions. This capability enables the execution of adaptive simulations on meshes of billions of elements running on  $O(100,000)$ 's of processing cores.

## 1.2 Organization

The organization of this thesis is as follows. Chapter 2 gives an introduction of the related work on mesh management components. This chapter introduces a general topology-based mesh representation and distributed mesh representation based on a partition model, followed by the discussions on main parallel mesh operations, such as mesh partition and mesh migration. Chapter 3 presents an on-going effort for efficient, generic software components for adaptive mesh-based analysis and demonstrates how they are applied in the mesh and geometric model to meet various needs of set, iterator and tagging functionalities. Chapter 4 extends the generic set component to fully support set operations of entity sets, and demonstrates the capability to deal with applications that have specific mesh entity grouping requirements such as boundary layer mesh adaptations in parallel. Chapter 5 presents a general technique of mesh matching to handle meshes on geometric models with periodic boundary conditions. Chapter 6 presents the multiple parts per process partitioning, which addresses the need to change the number of parts in a mesh partition to increase the number of processing cores in subsequent steps during adaptive simulations. Chapter 7 concludes the study by summarizing the results obtained, and discusses the future work.

### 1.3 Nomenclature

$V$  the model,  $V \in \{G, P, M\}$ , where  $G$  represents the geometric model,  $P$  represents the partition model, and  $M$  represents the mesh model.

$\{V\{V^d\}\}$  a set of topological entities of dimension  $d$  in model  $V$ .

$V_i^d$  the  $i^{th}$  entity of dimension  $d$  in model  $V$ .  $d = 0$  for a vertex,  $d = 1$  for an edge,  $d = 2$  for a face, and  $d = 3$  for a region.

$\{\partial(V_i^d)\}$  a set of entities on the boundary of  $V_i^d$ .

$\{V_i^d\{V^q\}\}$  a set of entities of dimension  $q$  in model  $V$  that are adjacent to  $V_i^d$ .

$V_i^d\{V^q\}_j$  the  $j^{th}$  entity in the set of entities of dimension  $q$  in model  $V$  that are adjacent to  $V_i^d$ .

$U_i^{d_i} \sqsubset V_j^{d_j}$  classification indicating the unique association of entity  $U_i^{d_i}$  with entity  $V_j^{d_j}$ ,  $d_i \leq d_j$ , where  $U, V \in \{G, P, M\}$  and  $U$  is lower than  $V$  in terms of a hierarchy of domain decomposition.

$P_i$  the  $i^{th}$  part in a distributed mesh.

$\mathcal{P}[V_i^d]$  residence part operator which returns a set of part id(s) where entity  $V_i^d$  exists,  $V \in \{P, M\}$ .

$S_i$  the  $i^{th}$  entity set.

For example,  $\{M\{M^3\}\}$  is the set of all the regions in a 3D mesh;  $\{M_i^1\{M^2\}\}$  include the mesh faces adjacent to mesh edge  $M_i^1$ ;  $M_i^3\{M^2\}_1$  is the 1<sup>st</sup> face adjacent to mesh region  $M_i^3$ ;  $M_i^2 \sqsubset G_j^2$  means that mesh face  $M_i^2$  is classified on geometric model face  $G_j^2$ .

## CHAPTER 2

### RELATED INVESTIGATIONS

This chapter provides the background technology in developing the new parallel capabilities of the later chapters.

This chapter first presents the data models involved with an unstructured mesh-based simulation in §2.1. Such a simulation often starts with a problem definition, including the geometric model and the attributes over the problem domain. Then the problem domain is discretized into a piecewise decomposition, i.e. a mesh. In a subsequent step in the simulation such as the error estimations, fields are defined over the domain discretization to make use of the solution properly.

Unstructured mesh methods have common requirements of representing and manipulating the mesh and associated data. One effective way to describe unstructured meshes is through a general topology-based mesh representation, in which a hierarchy of topological entities, including regions, faces, edges and vertices, are defined and their connections, in terms of adjacencies, are stored. The general topology-based mesh representation is presented in §2.2.

To solve computationally demanding problems, such as applications involved with complicated geometries with complex physics, parallel adaptive unstructured mesh methods are used. In addition to a specific mesh representation, parallel unstructured mesh methods introduce more complications, such as the need for *(i)* the mesh (re)distribution, *(ii)* data communications, and *(iii)* distributed mesh data operations.

A distributed mesh data structure supports a topological representation of the distributed mesh and efficient distributed mesh manipulation operations. A general distributed mesh representation built on top of a partition model is given in §2.3.

One common approach to perform the mesh distribution is through mesh partitioning, which separates the mesh into a number of parts. §2.4 discusses mesh partitioning techniques for unstructured meshes.

To move mesh entities from one part to another part in support of *(i)* mesh dis-

tribution to parts, *(ii)* mesh load balancing, or *(iii)* obtaining mesh entities needed for mesh modification operations, a parallel processing procedure, referred to as mesh migration, is introduced in §2.5.

## 2.1 Data Models in Mesh-Based Simulation

A mesh-based simulation begins with a mathematical problem definition, which consists of a description of the geometric domain<sup>1</sup> with appropriate analysis attributes [22, 23]. Then the geometric domain is decomposed into a set of small pieces, the mesh, and the mathematical problem, modeled by continuous partial differential equations (PDEs), is approximated on that mesh using mesh methods such as finite volume, finite difference, or finite element. Once the domain and PDEs are discretized, the numerical problems can be easily executed.

The data models used in such a mesh based simulation include: *(i)* the geometric model which houses the topological and shape description of the domain of the problem, *(ii)* attributes which describe the rest of information needed to define the physical problem, *(iii)* the mesh which is the discretized representation of the domain used by the analysis method, *(iv)* fields which house the distribution of numerical solution tensors over the domain of the problem and numerical systems resulting from the discretization processes.

### 2.1.1 Geometric model

The geometric model considered here is a subset of a three-dimensional space bounded by a collection of geometric entities (points, curves, surfaces and volumes) [24]. As the most common geometric representation in computer-aided design (CAD) systems, *boundary representations (b-reps)* are effective for representing geometric models [25, 26].

The geometric model is a data model which provides a functional interface to support the communication of geometry information to mesh-based applications.

***Geometric entities:*** the primary constituents of a geometric model. They are,

---

<sup>1</sup>The problem domain considered here may also include a temporal component if the solution changes with time [22].

in the Radial Edge Data Structure [27], *regions*, *shells*, *faces*, *loops*, *edges* and *vertices* and *use* entities for vertices, edges, loops and faces with a non-manifold model<sup>2</sup>.

**Adjacencies:** how geometric entities are connected to each other [28].

**Geometric interrogations:** provide specific information relating to the shape of geometric entities such as pointwise locations and shape coefficients [2, 25, 29].

**Geometric entity sets:** mechanism to group geometric entities for various purposes [9, 30]. The useful attributes and requirements of geometric entity set are: (i) entity uniqueness and entity insertion order preservation (ii) set population through entity addition or entity removal (iii) traversal through an iterator per entity type (iv) set binary operations (union, subtraction, intersection) (v) relationships among entity sets such as superset/subset and parent/child.

**Tags:** mechanism to attach arbitrary user data, termed as *tag data*, to geometric entity sets or geometric entities [9, 30]. Tag data is a single or array of a specific type where it could be of primary data type such as integer, double, geometric entity set, geometric entity, or arbitrary type represented as *void\**.

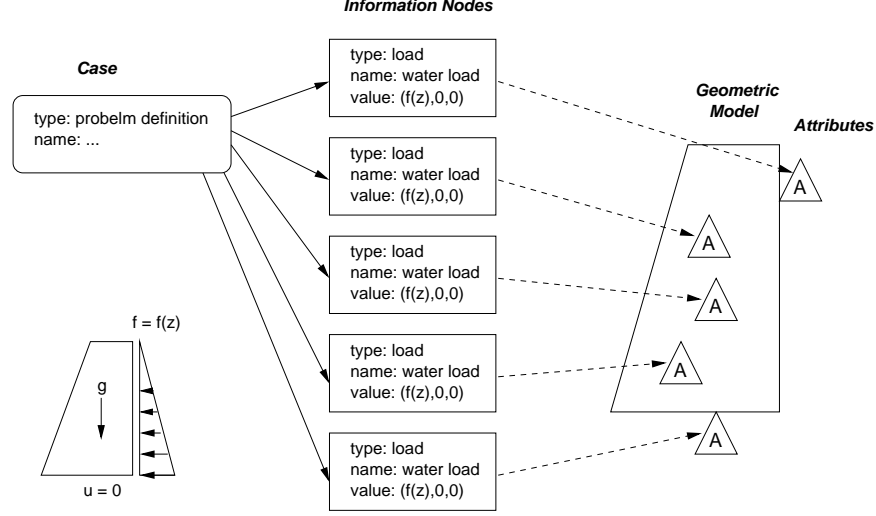
**Iterators:** mechanism to traverse geometric entities by type either in an entity set or in the entire model [9].

### 2.1.2 Attributes

In addition to the geometric model, the specification of a mesh-based application problem requires extra mathematical information, such as equations, material properties, forcing functions, boundary conditions, and initial conditions [23, 31]. For instance, a PDE solver may require a domain definition associated with the mathematical form controlling the simulation and parameters associated with the controlling mathematical equations [23]. Such kind of information is described in

---

<sup>2</sup>A non-manifold model consists of general combinations of solids, surfaces, and wires. Please refer to reference [27] for detailed discussions.



**Figure 2.1: Example of geometry-based problem definition [23].**

terms of attributes. Many attributes are tensorial in nature, and may vary in both space and time.

Attributes can be associated with a specific portion of the geometric domain being analyzed. In the problem definition, when attributes are primarily associated with boundary value problems, a natural choice is to associate attributes with the various topological geometric model entities in the boundary representation of the geometric model [31, 32, 33].

A simple example of a problem definition is illustrated in Figure 2.1. The problem being modeled here is a dam subjected to loads due to the gravity and the water behind the dam. The attribute case for the problem definition covers a set of attributed information nodes. If this case is associated with the geometric model, attributes (indicated by triangles with  $A$ 's inside of them) are created and attached to the individual geometric model entities on which they act [23, 34].

### 2.1.3 Mesh

The domain discretization is a piecewise decomposition of the problem domain, usually a mesh. Even though both spatial and temporal domains can employ different discretizations, typically the more complex of the two is the definition of the spatial mesh. Thus the focus herein is on the spatial mesh defined over the geometric domain. A single mesh can cover the entire geometric (spatial) domain,

while there are cases where more than one mesh is associated with a domain. For instance, different full geometry meshes can be used during different stages of the numerical solution, as in the case of adaptive mesh methods.

One effective way to describe an unstructured mesh is through a general topology-based mesh representation [15, 35, 36, 37, 38, 39, 40], which supports the ability to properly adapt the mesh fully accounting for the geometric domain and attributes as the mesh changes [35, 41]. A detailed discussion on a general topology-based mesh representation is given in §2.2.

The mesh is a data model which provides a description of the mesh information in a manner that mesh-based operations can be efficiently performed. A minimum set of functional requirements to support adaptive simulations includes:

**Mesh entities:** the constituents of a mesh. They are distinguished by their type, i.e. topological dimension<sup>3</sup>, (*vertex* (0D), *edge* (1D), *face* (2D), or *region* (3D)), and topology (for instance, triangle and quadrilateral for 2-dimensional entities, or tetrahedron or hexahedron for 3-dimensional entities) [42].

**Adjacencies:** how the topological mesh entities connect to each other. For an entity of dimension  $d$ , first-order adjacency returns all of the mesh entities of dimension  $q$ , which are either on the closure of the entity for a downward adjacency ( $d > q$ ), or which it is part of the closure for an upward adjacency ( $d < q$ ) [11, 15, 16, 35, 39].

**Geometric classification:** a relation that each mesh entity maintains to a geometric model entity for partial representation. Given a geometric model entity, the set of equal dimension mesh entities classified on geometric entity is termed as the *reverse classification* for the geometric entity [35].

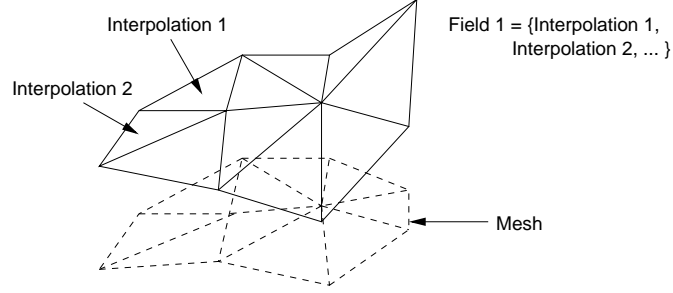
**Mesh entity sets:** mechanism to group of mesh entities for various purposes [12, 30]. There are various kinds of sets depending on the following criteria. The useful options of mesh entity set are the following.

- entity uniqueness

---

<sup>3</sup>In this thesis, entity *dimension* and entity *type* are interchangeable terms.





**Figure 2.2: Representation of a field defined over a mesh [23].**

- entity insertion order preservation
- entity type constraint

In addition, *(i)* set population through entity addition or entity removal *(ii)* traversal through an iterator per entity type and/or topology *(iii)* set binary operations such as subtraction, intersection, and union *(iv)* relationships among sets such as superset/subset and parent/child are needed for flexible set manipulation.

**Tags:** mechanism to attach arbitrary user data, termed as *tag data*, to a part, entity set or mesh entity [12, 30]. Tag data is a single or array of a specific type where it could be of primary data type such as integer, double, mesh entity set and mesh entity, or arbitrary type represented as *void\**.

**Iterators:** mechanism to traverse mesh entities in a specific range with various options [9, 12], such as *(i)* traversing entities by type and/or topology *(ii)* traversing entities classified on a specific geometric model entity, and so on.

#### 2.1.4 Solution fields

A solution field describes the variation of solution tensors over the mesh entities to quantify the distribution of physical parameters. The spatial variation of the field is defined in terms of interpolations defined over the mesh. A field is a collection of individual interpolations, each of which is interpolating the same quantity, and is associated with one or more mesh entities [23]. Figure 2.2 describes a field written in terms of  $C^0$  interpolating distribution functions over a patch of mesh entities.

Fields must be maintained in a form that can be used for queries and manipulations as required, such as the transfer of fields to other meshes during a multiphysics analysis step, or to maintain the description of the mesh on an adapted field [22, 23].

## 2.2 General Topology-Based Mesh Data Structure

One effective way to describe unstructured meshes is through a general topology-based mesh representation [15, 16, 35, 36, 37, 38, 39]. The mesh consists of a collection of topological entities of controlled size, shape and distribution. The relations of mesh entities defining the mesh are well described through topological adjacencies. A mesh data structure is a toolbox that is able to answer various queries about the mesh, and that provides the mesh-level services to higher level applications that create and manipulate the mesh data.

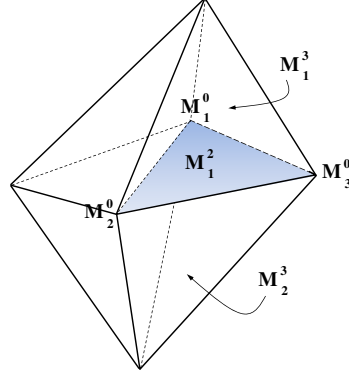
There are three functional requirements of a general topology-based mesh data structure [35], and they are *(i)* topological entities, *(ii)* adjacencies between entities, and *(iii)* geometric classification.

### 2.2.1 Topological entities and adjacencies

Each topological mesh entity of dimension  $d$ ,  $M_i^d$ , is bounded by a set of topological lower order mesh entities. The full set of mesh entities in a 3-D mesh are:  $\{\{M\{M^0\}\}, \{M\{M^1\}\}, \{M\{M^2\}\}, \{M\{M^3\}\}\}$ , where  $\{M\{M^d\}\}$ ,  $d = 0, 1, 2, 3$ , are, respectively, the set of vertices, edges, faces, and regions. A mesh can be represented with the basic 0 to  $d$  dimensional topological entities, where  $d$  is the dimension of the problem domain, with the following topological restrictions [35]:

- Regions and faces have no interior holes.
- Each entity of order  $d_i$  in a mesh,  $M_i^{d_i}$ , may use a particular entity of lower order,  $d_j$ ,  $M_j^{d_j}$ ,  $d_j < d_i$ , at most once.
- For any entity  $M_i^d$ , there is the unique set of entities of order  $d-1$ ,  $\{M_i^d\{M^{d-1}\}\}$  that are on the boundary of  $M_i^d$ .

The first restriction means that a region can be represented by the faces that bound the region, and a face can be represented by the edges that bound the face, and



**Figure 2.3: Example of 3D mesh adjacencies.**

so on. The second restriction supports the orientation of an entity in terms of its boundary entities. The third restriction means that a mesh entity can be uniquely specified by its bounding entities.

Note, mesh entities are distinguished by their type, i.e. topological dimension (*vertex* (0D), *edge* (1D), *face* (2D), or *region* (3D)), and topology (such as, triangle and quadrilateral for 2-dimensional entities, or tetrahedron or hexahedron for 3-dimensional entities) [11, 12, 42].

Entity adjacencies describe how mesh entities connect to each other. Adjacency definition includes both first-order and second-order adjacencies [11, 12, 15, 16, 35, 39]:

- *First-order adjacencies*

For an entity of dimension  $d$ , a first-order adjacency returns all of the mesh entities of dimension  $q$ , which are either on the closure of the entity for a downward adjacency ( $d > q$ ), or which it is part of the closure for an upward adjacency ( $d < q$ ).

- *Second-order adjacencies*

For an entity of dimension  $d$ ,  $M_i^d$ , second-order adjacencies describe all the mesh entities of dimension  $q$  that share any adjacent entities of dimension  $b$  with the entity  $M_i^d$ , where  $d \neq b$  and  $b \neq q$ .

A second-order adjacency indicates the set of topological entities of a given dimension that are adjacent to entities sharing common boundary entities of the specific dimension. Second-order adjacencies are useful for many applications to provide needed neighboring information. Second-order adjacencies can be derived from first-order adjacencies. For instance, in the small mesh in Figure 2.3, for mesh face  $M_1^2$  (the shaded plane), the regions on either side of  $M_1^2$  include mesh regions  $M_1^3$  and  $M_2^3$  (first order upward adjacencies,  $\{M_1^2\{M^3\}\}$ ); the vertices that bound  $M_1^2$  include mesh vertices  $M_1^0$ ,  $M_2^0$ ,  $M_3^0$  (first order downward adjacencies,  $\{M_1^2\{M^0\}\}$ ); the regions that share any vertex with mesh region  $M_1^3$  include all other three neighboring mesh regions in the mesh (second-order adjacencies).

### 2.2.2 Geometric classification

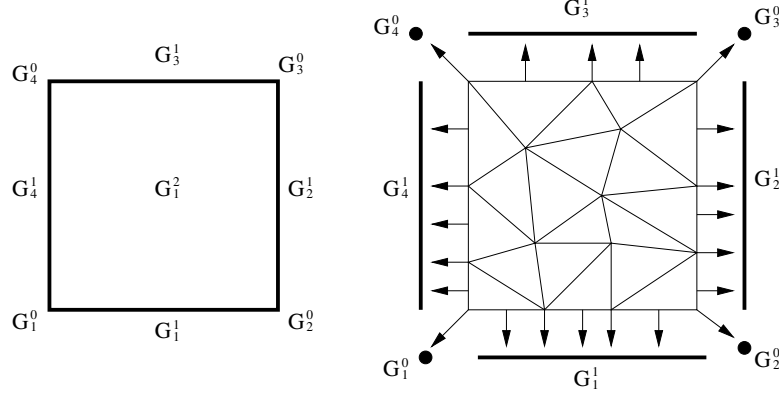
Geometric classification is used to describe the relationship of the mesh with the geometric model. The connection of the mesh to the geometric model is critical for mesh generation and adaptation procedures since it can be used to improve geometric approximations as the mesh is modified [35, 43].

The unique association of a mesh entity of dimension  $d_i$ ,  $M_i^{d_i}$  to a geometric model entity of dimension  $d_j$ ,  $G_j^{d_j}$  where  $d_i \leq d_j$ , is called geometric classification, and is denoted by  $M_i^{d_i} \sqsubset G_j^{d_j}$ , where the classification symbol,  $\sqsubset$ , indicates that the left-hand entity, or set, is classified on the right-hand entity. Multiple mesh entities (say  $M_i^{d_i}$ ) can be classified on one geometric model entity  $G_j^{d_j}$ . Mesh entities are classified with respect to the possible lowest-order geometric model entity [35].

Figure 2.4 illustrates an example of geometric classification. A mesh of a simple square model with entities labeled is shown with arrows, indicating the classification of mesh entities onto the model entities. All interior mesh faces, mesh edges, and mesh vertices are classified on model face  $G_1^2$ .

### 2.2.3 Mesh representation options

Depending on the levels of entities and adjacencies that are explicitly stored in the mesh representation, there are many options in the design of a mesh data structure. The mesh representation can be categorized into two criteria [15, 16, 35, 39]:



**Figure 2.4: Example of simple model(left) and mesh(right) showing their association via geometric classification [34].**

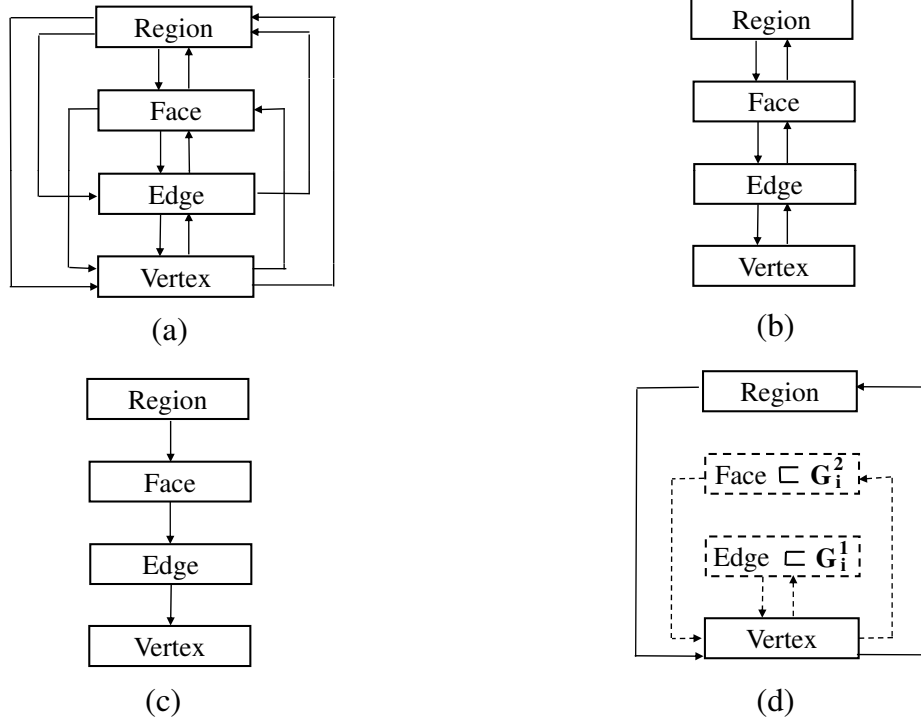
- *Full or reduced*

If a mesh representation stores all 0 to  $d$  level entities explicitly, it is a *full* representation. Otherwise, it is a *reduced* representation.

- *Complete or incomplete*

In terms of the cost of adjacency retrieval, if a mesh representation is able to provide any type of adjacencies requested in a constant time, it is a *complete* representation. If the time required to determine a mesh adjacency is a function of the number of entities in the mesh, or it can not be determined, it is an *incomplete* representation.

Figure 2.5 illustrates adjacency graphs of four 3D mesh representation options: (a) is the *greedy representation* where 4 levels of entities and all possible 12 adjacencies are stored [15, 37]; (b) is the *one-level adjacency representation* that maintains adjacencies between entities one dimension apart [15, 35]; (c) stores mesh entities of all dimensions and maintains only downward adjacencies between entities one dimension apart [15]; (d) is the *complete minimum sufficient representation* that consists of all mesh entities equally classified on the equal dimension model entities and all vertices (*minimum sufficient representation*, MSR [15, 39]), plus upward adjacencies from vertices to their bounding entities of dimension  $> 0$  [15, 36]. In Figure 2.5, solid boxes denote different dimension of mesh entities, and solid arrows denote various adjacencies from outgoing levels to incoming levels. In Figure 2.5d,



**Figure 2.5: Example of 3D mesh representation options.**

the boxes for mesh faces and edges and the lines for adjacency  $\{M^2\{M^0\}\}$  and  $\{M^1\{M^0\}\}$  are dotted since only equally classified faces and edges are stored and adjacency  $\{M^2\{M^0\}\}$  and  $\{M^1\{M^0\}\}$  are maintained only for the existing faces and edges.

For the properties of the mesh representations in Figure 2.5 in terms of full and complete criteria, (a) and (b) are full and complete due to all 0 to  $d$  levels of entities exist and the 12 adjacencies are obtainable in  $O(1)$  time either by direct access or local traversal. (c) is full and incomplete since it requires mesh level global search or traversal to get proper adjacencies. (d) is reduced and complete [15].

In the design of mesh representation for applications, the factors that must be considered are storage and computational efficiency. A mesh representation that explicitly store all topological mesh entities and all the adjacencies between them can perform mesh-level operations and queries efficiently, but it requires a large storage space to store all the information and high computational efforts to maintain proper

adjacencies as the mesh changes [15, 35, 37, 38, 39]. Although it is unnecessary to explicitly store all possible adjacencies in a mesh representation, a complete representation is needed to effectively support adaptive mesh-based applications, since mesh modifications associated with adapting the mesh require accessing the wide range of adjacencies and an  $O(1)$  time for efficiency is critical. In a reduced mesh representation, it is also possible to achieve complete adjacencies through extra software design efforts [15, 35, 36, 39].

## 2.3 Distributed Mesh Data Structure

To meet the requirements of adaptive simulations in a parallel computing environment, maintaining mesh topological adjacencies for a mesh distributed over a large number of processing cores, *processors*, is also needed.

A distributed mesh data structure is an infrastructure which executes underneath and supports all mesh based parallel functionalities, such as communications between mesh entities distributed over processors, and mesh data movement between processors [15, 20, 40, 44].

Key operations using the distributed mesh data structure are (i) dynamic mesh load balancing [44, 45, 46, 47] and (ii) parallel mesh adaptation [20, 21, 48, 49, 50, 51]. The focus here is a general distributed unstructured mesh data structure that is capable of handling general non-manifold models and effectively supporting automated adaptive analysis.

### 2.3.1 Part and part boundary

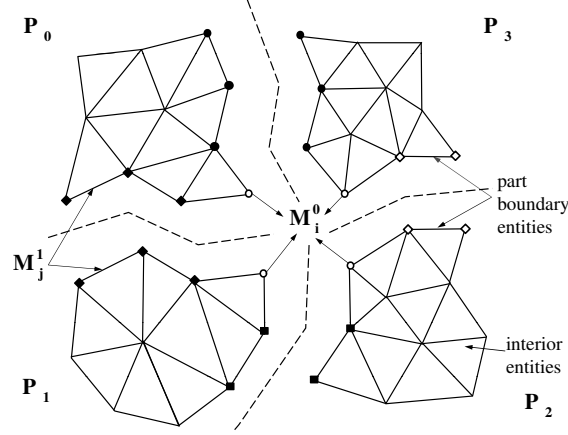
A common approach for unstructured mesh distribution across the processors is through mesh *partition*, which assigns sets of mesh entities, called *parts*<sup>4</sup>, to specific processors.

#### Definition *Part*

A part is a set of mesh entities that is a subset of the entire mesh, and it is uniquely identified by its part identifier (id), denoted by  $P_i$ , where  $0 \leq i \leq N$

---

<sup>4</sup>The original term in the FMDB is *partition* [15, 16], instead of *part*, assuming that one processor has only one part.



**Figure 2.6: A distributed mesh on four processors with one part per processor.**

and  $N$  is the total number of parts. A part will reside on a single processor during parallel computations.

Figure 2.6 depicts a 2D mesh that is distributed to four parts on four processors where each processor contains one part. The dashed lines represent part boundaries between parts.

Each part is treated as a serial mesh with the addition of part boundaries to describe groups of mesh entities on part boundaries, called *part boundary entities*. These mesh entities are duplicated on all parts for which they bound higher order mesh entities and are therefore used in adjacency relations on that part [15, 16]. Mesh entities that are not on any part boundary reside on a single part and are termed as *interior* mesh entities. In the 2D mesh illustrated in Figure 2.6, the part boundary entities include all the mesh vertices and edges duplicated on more than one parts for which they bound mesh faces. For instance, vertex  $M_i^0$  is common to four parts and duplicated on all the part, and mesh edges like  $M_j^1$  are common to two parts and duplicated on part  $P_0$  and part  $P_1$ .

In order to denote the part(s) that a mesh entity resides, an operator is defined to provide the residence parts of a mesh entity [15, 16].

**Definition** *Residence part operator*  $\mathcal{P}[M_i^d]$

An operator that returns a set of part id(s) where a mesh entity  $M_i^d$  exists.



The residence parts of  $M_i^d$ ,  $\mathcal{P}[M_i^d]$ , can be computed through *residence part equation* [15, 16]. If  $\{M_i^d\{M^q\}\} = \emptyset$ ,  $d < q$ , the mesh entity is not part of the boundary of any higher order mesh entities,  $\mathcal{P}[M_i^d] = \{P_i\}$  where  $P_i$  is the id of the single part where  $M_i^d$  exists. Otherwise,  $\mathcal{P}[M_i^d] = \cup \mathcal{P}[M_j^q \mid M_i^d \in \{\partial(M_j^q)\}]$  which indicates the set of part id's for all the mesh entities that  $M_i^d$  bounds.

In parallel adaptive unstructured mesh applications, the mesh and its distributions change all the time. Thus at the mesh level, it is required to maintain the connections between part boundary entities duplicated on different parts. Part boundary entities must be aware of where they are duplicated and what are their duplicated copies [15, 16].

**Definition** *Remote copy*

If a mesh entity  $M_i^d$  on part  $P_i$  is duplicated on another part  $P_j$ , where  $i \neq j$ , then the memory location of the entity on part  $P_j$  is a remote copy of the entity  $M_i^d$  on  $P_i$ , and vice versa.

For instance, in the 2D distributed mesh illustrated in Figure 2.6, the residence parts of mesh vertex  $M_i^0$  are  $\{P_0, P_1, P_2, P_3\}$ , which equals the union of residence parts of its bounding edges. The mesh vertices  $M_i^0$  on part  $P_0, P_1, P_2, P_3$  are remote copies to each other.

For duplicated copies of a part boundary entity, it is necessary to assign a specific copy on one part as the *owner* of the other copies, and let the owner be in charge of the communications or computations between the copies [15, 16]. The entity ownership of a part boundary entity can be determined through two strategies: (i) static ownership where the owner part is always fixed, and (ii) dynamic ownership where the owner part is specified dynamically during parallel computations. For the options to determine the entity ownership, please refer to references [15, 16, 46, 50].

In the case of parallel mesh adaptation applications, especially applications involved with local mesh modifications, communications always happen between mesh entities and their remote copies. To reduce the communication costs in parallel computations, it is useful to perform efficient neighborhood communications [52, 53] between *neighboring parts*.

**Definition** *Neighboring part*

A part  $P_i$  neighbors another part  $P_j$  if there exists a mesh entity on part  $P_i$  that has a remote copy on part  $P_j$ . In other word, part  $P_i$  neighbors another part  $P_j$ , if there exists a mesh entity whose residence parts contain both  $P_i$  and  $P_j$ .

**2.3.2 Partition model**

For the purpose of topological representation of a distributed mesh partitioning and efficient parallel mesh-level operations, a partition model is developed as a conceptual model existing between a geometric model and a mesh. Based on the fact that part boundary entities share a set of residence parts depending on the locations in a partition, a partition model consists of *partition model entities* [15, 16].

**Definition** *Partition (model) entity*

A topological entity in the partition model,  $P_i^d$ , which represents a group of mesh entities of dimension  $d$ , that have the same residence part(s)  $\mathcal{P}$ . Each partition model entity can be uniquely determined by its  $\mathcal{P}$ .

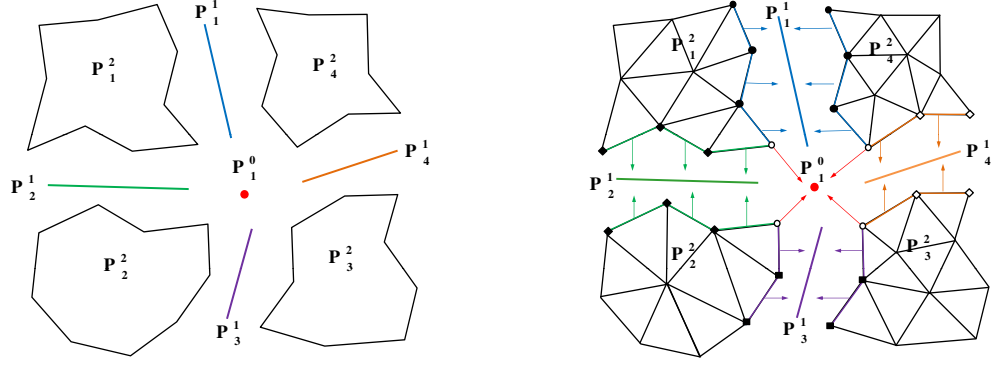
By keeping a proper relation from a mesh entity to a partition model entity, the required parallel mesh operations and inter-part communications on a distributed mesh can be easily supported.

**Definition** *Partition classification*

The unique association of a mesh topological entity of dimension  $d_i$ ,  $M_i^{d_i}$ , to the topological entity of the partition model of dimension  $d_j$ ,  $P_j^{d_j}$ , where  $d_i \leq d_j$ , on which it lies is termed partition classification and is denoted as  $M_i^{d_i} \sqsubset P_j^{d_j}$ .

**Definition** *Reverse partition classification*

For each partition model entity, the set of equal dimension mesh entities classified on that entity defines the reverse partition classification. The reverse partition classification is denoted as  $RC(P_j^d) = \{M_i^d | M_i^d \sqsubset P_j^d\}$ .



**Figure 2.7: Distributed mesh and its association with the partition model via partition classifications.**

Figure 2.7 depicts the 2D distributed mesh in Figure 2.6 and its association with the partition model. The mesh entity arrows indicate the partition classification of the mesh entities onto the partition model entities and its associated partition model. The mesh vertices  $M_i^0$  duplicated on four parts is classified on the partition vertex  $P_1^0$ . Other mesh vertices and edges (like mesh edge  $M_j^1$ ) on part boundaries are classified on partition edges. The remaining mesh entities are not part boundary entities, and they are classified on the partition faces. Note the reverse classification returns only the same dimension mesh entities. For instance, the reverse partition classification of partition edge  $P_1^1$  returns mesh edges located on the thick lines, and the reverse partition classification of partition face  $P_i^2$  returns mesh faces on that part.

## 2.4 Mesh Partitioning

The goal of mesh partitioning is to divide the computational work of a mesh into parts. The computational work is typically associated with specific mesh entities (vertices, edges, faces and regions) or groups of mesh entities and decompositions can be computed with respect to any of these entities or to a combination of the entities (such as, vertices and regions) [54]. To represent these entities, the term of *partition object* is defined.

**Definition** *Partition object*

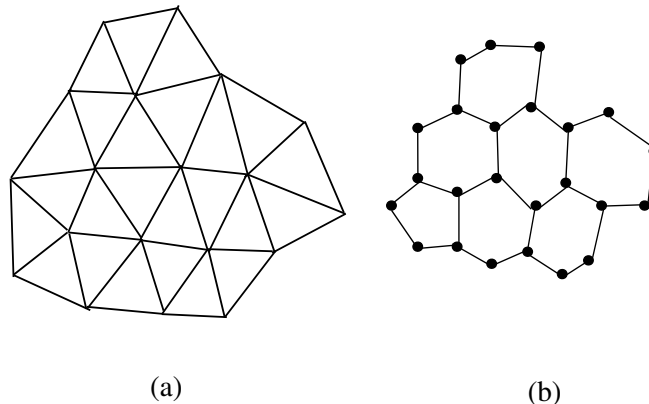
The basic unit to which a destination part id can be assigned through a par-

tioning algorithm. Partition objects are uniquely identifiable data objects in a partition.

For instance, the set of partition objects can be the set of mesh entities that are not part of the boundary of any higher order mesh entities [15, 16]. In the case of a non-manifold model, this includes all mesh regions, the mesh faces not bounded by any mesh regions, the mesh edges not bounded by any mesh faces, and mesh vertices not bounded by any mesh edges [15, 16] (for example, see Fig.2 in reference [16]). In case of a manifold model, partition objects are all mesh regions in 3D and all mesh faces in 2D. In the present work on process, mesh sets can also be considered as partition objects (see later §3.3 for detailed discussions). In the present work, each part contains a subset of the mesh, and the union of the mesh entities on all parts is equal to the entire mesh.

In terms of the computational models used by a partitioning algorithm, there are two commonly used partitioning categories: *coordinate*-based partition [55, 56, 57], and *graph/hypergraph*-based partition [13, 58, 59, 60] partition. Coordinate-based approaches, such as *Recursive Coordinate Bisection* [55] and *Space-Filling Curves* [56, 57], require geometric coordinates and only indirectly consider communication or data movement costs in the computational models. These approaches are fast but do not typically produce an optimal partition for an unstructured mesh.

Graph/hypergraph-based partitioning approaches model the computation and communication costs as a graph/hypergraph, and then divide the graph/hypergraph into weighted subgraphs/hypergraphs through some multilevel heuristic strategies [13, 14, 59, 60]. These methods are well suited for the partitioning of unstructured meshes [54, 59, 61, 62]. In this case, the mesh is used to construct a weighted graph  $G(V, E)$ , where graph nodes  $V$  represent partition objects, and graph edges  $E$  represent dependencies between graph nodes, such as dependencies defined by mesh entity adjacencies. The weights of graph nodes and graph edges represent computation cost and communication cost respectively. In the finite element calculations, defining graph edges built from face/edge type adjacencies was observed to provide a good set of graph edges for partitioning mesh regions/faces (partition objects) in a 3D/2D mesh. Figure 2.8 depicts a 2D mesh and its partition object diagram,



**Figure 2.8: Example of a 2D mesh (left) and its partition object diagram (right).**

where graph nodes are mesh faces, and graph edges are defined by edge adjacencies between mesh faces.

Graph based partitioning is restricted to symmetric data dependencies, while hypergraph partitioning is not. Hypergraph partitioning uses a hypergraph model to add additional dependency information to potentially produce higher quality partitions [58]. In this work, we use the Zoltan library which is a collection of tools for parallel partitioning and load-balancing [13, 14]. It includes a suite of partitioning tools, such as graph-based partitioner PARMETIS [60], Parallel Hypergraph Partitioner (PHG) [58, 59, 63].

For adaptive unstructured mesh applications, the mesh evolves as mesh entities are added and/or removed from the mesh. Dynamic repartitioning of the mesh, also called *dynamic load balancing* [53, 64, 65], is desired for several reasons of (i) balancing the work loads, (ii) minimizing the processor idle time, (iii) letting the work loads not exceed the physical memory limitations on each processor. Although dynamic load balancing has the same goals as general mesh partitioning, it has additional desired features [53] as follows:

- Operate on a already distributed mesh.

- Be as fast as possible as it is often performed frequently.
- Be incremental, in other word, small changes in the mesh only produce small changes in the decomposition.

The third feature is desired since the cost of the mesh redistribution is often the most significant part of a dynamic load balancing step.

## 2.5 Mesh Migration

As the mesh changes, mesh entities need to be periodically redistributed among the parts. The redistribution requires the mesh entities being transferred from one part to another part. This process is referred to as *mesh migration*.

An efficient mesh migration algorithm has been developed to support parallel adaptive mesh-based simulations in the FMDB [15, 16]. The original distributed mesh data structure was built on top of the partition model, assuming one part per processor [15, 16]. With the redistribution information from either (i) the partitioning result from a partitioning tool (herein, Zoltan library) or (ii) mesh modification operation request migration of specific entities, the pre-processing step converts the redistribution information into a list of partition objects to migrate and their destination part ids, referred to as *POsToMove*, as the input of the mesh migration algorithm. The migration procedure performs the following steps [15, 16]:

**Step 1:** Given the *POsToMove*, collect a set of entities to be updated (*EntitiesToUpdate*) and clear the partitioning data (partition classification) of them.

**Step 2:** Determine residence parts, update the partition classification of entities in *EntitiesToUpdate*, and collect entities to remove from the local parts(*EntitiesToRemove*).

**Step 3:** Exchange the entities contained in *EntitiesToUpdate* and update the remote copies of affected entities.

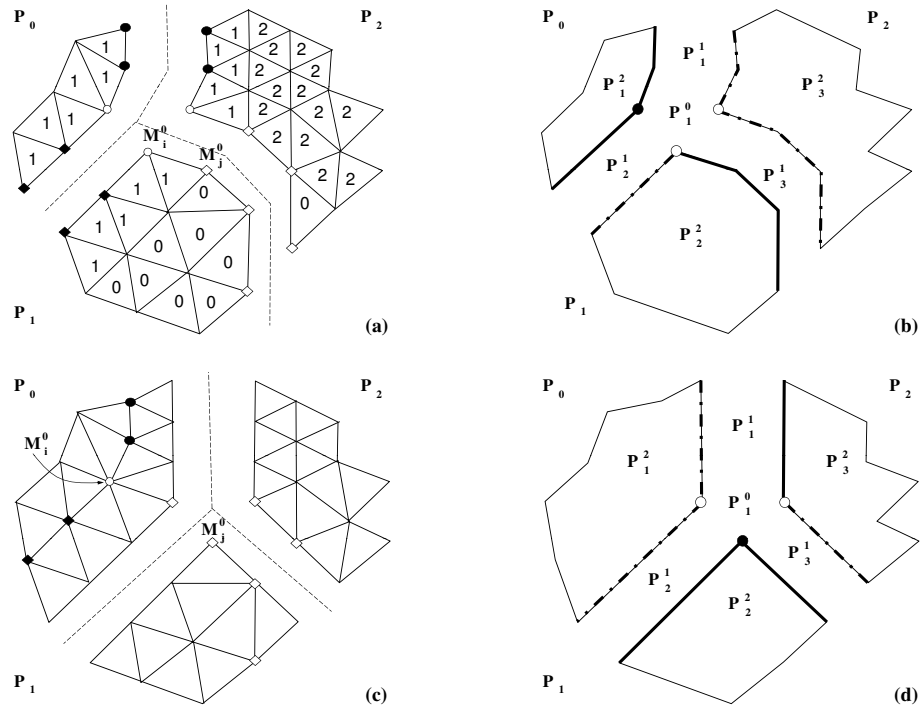
**Step 4:** Remove unused entities in *EntitiesToRemove*.

**Step 5:** Update the owner parts of partition model entities in the partition model.

Given a list of partition objects to migrate, *Step 1* collects the entities whose partitioning data (i.e., residence parts  $\mathcal{P}$ , partition classification) will be updated after migration. In *Step 2*, after the computation of residence parts  $\mathcal{P}$  of these entities, their partition classifications are updated to reflect the changes in the partition model. This step also determines and collects the entities to remove (*EntitiesToRemove*) from the local parts based on their residence parts  $\mathcal{P}$ . *Step 3* transfers the required entities to the destination parts. After exchanging entities, the adjacency relations associated with the newly created entities are properly maintained on destination parts, and the remote copy information of the affected part boundary entities are updated. In *Step 4*, entities collected in *EntitiesToRemove* are deleted from the local parts. The adjacency relations associated with the unused entities are deleted also. For any part boundary entity in *EntitiesToRemove*, it will be removed from other parts where it is kept as for remote copies. In the final step (*Step 5*), the owner parts of partition entities are updated. The technical aspects of doing this procedure efficiently in parallel for one part per processor are given in [15, 16].

Figure 2.9 illustrates a 2D partitioned mesh and its associated partition model to be used for an example of mesh migration. Figure 2.9a is the initial partitioned mesh where mesh faces (i.e. partition objects) are labeled with their destination parts. Figure 2.9b is the partition model associated with the initial mesh in Figure 2.9a, where the owner part of a partition model vertex (edge) is illustrated with a solid black circle (line). For instance, the owner part of partition vertex  $P_1^0$  is part  $P_0$ . Therefore  $P_1^0$  on  $P_0$  is depicted with a solid black circle, while  $P_1^0$  on  $P_1$  and  $P_2$  is depicted with a hollow circle. Part boundary vertices  $M_i^0$  and  $M_j^0$  are classified on partition entity  $P_1^0$  and  $P_3^1$  respectively. Figure 2.9c is the final partitioned mesh after migrating the labeled partition objects to the destination parts. Figure 2.9d is the final partition model associated with the mesh in Figure 2.9c. In Figure 2.9d, the owner parts of partition model entities are updated. For instance, the owner part of  $P_1^0$  is updated to  $P_1$ .  $M_i^0$  becomes an interior entity, and  $M_j^0$  becomes classified on the partition vertex  $P_1^0$ .

Mesh migration is critical in supporting of parallel adaptive unstructured mesh applications that include mesh modification operations [16, 20, 21, 51]. For instance,



**Figure 2.9: Example of 2D mesh migration: (a) initial mesh with partition objects labeled with their destination parts; (b) partition model of (a); (c) final mesh; (d) final partition model with ownership.**

when a mesh modification operation (such as, mesh coarsening or swapping and etc.) on part boundaries is desired, migrating the mesh entities within the polyhedron associated with the operation onto one part is required to apply the operation.

Migrating the user data associated with the mesh can be achieved through specifying callback functions [15, 51] in the migration. This mechanism allows the user to specify how to pack the user data within the message packing procedure and how to unpack and attach the user data received from remote parts.



## CHAPTER 3

### GENERIC COMPONENTS

In the traditional programming paradigm, data structures and algorithms are developed for specific data type and requirements. This leads to code redundancy and inflexibility thus not allowing effective code reuse for similar applications. One effective approach to increase code reuse is through generic programming, which focuses on the development of efficient, reusable software libraries through suitable abstractions for common requirements. In this chapter, we present how we applied generic programming to an on-going effort in support of mesh-based adaptive simulations on massively parallel computers. Three generic components, iterator, set and tag, were developed using design pattern, C++ template programming and the Standard Template Library (STL). The scaling studies on massively parallel supercomputer demonstrates the efficiency of the reusable, generic components which do not sacrifice the performance of the previous tools developed in the traditional object-oriented programming paradigm.

The rest of the chapter is organized as follows. §3.1 introduces the application of generic programming in the scientific computing software development. §3.2 analyzes the abstract data models of a mesh-based simulation, and summarizes a set of generic components for adaptive mesh-based simulations. Sections §3.3 - §3.5 present the three generic components developed, and describe how these generic components are used to support automated adaptive simulations. §3.6 presents the performance results of adaptive simulations on massively parallel computers, using the generic components.

### 3.1 Generic Programming in Scientific Computing Software

The generic programming [17, 66] paradigm has emerged as a methodology towards developing efficient, reusable component-based software libraries, and gained popularity through the success of the C++ *Standard Template Library* (STL) [17, 67, 68]. Generic programming has been applied to scientific computing software

packages such as:

- Matrix Template Library (MTL): a generic component library for high performance numerical linear algebra [69],
- Generic Message Passing framework (GMP): message communication library [70],
- Grid Algorithms Library (GrAL): a generic grid toolbox for reusable mesh-level components [71, 72],
- Computational Geometry Algorithms Library (CGAL): a library for general-purpose geometric data structures and algorithms [73],
- A generic grid interface of parallel and adaptive scientific computing implemented in DUNE [74, 75],
- A layer of generic software components used for parallelization of the finite element solver and for solver coupling in multi-physics applications [76].

The software components for an adaptive analysis of partial differential equations (PDEs) include [22, 23, 34]: *(i)* the geometric model which houses the topological and shape description of the domain of the problem, *(ii)* attributes describing the rest of parameters needed to define and quantify the problem, *(iii)* the mesh which describes the discretized representation of the domain used by the analysis method, and *(iv)* fields which describe the distribution of solution tensors over the mesh entities.

The most common approach for developing reusable simulation software is to create libraries for specific data components such as mesh, geometric model, and field and let them interact through well-defined API's to perform needed operations to accomplish scientific applications. The Interoperable Technologies for Advanced Petascale Simulations (ITAPS) center [9] has defined a set of common interfaces that support the abstract data model [9, 11, 12], including Geometry (*iGeom*), Mesh (*iMesh/iMeshP*), Field (*iField*), Data Relation Manager (*iRel*) and one that contains the utilities and definitions used by other interfaces (*iBase*). A common set of utilities used in the functional components are iterator, set and tag.

## 3.2 Simulation Data Model Analysis

The four data models central to the general numerical solution of PDEs with mesh based methods are:

- *Geometric model*: geometric model interface that supports the ability to interrogate solid models for topological adjacency and geometric shape information.
- *Mesh*: domain discretization that provides mesh-based operations.
- *Field*: representation of tensor fields to quantify the distribution of physical parameters over mesh entities.
- *Relationship manager*: utility used to manage the relationships between meshes and geometric models, tensor fields and meshes, and so on.

The design of reusable and efficient generic components with appropriate concepts for mesh-based simulations is availed from understanding the common requirements of essential data models for parallel mesh-based simulations as well as a suitable level of abstractions in the form of *concepts* [17] of such requirements with ultimate balance between commonality and specialization.

In addition to the four data models described above, adaptive simulations in a parallel computing environment place extra demands to represent and manipulate the distributed mesh data over a large number of processing cores (i.e. *processors*). The subsections that follow present the data model and functional requirements of the geometric model, and mesh, and distributed mesh as a first step towards identifying essential, reusable generic components.

Based on the data model and requirement analysis from §2.1 in Chapter 2, a set of generic components to support mesh level operations in adaptive mesh-based simulations on a massively parallel computing environment includes:

***Set***: component for grouping arbitrary data with common set requirements.

***Iterator***: component for iterating over a range of data.

***Tag***: component for attaching arbitrary user data to arbitrary data or set with common tagging requirements.

A generic component can be reused in various situations in which the concepts of the component are met as a minimal set of requirements and associated types. For instance, the set component can be used on geometric entities, or mesh entities, and so on. The iterator and tag components can be used on the geometric model, the mesh, or sets.

The following sections (Sections 3.3 - 3.5) present the design and implementation of the three components - set, iterator, and tag - which are in use as utilities. They also illustrate how the three generic components were used to implement various iterator, set and tagging needs in the Flexible distributed Mesh DataBase (FMDB) and Geometric Model Interface (GMI), a distributed mesh and geometric model data infrastructure, to support parallel adaptive simulations [15, 16, 23].

### 3.3 Set Component

A set is a collection of objects or a container of data objects where a set can contain other sets. Data objects from which a given set is composed are called elements or members of the set. Each non-set data member in a set can have relations to each other. A common relation useful in most applications is *ordering*. Depending on whether there is the need to preserve the insertion order for non-set data members or not, the two set types are defined [9, 12, 30]:

**Ordered set:** if any two data members are comparable in terms of the insertion ordering, a set is an ordered set, which can contain duplicate data members.

**Unordered set:** if the insertion ordering is not preserved, a set is an unordered set. An unordered set contains unique data members.

#### 3.3.1 Design and implementation

To support set needs of parallel adaptive simulations, the primary constituents of the set component include: (i) *set handle* for holding multiple arbitrary data of the same type, (ii) *set holder* for maintaining all active sets uniquely identified by the set handles, and (iii) *settable object* which models data containable in a set.

The syntax of set API's for basic set operations, such as set creation/deletion, set existence/type check, set data insertion/deletion and so on, is listed in the

following. Two Standard Template Library (STL) containers [17, 77], *std::list* and *std::vector*, are used as data types for the output arguments with multiple data.

```
// create set and store it in set holder
// type: ordered or unordered
template<typename Entity>
int SetHolder_CreateSet (SetHolder<Entity>*,
int type, Set<Entity>*);

// delete set and remove it from set holder
template<typename Entity>
int SetHolder_DelSet (SetHolder<Entity>*,
Set<Entity>*);

// check whether set exists in set holder
template<typename Entity>
int SetHolder_HasSet (SetHolder<Entity>*,
Set<Entity>*, int* exist);

// get a list of sets contained in set holder
template<typename Entity>
int SetHolder_GetSet (SetHolder<Entity>*,
vector<Set<Entity>* >&);

// get # sets contained in set holder
template<typename Entity>
int SetHolder_GetNumSet (SetHolder<Entity>*,
int* num);

// get type of set (ordered or unordered)
template<typename Entity>
int Set_GetType (Set<Entity>*, int* type);

// check whether set has data
```

```

template<typename Entity>
int Set_HasEnt (Set<Entity>*, Entity*, int* exist);

// get # data contained in set
template<typename Entity>
int Set_GetNumEnt (Set<Entity>*, int* num);

// insert data into set
template<typename Entity>
int Set_AddEnt (Set<Entity>*, Entity*);

// insert multiple data into set
template<typename Entity>
int Set_AddEntArr (Set<Entity>*, vector<Entity*>&);

// remove data from set
template<typename Entity>
int Set_RmvEnt (Set<Entity>*, Entity*);

// remove multiple data from set
template<typename Entity>
int Set_RmvEntArr (Set<Entity>*, vector<Entity*>&);

```

Herein, *Entity* is a concept modeling a piece of data in a data model. In the current unstructured mesh applications, *Entity* can be a mesh entity in a mesh, or a model entity in a geometric model. Given a desired set type (ordered or unordered), the function *SetHolder\_CreateSet* creates a set and stores its handle in the set holder object provided.

To implement the data uniqueness and order preservation characteristics of ordered/unordered sets efficiently, two STL containers, *std::set* and *std::list*, are ideal for an unordered set and an ordered set, respectively. The *factory method* design pattern [78], an object-oriented design method to define an interface for creating a class object with yielding instantiation to sub-classes with specialization, is used

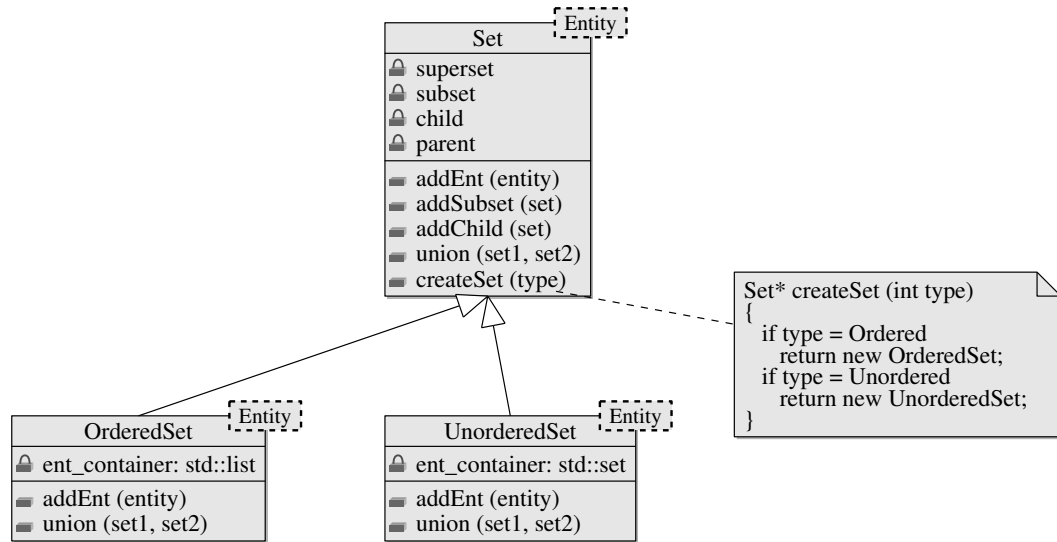


Figure 3.1: Class diagram of the generic set component.

to support the dynamic container data structure selection upon the set type input at runtime. As illustrated in Figure 3.1, the base class (*Set*) provides a uniform interface (function *createSet*) and the factory method enables creation of ordered or unordered set dynamically.

In addition to the general set functionality described above, it's also desirable to support set relations (parent/child and superset/subset) and set binary operations such as union, subtraction, and intersection for flexible set manipulation. The following are the API's for set binary operations.

```

// set operations: third argument is output set
template<typename Entity>
int Set_Unite (Set<Entity>*,
              Set<Entity>*, Set<Entity>*);

template<typename Entity>
int Set_Intersect (Set<Entity>*,
                  Set<Entity>*, Set<Entity>*);

template<typename Entity>
int Set_Subtract (Set<Entity>*,
                  Set<Entity>*, Set<Entity>*);

```

The following are the API's for set relations.

```
// superset-subset operations
template<typename Entity>
int Set_AddSuperSub (Set<Entity>*, Set<Entity>*);

template<typename Entity>
int Set_RmvSuperSub (Set<Entity>*, Set<Entity>*);

template<typename Entity>
int Set_IsSubOf (Set<Entity>*, Set<Entity>*, int* isSub);

template<typename Entity>
int Set_GetNumSuper (Set<Entity>*, int* num);

template<typename Entity>
int Set_GetNumSub (Set<Entity>*, int* num);

template<typename Entity>
int Set_GetSuper (Set<Entity>*, list<Set<Entity>*>&);

template<typename Entity>
int Set_GetSub (Set<Entity>*, list<Set<Entity>*>&);

// parent-child operations
template<typename Entity>
int Set_AddPrntChld (Set<Entity>*, Set<Entity>*);

template<typename Entity>
int Set_RmvPrntChld (Set<Entity>*, Set<Entity>*);

template<typename Entity>
int Set_IsChldOf (Set<Entity>*, Set<Entity>*,
                 int* isChld);
```



```

template<typename Entity>
int Set_GetNumChld (Set<Entity>*, int* num);

template<typename Entity>
int Set_GetNumPrnt (Set<Entity>*, int* num);

template<typename Entity>
int Set_GetChldn (Set<Entity>*, list<Set<Entity>*>&);

template<typename Entity>
int Set_GetPrnts (Set<Entity>*, list<Set<Entity>*>&);

```

### 3.3.2 Application

Due to the property of a mesh entity, which must be assigned to a part for management, there are two types of entity sets used in unstructured mesh applications depending on the number of parts associated with the entities contained in a set: (i) a set of entities residing on a single part, and (ii) a set with entities spanning multiple parts. In support of parallel anisotropic adaptive meshing based on mesh metric fields with adaptive boundary layer meshes [4], a stack of boundary mesh entities is required to be on a single part for mesh modification, partition and migration, and it is useful to differentiate an entity set with the single part restriction from an entity set without such a restriction. Please refer to Chapter 4 for the detailed discussions of mesh sets.

#### 3.3.2.1 Mesh entity set for iMesh:

Using the set component with a mesh instance as a *SetHolder*, all requirements of *iMesh* sets are easily supported [9]; (i) populating by addition or removal of entities into or from the set, (ii) traversal through an iterator with various conditions such as type, or topology of an entity, (iii) set binary operations (union, subtraction, intersection), and (iv) relationships among sets such as superset/subset and parent/child.

### 3.4 Iterator Component

An iterator is a generalization of a pointer, an object that points to another object used to traverse over a range of objects. When an iterator is within a range, the increment operator moves the iterator to the next object [17, 77]. In the STL, the concept *iterator* models an object which traverses over a single one-dimensional container [17, 77].

#### 3.4.1 Design and implementation

To support iterator needs of parallel adaptive simulations, in addition to the general iterator functionality such as initialization, advancement, reset, position check, and iterator deletion, the iterator component should support: *(i) filtering*: skipping unwanted data over traversal with various conditions specifiable by the user, and *(ii) resilience*: validity with data modification such as data insertion or deletion.

The following are the API's for iterator initialization, advancement, reset, the position check, and iterator deletion. The output of API functions is an integer value notifying success (*zero*) or failure (predefined non-zero error code) to the user.

```
// iterator initialization
template<typename Iterator, typename Entity>
int Iter_Init (const Iterator& first,
               const Iterator& last,
               int type, int topo, void* ptr,
               void (*Functor)(Iterator&, Iterator&,void*,int,int),
               Iter<Iterator, Entity>*);

// iterator advancement
template<typename Iterator, typename Entity>
int Iter_GetNext (Iter<Iterator, Entity>*, Entity*);

// iterator reset
template<typename Iterator, typename Entity>
int Iter_Reset (Iter<Iterator, Entity>*);
```

```

// iterator position check
template<typename Iterator, typename Entity>
int Iter_IsEnd (Iter<Iterator, Entity>*, int*);

// iterator deletion
template<typename Iterator, typename Entity>
int Iter_Del (Iter<Iterator, Entity>*);

```

Herein, *Entity* is a concept which is to represent a piece of data in a data model, for instance, it can be a mesh entity in a mesh, a model entity in a geometric model, or a mesh entity in a set. In a distributed mesh environment, mesh entity traversals only consider mesh entities within a single part with an iterator to avoid extra communication costs.

In *Iter\_Init* function, given the input arguments consisting of (i) a data range represented by an iterator pair  $[first, last)$ <sup>1</sup>, (ii) type (dimension), (iii) topology, (iv) *void\** type pointer (*ptr*), and (v) user-defined filtering function pointer (*Functor*), an iterator instance is created and returned. The entity data which satisfies the specified type, topology and filtering function pointer requirements within the data range is traversed with the iterator. *ptr* is reserved for any user-defined data structure that can be casted into a *void\** type pointer, and can be an empty *null* pointer if unnecessary.

The function *Iter\_GetNext* fetches the entity data pointed by the current iterator, and then advances the iterator to the next available entity data. If the iterator reaches to the end of the data range, the function *Iter\_GetNext* returns a specific pre-defined non-zero error code.

Unlike an STL iterator [77], the iterator component for adaptive unstructured mesh simulations should be able to traverse multiple containers through a single iterator since the topological model data is usually stored in multiple containers per type. For instance, mesh entities can be stored in four containers with vertices, edges, faces and regions being in separate containers. To support traversing multi-

---

<sup>1</sup>The notation  $[first, last)$  refers to all the iterators from *first* up to, but not including, last [77].

ple containers with a single iterator, a *linking* method is developed in the iterator advancement operator with which the end of the previous one-dimensional container is connected to the beginning of the next container.

The following is the pseudo-code to traverse multiple containers with a single iterator.

```
// cur_pos is the current iterator position
if cur_pos == current_container.end
    if (current_container.end != data_range.end)
        cur_pos = next_container.begin;
    else
        return;
else
    advance cur_pos;
```

### 3.4.2 Applications

To support parallel adaptive simulations, the minimum set of iterators needed in a mesh and geometric model includes *(i)* mesh entity traversal in a part by type and/or topology, *(ii)* mesh entity traversal by reverse geometric classification, *(iii)* part boundary mesh entity traversal, *(iv)* mesh entity traversal in a set by type and/or topology, and *(v)* geometric entity traversal in the entire geometry by type. For all cases, the iterator range for mesh entities is limited to entities on a single part or a set.

In the FMDB, a single part uses an *std::set* container to store mesh entities in terms of entity duplicate prevention. To be consistent with the *std::set* iterator specification [77], an iterator is guaranteed to work properly in the presence of data modifications with one exception, the case when the entity the iterator is currently pointed at is deleted. However, even in such cases, on deletion of a mesh entity currently being pointed by the iterator, the invalid iterator problem can be avoided through a simple approach, which is to advance the iterator on entity deletion to the next (not already deleted) entity member it would have proceeded to. In this way, an iterator is guaranteed to always point at a valid entity member or to the

end of the *std::set* container, in the case that the member being removed is the last one.

#### 3.4.2.1 Mesh entity traversal in part by type and/or topology:

An iterator for a mesh entity traversal is implemented by providing the first and one past the end of an entity container as for the data range based on the requested entity dimension and topology. If a multiple dimensional mesh entity traversal is needed, the linking method is used to traverse multiple mesh entity containers, in which mesh entities are stored per dimension.

To traverse mesh entities in a part by type and/or topology, the input of the iterator creation API only requires the entity dimension and entity topology, so various combinations of the type and topology pairs are possible for entity filtering. Users can also specify *all* types (*ALLTYPE*) and all topologies (*ALLTOPO*).

The following code illustrates how to initialize an iterator to implement a mesh entity traversal in a part. For each part, the entity container of dimension  $i$  is denoted as  $part \rightarrow container.[i]$  where  $i = \{\text{VERTEX, EDGE, FACE, REGION}\}$ . For each entity container of dimension  $i$ ,  $part \rightarrow container[i].begin$  and  $part \rightarrow container[i].end$  denote the first and one past the end of the container, respectively.  $pPart$  is a pointer type to a part, and  $mEntity$  is the class name of mesh entity.

```
typedef entity_container_iterator_type part_iter;
typedef Iterator<part_iter, mEntity>* pPartEntIter;

// iterator initialization
int FMDB_PartEntIter_Init (pPart part, int type,
                          int topo, pPartEntIter& iter)
{
    if type==ALLTYPE
        return Iter_Init (part->container[VERTEX].begin,
                          part->container[REGION].end, type, topo,
                          (void*)part, &EntityProcessFilter, iter);
    else
        return Iter_Init (part->container[type].begin,
```

```

        part->container[type].end, type, topo,
        (void*)part, &EntityProcessFilter, iter);
}

```

Herein, if the input entity type is *ALLTYPE*, the iterator input range is specified as all the mesh entities of all dimensions, denoted by *part*→*container*[*VERTEX*].*begin* and *part*→*container*[*REGION*].*end*. Otherwise, the iterator input range is specified as mesh entities of a specific dimension, denoted by *part*→*container*[*type*].*begin* and *part*→*container*[*type*].*end*.

For readability, the predefined data type of an entity container iterator on each part is *part\_iter* and *pPartEntIter* is a pointer to the template class *Iterator*<*part\_iter*, *mEntity*>. The following pseudo code illustrates the function *EntityProcessFilter*, which moves the iterator to the next proper position that is a pointer to the next mesh entity satisfying the type and topology criterion requested.

```

void EntityProcessFilter (part_iter& ibegin,
    part_iter& iend, void* ptr, int type, int topo)
{
    if ibegin==iend
        return;
    for each entity in range [ibegin, iend)
        if entity->type==type && entity->topology==topo
            ibegin = current_entity_position;
            return;
    ibegin=iend;
}

```

The following code illustrates how the iterator component is used to implement other iterator functionalities in a mesh entity traversal in a part.

```

// iterator advancement
int FMDB_PartEntIter_GetNext
    (pPartEntIter iter, mEnt* ent)
{

```

```

    return Iter_GetNext (iter, ent);
}

// iterator position check
int FMDB_PartEntIter_IsEnd
    (pPartEntIter iter, int* isEnd)
{
    return Iter_IsEnd(iter, isEnd);
}

// iterator deletion
int FMDB_PartEntIter_Del (pPartEntIter iter)
{
    return Iter_Del (iter);
}

// iterator reset
int FMDB_PartEntIter_Reset (pPartEntIter iter)
{
    return Iter_Reset (iter);
}

```

#### 3.4.2.2 Mesh entity traversal by reverse classification:

For the input geometric entity of dimension  $d$ , mesh entity traversal by reverse geometric classification is implemented using the iterator component with the filtering function that checks the geometric classification of each mesh entity on traversal. Note the cost of the reverse classification through iterator is  $O(n)$  where  $n$  is the number of mesh entities in a part, while the reverse classification can be obtained in  $O(1)$  cost using mesh adjacencies.

The following FMDB API illustrates how to initialize an iterator for a mesh entity traversal by reverse classification in a part.

```

// iterator initialization

```

```

int FMDB_PartEntIter_InitRevClas (pPart part,
    pGeomEnt geomEnt, int type, pPartEntIter& iter)
{
    if type==ALLTYPE
        return Iter_Init (part->container[VERTEX].begin,
            part->container[REGION].end, type, topo,
            (void*)part_ent, &GeomClasProcessFilter, iter);
    else
        return Iter_Init (part->container[type].begin,
            part->container[type].end, type, topo,
            (void*)part_ent, &GeomClasProcessFilter, iter);
}

```

Herein, *pGeomEnt* is a pointer type to a geometric entity. If the input entity type is *ALLTYPE*, the iterator input range is specified as the mesh entities of all dimensions. Otherwise, the iterator input range is specified as mesh entities of a specific dimension. Using a data structure *part\_ent* that is casted into *void\** type and contains a pair of local part and geometric entity, the input filtering function *GeomClasProcessFilter* is used to increment the iterator to the next proper position that points to a mesh entity classified on a given geometric entity *geomEnt* in the range. Note that the local part stored in the data structure *part\_ent* is used by the linking method to traverse multiple entity containers in a part. The following is the pseudo code of function *GeomClasProcessFilter*.

```

void GeomClasProcessFilter(part_iter& ibegin,
    part_iter& iend, void* ptr, int type, int topo)
{
    ent = (cast<part_geomEnt*> ptr)->second;
    if ibegin==iend
        return;
    for each entity in range [ibegin, iend)
        if entity->geometric_classification
            ==cast<pGeomEnt> ent
            ibegin = current_entity_position;
}

```



```

        return;
    ibegin=iend;
}

```

An example usage is to calculate the number of mesh entities classified on a specific geometric entity through the FMDB API *FMDB\_GeomEnt\_GetNumRevClas*. The pseudo code of Algorithm 1 illustrates the procedure, which consists of three main steps: (i) initializing an iterator based on the input part, geometric entity (*geomEnt*) and entity type, (ii) advancing the iterator one step forward in the entity traversal loop, and (iii) deleting the iterator to avoid memory leak.

```

Data: part, geomEnt, type
Result: store the number of mesh entities classified on geomEnt into numEnt
begin
    /* STEP 1: initialize an iterator */
    numEnt ← 0;
    iterEnd ← FMDB_PartEntIter_InitRevClas(part, geomEnt, type, iter);
    /* STEP 2: traverse mesh entities in a loop */
    while iterEnd = false do
        iterEnd ← FMDB_PartEntIter_GetNext(iter, meshEnt);
        if iterEnd = true then break;
        ++numEnt;
    end

    /* STEP 3: delete the iterator */
    FMDB_PartEntIter_Del(iter);
    return numEnt;
end

```

**Algorithm 1: Example of mesh entity traversal by reverse classification:** to calculate the number of mesh entities classified on a specific geometric entity in a part.

### 3.4.2.3 Part boundary mesh entity traversal:

A part boundary entity traversal is implemented using the iterator component with the filtering function which checks the duplicated copy existence of each mesh entity (also referred to as *remote copy* [15, 16]) on traversal. Given the input arguments consisting of entity type, topology and non-local part id, called *target\_part\_id*,

the iterator creation API initializes an iterator to traverse mesh entities duplicated on the part boundary between the local part and target part.

```
int FMDB_PartEntIter_InitPartBdry (pPart part,
    int target_part_id, int type, int topo,
    pPartEntIter& iter)
{
    part_pid=pair<part, target_part_id>;
    if type==ALLTYPE
        return Iter_Init (part->container[VERTEX].begin,
            part->container[REGION].end, type, topo,
            (void*)part_pid, &PartBdryProcessFilter, iter);
    else
        return Iter_Init (part->container[type].begin,
            part->container[type].end, type, topo,
            (void*)part_pid, &PartBdryProcessFilter, iter);
}
```

Using a data structure *part\_pid* that is casted into *void\** type and contains a pair of local part and target part id, the function *PartBdryProcessFilter* moves the iterator to the next proper position, which points to a mesh entity that is on the part boundary between the local part and target part with the help of partition classification.

#### 3.4.2.4 Mesh entity traversal in set by type and/or topology:

Similar to a mesh entity iterator in a part described in §3.4.2.1, a mesh entity iterator in an entity set is implemented using the iterator component with the input range  $[set \rightarrow container.begin, set \rightarrow container.end)$ , and the function *EntityProcessFilter*, where *set*→*container* denotes the entity container for a set.

#### 3.4.2.5 Geometric entity traversal by type:

In the Geometric Model Interface (GMI) [79], an *std::vector* container is used to store the model entities of a given dimension. Given the geometric model (*model*)

and entity dimension (*type*), the following GMI API illustrates how to initialize an iterator for a model entity traversal.

```
typedef model_entity_container_iterator_type model_iter;
typedef Iterator<model_iter, gEntity>* pGeomEntIter;

// iterator initialization
int GMI_GeomEntIter_Init (pGeomMdl model, int type,
                        pGeomEntIter& iter)
{
    if type==ALLTYPE
        return Iter_Init (model->container[VERTEX].begin,
                        model->container[REGION].end, type, 0,
                        (void*)model, &GEntityProcessFilter, iter);
    else
        return Iter_Init (model->container[type].begin,
                        model->container[type].end, type, 0,
                        (void*)model, &GEntityProcessFilter, iter);
}
```

Herein, *pGeomMdl* is a pointer type to a geometric model, and *gEntity* is a class of geometric entity. For readability, *model\_iter* is a predefined data type of an iterator of the geometric model's entity container. *pGeomEntIter* is a predefined data type of a pointer to the template class *Iterator<model\_iter, gEntity>*.

Similar to a mesh entity iterator in a part described in §3.4.2.1, the model entity container of dimension *i* is denoted as *model*→*container*[*i*] where *i*={VERTEX, EDGE, FACE, REGION}. If the input entity type is *ALLTYPE*, the iterator's range is specified as the mesh entities of all dimensions (*VERTEX* to *REGION*). Otherwise, the iterator's range is specified as the input. The geometric entity filtering function *GEntityProcessFilter* moves the iterator to the next proper position in the geometric model.

The following code illustrates how the iterator component is used to implement iterator advancement, reset, the position check, and deletion functionalities in the geometric model.

```

// iterator advancement
int GMI_GeomEntIter_GetNext
    (pGeomEntIter iter, pGeomEnt ent)
{
    return Iter_GetNext (iter, ent);
}

// iterator position check
int GMI_GeomEntIter_IsEnd
    (pGeomEntIter iter, int* isEnd)
{
    return Iter_IsEnd(iter, isEnd);
}

// iterator deletion
int GMI_GeomEntIter_Del (pGeomEntIter iter)
{
    return Iter_Del (iter);
}

// iterator reset
int GMI_GeomEntIter_Reset (pGeomEntIter iter)
{
    return Iter_Reset (iter);
}

```

### 3.5 Tag Component

Tags are used as containers of arbitrary user-defined data that can be attached to the geometric model, geometric model entities, mesh instance, part, mesh entities, sets, and fields. Different values of a particular tag can be associated with different data models, entities or sets [9, 12, 30].

### 3.5.1 Design and implementation

The tag component consists of (i) *tag data* for representing arbitrary user data, (ii) *tag handle* for holding a unique tag identifier attachable to primary data, (iii) *tag holder* for maintaining all active tags identifiable with handle, and (iv) *taggable object* which models the primary data to which tag data is attached with a tag handle.

Each tag handle is uniquely identified by a pair of belonging tag holder and string tag name and has two attributes, (i) *tag data type* which is primary type (integer, double, entity and set) or arbitrary type data, and (ii) *tag size* which specifies the number of data in tag data. If the tag size is 1, the tag data holds one single piece of data of given tag type. If the tag size is greater than 1, the tag data holds an array of data of given tag type.

The following are the API's for tag handle creation/deletion and various queries for tag handles within a tag holder.

```
// given tag name, tag type, and tag size,
// create tag handle and store it in tag holder
TagHandle* TagHolder_CreateTag
(TagHolder*, const char* name, int type, int size);

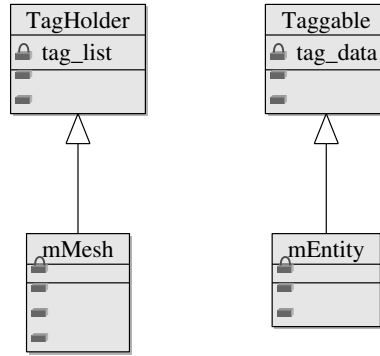
// delete tag handle and remove it from tag handle
int TagHolder_DelTag (TagHolder*, TagHandle*);

// remove all tags from tag handle
void TagHolder_ClearTag (TagHolder*);

// check tag type matches given type info
int TagHolder_CheckTag (TagHolder*, TagHandle*,
    int tag_type);

// check tag exists in tag holder
int TagHolder_HasTag (TagHolder*, TagHandle*,
    int *exist);
```





**Figure 3.2:** Class diagram of the mesh and mesh entity inherited from the tag component's classes.

```

template<typename Type>
void Taggable_GetByteArrData (Taggable*, TagHandle*,
void** data, int* data_size);
  
```

### 3.5.2 Application

To support tag handles created per mesh instance, the mesh class, *mMesh*, inherits from the class *TagHolder*. To support efficient tag data access and automatic tag data removal along the taggable object deletion, part, mesh set and mesh entity classes inherit from the class *Taggable*. In Figure 3.2, class diagram illustrates *mMesh* (the mesh class) inherited from *TagHolder* and *mEntity* (the mesh entity class) inherited from *Taggable*, which are to support tag handles per mesh instance and tagging functionality to mesh entities, respectively.

The following is the pseudo code to create a tag handle of specific name, type and size. *pMeshMdl* is a pointer type to mesh data which is identical to *mMesh\**.

```

int FMDB_Mesh_CreateTag (pMeshMdl mesh,
    const char* tag_name, int tag_type,
    int tag_size, TagHandle* tag)
{
    if tag with given tag_name exists in mesh
        tag = existing_tag;
    else
        tag = TagHolder_CreateTag(cast<TagHolder*>(mesh),
  
```

```

        tag_name, tag_type, tag_size);
    return SUCCESS;
}

```

Herein, if a tag handle of the specific name, type and size already exists in the tag holder object, i.e. the mesh, the function just returns the existing tag handle. Otherwise, it creates a new tag handle and stores it in the mesh.

The following is the pseudo code to delete a specific tag handle from the mesh.

```

int FMDB_Mesh_DelTag (pMeshMdl mesh, TagHandle* tag)
{
    TagHolder_DelTag(cast<TagHolder*>(mesh), tag);
    delete tag;
    return SUCCESS;
}

```

Herein, the function also deallocates the tag handle pointer to avoid the memory leak.

The following is the pseudo code to search and get tag handle(s).

```

// for input string name, find tag handle from mesh
int FMDB_Mesh_FindTag (pMeshMdl mesh, char* name,
    TagHandle* tag)
{
    return TagHolder_FindTag (cast<TagHolder*>(mesh),
        name, tag);
}

// get all tag handles in mesh
int FMDB_Mesh_GetTag (pMeshMdl mesh,
    vector<TagHandle*>& tags)
{
    return TagHolder_GetTag (
        cast<TagHolder*>(mesh), tags);
}

```



The code to set/get integral tag data to a part, and delete tag data attached to a part is illustrated below.

```
// set integral tag data to a part
int FMDB_Part_SetIntTag (pPart part, TagHandle* tag,
    const int data)
{
    return Taggable_SetData<int>(cast<Taggable*>(part),
        tag, &data);
}

// get integral tag data from a part
int FMDB_Part_GetIntTag (pPart part, TagHandle* tag,
    int* data)
{
    return Taggable_GetData<int>(cast<Taggable*>(part),
        tag, data);
}

// delete tag data attached to a part
int FMDB_Part_DelTag (pPart part, TagHandle* tag)
{
    return Taggable_DelTag (cast<Taggable*>(part), tag);
}
```

The code to set/get an array of integral tag data with an entity set is given below. *pEntSet* is a pointer type to a mesh entity set.

```
// get integral tag array data to a set
int FMDB_Set_SetIntArrTag (pEntSet set, TagHandle* tag,
    const int* data, int data_size)
{
    if data_size!=tag->size return ERROR;
    return Taggable_SetData<int>(cast<Taggable*>(set),
        tag, data);
}
```

```

}

// get integral tag array data from a set
int FMDB_Set_GetIntArrTag (pEntSet set, TagHandle* tag,
    int** data, int* data_size)
{
    *data_size = tag->size;
    return Taggable_GetData<int>(cast<Taggable*>(set),
        tag, *data);
}

```

Contrary to the code to set/get primary type tag data in which accessing the tag data is done in one step, the implementation to set/get *byte* type (*void\**) tag data is composed of two steps: (i) for a given tag handle, retrieve tag type and transform the *void\** tag data to a specific (primary) type data if necessary and (ii) if primary type, call single or array type set/get tag functions based on tag size information. For instance, part of the code to set *void\** tag data with a mesh entity (*ent*) is given below.

```

if tag->type==byte
    out = Taggable_SetByteArrayData<char>
        (cast<Taggable*>(ent), tag, data, data_size);

if tag->type==integer
    if tag->size==1 // single integer
        out = Taggable_SetByteData<int>
            (cast<Taggable*>(ent), tag, data, data_size);
    else // multiple integers
        out = Taggable_SetByteArrayData<int>
            (cast<Taggable*>(ent), tag, data, data_size);

```

The code to set *void\** tag data with other primary type, such as *double*, *pMesh-Ent* or *pEntSet*, is implemented by replacing *integer* to other primary data type in the code above. As aforementioned, tagging for a part, entity and entity set can

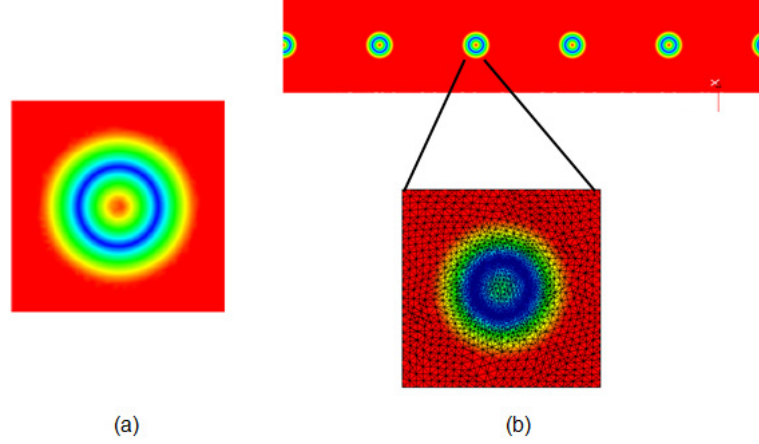
be implemented easily by reusing the tag component through class inheritance and template mechanism.

In parallel computations using the FMDB, migrating the tag data attached to an entity along with the migration is achieved through specifying callback routines [15, 51] in the mesh migration, which is the mechanism to allow the user to specify how to pack the tag data within the entity message packing procedure and how to unpack and attach the tag data received from remote parts. If no callback routine is specified, tag data is ignored during the migration and removed automatically when an entity is eliminated from the part.

### 3.6 Scaling Studies on Mesh Adaptation Using Generic Components

This section presents an example in parallel mesh adaptation developed using the geometric model and mesh (Geometric Model Interface (GMI) [79] and FMDB) with the three generic components used underneath as utilities. The mesh adaptation procedure developed in SCOREC [3, 20] is carried out on a flow simulation example to compare the performance of mesh adaptation between traditional object-oriented and generic programming paradigm. The mesh adaptation procedure is chosen for testing, since it relies heavily on entity iteration and tag data association. The iterator component is used to support traversing over either geometric model entities or mesh entities, and the tag component is to support data association with either of these entities. Associated data can be arbitrary, such as a solution-based mesh size field.

In the adaptive flow simulation, a straight pipe model with air bubbles distributed in the pipe is used (see Figure 3.3). Figure 3.3a shows the mesh size field which represents the motion of air bubbles in the geometric model. The smaller size field is shown in blue, implying a fine mesh (or high resolution), and the relative large size field is shown in red, implying a coarse mesh. In multi-phase flow simulations, fine meshes at phase boundaries are needed to capture the complicated physical phenomena at the interface [1]. Figure 3.3b shows a segment of the straight pipe model which involves the motion of five air bubbles by a distance of  $1/5$  of their



**Figure 3.3: Defined mesh size field (left) and a segment of straight pipe model with air bubbles (right).**

radius. A zoomed bubble in the mesh is colored by the magnitude of size field in Figure 3.3b.

The mesh adaptation procedure starts with an initial uniform tetrahedral mesh with 165 million tetrahedra, and obtains an adapted mesh of 188 million tetrahedra. All the test cases were run on the ANL *Intrepid* (IBM BG/P system) [80]. The test cases were executed on 1,024 up to 32,768 cores using 512MB per core memory. The execution time of the mesh adaptation procedure for all the test cases are collected, and scaling factors are computed based on the execution time on 1,024 cores. The scaling factor is defined as

$$s\text{-factor} = (np_{base} * t_{base}) / (np_i * t_i), \quad (3.1)$$

where  $t$  represents the execution time and  $np$  represents the number of cores. For instance,  $np_{base}$  represents the number of cores in the base case (here is the test case running on 1,024 cores), and  $t_{base}$  represents the execution time of the base case.

The performance results are summarized in Table 3.1, which compares the performance of two methods including (i) the mesh adaptation procedure using the iterator and tag components, and (ii) the mesh adaptation procedure using the traditional object-oriented programming paradigm. As shown in the last column of the

**Table 3.1: Scaling results of air-bubble mesh adaptation on ANL Intrepid using two methods: mesh adaptation without using the generic components (*adapt without generic*), and mesh adaptation using the generic components (*adapt with generic*).**

num of cores	<i>adapt without generic</i>		<i>adapt with generic</i>		time increase(%)
	time	s-factor	time	s-factor	
1,024 (base)	475.90	1	476.07	1	0.04
2,048	329.94	0.72	330.81	0.72	0.26
4,096	220.17	0.54	220.99	0.54	0.37
8,192	107.78	0.55	110.32	0.54	2.36
16,384	74.46	0.40	74.72	0.40	0.35
32,768	44.15	0.34	44.33	0.34	0.41

table, the mesh adaptation procedure using the generic components requires at most 2.36% more time than the one without the generic components, and does not affect the scaling. This time increase may due to the conditional statements implemented for P-sets during mesh migration (refer to the P-set discussion in Chapter 4). In summary, the generic components achieve code reusability and flexibility without sacrificing the performance of mesh adaptation, compared to the traditional object-oriented programming.

## CHAPTER 4

### ENTITY SET

The main purpose of using entity sets is to support the effective grouping of entities for various applications. This chapter first introduces the mathematics definitions of sets, and analyzes the entity set definitions and functionality. Then this chapter extends the generic set component to deal with specific requirements of mesh entity grouping for some unstructured mesh applications, especially boundary layer mesh applications.

#### 4.1 Introduction to Set Theory

A set is a collection of objects or a container of objects. Objects from which a given set is composed are called elements or members of the set. An object  $x$  belongs to a set  $A$  can be symbolically represented as  $x \in A$  [81, 82]. It is also assumed that a set has common properties associated with a collection of objects. A set can be described using a number of different ways, such as (i) to list up all the members of a set, (ii) to describe the common properties for membership in a set, or (iii) to give a recursive definition for membership of a set. Basic definitions used to build up the fundamental axiom system for set theory are given in references [81, 82]. Key definitions important to the development of mesh entity sets are given here:

**Definition** *Empty set*

The (unique) set with no elements is called the empty set, denoted by  $\emptyset$ .

**Definition** *Subset*

Set  $A$  is a subset of set  $B$ , if every element in  $A$  is also in  $B$ . Formally,  $A$  is a subset of  $B$ , denoted by  $A \subseteq B$ , if  $\forall x, x \in A$  implies  $x \in B$ .

Correspondingly, set  $B$  is a *superset* of set  $A$ , denoted by  $B \supseteq A$ .

**Definition** *Set equality*

Two sets are equal if and only if they have the same elements. More formally,

for any sets  $A$  and  $B$ ,  $A = B$  if and only if  $\forall x, x \in A$  implies  $x \in B$ , and vice versa.

**Definition** *Set size*

If a set  $A$  has  $n$  distinct elements for some natural number  $n$ ,  $n$  is the set size<sup>1</sup> of  $A$  and  $A$  is a finite set. The size of  $A$  is denoted by  $|A|$ .

For example, the size of the set  $\{1, 2, 3, 4\}$  is 4.

Sets can be combined in a number of different ways to produce another sets. Simple set-theoretic operations (union, intersection, difference, and etc) are introduced below [81, 82].

**Definition** *Union*

The union of sets  $A$  and  $B$ , denoted by  $A \cup B$ , is the set defined as  $A \cup B = \{x | x \in A \text{ and } x \in B\}$ .

**Definition** *Intersection*

The intersection of sets  $A$  and  $B$ , denoted by  $A \cap B$ , is the set defined as  $A \cap B = \{x | x \in A \text{ or } x \in B\}$ .

**Definition** *Subtraction (Difference)*

The difference of sets  $A$  and  $B$ , denoted by  $A - B$ , is the set defined as  $A - B = \{x | x \in A \text{ and } x \notin B\}$ .

In general,  $A - B \neq B - A$ .

A binary relation defined on a set can be used to reflect a binary relation between the elements of the set, such as an ordering of the elements of a set. If any two elements of a set  $A$  are *comparable* in terms of an ordering relation ( $\leq$ ), i.e. either  $a \leq b$  or  $b \leq a$  exists where  $a, b \in A$ , then set  $A$  is a *totally ordered set* [81].

## 4.2 Entity Set Definitions

In a mesh-based numerical analysis environment, an entity set is an arbitrary collections of entities for applications [12, 30]. Herein, entities can be either mesh

---

<sup>1</sup>For a more common and extensive term *set cardinality*, refer to set theory textbooks [81, 82].

entities in a mesh, or geometric model entities in a geometric model. For instance, an entity set could be used to represent the set of mesh faces classified on a geometric model face. More formally, an entity set can be denoted as follows:

$$S_i = \{x | x \in V \text{ and } P(x)\}, \quad (4.1)$$

where  $V \in \{G, M, F\}$ ,  $G$ ,  $M$  and  $F$  represents the geometric model, the mesh and fields separately, and  $P(x)$  represents the common properties of entity members in the set. A mesh (entity) set that contains unique mesh entities of dimension  $d$  is denoted by

$$S_i = \{M_i^d \mid M_i^d \in M \text{ and } (\forall x, y \in S_i, x \neq y)\}. \quad (4.2)$$

An entity set deals with entity members, and it supports general set properties and operations defined in §4.1.

Relations between two entity sets  $S_1$  and  $S_2$  is a subset of the product of  $S_1 \times S_2$ . Entity set relations [12, 30] can be used to represent (i) superset and subset relations, or (ii) logical relations between two sets, also referred to as parent and child relation.

A logical relation can be used to represent entity sets in multigrid and adaptive mesh sequences. A parent-child relation between two entity sets  $S_i$  and  $S_j$  can be denoted by

$$S_i \rightarrow S_j, \quad (4.3)$$

where the relation  $\rightarrow$  indicates the direction from  $S_i$  to  $S_j$ .  $S_i$  is a parent, and  $S_j$  is a child.

In an entity set  $S_i$ , applications can define relations between entity members (non-set members). Entity member relations will be subset of the product  $S_i \times S_i$ . A relation  $R$  defined on entity members in a set  $S_i$  is denoted by

$$(S_i, R). \quad (4.4)$$

A common relation between entity members useful in many applications is *ordering*. An entity set can be either a partially ordered set or totally ordered set. A totally



ordered entity set is denoted by

$$(S_i, \leq), \quad (4.5)$$

where  $\leq$  represents a total ordering between any two entity members in the set  $S_i$ .

Depending on whether there is the need to preserve the insertion order for non-set entity members or not, we define two types of entity sets [9, 12, 30]:

**Definition** *Ordered set*

If any two entity members are comparable in terms of the entity insertion order, a set is a (totally) ordered set, which can contain duplicate entity members.

**Definition** *Unordered set*

If no such ordering is preserved, a set is an unordered set.

In this work, we only consider (i) totally ordered entity set that preserves the entity insertion order in the set, and (ii) unordered entity set that contains unique entity members. For example, given two entity members ( $V_i^{d_i}$  and  $V_j^{d_j}$ ) in an ordered set  $S_i$ ,  $V_i^{d_i} \leq V_j^{d_j}$  means that  $V_i^{d_i}$  is inserted earlier than that  $V_j^{d_j}$  is. On the other hand, for any two elements  $x$  and  $y$  in an unordered set  $S_j$ , it can be concluded that  $x \neq y$ .

If an entity set contains another entity set, set contents can be queried recursively or nonrecursively. For example, if set  $S_i$  is a subset of set  $S_j$ ,  $S_i \subseteq S_j$ , a recursive request for the contents of  $S_j$  will include (i) the entities in  $S_j$  and (ii) the entities in the sets contained in  $S_j$ , such as entities in set  $S_i$ .

### 4.3 Entity Set Functionality

For illustration purposes, mesh sets, sets which contain only mesh entities, are used as an example to illustrate the entity set functionality. The mesh entity set functionality considered here is based on the ITAPS specification [12, 30, 42].

The mesh set functionality is divided into three parts: (i) operations on the mesh level, (ii) operations on the set level and (iii) operations on the entity level. Operations on the mesh level, summarized in Table 4.1, include querying to determine the number of sets in a mesh; retrieving all the sets in a mesh; and iteration

**Table 4.1: Set functionality at mesh level.**

Operation	Description
Iteration over sets	Loop through all the sets in a mesh
GetNumSet	Get the number of sets in a mesh
GetSet	Get all the sets in a mesh

over all the sets in a mesh. In general, iteration operations include initiating an iterator based on the application's requirements, retrieving the next iterator, and resetting an iterator.

Operations on the set level are further divided into three parts: *(i)* basic set functionality, *(ii)* hierarchical set relation, and *(iii)* set binary operations. The basic set functionality, summarized in Table 4.2, includes creating and destroying sets, adding and removing entities and sets into and from a set, and set specific query functions. Sets are created empty. Sets can be of various types, thus the specific type of a set must be specified when the set is created and can be queried later. Entities can be added to or removed from a set individually or in blocks. If a set allows duplicated members, the most recently added copy of duplicated members will be the first to be deleted. It is allowed to add sets to or remove sets from each other to represent relations such as subset and superset. A set can also be queried to determine the number of member entities and sets that it contains, and to determine whether a given entity or set belongs to that set. Entity traversal over a set is also supported based on entity type and topology conditions.

Set relation operations, summarized in Table 4.3, include adding, removing, counting and identifying parents and children, and deciding if one set is a child of another.

Set binary operations (intersection, union and subtraction) are summarized in Table 4.4. Set binary operations apply to both entity members and set members, but do not preserve any set relations. In other words, the resulting set will not have any relationships assigned automatically. Applications can later define any relation(s) on the resulting set through set relation operations.

For all set binary operations, if and only if both input sets are ordered sets,

**Table 4.2: Basic set functionality at set level.**

Operation	Description
Create/Delete	Create or delete a set of various types
AddEnt/RmvEnt	Append or remove an entity to or from a set
AddEnts/RmvEnts	Append or remove an array of entities to or from a set
AddSet/RmvSet	Append or remove a set to or from a set
GetType	Decide the type of a set
GetNumEnt	Get the number of entities in a set
GetNumSet	Get the number of sets in a set
HasEnt	Decide if a given entity belongs to a set
HasSet	Decide if a given set belongs to a set
Iteration over entities	Loop through entities in a set with specific type and topology requirements

**Table 4.3: Hierarchical set functionality at set level.**

Operation	Description
AddPrntChld	Create a parent to child relationship
RmvPrntChld	Remove a parent to child relationship
IsChldOf	Decide if a set is a child of another set
GetNumChld	Get the number of children of a set
GetNumChld	Get the number of children of a set
GetChldn	Get an array of children for a set
GetPrnts	Get an array of parents for a set

**Table 4.4: Set binary functionality at set level.**

Operation	Description
Intersect	Set intersection of two sets
Union	Set union of two sets
Subtract	Get set difference of two sets

the ordering of entity members in the resulting set is considered, and the resulting set is an ordered set. The rules of three binary set operations (union, intersection and subtraction) are follows<sup>2</sup> (we assume that for any duplicated entity, the first input set contains  $m$  copies of the entity, and the second set contains  $n$  copies of the entity, where  $m, n \geq 0$ ):

**Intersection:** given two ordered sets, for any duplicated entity, only  $\min(m, n)$  copies of the entity will be contained in the resulting set. The copies and other non-duplicate entities will appear in the same order as in the first input set, keeping the first number of copies. Otherwise, as long as one input set is unordered, both input sets are seen as unordered, and the intersection is to collect all the common entity members in the two sets.

**Union:** given two ordered sets, union is just to concatenate the input sets. Otherwise, as long as one input set is unordered, both input sets are seen as unordered, and the union is to collect all the entity members in the two sets.

**Subtraction:** given two ordered sets, for any duplicated entity, only  $\max(m - n, 0)$  copies of the entity will be contained in the resulting set. The copies and other non-duplicate entities will appear in the same order as in the first input set, keeping the first number of copies. Otherwise, as long as one input set is unordered, both input sets are seen as unordered, and the subtraction is to collect entity members that are in the first set and not in the second set.

For example, for two ordered sets,  $(S_1, \leq) = \{abacdbbc\}$  and  $(S_2, \leq) = \{dabae\}$ , and two unordered sets,  $S_3 = \{abcde\}$  and  $S_4 = \{abc\}$ , consider three cases: (i) both input sets are ordered such as  $S_1$  and  $S_2$ , (ii) both input sets are unordered such as  $S_3$  and  $S_4$ , and (iii) one input set is ordered while the other is unordered such as  $S_1$  and  $S_3$ . Respectively, the resulting set of a binary set operation is:

**Intersection :**

- (i) an ordered set,  $S_1 \cap S_2 = \{abad\}$ ,
- (ii) an unordered set,  $S_3 \cap S_4 = \{abc\}$
- (iii) an unordered set,  $S_1 \cap S_3 = \{abcd\}$ .

---

<sup>2</sup>These rules are consistent with the arbitrary rules defined in the ITAPS specification [12, 42].

**Table 4.5: Set functionality at entity level.**

Operation	Description
IsInSet	Decide if an entity belongs to a set
GetSet	Get all the sets associated with an entity

**Union :**

(i) an ordered set,  $S_1 \cup S_2 = \{abacdbcdabae\}$ , (ii) an unordered set,  $S_3 \cup S_4 = \{abcde\}$ , (iii) an unordered set,  $S_1 \cap S_3 = \{abcde\}$ .

**Subtraction :**

(i) an ordered set,  $S_1 - S_2 = \{bcc\}$ , (ii) an unordered set,  $S_3 - S_4 = \{de\}$ , (iii) an unordered set,  $S_1 - S_3 = \emptyset$ .

Regardless of whether the entity members of a set are ordered or unordered, the set members after a binary operation are always unordered and unique.

Two entity level operations are summarized in Table 4.5<sup>3</sup>. One is to determine if a given entity is inside any set, and the other is to get all the sets associated with a given entity (it returns an empty array if the entity is not contained in any set). These operations are useful for practical adaptive applications which require retrieving the relation from an entity to a set, but they may be not efficient in some cases if a set contains the whole mesh.

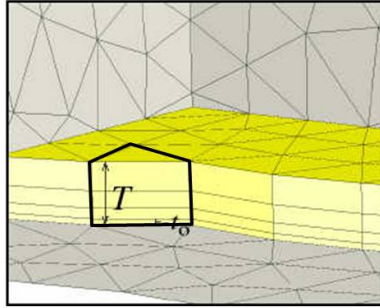
## 4.4 Mesh Sets to Deal with Boundary Layer Meshes

Due to the property of a mesh entity, which must be assigned to a part for management, there are two types of entity sets used in unstructured mesh applications depending on the number of parts associated with the entities contained in a set: (i) a set of entities residing on a single part, and (ii) a set with entities spanning multiple parts.

In support of parallel anisotropic adaptive meshing based on mesh metric fields with adaptive boundary layer meshes [4], a stack of boundary mesh entities

---

<sup>3</sup>These operations are not defined by the ITAPS requirement [12, 42].



**Figure 4.1:** Part of a boundary layer mesh (the stacks of mesh entities considered as entity sets).

is required to be on a single part for mesh modification, partition and migration, and it is useful to differentiate an entity set with the single part restriction from an entity set without such a restriction. In Figure 4.1, mesh entities contained in the black polygon illustrate a stack of entities to be treated with a P-set.

**Definition** *Part set (P-set)*

A mesh entity set with the single part restriction that all mesh entities in the set belong to the same part.

**Definition** *N-Part set (NP-set)*

A mesh entity set without the single part restriction, another word, a mesh entity set spanning  $N$  parts, where  $N \geq 1$ .

P-set is a special case of NP-set, where  $N = 1$ . Given a P-set, if it contains at least one mesh entity not part of the boundary of any higher-order mesh entities, the user can designate the set as a partition object so migrate all entities contained in the set to the destination part during the migration stage.

NP-sets allow arbitrary grouping of mesh entities spanning more than one parts and bring extra complexities in parallel computations, especially when the mesh changes. Since the current goal is to support adaptive unstructured mesh based applications, the focus herein is on the discussions of P-sets.

An important example of using mesh sets in unstructured mesh applications is to support boundary layer mesh adaptation, which requires the following of a mesh set.

1. Mesh entities contained in a set are unique.
2. Mesh entities are ordered in a set based on their insertion order.
3. Mesh entities contained in a set are not part of the boundary of any higher dimension mesh entities.
4. User can attach arbitrary data to a set.
5. Iterating over mesh entities in a set.
6. Migrating a set and constituting entities to another part.
7. Migrating tag data along the set migration.

Items 1-5 are supported by using the generic set component with a part as a *SetHolder*. For Item 6 and 7, the user can designate a set as a partition object so all the mesh entities contained in the set and tag data attached to the set are migrated along when the set is migrated.

Note that the following discussions assume P-sets with the above requirements, and also assume that P-sets do not contain any other set.

#### 4.4.1 Partition objects

Partition objects are determined by the application. A partition object can be either a mesh entity or a P-set. A partition object mesh entity is not bounded by any higher dimension mesh entity and not in any P-set. Figure 4.2a depicts a distributed 2D mesh with three P-sets, each of which consists of mesh faces and is designated as a partition object. Figure 4.2b illustrates its corresponding partition object diagram, where graph nodes (circles) represent partition objects and graph edges (lines) represent entity adjacency information between partition objects.

Algorithm 2 illustrates the procedure to return the partition objects in a part. In the first step, the algorithm traverses the mesh entities of all dimensions (say,  $M_i^d$ ) in the part, and checks if a mesh entity  $M_i^d$  is not bounded by any higher order mesh entity and does not belong to any P-set. If both are satisfied, the entity  $M_i^d$





have the operation target in the middle of the function name and the operation performed on the target data in the end. For instance, (i) the function with *Ent* in the middle of the name, such as *FMDB\_Ent\_IsInSet* and *FMDB\_Ent\_GetSet*, is performed on a specific mesh entity, (ii) the function with *Part* in the middle of the name is performed on a specific part, and (iii) the function with *Set* in the middle of the name, such as *FMDB\_Set\_AddEnt* and *FMDB\_Set\_Create*, is performed on a specific set.

#### 4.4.2 Pre-processing for graph-based mesh partitioning

For a graph/hypergraph based mesh partitioning, graph nodes are defined as partition objects in a mesh, and graph edges are often built through face adjacencies between neighboring partition objects. It is required to find neighboring partition objects of a partition object. To reduce the entity traversal and communication costs during the process of finding neighboring partition objects for a P-set, two preprocessing procedures are used. They are: (i) get boundary entities for a P-set (the procedure *FMDB\_Set\_GetBdryEnt*), and (ii) exchange partition object information between part boundary entities (the procedure *FMDB\_Part\_ExchPtnObj*).

The procedure *FMDB\_Set\_GetBdryEnt* returns the boundary entities that bound a given P-set ( $S_i$ ), formally  $M_i^d \in \partial(S_i)$ . Algorithm 3 is the pseudo code to illustrate this procedure. The algorithm traverses all member entities ( $M_i^d$ ) in a P-set ( $S_i$ ) and retrieves the boundary entities ( $M_j^{d-1}$ ) of each member entity  $M_i^d$ . If the entity  $M_j^{d-1}$  is on any part boundary, it stores  $M_j^{d-1}$  into the return list. Otherwise, if the entity  $M_j^{d-1}$  is adjacent to another partition object not equal to  $S_i$  (either an entity or another set),  $M_j^{d-1}$  is also added to the return list.

If two partition objects reside on two separate parts in a distributed mesh, one round of communication is needed to collect neighboring partition objects for each other on the remote parts. To reduce the accumulated communication costs of exchanging small messages between parts, it is necessary to pre-store neighboring partition objects on the remote parts in part boundary entities. Algorithm 4 illustrates the procedure *FMDB\_Part\_ExchPtnObj*. The algorithm first collects part boundary entities in a part, and then exchanges the information of neighboring par-

```

Data: a P-set  $S_i$  in a part M
Result: store all boundary entities that bound  $S_i$  into a list bdryEnt
begin
  for each  $M_i^d \in S_i$  do
    for each  $M_j^{d-1} \in \partial(M_i^d)$  do
      if FMDB_Ent_IsOnPartBdry( $M_j^{d-1}$ ) then
        store  $M_j^{d-1}$  into bdryEnt;
        continue;
      end
      /* check if  $M_j^{d-1}$  bounds another partition object */
      for each  $M_k^d \in \{M_j^{d-1}\{M^d\}\}$  do
        if FMDB_Ent_IsInSet( $M_k^d$ ) = false then
          store  $M_j^{d-1}$  into bdryEnt;
          break;
        else if  $S_l \leftarrow \text{FMDB\_Ent\_GetSet}(M_k^d)$  and  $S_l \neq S_i$  then
          store  $M_j^{d-1}$  into bdryEnt;
          break;
        end
      end
    end
  end
  return bdryEnt;
end

```

**Algorithm 3:** Get boundary entities for a P-set (`FMDB_Set_GetBdryEnt`).

tition objects between part boundary entities and their remote copies through one round of communication. The information of neighboring partition objects is then associated with part boundary entities through data tagging.

By using the two preprocessing steps, the pseudo code of Algorithm 5 illustrates the procedure to find neighboring partition objects for the partition objects in a part (the procedure *FMDB\_Part\_GetNborPtnObj*). Given a partition object in a part, the algorithm traverses the boundary entities of the partition object, formally either  $M_j^{d-1} \in \partial(M_i^d)$  or  $M_j^{d-1} \in \partial(S_i)$ , and retrieves the partition objects (not equal to the given partition object) adjacent to each boundary entity ( $\neq M_i^d$  or  $\neq S_i$ ). For example, given a P-set  $S_i$ , the algorithm retrieves its pre-stored boundary entities through the procedure *FMDB\_Set\_GetBdryEnt*. For each boundary entity ( $M_j^{d-1}$ ) of

```

Data: a  $d$ -dimension ( $d \in \{2, 3\}$ ) manifold mesh  $M$  in a part
Result: get adjacent partition object (PO) for each part boundary entity of
dimension  $d - 1$ , and exchange them between neighboring parts

begin
  /* STEP 1:  get all part boundary entities  $\{M\{M^{d-1}\}\}$           */
  for each  $M_i^{d-1} \in \{M\{M^{d-1}\}\}$  do
    if FMDB_Ent_IsOnPartBdry( $M_i^{d-1}$ ) = true then
      store  $M_i^{d-1}$  into entOnPartBdry;
    end
  /* STEP 2:  exchange PO between part boundary entities          */
  for each  $M_i^{d-1} \in \text{entOnPartBdry}$  do
    /* get the PO that  $M_i^{d-1}$  bounds          */
    get  $M_j^d$  that  $M_i^{d-1} \in \partial(M_j^d)$ ;
    if FMDB_Ent_IsInSet( $M_j^d$ ) then
       $S_k \leftarrow \text{FMDB\_Ent\_GetSet}(M_j^d)$  ;
      pack  $S_k$  in message A;
    else pack  $M_j^d$  in message A;
    for each remote copy  $M_i^{d-1}$  on remote part  $P_i$  do
      send message A ( $S_k$  or  $M_j^d$  on  $P_{local}$ ,  $M_i^{d-1}$  on  $P_i$ ) to  $P_i$ ;
    end
  end
  while  $P_i$  receives message A from  $P_{sender}$  do
    FMDB_Ent_SetTagData( $M_i^{d-1}$ , bdryEntTag, the GID of  $S_k$  or  $M_j^d$  on
       $P_{sender}$ );
  end
end

```

**Algorithm 4: Exchange partition object information between part boundary entities (FMDB\_Part\_ExchPtnObj).**

the P-set  $S_i$ , if  $M_j^{d-1}$  is on a part boundary, the remote partition object information (pre-stored through the procedure *FMDB\_Part\_ExchPtnObj*) is retrieved directly through the tagged data. Otherwise, the algorithm retrieves the higher dimension entity ( $M_k^d$ ) bounded by entity  $M_j^{d-1}$  through adjacencies, and decides the neighboring partition object (either  $M_k^d$  or its belonging P-set) adjacent to the current partition object.

For example, using edge adjacencies on the 2D mesh of Figure 4.2a, the P-set  $S_1$  neighbors two partition objects: faces  $M_2^2$  and  $M_4^2$  (see the two circles connected to the  $S_1$  circle in Figure 4.2b), while the mesh face  $M_2^2$  neighbors three partition

**Data:** a  $d$ -dimension ( $d \in \{2, 3\}$ ) mesh  $M$  of a part

**Result:** store neighboring partition objects (PO) in `nborPtnObj` for each PO in  $M$

```

begin
  /* STEP 1:  traverse all P-sets in M                                     */
  for each P-set  $S_i \in M$  do
    bdryEnt  $\leftarrow$  FMDB_Set_GetBdryEnt( $S_i$ );
    for each  $M_j^{d-1} \in \text{bdryEnt}$  do
      if FMDB_Ent_IsOnPartBdry( $M_j^{d-1}$ ) then
        adjPO  $\leftarrow$  FMDB_Ent_GetTagData( $M_j^{d-1}$ , bdryEntTag);
        store adjPO into nborPtnObj;
        continue;
      end
      store each  $M_j^{d-1}$ 's adjacent PO (either entity or P-set,  $\neq S_i$ ) into
       $M_i^d$ 's nborPtnObj;
    end
    make unique the elements in nborPtnObj of  $S_i$ ;
  end
  /* STEP 2:  traverse all entities of dim  $d$  in M                       */
  for each  $M_i^d \in \{M\{M^d\}\}$  do
    if FMDB_Ent_IsInSet( $M_i^d$ ) then continue;
    for each  $M_j^{d-1} \in \partial(M_i^d)$  do
      if FMDB_Ent_IsOnPartBdry( $M_j^{d-1}$ ) then
        adjPO  $\leftarrow$  FMDB_Ent_GetTagData( $M_j^{d-1}$ , bdryEntTag);
        store adjPO into nborPtnObj;
        continue;
      end
      store each  $M_j^{d-1}$ 's adjacent PO (either entity or P-set,  $\neq M_i^d$ ) into
       $M_i^d$ 's nborPtnObj;
    end
    make unique the elements in nborPtnObj of  $M_i^d$ ;
  end
end

```

**Algorithm 5:** Get neighboring partition objects for each partition object in a part (FMDB\_Part\_GetNborPtnObj).

objects: P-set  $S_1$ , faces  $M_1^2$  and  $M_3^2$  (see the three circles connected to the  $M_2^2$  circle in Figure 4.2b).

For illustration purposes, Algorithm 4 and Algorithm 5 assume a  $d$ -dimension manifold mesh. In a 3D (2D) manifold mesh, a mesh face (edge) is adjacent to at most two mesh regions (faces). These algorithms can be extended to handle non-manifold meshes through adding extra conditional statements.

Note that the pre-stored tagged data associated with sets and entities will be removed immediately after mesh partitioning to avoid memory leaking.

#### 4.4.3 Mesh migration algorithm

To support boundary layer mesh adaptation in parallel, the mesh migration algorithm developed in Reference [15, 16] was improved to migrate both mesh entities and P-sets. Given an array of entities of dimension  $d$  to migrate, the pseudo code of Algorithm 6 illustrates the mesh entity exchange procedure to move mesh entities and P-sets to destination parts in mesh migration. Note each P-set handle is attached (tagged) to each consisting entity to expedite the P-set manipulation and migration.

The main steps in Algorithm 6 are listed below:

**Step 1:** packs messages and sends them to destination parts. Before sending a message ( $A$ ) of a mesh entity ( $M_i^d$ ) in *EntitiesToUpdate*[ $d$ ], check if the entity is in a set (say,  $S_i$ ), and pack the message  $A$  with a variable (*isInSetFlag*) to indicate whether the message contains set data or not. If yes, i.e. *isInSetFlag* equals *true*, pack the set data within the message  $A$  also.

**Step 2:** initializes a set map to keep track of P-sets. On each process, create a set map *PSetMap* to store the pairwise relations between an original set ( $S_i$ ) from the sending part ( $P_i$ ) and its local copy ( $S_j$ ).

**Step 3:** unpacks messages and creates entities and P-sets. When a part  $P_i$  receives the message  $A$  from the sender ( $P_{sender}$ ), it creates a new mesh entity  $M_i^{d'}$ . Then it checks the variable *isInSetFlag* contained in the message  $A$ . If necessary, i.e. *isInSetFlag* equals *true*, it searches *PSetMap* for a local copy of the

set  $S_i$  from the part  $P_{sender}$ . If a local copy  $S_j$  exists, the new entity is added into the existing set  $S_j$ . Otherwise, a new set  $S_i'$  is created, a new entry for the set  $S_i'$  is added into  $PSetMap$ , and  $M_i^{d'}$  is added into the new set  $S_i'$ .

```

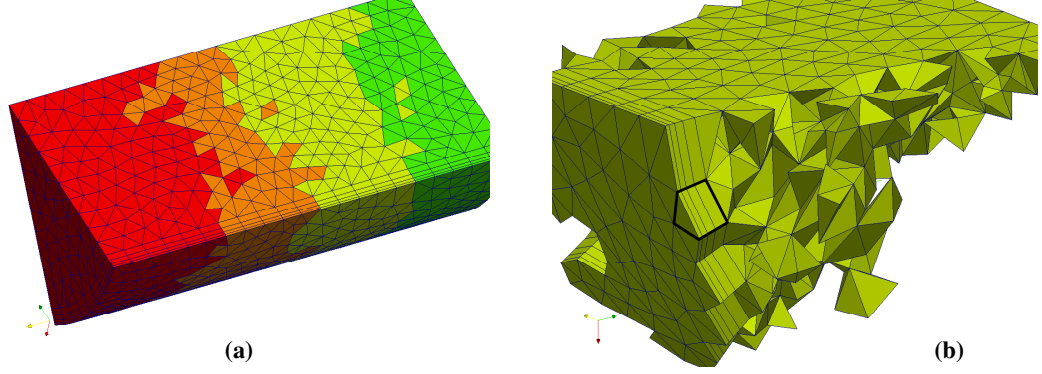
Data: EntitiesToUpdate[d]
Result: entities and P-sets are migrated to destination parts
begin
  /* STEP 1:  pack messages and send to destination parts      */
  for each  $M_i^d \in \text{EntitiesToUpdate}[d]$  do
    pack message A (entity info of  $M_i^d$ , ...);
    if FMDB_Ent_IsInSet( $M_i^d$ ) = false then
      isInSetFlag  $\leftarrow$  false;
      append (isInSetFlag) to message A;
    else
      isInSetFlag  $\leftarrow$  true;
       $S_i \leftarrow \text{FMDB\_Ent\_GetSet}(M_i^d)$ ;
      entPos  $\leftarrow \text{FMDB\_Set\_GetEntOrder}(S_i, M_i^d)$ ;
      append (isInSetFlag,  $S_i$ , entPos) to message A;
    end
     $P_{local}$  sends message A to  $P_i$ ;
  end

  /* STEP 2:  initialize a map to track P-sets                  */
  PSetMap  $\leftarrow$  map<pair<  $P_i, S_i$  >,  $S_j$  >;

  /* STEP 3:  unpack messages and create entities and P-sets   */
  while  $P_i$  receives message A from  $P_{sender}$  do
     $M_i^{d'} \leftarrow \text{FMDB\_Ent\_Create}(P_i, \text{entity info})$ ;
    if isInSetFlag = false then continue;
    if PSetMap [pair<  $P_{sender}, S_i$  >] exists then
       $S_j \leftarrow \text{PSetMap}$  [pair<  $P_{sender}, S_i$  >];
      FMDB_Set_AddEnt( $S_j, M_i^{d'}, \text{entPos}$ );
    else
       $S_i' \leftarrow \text{FMDB\_Set\_Create}(P_i, \text{PSET})$ ;
      PSetMap [pair<  $P_{sender}, S_i$  >]  $\leftarrow S_i'$ ;
      FMDB_Set_AddEnt( $S_i', M_i^{d'}, \text{entPos}$ );
    end
  end
end

```

**Algorithm 6:** The mesh entity exchange procedure to move mesh entities and P-sets to destination parts.



**Figure 4.3: Distributed boundary layer mesh on four parts.**

In Algorithm 6, since the entity insertion order ( $entPos$ ) is retrieved, the tagged data of a set can be migrated along with the first entity in the set.

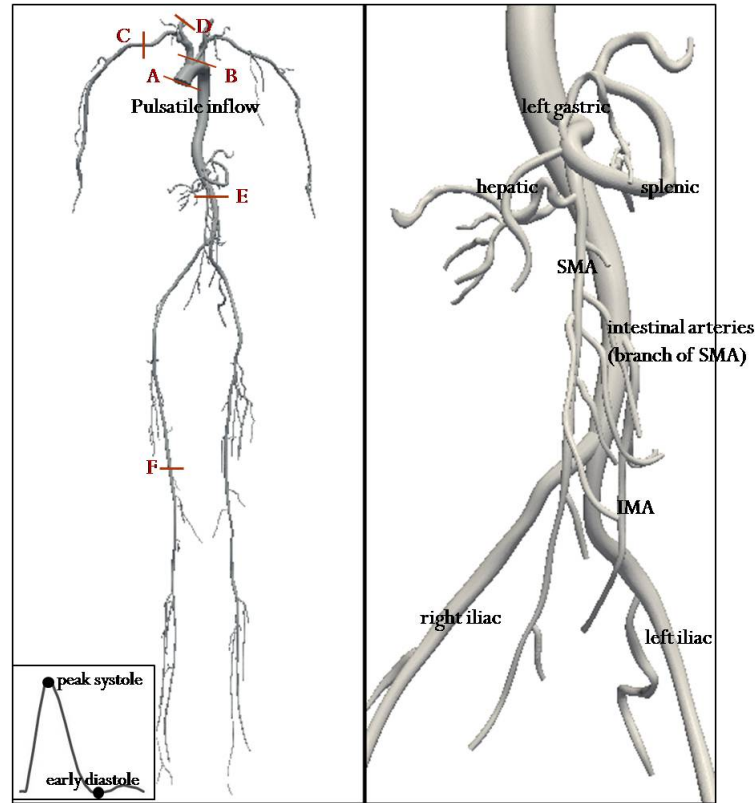
## 4.5 Examples and Applications

In a boundary layer mesh, a stack of mesh entities is considered as a P-set and is treated as a partition object during mesh partition and migration. The graph-based partitioner, ParMetis [60] is applied here for boundary layer mesh partitioning.

Two examples are considered here. The first example investigates the performance of mesh partition and migration with P-sets within the FMDB on two boundary layer meshes. The second example shows the results of parallel boundary layer mesh adaptation using the FMDB with P-sets. The two examples demonstrate the capability of the P-set implementation to effectively preserve boundary layers in anisotropic boundary layer mesh adaptive applications, especially in parallel.

### 4.5.1 Mesh partition and migration with P-sets

In the first case, a small mesh with 9,705 regions (9K mesh) on a simple geometry, which is a sector part of a pipe model (see the mesh in Figure 4.3), is considered. On the initial 9K mesh, a stack of wedges contains four layers of wedges and is considered as a P-set, and 424 P-sets are constructed from the curved bottom boundary of the geometric model. The initial serial mesh was partitioned into a distributed mesh over four parts, as shown in Figure 4.3a, where different



**Figure 4.4:** Whole body model composed of 78 arteries along with labels for sections (left) and labels of main arteries in the magnified view (right) [8].

colors represent different parts. The partitioning process was carried out on the SCOREC Piglet cluster [84], an AMD Opteron machine with eight quad-core nodes and 64GB in total of memory. As shown in one part of the distributed mesh (in Figure 4.3b), the four wedges in each stack were kept together as a unit, during the process of mesh partitioning and migration.

In the second case, a larger mesh with 7,094,564 regions (7M mesh) on a complex geometry that simulates the blood flow in a 3D whole body (see the model in Figure 4.4) is considered. A stack of wedges (see the detail of stacks in Figure 1.1) is treated as a P-set, and 648,783 P-sets are constructed in the initial 7M mesh. Note that it took 9.91 seconds to construct 648,783 P-sets through mesh adjacencies on the SCOREC Piglet cluster.

The second test case includes three steps: (1) partitioning the initial 7M mesh to 16 parts on the SCOREC Piglet cluster, (2) repartitioning the resulting

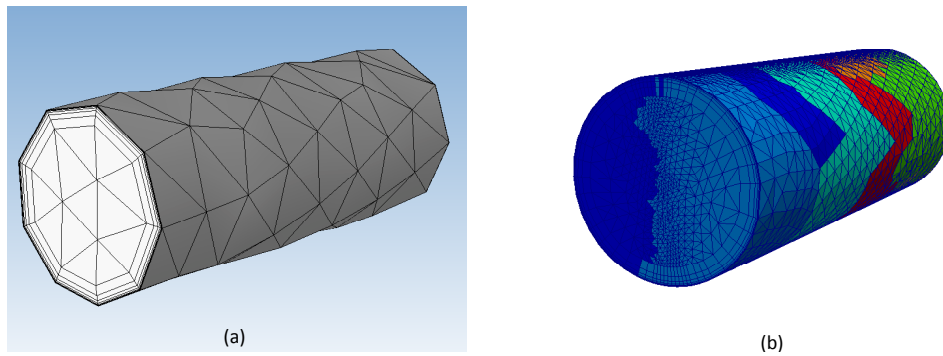


**Table 4.6: The average execution time (in seconds) of mesh partition, migration and total execution on the 7M mesh on 256 processors on NERSC Hopper through the global partitioning with ParMetis.**

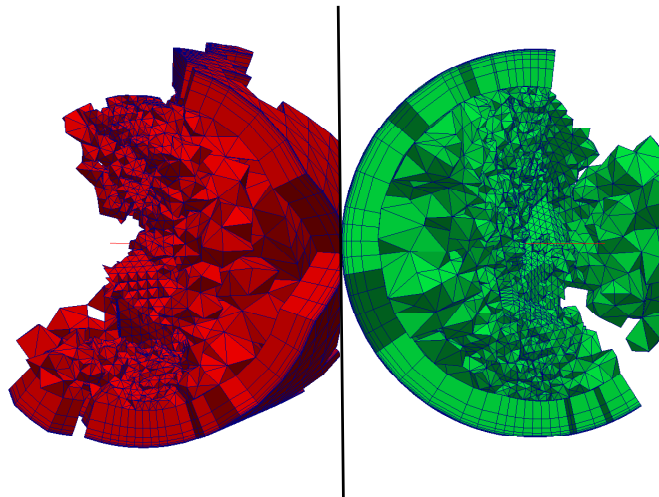
Total #parts	#RgnsToMove	#SetsToMove	Partition (sec)	Migration (sec)	Total (sec)
512	7,088,837	648,225	0.39	4.05	4.44
1,024	7,082,937	647,811	0.49	3.97	4.46
2,048	7,092,442	648,583	0.65	4.13	4.78
4,096	7,093,698	648,718	0.83	4.22	5.05

partitioned mesh of 16 parts (from the first step) to 256 parts on 16 processors, and (3) repartitioning the resulting partitioned mesh of 256 parts (from the second step) to various partitions containing 512 up to 4,096 parts on 256 processors. The repartitioning in the last two steps was carried out through the capability of multiple parts per process in the FMDB on the NERSC Hopper system, using 12 cores per node with 2.58GB of memory for each core. After the repartitioning, each processor contains multiple parts. For example, in the 4,096 part partition, each processor contains 16 parts.

Table 4.6 lists the average execution time (in seconds) of the mesh *Partition* stage (in the 4th column), mesh *Migration* stage (in the 5th column) and *Total* partitioning process (in the 6th column) to obtain different partitions on 256 processors on the NERSC Hopper, along with the total number of regions to migrate (*RgnsToMove*) and the total number of P-sets to migrate (*SetsToMove*) during the migration stage. As shown in the 2nd and 3rd columns, due to the global partitioning, almost all the partition objects (both regions and P-sets) were migrated. Since the average number of regions on a part is small (eg. in the 512 part partition, average number of regions on a part is 13,857), the repartitioning from 256 parts to 512 up to 4,096 parts takes about 5 seconds, and does not increase much time. In the subsequent simulation, the scaling studies can be performed on 256 up to 4,096 processors on these partitioned boundary layer meshes containing 256 up to 4,096 parts, respectively.



**Figure 4.5: Anisotropic adaptivity, preserving boundary layer mesh structure from an initial mesh of 2K regions to an adapted and partitioned mesh of 556K regions.**

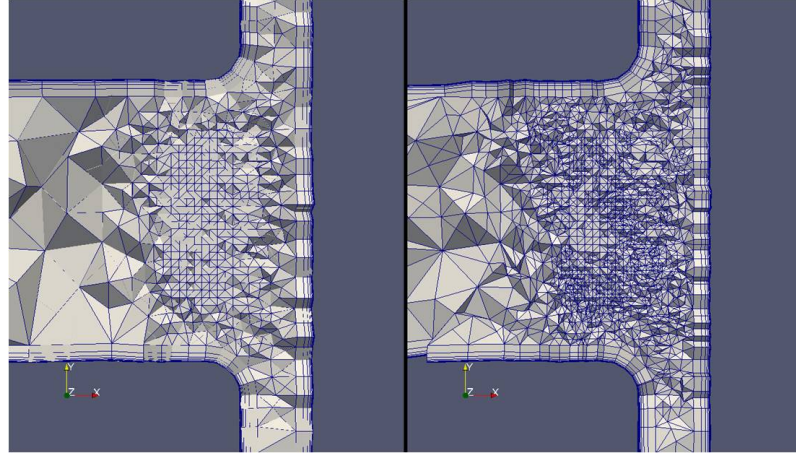


**Figure 4.6: Two example parts of the partitioned mesh of 556K regions adapted from the anisotropic adaptivity.**

#### 4.5.2 Support boundary layer mesh adaptation

This section presents the example usage of the parallel boundary layer mesh adaptation results developed on top of the software tools, including the geometric model interface (GMI), the mesh adaptation procedure developed in SCOREC [85], and the mesh database FMDB with the P-set implementation.

The first case uses a small boundary layer mesh of about 2,000 (2K) regions on a simple pipe model. The anisotropic boundary layer adaptation is carried out on this pipe model based on a specified analytical mesh size field [20]. The initial serial



**Figure 4.7: Clip-plane mesh view of a boundary layer adapted mesh of 4.5M regions [85].**

mesh (see Figure 4.5a) was partitioned and then adapted to a partitioned mesh (see Figure 4.5b) of 556,000 (556K) regions on 8 parts. The whole adaptation process was performed on Piglet machine. As shown in the two parts of the partitioned mesh in Figure 4.6, the boundary layers are preserved after the mesh partition, migration and adaptation stages.

The second case uses a larger boundary layer mesh of about 450,000 (450K) regions on a manifold heat exchanger model, in which a large flow rate comes in a larger tube and dumps into a thin rectangular geometry where the flow is distributed into smaller pipes [85]. The solution based anisotropic boundary layer adaptation is carried out on this model to capture the flow features. The resulting adapted boundary layer mesh contains about 4.5 million (4.5M) regions. The clip-plane views of the interior boundary layer mesh structures before and after the boundary layer mesh adaptation are shown in Figure 4.7.

## CHAPTER 5

### MESH MATCHING

When solving differential equations using numerical methods, like finite element methods, there are some cases where the boundary conditions are periodic, in which case the solution repeats itself on specific matched boundary entities. To most effectively apply periodic boundary conditions the mesh entities on matched boundaries should match topologically and geometrically. Reference [86] presented a general methodology for applying periodic boundary conditions that requires maintaining all mesh entities identical on periodic faces, with a periodic master of each mesh entity on a selected periodic face, defined to control all the match periodic faces. Reference [87] presented a technique to build adaptive meshes on periodic domains through moving the current periodic boundaries towards the interior of the mesh, and thus to enable adapting periodic meshes using standard non-periodic mesh adaptation routines.

These references clearly indicate that applying periodic boundary conditions can be most easily handled by the *matching* of the meshes on the periodic boundaries, where *matching* means the mesh on one periodic boundary is identical to the other once it is transformed. Doing this allows the simple matching of unknowns in the solution. This chapter addresses the specific requirement of mesh matching and presents a general technique that maintains the entire mesh topology on 3D models for a general set of matching requirements.

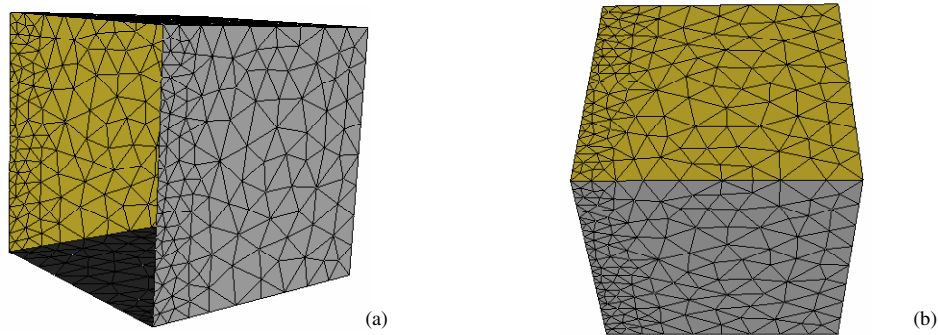
We start from a geometric model and a mesh where the mesh entities on periodic boundaries are identical and aware of each other. The initial mesh is generated by a mesh generation tool such as Simmetrix [34], and the information of identical entities in the initial mesh is also given by the mesh generation tool. This initial mesh is converted into the FMDB format mesh through file import/export. All the mesh operations discussed in this chapter are performed on the geometric model and the converted FMDB mesh. The goal is to maintain the matching information for identical entities classified on periodic boundaries as the mesh changes, especially

in parallel operations.

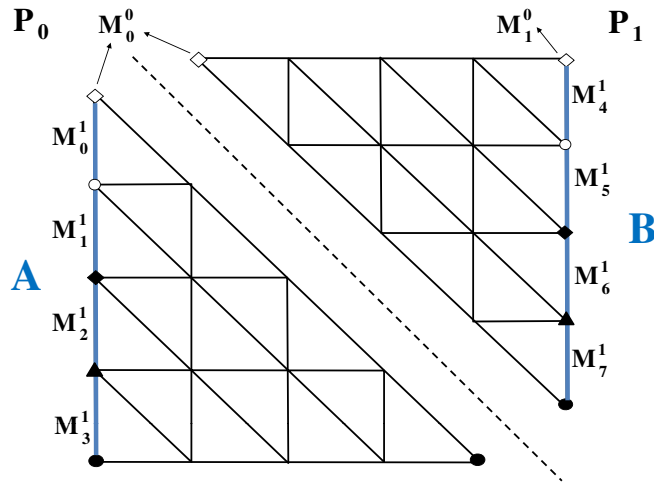
In this chapter, the relation between periodic boundary conditions and the geometric model is presented in §5.1. The definitions and functionality of matched mesh and matched entity are given in §5.2, followed by the implementation of the mesh entity data structure. §5.3 presents the conversion from an initial matched mesh generated from a mesh generation tool to an FMDB format mesh. §5.4 and §5.5 investigate the parallel operations to deal with matched meshes, mesh partitioning and migration, respectively. §5.6 presents the results of the parallel operations in the FMDB and the mesh adaptation procedure on matched meshes.

## 5.1 Geometric Model with Periodic Boundary Conditions

When boundary conditions involve periodicity, the solution repeats itself after a certain spatial interval, in a given direction. This fact can be used to equate the solution variables at the ends of the periodic interval, which can be any combination of translation and rotation. For example, Figure 5.1a is part of a mesh on a cube geometry with transformed periodic boundary conditions, the left yellow face and the right grey face of the geometric model are specified as periodic boundaries (the front face of the geometry is hidden to show the yellow model face). Figure 5.1b is another example with axisymmetry periodicity. The upper yellow face and the lower grey face of the geometric model are periodic boundaries. This match is determined by rotating a 90-degree angle between the two periodic faces.



**Figure 5.1: Two types of periodic boundary conditions: transformed (left) and rotated (right) periodic boundary conditions [34].**



**Figure 5.2:** Example of a distributed 2D matched mesh on two parts, left model boundary A and right model boundary B are periodic boundaries.

In the case of automatic adaptive finite element analysis, the mesh may be varied over the course of several analysis runs, but the geometric model, the topological description of the physical domain, remains fixed. Thus boundary condition attributes need to be associated with the geometric model rather than the mesh, and this association provides a reliable approach to their application [33]. Since mesh-model classification [35] provides the unique association of mesh entities on the geometric model entities, classification can be used to transfer the desired boundary condition attributes from model entities to the mesh entities that are classified on them. Thus periodic boundary conditions are defined as attributes assigned at the model level, and then model entities apply the attributes to the mesh entities classified on them [88].

## 5.2 Matched Mesh Definition and Functionality

Mesh entities classified on periodic boundaries on a geometric model are identical to each other, respectively. For example, Figure 5.2 depicts a 2D mesh on a plate model distributed on two parts  $P_0$  and  $P_1$ . The dashed line represents the part boundary between the two parts. The model edges A and B (in thick blue

lines) are periodic edges. Thus mesh edges  $M_0^1 - M_3^1$  classified on the model edge  $A$  are matched to identical, but translated, mesh edges  $M_4^1 - M_7^1$  classified on the model edge  $B$ , respectively. The mesh vertices (in identical shapes) classified on boundaries  $A$  and  $B$  are also matched to each other. Such mesh entities classified on periodic boundaries are called matched entities, and a mesh which contains such matched entities is called a matched mesh.

Information must be maintained so that the matched entities know what they are matched to, and which parts they exist on. To maintain the entire mesh topology, matched entities can be classified on spatially unconnected geometric model entities. Moreover, if multiple periodic boundary conditions exist, a matched entity can match multiple mesh entities, and should know all the entities that it matches. In a distributed environment, a matched mesh is distributed across a number of parts and matched entities can reside on different parts, thus a matched entity needs to know which part on which its matched entity exists. If a matched entity is on a part boundary, its remote copy (or copies) and matched entity (or entities) should know each other. An entity can have multiple matched entities from a single part. For example, in the 2D mesh in Figure 5.2, part boundary mesh vertex  $M_0^0$  matches mesh vertex  $M_1^0$  on part  $P_1$ . Two remote copies of vertex  $M_0^0$  know vertex  $M_1^0$  on part  $P_1$ , and vertex  $M_1^0$  knows the two copies.

In summary, at the mesh entity level, the following functions are required:

- *FMDB\_Ent\_IsMatch*

Check if a given entity is a matched entity. If yes, this returns true, otherwise, this returns false.

- *FMDB\_Ent\_GetMatch*

Given a mesh entity  $M_i^d$ , return the mesh entity  $M_j^d$  that matches  $M_i^d$  and the part id where  $M_j^d$  exists. If more than one entity matches  $M_i^d$ , return a list of pairs, containing all the matched entities and the part id's where they exist.

- *FMDB\_Ent\_GetNumMatch*

Given a mesh entity  $M_i^d$ , return the number of mesh entities that match  $M_i^d$ . If there is no such entity, this returns zero.

As the mesh changes, it is necessary to rebuild or remove matching connections between a set of matched entities. Thus two more functions are required:

- *FMDB\_Ent\_AddMatch*

Given two matched entities, add the second entity and its part id into the matched entity information stored in the first entity.

- *FMDB\_Ent\_RmvMatch*

Given two matched entities, remove the second entity and its part id from the matched entity information stored in the first entity.

In the implementation of mesh entity data structure, matched entity information is stored as an STL *multimap* [77]. Each entry in the *multimap* represents one matched entity, including the key value of the part id and the mapped value of the entity address. By using STL *multimap*, a matched entity allows different matched entities from a single part. For example, in the mesh of Figure 5.2, vertex  $M_0^0$  on part  $P_0$  matches vertex  $M_1^0$  on part  $P_1$ . The *multimap* of matched entity information stored in vertex  $M_1^0$  on part  $P_1$  includes two entries: vertex  $M_0^0$  on  $P_0$ , and vertex  $M_0^0$  on  $P_1$ , this is also the output of the function *FMDB\_Ent\_GetMatch* for  $M_1^0$ .

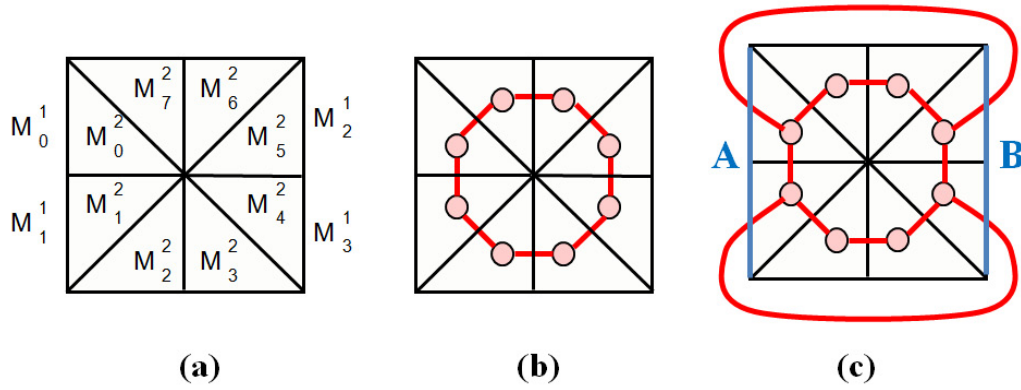
### 5.3 Construct Matched Mesh from Mesh File

The mesh import/export procedures let the user export the mesh into a mesh file and recover the mesh later from the file. Since an initial matched mesh generated from a mesh generation tool is not recognized by the FMDB, it is required to convert the initial mesh into the FMDB format mesh file.

Based on the FMDB mesh file format, matched entity information obtained from a mesh generator is stored at the end of the mesh file. The extra information includes:

- Starting with the keyword 'Matching' in a single line.
- The number of groups for each entity dimension (0 to 2 for vertex/edge/face), where mesh entities match each other in each group.





**Figure 5.3: Example of a graph-based mesh partitioning, (a) is a 2D mesh, (b) is the dual graph of the mesh without considering periodic boundaries, and (c) is the dual graph of the mesh with A and B being periodic boundaries.**

- The information of each matched entity group, including the number of mesh entities in the group, the number of vertices in each entity, and the information of each entity.

Note that matched edges/faces are adjacent to the same number of mesh vertices. If a matched entity is a vertex, the entity information is the vertex id. Otherwise, the entity information includes the vertex id's of all the vertices that bound the current entity. Herein, a vertex id is the ordering number of a mesh vertex  $M_i^0$  visited in the previous part of the mesh file, starting from number 1. For each edge/face, vertices are listed in the order consistent with downward adjacencies.

## 5.4 Pre-processing for Graph-based Mesh Partitioning

In a graph-based mesh partitioning, graph nodes represent partition objects, and graph edges between graph nodes represent dependencies between partition objects. In a matched mesh partitioning, two dependencies exist: entity adjacency and matched entity relation. Besides adjacency type dependency, if two partition objects (e.g. regions) are bounded by a couple of matched entities (e.g. faces), a graph edge should be built between them. For example, in the 2D mesh in Figure 5.3a, without considering periodic boundaries, graph nodes are mesh faces, graph edges are built through edge adjacencies. The resulting graph is shown in Figure 5.3b. On

the other hand, if model edges  $A$  and  $B$  are periodic boundaries, two extra graph edges are built to reflect matched entity dependencies, as shown in Figure 5.3c.

To build graph edges, an important step is to find all the partition objects that have dependencies to a given partition object. To reduce entity traversal and communication cost, a pre-processing procedure is used to exchange partition object information between matched entities.

Algorithm 7 is the pseudo code to illustrate this pre-processing procedure. Given a mesh on a part  $P_i$ , the algorithm includes the following main steps:

**Step 1:** defines a tag handle *matchEntTag* to hold a unique tag identifier attachable to any matched entity.

**Step 2:** traverses all the mesh entities of dimension  $d - 1$  on the part, collects all the matched entities of dimension  $d - 1$ , and stores them into an entity list (*matchEnts*).

**Step 3:** for each matched entity  $M_i^{d-1}$  in the list *matchEnts*:

**Step 3.1:** initiates a partition object list *tagPO<sub>i</sub>* and attach it to entity  $M_i^{d-1}$  with the tag *matchEntTag*;

**Step 3.2:** retrieves the partition object ( $M_i^d$ ) that  $M_i^{d-1}$  bounds;

**Step 3.3:** traverses each matched entity (say,  $M_j^{d-1}$  on part  $P_j$ ) of the current entity  $M_i^{d-1}$ , and check if  $M_j^{d-1}$  is on the local part ( $P_i$ ). If yes, store the partition object ( $M_j^d$ ) that  $M_j^{d-1}$  bounds into *tagPO<sub>i</sub>* attached to  $M_i^{d-1}$ . Otherwise, send a message  $A$  to the matched entity  $M_j^{d-1}$  on part  $P_j$ . The message  $A$  includes the address of  $M_j^{d-1}$  and the partition object  $M_i^d$ .

**Step 4:** When part  $P_j$  receives the message  $A$ , it retrieves the attached partition object list (*tagPO<sub>j</sub>*) from the received matched entity  $M_j^{d-1}$ , and stores the received partition object  $M_i^d$  into the list *tagPO<sub>j</sub>*.

In Algorithm 7, the attached partition object list (*tagPO<sub>i</sub>*) of a matched entity ( $M_i^{d-1}$ ) should include unique partition objects that are not equal to the partition object ( $M_i^d$ ) bounded by the current entity  $M_i^{d-1}$ .

**Data:** a  $d$ -dimension ( $d \in \{2, 3\}$ ) mesh  $M$  on a part  $P_i$   
**Result:** exchange partition object information between matched faces  
**begin**

```

    /* STEP 1:  initiate matchEntTag                                */
    /* STEP 2:  collect all matched entities in M                  */
    for each  $M_i^{d-1} \in \{M\{M^{d-1}\}\}$  on part  $P_i$  do
        if FMDB_Ent_IsMatch( $M_i^{d-1}$ ) = true then
            store  $M_i^{d-1}$  into matchEnts;
        end
    end
    /* STEP 3:  send message to matched entity                      */
    for each  $M_i^{d-1} \in$  matchEnts do
        initiate tagPO $_i$ ;
        FMDB_Ent_SetTagData( $M_i^{d-1}$ , matchEntTag, tagPO $_i$ );
        get  $M_i^d$  that  $M_i^{d-1} \in \partial(M_i^d)$ ;
        for each matched entity  $M_j^{d-1}$  on part  $P_j$  do
            if  $P_i = P_j$  then
                get  $M_j^d$  that  $M_j^{d-1} \in \partial(M_j^d)$ ;
                if  $M_j^d \notin$  tagPO $_i$  then
                    store  $M_j^d$ 's GID into  $M_i^{d-1}$ 's tagPO $_i$ ;
                end
            else
                send a message  $A(M_j^{d-1}, M_i^d)$  to part  $P_j$ ;
            end
        end
    end
    /* STEP 4:  receive message from matched entity                */
    while part  $P_j$  receives message  $A$  from  $P_i$  do
        tagPO $_j \leftarrow$  FMDB_Ent_GetTagData( $M_j^{d-1}$ , matchEntTag);
        if  $M_i^d \notin$  tagPO $_j$  then
            store  $M_i^d$ 's GID into  $M_j^{d-1}$ 's tagPO $_j$ ;
        end
    end
end

```

**Algorithm 7:** Exchange partition object information between matched entities.

```

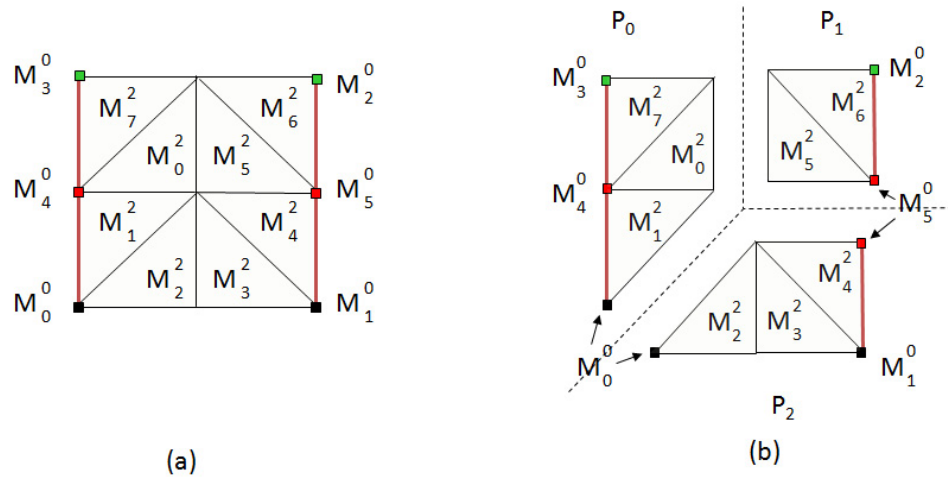
Data: a  $d$ -dimension ( $d \in \{2, 3\}$ ) mesh  $M$  on a part  $P_i$ 
Result: store dependent partition objects (PO) in  $\text{dependPtnObj}$  of each PO
           in  $M$ 
begin
  retrieve  $\text{matchEntTag}$  and  $\text{bdryEntTag}$ ;
  /* traverse all partition objects in  $M$  */
  for each  $M_i^d \in \{M\{M^d\}\}$  on part  $P_i$  do
    for each  $M_j^{d-1} \in \partial(M_i^d)$  do
      get  $M_l^d$  that  $M_j^{d-1} \in \partial(M_l^d)$ ;
      if  $M_l^d \neq M_i^d$  then
        store  $M_l^d$ 's GID into  $M_i^d$ 's  $\text{dependPtnObj}$ ;
      end
      if  $\text{FMDB\_Ent\_IsMatch}(M_j^{d-1})$  then
         $\text{tagPO}_j \leftarrow \text{FMDB\_Ent\_GetTagData}(M_j^{d-1}, \text{matchEntTag})$ ;
        for each  $M_k^d \in \text{tagPO}_j$  do
          store  $M_k^d$ 's GID into  $M_i^d$ 's  $\text{dependPtnObj}$ ;
        end
      end
    end
    if  $\text{FMDB\_Ent\_IsOnPartBdry}(M_j^{d-1})$  then
       $M_j^d \leftarrow \text{FMDB\_Ent\_GetTagData}(M_j^{d-1}, \text{bdryEntTag})$ ;
      store  $M_j^d$ 's GID into  $M_i^d$ 's  $\text{dependPtnObj}$ ;
    end
  end
end
end

```

**Algorithm 8:** Get all partition objects that depends on a partition object in a part.

Algorithm 8 illustrates the procedure to collect the partition objects that have dependencies to a partition object on a part. Given a partition object  $M_i^d$  on a part  $P_i$ , the algorithm stores all its dependent partition objects into a list  $\text{dependPtnObj}$ . The algorithm first traverses all the boundary entities of the partition object, i.e.  $M_j^{d-1} \in \partial(M_i^d)$ . For each boundary entity  $M_j^{d-1}$ , it performs the following steps:

- Retrieve the partition object bounded by  $M_j^{d-1}$  through adjacencies, and store it into the list  $\text{dependPtnObj}$ .
- If  $M_j^{d-1}$  is a matched entity, get the attached partition object list  $\text{tagPO}_j$  (pre-stored through Algorithm 7), store each partition object in  $\text{tagPO}_j$  into



**Figure 5.4: Example of a 2D mesh migration: before (left) and after (right) mesh migration with matched entities**

the list *dependPtnObj*;

- If  $M_j^{d-1}$  is on a part boundary, get the attached remote partition object (pre-stored through the procedure *FMDB\_Part\_ExchPtnObj* in §4.4.2), store it into the list *dependPtnObj*;

Note for illustration purposes, Algorithm 7 and Algorithm 8 consider only meshes on manifold models without P-sets, where each mesh face (or edge) is adjacent to at most two mesh regions in a 3D (or faces in a 2D) mesh.

## 5.5 Mesh Migration Algorithm

The mesh migration procedure moves mesh entities between parts. Figure 5.4 shows an example of the mesh entity migration process in a 2D matched mesh. Figure 5.4a depicts the initial 2D mesh on part  $P_0$ . The geometric model includes two periodic model edges (in thick red lines). Mesh faces  $M_5^2$  and  $M_6^2$  will migrate to part  $P_1$ , and mesh faces  $M_2^2$ ,  $M_3^2$ , and  $M_4^2$  will migrate to part  $P_2$ . Figure 5.4b depicts the distributed mesh on three parts after mesh migration. The dashed lines represent the part boundaries between parts. The mesh edges (in thick lines) classified on periodic edges match each other respectively, as well as the vertices (of the small squares). Matched entities such as mesh vertices  $M_0^0$  and  $M_5^0$  exist on the

part boundaries, and their remote copies and matched entities will know each other. For example, vertex  $M_4^0$  on part  $P_0$  knows the two copies of vertex  $M_5^0$ .

Given the partition objects to migrate and their destination parts (*POsToMove*), the following are the main steps of the overall mesh migration procedure:

**Step 1:** given the *POsToMove*, collects entities to process (*entitiesToUpdate*) and clears their partitioning data (residence parts and partition classification).

**Step 2:** determines the residence parts of entities in *entitiesToUpdate*, updates partition classification and the partition model, and collects entities to remove (*entitiesToRemove*).

**Step 3:** exchanges mesh entities in *entitiesToUpdate*, updates their remote copy and matched entity information.

**Step 4:** removes unused entities in *entitiesToRemove* from local parts, updates related remote copy and matched entity information.

**Step 5:** updates the ownership of partition entities.

*Step 1, 2, 5* are the same as given in references [15, 16]. *Step 3*, exchange entities, and *Step 4*, remove unused entities, must explicitly consider matched mesh information and are discussed as follows.

### 5.5.1 Exchange entities

In *Step 3* of the mesh migration algorithm, mesh entities are exchanged from low to high dimension, since an entity of dimension  $> 0$  is bounded by lower dimension entities [15, 16]. Mesh entities in *entitiesToUpdate* are transferred from source parts to destination parts, and new entities are created on the destination parts.

Algorithm 9 is the pseudo code to illustrate the procedure of exchanging the mesh entities in *entitiesToUpdate[d]*. The following are the main steps of Algorithm 9:

**Step 3.1:** sends messages to destination parts to create new mesh entities. For each entity  $M_i^d$  in *entitiesToUpdate[d]*, if  $M_i^d$  resides on more than one part,

```

Data: entitiesToUpdate [d]
Result: create entities on destination parts and update remote copy and
           matched entity information
begin
  /* STEP 3.1: send message to destination part */
  for each  $M_i^d \in \text{entitiesToUpdate}[d]$  do
    choose  $P_{bc}$  to send message  $A$  ( $M_i^d$  on  $P_{bc}$ , information of  $M_i^d$ ) to  $P_i$ 
    where  $P_i \in \mathcal{P}[M_i^d]$  and no copy exists on  $P_i$ ;
  end
  /* STEP 3.2: create new entity and report to broadcaster */
  while  $P_i$  receives message  $A$  from  $P_{bc}$  do
    create  $M_i^d$  on  $P_i$ , add remote copy or matched entity information if
    necessary;
    if  $M_i^d \neq \text{partition object}$  then send message  $B$  ( $M_i^d$  on  $P_{bc}$ , new  $M_i^d$ 
    on  $P_i$ ) to  $P_{bc}$ ;
  end
  /* STEP 3.3: broadcaster forward the new entity */
  while  $P_{bc}$  receives message  $B$  from  $P_i$  do
     $M_i^d$  saves the  $M_i^d$  on  $P_i$  as a remote copy;
    for each remote copy of  $M_i^d$  on part  $P_{remote}$  do
      isRemoteCopyFlag  $\leftarrow$  true;
      send message  $C$  (isRemoteCopyFlag,  $M_i^d$  on  $P_{remote}$ ,  $M_i^d$  on  $P_i$ ) to
       $P_{remote}$ ;
    end
    for each matched entity  $M_j^d$  on part  $P_{remote}$  do
      isRemoteCopyFlag  $\leftarrow$  false;
      send message  $C$  (isRemoteCopyFlag,  $M_j^d$  on  $P_{remote}$ ,  $M_i^d$  on  $P_i$ ) to
       $P_{remote}$ ;
    end
  end
  /* STEP 3.4: update remote copy or matched entity */
  while  $P_{remote}$  receives message  $C$  from  $P_{bc}$  do
    if isRemoteCopyFlag = true then
      FMDB_Ent_AddCopy( $M_i^d$ ,  $M_i^d$  on  $P_i$ );
    else
      FMDB_Ent_AddMatch( $M_j^d$ ,  $M_i^d$  on  $P_i$ );
    end
  end
end

```

**Algorithm 9:** Exchange mesh entities and update remote copy and matched entity information.

to reduce the communication cost between parts, the part with the minimum part id where  $M_i^d$  exists is assigned as the *broadcaster* ( $P_{bc}$ ). Then the broadcaster  $P_{bc}$  sends a message  $A$  to the destination part ( $P_i$ ) where  $P_i$  is in the residence parts of  $M_i^d$  (i.e.  $\in \mathcal{P}[M_i^d]$ ) and no copy exists on  $P_i$ . The message  $A$  to send includes: entity shape information, required adjacencies, geometric classification, partitioning data, remote copy and matched entity information.

**Step 3.2:** when the destination part  $P_i$  receives the message  $A$ , a new entity  $M_i^d$  is created on  $P_i$ . If the message  $A$  contains remote copy or matched entity information, the new entity  $M_i^d$  will store the information. If the new entity  $M_i^d$  is not a partition object,  $P_i$  will send the address of  $M_i^d$  back to the broadcaster ( $M_i^d$  on  $P_{bc}$ ). The message  $B$  to send back to  $P_{bc}$  consists of the address  $M_i^d$  on  $P_{bc}$  and the address of the newly created entity on  $P_i$ .

**Step 3.3:** when the broadcaster  $P_{bc}$  receives the message  $B$  from part  $P_i$ , it saves the address of  $M_i^d$  on  $P_i$  as a remote copy of local  $M_i^d$ , and forwards the message to any remote copy or matched entity on remote part ( $P_{remote}$ ). It uses a flag (*isRemoteCopyFlag*) to indicate whether the message is for remote copy or for matched entity. The forwarded message  $C$  to send includes the flag *isRemoteCopyFlag*, the address of  $M_i^d$  on  $P_i$ , and the address of the entity (either remote copy  $M_i^d$  or matched entity  $M_j^d$ ) on  $P_{remote}$ .

**Step 3.4:** when the part  $P_{remote}$  receives the message  $C$  from the broadcaster  $P_{bc}$ , it checks the flag *isRemoteCopyFlag*. If the message is for a remote copy, the remote copy  $M_i^d$  on  $P_{remote}$  is updated to include the address of  $M_i^d$  on  $P_i$ . If the message is for a matched entity, the matched entity  $M_j^d$  on  $P_{remote}$  is updated to include the address of  $M_i^d$  on  $P_i$ .

In Algorithm 9, a broadcaster (part  $P_{bc}$ ) is in charge of updating the remote copy and matched entity information of an entity ( $M_i^d$ ) over all parts.

### 5.5.2 Remove unused entities

In Step 4 of the migration algorithm, unused mesh entities collected in the previous step are removed from local parts. If the entity to remove is a part boundary



entity, it must be removed from other parts where it is kept as remote copies. If the entity to remove is a matched entity, it must be removed from other parts where it is kept as matched entities. Mesh entities are removed from high to low dimension, in the opposite direction of entity creation. Algorithm 10 is the pseudo code to illustrate this procedure.

Given mesh entities to remove (*entitiesToRemove*), Algorithm 10 includes the following main steps:

**Step 4.1:** sends messages to remote parts to update remote copy and matched entity information. For each entity  $M_i^d$  in *entitiesToRemove*[ $d$ ] on part  $P_i$ , if  $M_i^d$  has a remote copy or matched entity on a remote part  $P_{remote}$ , a message ( $A$ ) is sent to the remote part  $P_{remote}$ . The message  $A$  uses a flag (*isRemoteCopyFlag*) to indicate whether the message is for remote copy or matched entity. Along with that flag, the message includes the address of  $M_i^d$  on  $P_i$  and the address of the entity (either remote copy  $M_i^d$  or matched entity  $M_j^d$ ) on  $P_{remote}$ .

**Step 4.2:** when the part  $P_{remote}$  receives the message  $A$  from the part  $P_i$ , it checks the flag *isRemoteCopyFlag*. If the message is for a remote copy, the remote copy  $M_i^d$  on  $P_{remote}$  is updated to remove the address of  $M_i^d$  on  $P_i$ . If the message is for a matched entity, the matched entity  $M_j^d$  on  $P_{remote}$  is updated to remove the address of  $M_i^d$  on  $P_i$ .

**Step 4.3:** removes all the mesh entities in *entitiesToRemove* from high to low dimension on local parts.

## 5.6 Examples and Applications

The first example is to investigate the capability of parallel mesh operations in the FMDB, mesh partition and migration, to deal with matched meshes. The second example is to demonstrate the mesh matching capability to support mesh adaptation on matched meshes. All the initial meshes are generated by the mesh generation tool Simmetrix [34], and then converted into the FMDB matched meshes.

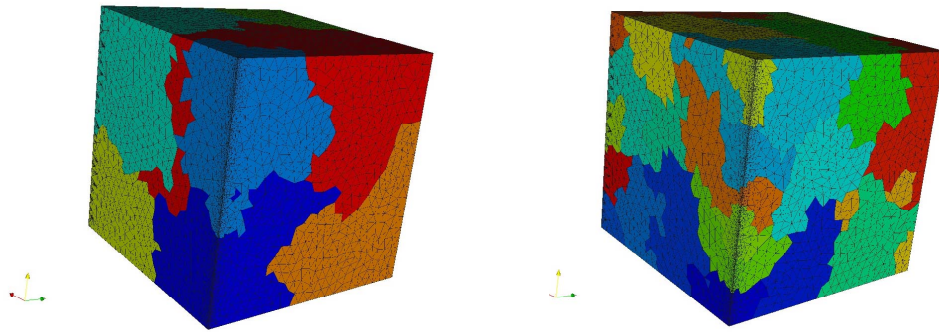
**Data:** a mesh  $M$  of dimension  $\dim$  on a part  $P_i$ , entitiesToRemove  
**Result:** remove unused entities and update remote copy and matched entity information

```

begin
  /* STEP 4.1: send message to remote copy or matched entity
  */
  for  $d \leftarrow \dim - 1$  to 0 do
    for each  $M_i^d \in \text{entitiesToRemove}[d]$  on part  $P_i$  do
      for each remote copy of  $M_i^d$  on part  $P_{\text{remote}}$  do
        isRemoteCopyFlag  $\leftarrow$  true;
        send message  $A$  (isRemoteCopyFlag,  $M_i^d$  on  $P_{\text{remote}}$ ,  $M_i^d$  on  $P_i$ )
        to  $P_{\text{remote}}$ ;
      end
      for each matched entity  $M_j^d$  on part  $P_{\text{remote}}$  do
        isRemoteCopyFlag  $\leftarrow$  false;
        send message  $A$  (isRemoteCopyFlag,  $M_j^d$  on  $P_{\text{remote}}$ ,  $M_i^d$  on  $P_i$ )
        to  $P_{\text{remote}}$ ;
      end
    end
  end
  /* STEP 4.2: update remote copy or matched entity */
  while  $P_{\text{remote}}$  receives message  $A$  from  $P_i$  do
    if isRemoteCopyFlag = true then
      FMDB_Ent_RmvCopy( $M_i^d$ ,  $M_i^d$  on  $P_i$ );
    else
      FMDB_Ent_RmvMatch( $M_j^d$ ,  $M_i^d$  on  $P_i$ );
    end
  end
  /* STEP 4.3: remove unused entities */
  for  $d \leftarrow \dim$  to 0 do
    for each  $M_i^d \in \text{entitiesToRemove}[d]$  on part  $P_i$  do
      FMDB_Part_RmvEnt( $M$ ,  $M_i^d$ );
    end
  end
end

```

**Algorithm 10:** Remove unused mesh entities and update related remote copy and matched entity information from a part.



**Figure 5.5:** The partitioned meshes for 8 parts (left) and 32 parts (right) obtained from mesh partition and migration on the initial 71,966-region mesh on the CCNI BG/L. The colors represent various parts.

### 5.6.1 Mesh partition and migration

A matched mesh of 71,966 regions on a cube geometric model is considered. Two opposite model faces are periodic faces. This mesh contains 1,074 matched vertices, 2,944 matched edges, and 1,871 matched faces.

In the first case, the initial mesh is first loaded and partitioned into 8 parts, and then all mesh entities are migrated back to a single part to write out another mesh. The entity ordering in the mesh file is preserved. The initial mesh file and the output mesh file contain the same matched entity information, this demonstrates that parallel mesh operations can maintain matched entity information properly. All the steps are carried out on 8 processors on the SCOREC Piglet cluster [84], an AMD Opteron machine with eight quad-core nodes and 64GB in total of memory.

In the second case, the initial mesh is loaded and partitioned on the CCNI BlueGene/L system [89], using 8 up to 256 processing cores with 512MB per core memory. The mesh partitioning is carried out through the graph-based partitioner, ParMetis [60]. Figure 5.5 depicts the partitioned meshes for 8 parts and 32 parts, respectively. Table 5.1 lists the average execution time (in seconds) of the mesh partition stage (*Partition* in the 3rd column), mesh migration stage (*Migration* in the 4th column), and total execution (*Total* in the 5th column), along with the total number of regions to migrate (*RgnsToMove* in the 2nd column) during the migration stage. Since the initial mesh is small, the total execution time does not

**Table 5.1:** The average execution time (in seconds) of mesh partition, migration, total execution on the 71,966-region mesh of the cube model running on up to 256 processors on the CCNI BG/L.

Total #parts	#RgnsToMove	Partition (sec)	Migration (sec)	Total (sec)
8	62,984	2.15	29.56	31.71
16	67,470	2.11	29.61	31.72
32	69,716	2.12	30.00	32.12
64	70,839	2.13	30.00	32.13
128	71,401	2.17	30.00	32.17
256	71,683	2.29	30.44	32.73

increase much as the total number of parts increases.

### 5.6.2 Support matched mesh adaptation

The mesh adaptation procedure through local mesh modifications [3], including mesh refinement, coarsening and projecting mesh vertices to geometric model boundaries (called snap operation), is carried out on two types of geometries. All the tests are carried out on one processor on the SCOREC Piglet cluster [84].

The first is a sector of a pipe geometry with *(i)* the front and back model faces being periodic, and *(ii)* the left and right side model faces being periodic. Figure 5.6 depicts an initial coarse mesh and an adapted mesh. After the mesh adaptation procedure, the entities on the front and back periodic boundaries match each other, and the entities on the left and right side periodic boundaries match each other.

The second is a pipe model with two opposite model faces being periodic boundaries. Figure 5.7 depicts an example of an initial 17,976 region mesh to an adapted 735,651 region mesh on the pipe model. After mesh adaptation with an analytical size field [20], as shown in the circle of Figure 5.7, the matched entities classified on two periodic boundaries are overlapped and still match each other.

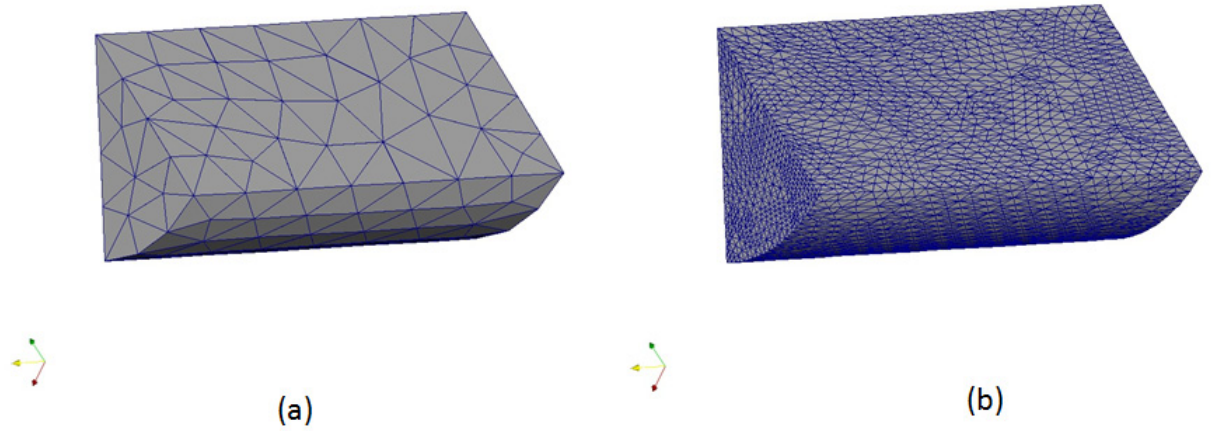


Figure 5.6: Example of mesh adaptation on a geometric model with double periodic boundary conditions: (a) is the initial coarse mesh, and (b) is the adapted mesh going through mesh refinement, coarsening and snap operations.

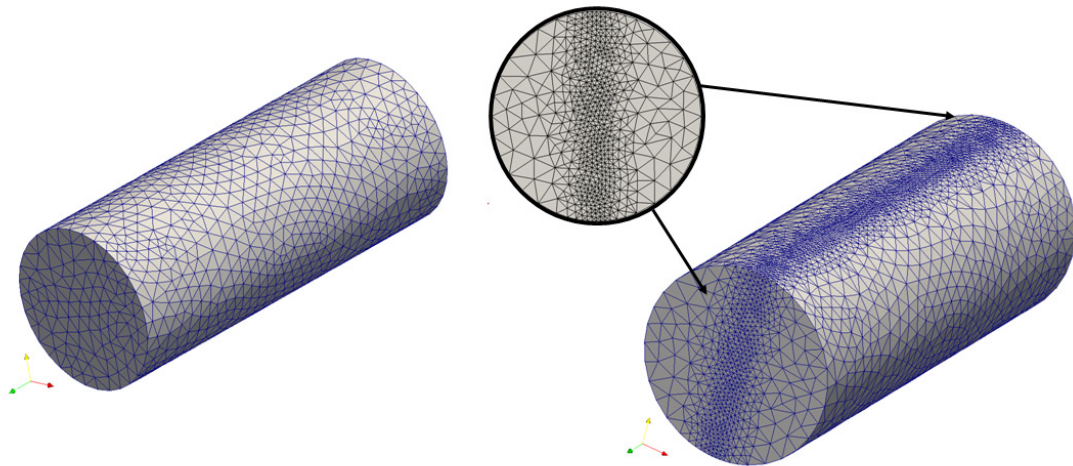


Figure 5.7: Example of mesh adaptation on a coarse mesh on a pipe model with periodic boundary conditions: (a) is the initial mesh of 17,976 regions, and (b) is the adapted mesh of 735,651 regions.

## CHAPTER 6

### MULTIPLE PARTS PER PROCESS

A parallel adaptive mesh-based simulation typically starts with a relative coarse initial mesh, and solution based mesh adaptation is carried out to improve the mesh resolution to efficiently resolve the problem. During this process, the number of mesh entities in the unstructured mesh can increase by orders of magnitude, consequently the number of processing cores, i.e., processors, to effectively solve the problem needs to be increased to account for the mesh size increase.

The original Flexible distributed Mesh Database (FMDB) [15, 16] allows only one part per process in the mesh partition, which constrains the simulation to a fixed number of processors and is not ideal for an adaptive simulation where the mesh size increases dramatically, and thus processors will eventually run out of memory, when the number of mesh entity increase on a part becomes too large. Also, it is not efficient to start with too many processors in the beginning of a simulation loop, especially since a *priori* estimation of the number of mesh entities (and therefore processors) needed is not possible [8].

To address the resulting need to be able to change the number of parts and processors during an adaptive simulation, the capability of multiple parts per process was developed. Allowing multiple parts per process is also useful as a preprocessing stage for partition redistribution. In such a case, the simulation may be performed on various computer architectures with different amount of per core memory, therefore the number of processors required will be different even for a fixed-size mesh problem [8]. For instance, each processing core at NERSC *Hopper* [90] (Cray XE6) can have 2.58GB memory, while the one at ANL *Intrepid* [80] (IBM BG/P) can have 512MB. The simulation which fits in 1,024 cores on Hopper may require more than 4,096 cores on Intrepid.

The objective is to support changing (especially increasing) the number of parts during mesh partitioning. This was implemented as an extension of the FMDB. The distributed mesh data structure based on an enhanced partition model is intro-

duced in §6.1. The global and local partitioning strategies for graph/hypergraph-based mesh partitioning are presented in §6.2, followed by the mesh migration procedure in §6.3. In the end, the performance results of the multiple parts per process partitioning are discussed in §6.4.

## 6.1 Distributed Mesh Data Structure

In parallel adaptive simulations, a mesh is distributed to a set of parts over a number of processes. A distributed mesh data structure supports a topological representation of the distributed mesh and efficient distributed mesh manipulation functions. A distributed mesh representation, referred to as a *partition model*, that will support the needed capabilities includes the information on parts, and part boundaries, and supports mesh migration and mesh partitioning [15, 16].

### 6.1.1 Part and part id

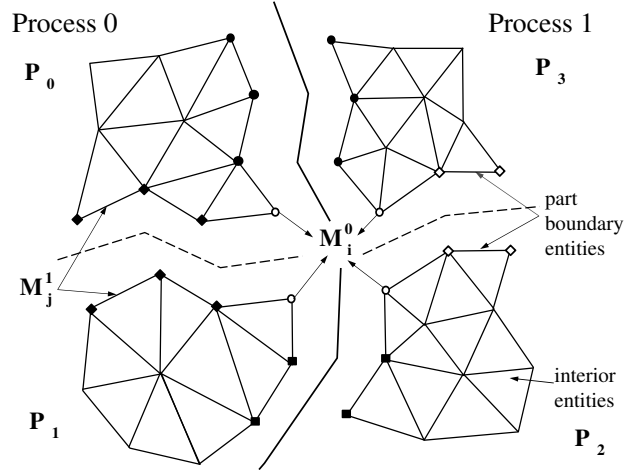
When a mesh is distributed to  $N$  parts, each part is assigned to a process, and more than one part can be assigned to a single process. Herein, a process can be seen as a processing unit with a single (unique) memory space in the parallel computation, and a process can be identified through its unique process rank, denoted by  $C_i$ .

A part is a subset of mesh entities of the entire mesh, uniquely identified by its part handle or id, denoted by  $P_i$ ,  $0 \leq i < N$ . Figure 6.1 depicts a 2D mesh that is distributed on four parts over two processes where each process contains two parts. The dashed lines represent intra-process part boundaries within a process and the solid black lines represent inter-process part boundaries between processes.

A part ( $P_i$ ) can have two kinds of part id's:

- Global part id, denoted by  $P_i$ , which is globally unique over all the processes.
- Local part id, denoted by  $lP_i$ , which is local and unique to only one process where the part resides.

Note, without specification, part id is short for global part id in the following chapter.



**Figure 6.1: Example of a 2D distributed mesh on two processes with two parts per process.**

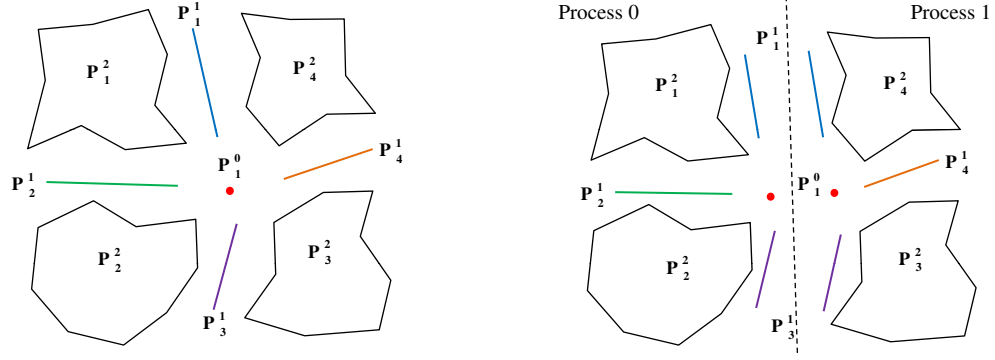
In practice, for representing global part id's, the use of single and continuous integers may be not ideal for applications of adaptive mesh methods, which require (i) the creation of multiple parts per process, (ii) the movement of mesh entities between parts, and (iii) the movement of parts between processes. For instance, one way to represent a global part id is to use a pair of integers, including the local part id and process rank where the part resides, denoted by  $(lP_i, C_i)$ .

Since single integers allow simple storage and modification compared to pairs of integers, this work uses single, non-continuous integers to represent global part id's. The global part id's are calculated based on a default base number (say,  $maxNumPart$ ), which defines the maximum number of parts that a process can contain. The base number  $maxNumPart$  is positive and consistent over all the processes, and can be redefined by the application. If local part id's are assumed to be continuous on each process and all the numbers start from zero, given a part id  $P_i$  on a process rank  $C_i$ , the global part id ( $P_i$ ) and local part id ( $lP_i$ ) can be converted to each other as follows.

$$lP_i = P_i \% maxNumPart; \quad (6.1)$$

$$P_i = C_i \times maxNumPart + lP_i. \quad (6.2)$$





**Figure 6.2:** The partition model of the distributed mesh in Figure 6.1 (left) and the partition model viewed from the process level (right).

The process rank  $C_i$  can be also calculated through the global part id  $P_i$ .

$$C_i = P_i \div \text{maxNumPart}. \quad (6.3)$$

### 6.1.2 Enhanced partition model

A partition model is a topological representation of a mesh partitioning, and consists of partition entities. Through partition classification and reverse partition classification, a partition model defines the relations between mesh entities and partition entities [15, 16].

An enhanced partition model allows an arbitrary number of parts existing on a single process, and contains two levels of domain decomposition views including: (i) the partition view at the part level and (ii) the partition view at the process level. For the 2D distributed mesh in Figure 6.1, Figure 6.2 depicts the whole partition model and its partition view from the process level .

In the part level view, each part is treated as a serial mesh with the addition of part boundary entities and their remote copies. Each part is uniquely identified by a part handle or id, denoted by  $P_i$ . Duplicated part boundaries can exist on a process. The partition classification is maintained for each mesh entity on a part.

In the process level view, each process stores (i) all parts that reside on the process, and (ii) all partition entities on which the on-process mesh entities are classified. If a partition entity  $P_i^d$  whose residence parts contain a part  $P_i$  on a

process  $C_j$ , i.e.  $P_i \in \mathcal{P}[P_i^d]$ , partition entity  $P_i^d$  will be stored on the process  $C_j$ .

In practice, for effective manipulation of multiple parts per process, a *mesh instance* is defined on each process to contain all the part handles on that process. Part handles are accessible only through the mesh instance on a process. The number of part handles stored in a mesh instance decides the existing number of parts residing on the process. The ordering of part handles stored in a mesh instance decides the continuous local part id's of the parts on the process, and thus their global part id's.

### 6.1.3 Partition entities

As the mesh changes in parallel computations such as mesh (re)partitioning, the partition model changes, as well as its associated partition entities. For the sake of memory efficiency, a partition entity does not store reverse partition classification explicitly. In addition to the dimension, id, residence parts and the owning part [15, 16], a partition entity stores an entity counter to record the number of existing mesh entities classified on the partition entity. The entity counters help to clean up unused partition entities in the partition model, and thus help to calculate neighboring parts on-the-fly in any stage of mesh modification.

The rules to calculate an entity counter in a partition entity  $P_i^d$  include:

- When the partition entity  $P_i^d$  is first created, the counter is set as zero.
- If a mesh entity is classified on the partition entity  $P_i^d$ , the counter is increased by one.
- If a mesh entity changes its partition classification from the partition entity  $P_i^d$  to another partition entity  $P_j^q$ , then the counter in  $P_j^q$  is increased by one and the one in  $P_i^d$  is decreased by one.
- If a mesh entity classified on the partition entity  $P_i^d$  is deleted from the mesh, the counter in  $P_i^d$  is decreased by one.
- If the counter is decreased to zero, the partition entity  $P_i^d$  will be deleted from the partition model.

```

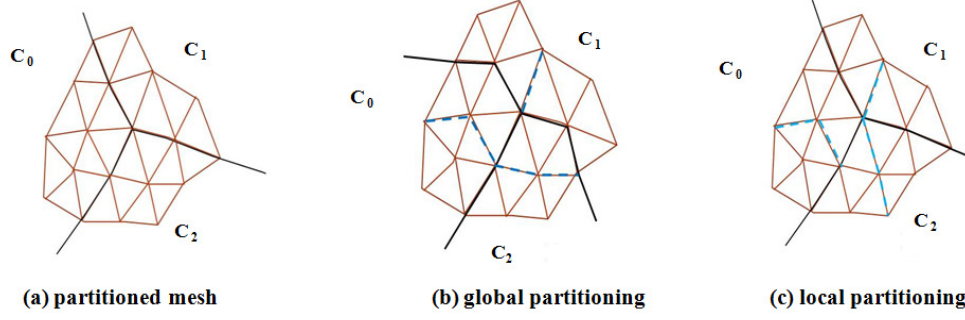
Data: a part  $P_i$  on a process  $C_i$ 
Result: store all neighboring parts for the part  $P_i$  into  $nborParts$ 
begin
  /* STEP 1: traverse all partition entities on process */
  for each partition entity  $P_i^d$  stored on  $C_i$  do
    for each part  $P_j \in \mathcal{P}[P_i^d]$  do
      if  $P_j = P_i$  then
        store  $P_i^d$  into  $ptnEnts$ ;
        break;
      end
    end
  end
  /* STEP 2: traverse partition entities in  $ptnEnts$  */
  for each  $P_i^d \in ptnEnts$  do
    for each part  $P_j \in \mathcal{P}[P_i^d]$  do
      if  $P_j \neq P_i$  then store  $P_j$  into  $nborParts$ ;
    end
  end
  /* STEP 3: make neighboring parts unique */
  make the neighboring parts unique in  $nborParts$ ;
  return  $nborParts$ ;
end

```

**Algorithm 11: Get neighboring parts of a specific part on a process.**

Algorithm 11 illustrates the procedure to collect the neighboring parts for a specific part ( $P_i$ ) on a process ( $C_j$ ). In the first step, the algorithm traverses all partition entities stored on the process, and collects all partition entities where the mesh entities on the current part  $P_i$  are classified, through checking if the residence parts of a partition entity  $P_i^d$ , i.e.  $\mathcal{P}[P_i^d]$ , contain the part  $P_i$ . If yes, the partition entity will be stored into a list  $ptnEnts$ . In the second step, the algorithm traverses all partition entities in the list  $ptnEnts$ . For each partition entity, it traverses its residence parts, and stores the residence parts except the current part  $P_i$  into the neighboring part list  $nborParts$ . The third step is to make sure that the collected neighboring parts of the part  $P_i$  are unique in the return list  $nborParts$ .

Since the number of partition entities stored on a process is much smaller than the mesh entities on the process, Algorithm 11 will be more efficient than an algorithm which requires traversing all mesh entities on any part boundary to obtain



**Figure 6.3:** Example of a mesh distribution from three parts to six parts through the global (b) and local (c) partitioning strategies.

the neighboring parts of a given part.

## 6.2 Global and Local Graph-based Partitioning

To obtain the destination part labeling for partition objects, two partitioning strategies can be used to provide data to the graph/hypergraph based partitioning tools: the global partitioning and local partitioning strategies. The global partitioning considers all the graph nodes and edges over all processes, and provides a balanced partition with the global minimum inter-part communication. On the other hand, the local partitioning considers only the on-process graph nodes and edges without knowing the existence of other graph nodes and edges on other processes. Thus the local partitioning can be carried out independently on each process.

Figure 6.3 shows an example of a mesh distribution from three parts to six parts. Figure 6.3a is an initial 2D distributed mesh on three parts. The redistribution of the 2D mesh can be performed by the global or local partitioning. The black solid lines represent the inter-process part boundaries, and the dashed lines represent the on-process part boundaries. In the global partitioning in Figure 6.3b, the original three inter-process part boundaries are changed. On the other hand, in the local partitioning in Figure 6.3c, each previous part is split into two smaller parts, thus the original three inter-process part boundaries stay fixed.

Depending on the requirements of applications, the two partitioning strategies can be used separately or in combination. Since it considers only graph nodes and edges on a process, the local partitioning requires less computational resources

(both time and memory) compared to the global partitioning, especially with a large number of processes. However, due to the limited local information, the quality of the partition will be reduced if the local partitioning is used repeatedly.

The current study uses the Zoltan partitioning library [14] to perform the graph/hypergraph-based partitioning for unstructured meshes. Both the global and local partitioning through the graph/hypergraph-based partitioning tools require the application provide (i) all partition objects on a process, and (ii) the neighboring partitioning objects for all partition objects on a process.

### 6.2.1 Partition objects

In a mesh partitioning for an unstructured mesh, partition objects can be either mesh entities not on the boundary of any higher dimension mesh entities and not in any P-sets or P-sets (please refer to the discussions in §4.4.1).

```

Data: a mesh  $M$  with  $N$  parts on a process  $C_i$ 
Result: store all partition objects in  $M$  into a partition object list ptnObj
begin
  for  $i \leftarrow 0$  to  $N - 1$  do
     $P_i \leftarrow \text{FMDB\_Mesh\_GetPart}(M, i);$ 
     $\text{FMDB\_Part\_GetPtnObj}(P_i, \text{ptnObj});$ 
  end
  return ptnObj;
end

```

**Algorithm 12:** Get all partition objects in the mesh instance on a process.

The pseudo code of Algorithm 12 illustrates the procedure to collect all partition objects on all parts in a process. To traverse all partition objects on each part in the process (by calling the procedure *FMDB\_Part\_GetPtnObj*), a loop over the number of parts on the process is required in the algorithm. The procedure *FMDB\_Part\_GetPtnObj*, as illustrated in Algorithm 2 in §4.4.1, is to collect all partition objects on a part.

### 6.2.2 Collecting neighboring partition objects for global partitioning

Since two neighboring partition objects can exist on two separate parts, it is necessary to pre-store neighboring partition object information on the remote copies of part boundary entities to reduce the communication costs. In §4.4.2, the procedure *FMDB\_Part\_ExchPtnObj*, as illustrated in Algorithm 4, is to exchange the adjacent partition object information between part boundary entities and their remote copies in a part, storing the remote partition object information in part boundary entities, and the procedure *FMDB\_Part\_GetNborPtnObj*, as illustrated in Algorithm 5, is to collect all neighboring partition objects for all partition objects on a part.

**Data:** a mesh instance  $M$  with  $N$  parts on a process  $C_i$   
**Result:** store the neighboring partition objects in `nborPtnObj` of all partition objects in  $M$

```

begin
  for  $i \leftarrow 0$  to  $N - 1$  do
     $P_i \leftarrow \text{FMDB\_Mesh\_GetPart}(M, i);$ 
     $\text{FMDB\_Part\_GetNborPtnObj}(P_i, \text{nborPtnObj});$ 
  end
  return nborPtnObj;
end

```

**Algorithm 13:** Get neighboring partition objects for each partition object on a process.

The pseudo code of Algorithm 13 illustrates the procedure to collect all neighboring partition objects for all partition objects on a process for the global graph-based partitioning. Compared to Algorithm 5, Algorithm 13 adds a loop which traverses all parts on the process, and calls the procedure *FMDB\_Part\_GetNborPtnObj* on each part.

### 6.2.3 Collecting neighboring partition objects for local partitioning

Since a local mesh partitioning only considers partition objects and their on-process neighboring partition objects on each process, a part boundary entity only needs to communicate with its on-process remote copies to exchange the adjacent partition object information. The pseudo code of Algorithm 14 illustrates the pro-

```

Data: a  $d$ -dimension ( $d \in \{2, 3\}$ ) mesh instance on a process  $C_i$ 
Result: exchange the on-process partition object information between part
           boundary entities that are adjacent to partition objects.
begin
  /* STEP 1:  get all part boundary entities adjacent to
  partition objects                                     */
  for each  $M_i^{d-1} \in \{M\{M^{d-1}\}\}$  do
    if FMDB_Ent_IsOnPartBdry( $M_i^{d-1}$ ) = true then
      store  $M_i^{d-1}$  into entOnPartBdry;
    end
  /* STEP 2:  exchange between part boundary entities                                     */
  for each  $M_i^{d-1} \in \text{entOnPartBdry}$  do
     $M_j^d \leftarrow M_i^{d-1} \in \partial(M_j^d)$ ;
    if FMDB_Ent_IsInSet( $M_j^d$ ) then
      adjPtnObj  $\leftarrow$  FMDB_Ent_GetSet( $M_j^d$ ) ;
    else adjPtnObj  $\leftarrow M_j^d$ ;
    /* tag data to only on-process remote copies                                     */
    for each remote copy  $M_i^{d-1'}$  of  $M_i^{d-1}$  on remote part  $P_{\text{remote}}$  do
      FMDB_Ent_SetTagData( $M_i^{d-1'}$ , adjPtnObj);
    end
  end
end
end

```

**Algorithm 14:** Exchange the on-process partition object information between part boundary entities on a process for the local partitioning.

cedure to exchange the adjacent partition object information. The input of this algorithm is a mesh instance on a process. In the first step, the algorithm traverses all mesh entities to collect part boundary entities that bound partition objects on the process. In the second step, for each entity  $M_i^{d-1}$  in the collected part boundary entity list *entOnPartBdry*, the algorithm gets the adjacent partition object (*adjPO*, either an entity or a P-set) of  $M_i^{d-1}$  through adjacencies, then tags the adjacent partition object to its on-process remote copies. The remote copies can be accessed without message passing through the share memory space.

Once the adjacent partition objects are pre-stored in part boundary entities, Algorithm 13 can be used to collect all neighboring partition objects for all partition objects on a process.

For illustration purposes, Algorithm 14 only considers neighboring partition objects through edge/face adjacencies in a 2D/3D manifold mesh in which partition objects are mesh faces/regions.

#### 6.2.4 Post-processing

A mesh partitioning procedure indicates which partition object should go to which destination part, and a mesh migration procedure moves mesh entities from one part to another part. It is necessary to construct the input of the migration from the partitioning result.

In the message passing implementation, when one part sends a message and another part receives the message, only the process rank of the sending part, instead of the part id, can be obtained automatically. However, to build the relations between part boundary entities and their remote copies through message passing, both the destination and source parts are required. Since it is not efficient to access the source part of a mesh entity through either (i) storing the local part id (of the source part) inside each mesh entity on a part or (ii) traversing the whole mesh on a process, it is required to construct the input of the migration procedure as a list of partition objects to migrate and their destination and source part id's, referred to as *POsToMove*.

In the implementation, *POsToMove* can be constructed directly through the partitioning result obtained from the graph/hypergraph based partitioners in the Zoltan library, which can contain partition objects, the destination part id's, and the source part id's tagged to the partition objects. However, since the partitioning result returns single, continuous destination part id's, a conversion may be required to change the output destination part id's to the single, non-continuous part id's, especially with the various number of parts setting on each process.

### 6.3 Mesh Migration Algorithm

#### 6.3.1 Entity migration algorithm

A mesh migration algorithm is designed to move mesh entities between parts. Algorithm 15 illustrates the procedure of migrating mesh entities (the procedure



```

Data: a mesh instance on a process M, POsToMove
Result: migrate partition objects in POsToMove
begin
  /* STEP 1:  create new empty parts on process */
  if curNumPart < tgtNumPart then
    for  $i \leftarrow \text{curNumPart}$  to  $\text{tgtNumPart}$  do FMDB_Mesh_CreatePart(M);
  end
  /* STEP 2:  collect entitiesToUpdate and entitiesToRemove */
  for each  $M_i^d \in \text{POsToMove}$  do
    insert  $M_i^d$  into entitiesToUpdate[d];
    for each  $M_j^q \in \{\partial(M_i^d)\}$  do insert  $M_j^q$  into entitiesToUpdate[q];
  end
  FMDB_Ent_SetResidencePart(POsToMove, entitiesToUpdate[q]);
  for  $d \leftarrow 3$  to 0 do
    for each  $M_i^d \in \text{entitiesToUpdate}[d]$  do
      if  $P_{\text{local}} \notin \mathcal{P}[M_i^d]$  then insert  $M_i^d$  into entitiesToRemove[d];
    end
  end
  /* STEP 3:  exchange entities in entitiesToUpdate */
  for  $d \leftarrow 0$  to 3 do
    FMDB_Mesh_ExchEnt(M, entitiesToUpdate[d]);
  end
  /* STEP 4:  remove unused entities in entitiesToRemove */
  for  $d \leftarrow 3$  to 0 do
    for each  $M_i^d \in \text{entitiesToRemove}[d]$  do
      if FMDB_Ent_IsOnPartBdry( $M_i^d$ ) = true then
        remove copies of  $M_i^d$  on remote parts;
      end
      remove  $M_i^d$ ;
    end
  end
  /* STEP 5:  remove unused parts on process */
  if  $\text{tgtNumPart} < \text{curNumPart}$  then
    for  $i \leftarrow \text{tgtNumPart}$  to  $\text{curNumPart}$  do FMDB_Mesh_DelPart(M);
  end
  /* STEP 6:  update ownerships of partition entities */
  for each  $P_i^d$  on the process do update the owning part of  $P_i^d$ ;
end

```

**Algorithm 15:** Migrate partition objects in POsToMove (FMDB\_Mesh\_MigrateEnt).

*FMDB\_Mesh\_MigrateEnt*). The input of the algorithm includes (i) the mesh instance (*Mesh*) and (ii) a list of partition objects to migrate with their source and destination parts (*POsToMove*).

For illustration purposes, only mesh entities that do not bound any other higher dimension mesh entities are considered as partition objects in this algorithm.

The following are the main steps of the migration procedure in Algorithm 15:

**Step 1:** creates new empty parts on a process. It compares the current existing number of parts on a process (*curNumPart*) and the target number of parts on the process (*tgtNumPart*). If *tgtNumPart* is greater than *curNumPart*, new empty part(s) are created on the process and added to the end of the mesh instance (*Mesh*). The number of new part(s) equals the result of subtracting *curNumPart* from *tgtNumPart*.

**Step 2:** based on *POsToMove*, collects the entities to process (*entitiesToUpdate*) whose partitioning data (residence parts  $\mathcal{P}$  and partition classification) will be updated after migration. The residence parts of these entities (*entitiesToUpdate*) are calculated, and the partition classification is updated to reflect the changes in the updated partition model. Based on the calculated residence parts  $\mathcal{P}$ , this step also determines and collects the entities to remove (*entitiesToRemove*) from the local parts.

**Step 3:** migrates required mesh entities in *entitiesToUpdate* to the destination parts. The adjacencies associated with the newly created entities are properly maintained on the destination parts, and remote copy information of the affected part boundary entities are updated.

**Step 4:** deletes mesh entities collected in *EntitiesToRemove* from the local parts. The adjacencies associated with the unused entities are deleted also. For any entity in *EntitiesToRemove* that is on a part boundary, it will be removed from other parts where it is kept as remote copies.

**Step 5:** removes the unused parts stored at the end of the mesh instance (*Mesh*), if *tgtNumPart* is less than *curNumPart* on the process. The number of deleted

part(s) equals the result of subtracting  $tgtNumPart$  from  $curNumPart$ .

**Step 6:** The ownerships of the partition entities on a process are updated.

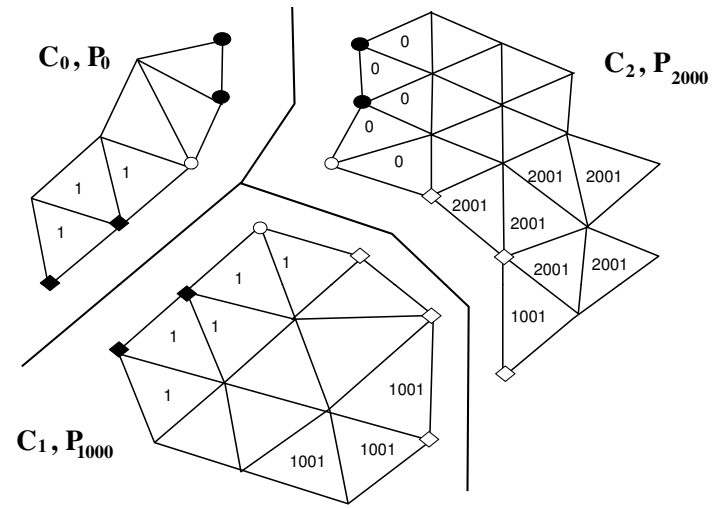
Herein, the current existing number of parts on a process ( $curNumPart$ ) is equal to the size of part handles stored in a given mesh instance ( $Mesh$ ) on a process, and the target number of parts that should exist on the process after the migration ( $tgtNumPart$ ) is defined by the application that requests the mesh migration procedure.

Figure 6.4 and Figure 6.5 illustrate the entity migration procedure. The initial mesh of Figure 6.4a is a 2D mesh distributed on 3 parts over 3 processes. The solid lines represent the inter-process part boundaries. As illustrated in Figure 6.4a, the partition objects (mesh faces) to migrate are labeled with their destination part id's. The objective is to migrate mesh entities to destination parts, creating 6 parts on 3 processes with 2 parts per process. First, since the target number of parts ( $tgtNumPart = 2$ ) is greater than the current number of parts ( $curNumPart = 1$ ) on each process, a new empty part (see the shaded plane in Figure 6.4b) is created and added to the end of the mesh instance on each process. Then, all affected entities are transferred to their destination parts, and all unused entities are removed from the local parts. The adjacencies and remote copies of these mesh entities are updated, as well as the partition model and partition entities. Figure 6.5c depicts the resulting mesh, in which the dashed lines represent the intra-process part boundaries. The partition model of the resulting mesh is presented in Figure 6.5d. As illustrated in the two figures, the part id's are labeled based on the number of 1,000.

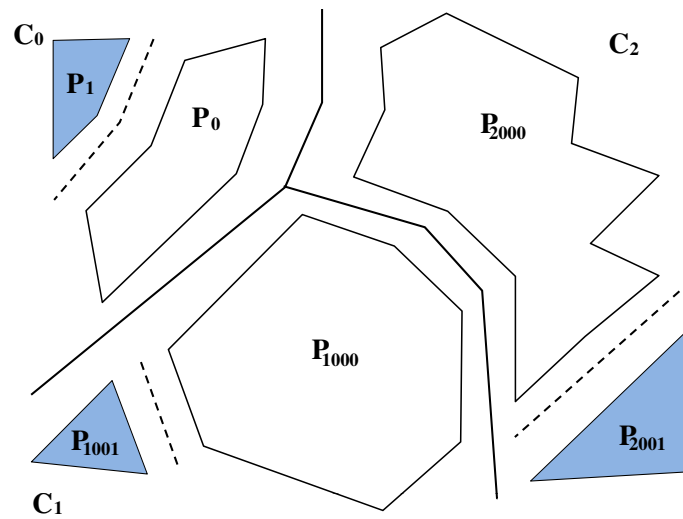
In Algorithm 15, *Step 1* and *Step 5* are easy to understand. For detailed explanations on *Steps 2, 4, 6* in Algorithm 15, please refer to the migration algorithm in references [15, 16]. *Step 3* will be discussed in the next subsection.

### 6.3.2 Communications between part boundary entities

*Step 3* of the migration procedure is used to demonstrate how to handle communications between part boundary entities during the migration (the procedure  $FMDB\_Mesh\_ExchEnt$ ), since it involves three rounds of communications.



(a) initial mesh with partition objects and destinations



(b) view of parts after creating empty parts

Figure 6.4: Example of 2D mesh migration from 3 parts to 6 parts on 3 processes (part 1).

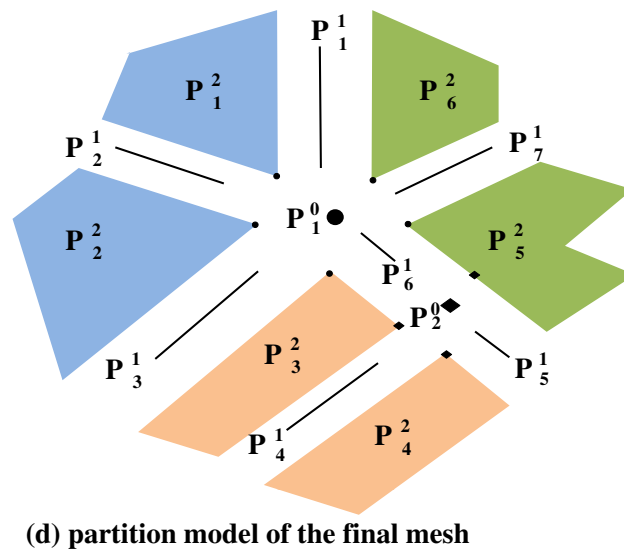
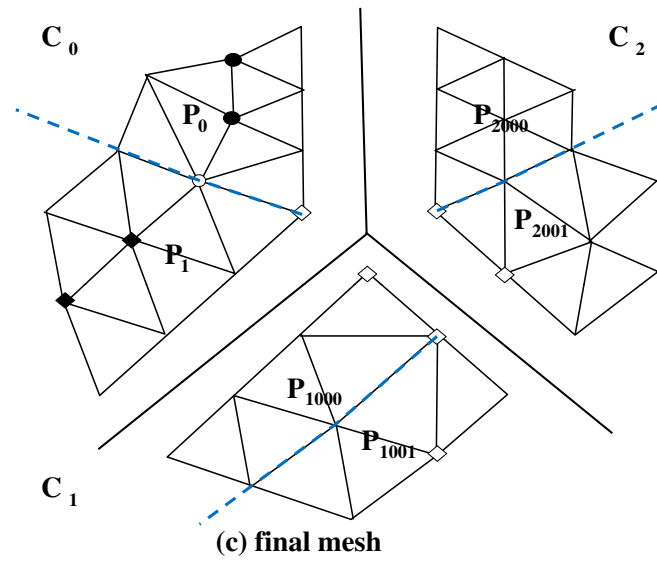


Figure 6.5: Example of 2D mesh migration from 3 parts to 6 parts on 3 processes (part 2).

```

Data: a mesh instance  $M$ , entitiesToUpdate[d]
Result: create entities on destination parts and update remote copies
begin
  /* STEP 3.1: send message to destination parts */
  for each  $M_i^d \in \text{entitiesToUpdate}[d]$  do
    if  $P_{src} \neq \text{minimum part id where } M_i^d \text{ exists}$  then continue;
    for each part id  $P_i \in \mathcal{P}[M_i^d]$  do
      if  $M_i^d$  has remote copy on  $P_i$  then continue;
       $C_i \leftarrow \text{FMDB\_Part\_GetRank}(P_i)$ ;
      send message A (address of  $M_i^d$  on  $P_{src}$ , entity information of  $M_i^d$ ,
         $P_i$  and  $P_{src}$ ) to  $C_i$ ;
    end
  end
  /* STEP 3.2: create entities and report to broadcaster */
  while  $C_i$  receives message A from  $P_{bc}$  do
    part handle  $P_i \leftarrow \text{FMDB\_Mesh\_GetPart}(M, P_i)$ ;
    create  $M_i^d$  on  $P_i$  with the entity information of  $M_i^d$ ;
    if  $M_i^d \neq \text{partition object}$  then
       $C_{bc} \leftarrow \text{FMDB\_Part\_GetRank}(P_{bc})$ ;
      send message B (address of  $M_i^d$  on  $P_{bc}$ , address of  $M_i^d$  created,  $P_i$ 
        and  $P_{bc}$ ) to  $C_{bc}$ ;
    end
  end
  /* STEP 3.3: broadcaster forwards new entity information */
  while  $C_{bc}$  receives message B from  $P_i$  do
     $M_i^d \leftarrow \text{entity located in the address of } M_i^d \text{ on } P_{bc}$ ;
     $M_i^d$  saves the address of  $M_i^d$  on  $P_i$  as for the remote copy on  $P_i$ ;
    for each remote copy of  $M_i^d$  on remote part  $P_{remote}$  do
       $C_{remote} \leftarrow \text{FMDB\_Part\_GetRank}(P_{remote})$ ;
      send message C (address of  $M_i^d$  on  $P_{remote}$ , address of  $M_i^d$  on  $P_i$ ,  $P_i$ 
        and  $P_{remote}$ ) to  $C_{remote}$ ;
    end
  end
  /* STEP 3.4: update remote copies */
  while  $C_{remote}$  receives message C from  $P_{bc}$  do
     $M_i^d \leftarrow \text{entity located in the address of } M_i^d \text{ on } P_{remote}$ ;
     $M_i^d$  saves the address of  $M_i^d$  on  $P_i$  as for the remote copy on  $P_i$ ;
  end
end

```

**Algorithm 16:** Exchange mesh entities in the migration procedure (FMDB\_Mesh\_ExchEnt).

In *Step 3* of the migration procedure in Algorithm 15, mesh entities are exchanged from low to high dimensions, because a higher dimension mesh entity is bounded by lower dimension mesh entities [15, 16]. This step exchanges entities from dimension 0 to 3, transfers entities to the destination parts and updates the remote copies of the affected part boundary entities. Algorithm 16 illustrates the procedure that exchanges the entities contained in  $entitiesToUpdate[d]$ , where  $d = 0, 1, 2, 3$ . The input of this algorithm includes (i) the mesh instance on a process (*Mesh*) and (ii)  $entitiesToUpdate[d]$  which consists of a list of mesh entities and their source parts. The source parts of the mesh entities to process are derived from their bounding partition objects in the previous *Step 2* in the migration.

The following are the main steps of the entity exchanging procedure in Algorithm 16:

**Step 3.1:** sends a message to the destination parts. For each entity  $M_i^d$  in  $entitiesToUpdate[d]$ , the part with the minimum part id where  $M_i^d$  exists is assigned as the *broadcaster* ( $P_{bc}$ , i.e.  $P_{src}$ ), and sends a message to the destination parts ( $P_i$ ) to create new mesh entities. Before sending a message to  $P_i$ , the source part  $P_{src}$  first checks if  $M_i^d$  already exists on  $P_i$  through its remote copies, and then determines on which process the part  $P_i$  exists ( $C_i$ ). If  $M_i^d$  has a copy on  $P_i$  already, it will not send the message. For each entity  $M_i^d$  to migrate,  $P_{src}$  sends to process  $C_i$  a message  $A$ , composed of (i) the address of  $M_i^d$  on  $P_{src}$ , (ii) the information of  $M_i^d$  necessary for entity creation, (iii) the source part id  $P_{src}$  and (iv) the destination part id  $P_i$ .

**Step 3.2:** creates a new entity and reports the information to the broadcaster. When the process  $C_i$  receives the message  $A$  from the part  $P_{bc}$ , it analyzes the message and decides that the part  $P_i$  is the destination part. Then a new entity  $M_i^d$  is created on  $P_i$ . If the newly created entity  $M_i^d$  is not a partition object,  $P_i$  should send the address of  $M_i^d$  back to the sender ( $M_i^d$  on  $P_{bc}$ ) to update the remote copy relation. The message  $B$  is sent back to the process where  $P_{bc}$  exists ( $C_{bc}$ ), and is composed of (i) the address of  $M_i^d$  on  $P_{bc}$ , (ii) the address of  $M_i^d$  created on  $P_i$ , (iii) the source part id  $P_i$  and (iv) the destination part id  $P_{bc}$ .

**Step 3.3:** forwards the new entity information from the broadcaster. When the process  $C_{bc}$  receives the message  $B$  from the part  $P_i$ , the address of  $M_i^d$  on  $P_i$  is saved as the remote copy of  $M_i^d$  on  $P_{bc}$ . The message  $C$  is then forwarded to all other remote copies of  $M_i^d$  on remote parts ( $P_{remote}$ ). Before sending the message to each remote part,  $P_{bc}$  determines on which process the part  $P_{remote}$  exists.

**Step 3.4:** updates the remote copies. When the process  $C_{remote}$  receives the message from  $P_{bc}$ , it updates the remote copy of  $M_i^d$  on the part  $P_{remote}$  to include the address of  $M_i^d$  on  $P_i$ .

In the message passing, when a part  $P_i$  sends a message to another part  $P_j$ , it needs to determine on which process the part  $P_j$  exists and then sends the message to the process. In the implementation, if two parts are on the same process, they can get the information directly through the shared memory space. The function *FMDB\_Part\_GetRank* performs the conversion between a part id and its corresponding process rank through Equation 6.3. On the contrary, whenever a process  $C_i$  receives a message, to set up remote copy relations, it needs to figure out which part sends the message (the source part) and which part should handle the message (the destination part). As illustrated in Algorithm 16, the current approach is to pack the part id's of both the source and the destination parts within a message. Thus when a process receives a message, the process unpacks the message and knows the destination part, and then the destination part will handle the remains of the message and build remote copy connections with the entity from the source part.

### 6.3.3 Entire part migration algorithm

There are applications where multiple parts are defined on one process and then specific parts are migrated to another process. This procedure can be more efficient than the general mesh entity migration procedure, since the only data that is changed is the partition model and its associated partition entities. Migrating an entire part can be viewed as two steps: (i) creates a new part on the destination process, and then (ii) migrates all the mesh entities from a given source part to the destination part. The second step can call the mesh entity migration procedure.



Algorithm 17 illustrates the procedure of migrating an entire part from one process to another process. The input of this algorithm includes the mesh instance on a process ( $Mesh$ ), the source part to migrate with part id  $P_i$ , and the destination process  $C_j$ . It is assumed that the destination part will be added to the mesh instance on the destination process  $C_j$  as the last part.

The following are the main steps of the part migration procedure in Algorithm 17:

**Step 1:** sets up the target number of parts on processes. On the process where the source part  $P_i$  exists ( $C_i$ ), the target number of parts is set to the existing number of parts on the process minus one ( $curNumPart-1$ ). On the destination process  $C_j$ , the target number of parts is set to the existing number of parts on the process plus one ( $curNumPart+1$ ).

**Step 2:** creates a new empty part on the destination process  $C_j$ . Then  $C_j$  sends a message  $A$  (the part id of the new part  $P_j$ ) to the process  $C_i$  where the source part  $P_i$  exists.

**Step 3:** receives the message and prepares a list of partition objects to migrate. When the process  $C_i$  receives the message  $A$  from  $C_j$ , it traverses all partition objects on the part  $P_i$ , and constructs a list of partition objects to migrate and their destination and source parts ( $POsToMove$ ), where the destination parts are  $P_j$  and the source parts are  $P_i$ .

**Step 4:** calls the mesh entity migration procedure with the mesh instance  $Mesh$  and  $POsToMove$  as the input.

In the mesh entity migration in *Step 4*, since  $tgtNumPart$  is less than  $curNumPart$  on the process  $C_i$ , the unused part  $P_i$  will be deleted automatically. For illustration purposes, Algorithm 17 only considers partition objects that are mesh entities.

**Data:** a mesh instance  $M$ , the source part  $P_i$ , the destination process  $C_j$   
**Result:** migrate the entire part  $P_i$  to the process  $C_j$   
**begin**

```

    /* STEP 1:  set tgtNumPart on affected processes          */
     $C_i \leftarrow \text{FMDB\_Part\_GetRank}(P_i)$ ;
    if the current process is  $C_i$  then
         $\text{tgtNumPart} \leftarrow \text{curNumPart} - 1$ ;
    end
    if the current process is  $C_j$  then
         $\text{tgtNumPart} \leftarrow \text{curNumPart} + 1$ ;
    end
    /* STEP 2:  create new part and send message from the    */
    destination
    if the current process is  $C_j$  then
         $N \leftarrow \text{FMDB\_Mesh\_GetNumPart}(M)$ ;
         $P_j \leftarrow \text{FMDB\_Mesh\_CreatePart}(M, N)$ ;
        send a message  $A$  (part id  $P_j$ ) to  $C_j$ ;
    end
    /* STEP 3:  receive message and prepare POsToMove on the */
    source
    while  $C_i$  receives message  $A$  from  $C_j$  do
        for each entity  $M_i^d$  on  $P_i$  do
            if  $M_i^d$  does not bound any higher order mesh entity then
                initialize a pair of integers pid;
                 $\text{pid.first} \leftarrow P_j$ ;           // the destination part id
                 $\text{pid.second} \leftarrow P_i$ ;         // the source part id
                 $\text{POsToMove}[M_i^d] \leftarrow \text{pid}$ ;
            end
        end
    end
    /* STEP 4:  call entity migration procedure              */
     $\text{FMDB\_Mesh\_MigrateEnt}(M, \text{POsToMove})$ ;
end

```

**Algorithm 17:** Migrate an entire part from one process to another process.

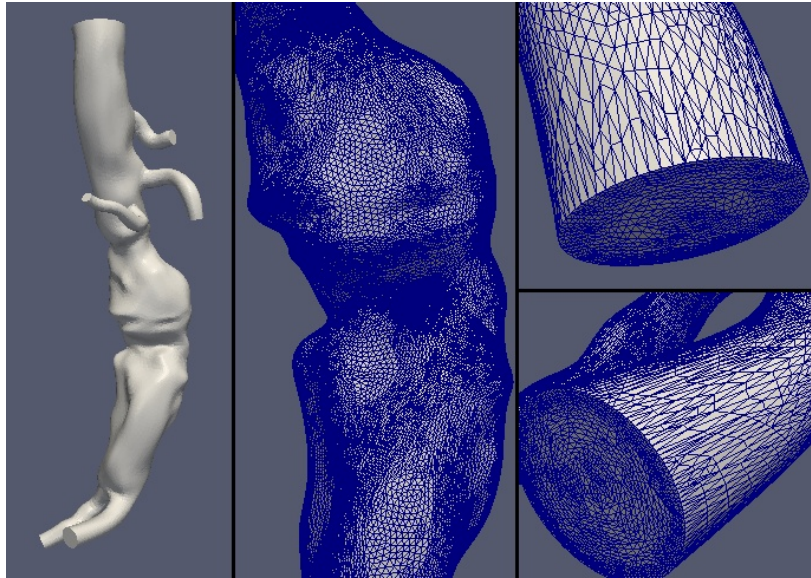


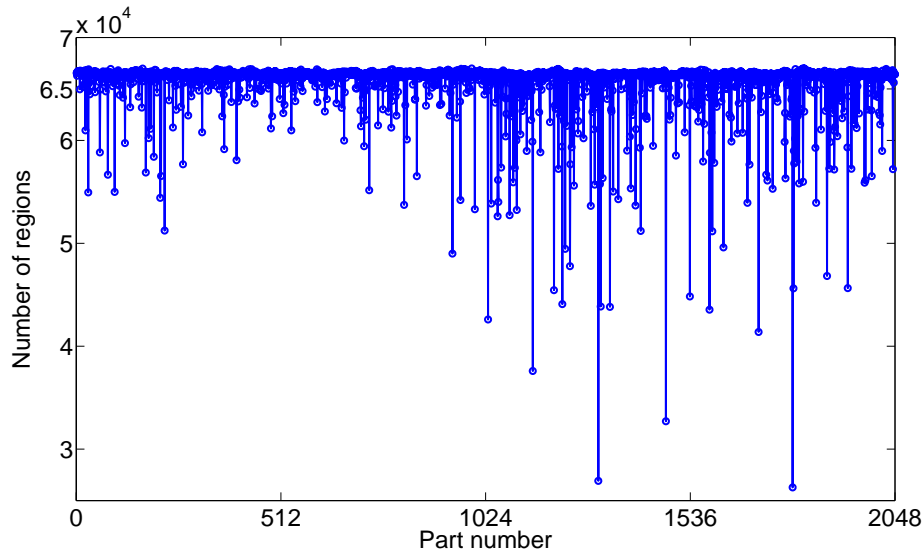
Figure 6.6: Geometry and a mesh of an AAA model [1].

## 6.4 Examples and Applications

This section demonstrates the capability and efficiency of multiple parts per process procedure in operations of importance to parallel adaptive simulations. In the following examples, meshes on a patient-specific abdominal aortic aneurysm (AAA) model are considered (see the model of Figure 6.6). The meshes are obtained through adaptive refinement on an anisotropic mesh obtained from previous adaptation cycles.

The first and second examples (in §6.4.1 and §6.4.2) investigate the performance of global and local partitioning strategies to create partitions containing multiple parts per process. The graph-based partitioner ParMetis [60] (the latest version 4.0.2 [91]) is used in both examples. The time usage of the two main stages of mesh partition and mesh migration is measured. Section §6.4.3 gives one adaptive analysis usage of multiple parts per process, combined with predictive load balancing [1]. §6.4.4 presents an actual use case of a large-scale adaptive fluid simulation on partitioned meshes of billions of elements.

The quality of a partition is measured in terms of region (partition object) imbalance ratio of a partition, which is defined as the maximum number of mesh regions on a part over the average number of regions per part across the partition.



**Figure 6.7: Number of regions on each part of a 133M region mesh for a partition with 2,048 parts.**

An input to the load balancing procedure is the imbalance tolerance (e.g., 1.03) [92]. If the number of mesh partition objects on each part over the average number of mesh regions across the partition is smaller than the imbalance tolerance, the balance of the partition is satisfied. For example, a mesh with 133 million regions is distributed on 2,048 parts, and the average number of regions on each part is  $133,973,440/2,048 = 65,417$ . If a part of 66,937 regions contains more regions than all the other 2,047 parts, then the region imbalance ratio of this partition equals  $66,937/65,417 = 1.023$  (i.e., 2.3% region imbalance). Given the imbalance tolerance of 1.03, this partition is measured as a balanced partition. Figure 6.7 depicts the number of regions on each part of this 2,048 part mesh. X-axis represents the part number, and y-axis represents the number of regions on a part. There are some parts contains a small number of regions in the partition. Since the heavily loaded parts in this partition dictate the scaling performance while a small number of lightly loaded parts has a very small influence, it is only the most heavily load part that is used in defining the region imbalance [8].

### 6.4.1 AAA model with $O(10^8)$ elements on 512 processors

A well-balanced 133,973,440 (133M where M denotes millions) mesh of the AAA model, distributed on 512 parts with a 1.2% region imbalance is considered. The global and local partitioning with different region imbalance tolerance numbers are applied to produce various partitions containing 1,024 parts up to 16,384 parts respectively. The partitioning is carried out on 512 processors at NERSC Hopper (Cray XE6) system [90], using 24 cores per node with 1.29GB of memory per core. After repartitioning, each processor will contain multiple parts. For instance, each processor has 32 parts in the 16,384 part partition.

This example consists of three test cases using three methods, respectively. In §6.4.1.1, the global partitioning is applied, and the time usage and region imbalance for each obtained partition are presented. In §6.4.1.2, the local partitioning is applied, and the time usage and region imbalance for each obtained partition are presented. In §6.4.1.3, a combined approach is applied, including the local partitioning with ParMetis, followed by running ParMA [54], an iterative improvement procedure for balancing multiple entity types. The time usage and region imbalance for each obtained partition are presented. §6.4.1.4 analyzes the result of these three methods, and demonstrates that the local partitioning related methods are much faster (at least 1.5 times faster) than the global partitioning, to obtain partitions of region imbalance with the requested 3% imbalance.

#### 6.4.1.1 Graph-based global partitioning

The global partitioning strategy is applied with the graph-based partitioner, ParMetis [60]. The imbalance tolerance is set at 3% for each mesh partition. Table 6.1 lists the average execution time of mesh partition stage (*Partition* in the 3rd column), mesh migration stage (*Migration* in the 4th column) and total execution (*Total* in the 5th column) time) in seconds, along with the total number of regions to migrate (*RgnsToMove*) during the migration stage. As the total number of parts increases, it requires more time to perform mesh partition, migration and thus the total execution. As shown in the 2nd column of the table, during the migration stage, nearly all the partition objects (regions) are migrated globally. Table 6.2

**Table 6.1:** The average execution time (in seconds) of mesh partition, migration and total execution on the 133M mesh of the AAA model on 512 processors on NERSC Hopper through the global partitioning strategy with ParMetis.

Total#parts	#RgnsToMove	Partition(sec)	Migration(sec)	Total(sec)
1,024	133,948,240	37.86	36.87	74.73
2,048	133,843,442	39.05	40.25	79.30
4,096	133,973,432	40.97	40.52	81.49
8,192	133,961,971	42.66	42.11	84.77
16,384	133,973,355	45.42	45.70	91.12

**Table 6.2:** The average number of regions on each part of various partitions (with region imbalance in percentage) on the 133M mesh of the AAA model on 512 processors on NERSC Hopper through the global partitioning strategy with ParMetis.

Total#parts	Local#parts	Ave.#rgns	Rgn. imb.
1,024	2	130,833	2.3
2,048	4	65,417	2.1
4,096	8	32,708	2.1
8,192	16	16,354	1.9
16,384	32	8,177	2.2

lists the average number of regions on each part (the 3rd column) in each partition, along with the region imbalance in percentage (the 4th column). Due to the global partitioning, the region imbalance of each resulting partition is at most 2.3% and satisfies the imbalance tolerance globally.

#### 6.4.1.2 Graph-based local partitioning

The local partitioning is applied with the graph-based partitioner, ParMetis [60]. The imbalance tolerance is set at 1% for each mesh partition. The tolerance set in this case is slightly smaller than the 3% set for the global partitioning. The reason is that the region imbalance ratio in the local partitioning could easily go up due to the compounding effect, since the local partitioning considers only the average number

**Table 6.3: The average execution time (in seconds) of mesh partition, migration and total execution on the 133M mesh of the AAA model on 512 processors on NERSC Hopper through the local partitioning strategy with ParMetis.**

Total#parts	#RgnsToMove	Partition(sec)	Migration(sec)	Total(sec)
1,024	66,982,773	7.26	19.22	26.48
2,048	100,480,328	7.31	28.79	36.10
4,096	117,219,929	7.34	34.55	41.89
8,192	125,600,240	7.38	39.00	46.38
16,384	129,786,485	7.46	44.07	51.53

of regions across the parts on a process instead of the average number of regions across all the parts in the whole partition. For example, given the 512 part mesh of 1.2% initial region imbalance and the region imbalance tolerance of 3%, the result region imbalance ratio obtained from the local partitioning could be  $1.2\% + 3.0\% = 4.2\%$  region imbalance. If the imbalance tolerance is set at 1%, the expected region imbalance is  $1.2\% + 1.0\% = 2.2\%$ .

The execution time (in seconds) of the mesh partition stage (*Partition* in the 3rd column), mesh migration stage (*Migration* in the 4th column) and total execution (*Total* in the 5th column) time) to obtain the various partitions is collected in Table 6.3, as well as the total number of regions to migrate (*RgnsToMove*) during the migration stage. As the total number of parts increases, the total number of regions to migrate increases from almost half to all the regions in the whole mesh, and the time to do migration increases, as well as the total execution time. The time to do partition does not increase much, since the local partitioning considers only local information on each process in each partition. Table 6.4 lists the average number of regions on each part (the 3rd column) in each partition, along with the region imbalance in percentage (the 4th column). The region imbalance of each resulting partition is less than or equal to 2.3%, almost as expected.

**Table 6.4:** The average number of regions on each part of various partitions (with region imbalance in percentage) on the 133M mesh of the AAA model on 512 processors on NERSC Hopper through the local partitioning strategy with ParMetis.

Total#parts	Local#parts	Ave.#Rgns	Rgn. imb.
1,024	2	130,833	1.3
2,048	4	65,417	2.0
4,096	8	32,708	2.3
8,192	16	16,354	2.2
16,384	32	8,177	2.3

#### 6.4.1.3 Graph-based local partitioning followed by ParMA

The local partitioning is applied first on the 133M mesh, distributed on 512 parts, to obtain various partitions containing 1,024 up to 16,384 parts. The region imbalance tolerance for each partition is set at 3%. Then the ParMA tool is applied on each partition to improve the region balance. ParMA is an under-development tool for balancing multiple mesh entity types by migrating selected mesh entities from heavily loaded part to lightly loaded part, globally and iteratively, also aiming to reduce the inter-part communication costs [54]. In each iteration step of ParMA, a greedy algorithm is applied to select mesh entities to migrate in the part neighborhood, and the mesh entity migration algorithm developed in this chapter is used as the underlying mesh migration procedure. In the ParMA of this case, the region imbalance tolerance is set at 2% for each partition and the number of iteration steps is set at 50.

Table 6.5 lists the average execution time (in seconds) of mesh partition stage (*Partition* in the 3rd column), mesh migration stage (*Migration* in the 4th column), ParMA stage (in the 5th column), and total execution (*Total* in the 6th column), along with the total number of regions to migrate (*RgnsToMove*) during the migration stage. As the total number of parts increases, the time for migration and ParMA increases, as well as the total execution time. Due to the local partitioning, the time for partitioning does not increase much in each partition.

Table 6.6 lists the average number of regions per part (the 3rd column) in



**Table 6.5:** The average execution time (in seconds) of mesh partition, migration, ParMA and total execution on the 133M mesh of the AAA model on 512 processors on NERSC Hopper through the local partitioning strategy (with ParMetis) followed by ParMA.

Total #parts	#RgnsToMove	Partition (sec)	Migration (sec)	ParMA (sec)	Total (sec)
1,024	66,975,240	7.24	19.13	0.90	27.27
2,048	100,490,369	7.36	28.80	0.41	36.57
4,096	117,222,748	7.40	34.35	1.01	42.76
8,192	125,600,685	7.41	38.39	4.50	50.30
16,384	129,786,233	7.48	43.34	8.13	58.95

**Table 6.6:** The average number of regions on each part of various partitions with region imbalance (in percentage) on the 133M mesh of the AAA model on 512 processors on NERSC Hopper through the local partitioning strategy (with ParMetis) followed by ParMA.

Total#parts	Local#parts	Ave.#Rgns	Rgn. imb. (Local)	Rgn. imb. (ParMA)
1,024	2	130,833	4.3	2.9
2,048	4	65,417	2.9	2.0
4,096	8	32,708	3.6	2.8
8,192	16	16,354	4.0	3.0
16,384	32	8,177	4.3	3.0

each partition, along with the region imbalance (in percentage) after running the local partitioning and ParMA (in the 4th and 5 columns), respectively. As shown in Table 6.6, with the region imbalance tolerance set at 3%, the region imbalance in most partitions obtained from the local partitioning is above 3%. By running ParMA, the region imbalance in each partition is down to 3%.

#### 6.4.1.4 Compare the three methods

This subsection compares the time usage and region imbalance for each partition through three methods: the global partitioning with the region imbalance

**Table 6.7:** Compare the average total execution time (in seconds) of mesh repartition on the 133M mesh of the AAA model on 512 processors on NERSC Hopper through three methods: Global partitioning, Local partitioning and Combined method (local partitioning followed by ParMA).

Total #parts	Global <i>tol=3%</i>	Local <i>tol=1%</i>	Global/ Local	Combined <i>tol=3%,tol=2%</i>	Global/ Combined
1,024	74.73	26.48	2.8	27.27	2.7
2,048	79.30	36.10	2.2	36.57	2.2
4,096	81.49	41.89	1.9	42.76	1.9
8,192	84.77	46.38	1.8	50.30	1.7
16,384	91.12	51.53	1.8	58.95	1.5

tolerance of 3% (*Global*), the local partitioning with the region imbalance tolerance of 1% (*Local*), and the combined method of the local partitioning with region imbalance tolerance of 3% followed by *ParMA* with the region imbalance tolerance of 2% (*Combined*).

Table 6.7 compares the average total execution time (in seconds) for each partition through the three methods: *Global* (in the 2nd column), *Local* (in the 3rd column), and *Combined* (in the 5th column). The 4th column lists the time ratio of the global partitioning over the local partitioning in each partition, and the 6th column lists the time ratio of the global partitioning over the combined method in each partition. As shown in the 4th and 6th columns, to produce various partitions, the local partitioning is at least 1.8 times faster than the global partitioning, and the combined method is at least 1.5 times faster than the global partitioning.

Table 6.8 compares the region imbalance (in percentage) of various partitions through the three methods: *Global* (in the 2nd column), *Local* (in the 3rd column), and *Combined* (in the 4th column). The region imbalance in each partition is less than or equal to 3%, but the region imbalance obtained from the combined method is a currently slightly higher than that from the other two methods. Note *ParMA* is under development, thus the results are likely to change as the development continues.

**Table 6.8: Compare the region imbalance (in percentage) of various partitions on the 133M mesh of the AAA model on 512 processors on NERSC Hopper through three methods: Global partitioning, Local partitioning, and Combined method (local partitioning followed by ParMA).**

Total#parts	Global	Local	Combined
1,024	2.3	1.3	2.9
2,048	2.1	2.0	2.0
4,096	2.1	2.3	2.8
8,192	1.9	2.2	3.0
16,384	2.2	2.3	3.0

#### 6.4.2 AAA model with $O(10^8)$ elements on 2,048 processors

Two well-balanced 133,973,440 (133M where M denotes millions) meshes of the AAA model distributed on 2,048 parts are considered: one is of 2.3% region imbalance and the other is of 1.0% region imbalance. The global and local partitioning are applied on the two meshes to create various partitions containing 4,096 parts up to 131,072 parts respectively. The partitioning is carried out on 512 processors at NERSC Hopper (Cray XE6) system [90], using 24 cores per node with 1.29GB of memory per core. After repartitioning, each processor will contain multiple parts. For instance, each processor has 8 parts in the 16,384 part partition.

This example consists three test cases using three methods: (i) the global graph-based partitioning on the 2,048 part mesh of 2.3% region imbalance (in §6.4.2.1); (ii) the local graph-based partitioning on the 2,048 part mesh of 1.0% region imbalance (in §6.4.2.2); (iii) the local hypergraph-based partitioning on the 2,048 part mesh of 1.0% region imbalance (in §6.4.2.3). §6.4.2.4 compares the time usage and region imbalance of the three methods, and demonstrates that both the graph-based and hypergraph-based local partitioning are much faster (at least 2 times) than the global partitioning, to obtain partitions of region imbalance with the requested 3% imbalance.

The initial 2,048-part meshes are obtained through the graph-based global partitioning on the previous 133M mesh, distributed on 512 parts on 512 processing

**Table 6.9:** The average execution time (in seconds) of mesh partition, migration and total execution and the region imbalance (in percentage) of the 2,048 part partition on the 133M mesh of the AAA model on 512 processors on NERSC Hopper through the global partitioning strategy with ParMetis.

Total#parts	RgnImbTol (%)	ResultRgnImb (%)	Partition (sec)	Migration (sec)	Total (sec)
2,048	1.0	1.0	39.58	40.00	79.58

cores on NERSC Hopper, with different region imbalance tolerance set for each partition. With the imbalance tolerance set at 3%, the partitioning obtains a 2,048 part mesh of 2.3% region imbalance, in about 79.30 seconds (in the 5th column of Table 6.1). With the imbalance tolerance set at 1%, the partitioning obtains a 2,048 part mesh of 1.0% region imbalance, in about 79.58 seconds (in the 6th column of Table 6.9). The total partitioning time spent in the two partitions are very close, and the partitioning of 1% region imbalance tolerance is slightly slower (about 0.4%) than that of 3% region imbalance tolerance.

#### 6.4.2.1 Graph-based global partitioning

The global partitioning with the graph-based partitioner, ParMetis [60] is applied on the 2,048 part mesh of 2.3% region imbalance. The imbalance tolerance is set at 3% for each partition. The repartitions to 65,536 and 131,072 parts failed and reported the error of running out of memory from the Hopper system. Table 6.10 lists the average execution time (in seconds) of the mesh partition stage (*Partition* in the 3rd column), mesh migration stage (*Migration* in the 4th column) and total execution (*Total* in the 5th column) time, along with the total number of regions to migrate (*RgnsToMove*) during the migration stage. As shown in this table, as the total number of parts increases, the total execution time increases as well as the time for mesh partition. The time for mesh migration does not increase much, around 10 seconds, since nearly all the partition objects (regions) are migrated during this stage. Table 6.11 lists the average number of regions on each part (the 3rd column) in each partition, along with the region imbalance (the 4th column). Due to the

**Table 6.10:** The average execution time (in seconds) of mesh partition, migration and total execution on the 133M mesh of the AAA model on 2,048 processors on NERSC Hopper through the global partitioning strategy with ParMetis.

Total#parts	#RgnsToMove	Partition	Migration	Total Exec.
4,096	133,950,085	14.33	9.69	24.02
8,192	133,952,150	16.06	9.87	25.93
16,384	133,973,348	18.29	10.00	28.29
32,768	133,971,242	24.27	10.85	35.12

**Table 6.11:** The average number of regions on each part of various partitions (with region imbalance in percentage) on the 133M mesh of the AAA model on 2,048 processors on NERSC Hopper through the global partitioning strategy with ParMetis.

Total#parts	Local#parts	Ave.#Rgns	Rgn. imb.
4,096	2	32,708	2.2
8,192	4	16,354	2.4
16,384	8	8,177	2.2
32,768	16	4,088	2.2

global partitioning, the region imbalance (partition objects) of each partition is at most 2.4% and satisfies the imbalance tolerance globally.

#### 6.4.2.2 Graph-based local partitioning

The local partitioning with the graph-based partitioner, ParMetis [60] is applied on the 2,048 part mesh of 1.0% region imbalance. The imbalance tolerance is set at 1% or 2% for each mesh partition. The partition result with better region imbalance is listed in the following tables. Note this case considers the 2,048 part mesh of 1.0% region imbalance, instead of the one of 2.3% region imbalance, since the goal is to obtain various partitions with the region imbalance within 3%. Otherwise, given the 2,048 part mesh of 2.3% region imbalance in the local partitioning and 1% region imbalance tolerance, the region imbalance in a resulting partition could easily go up to  $2.3\% + 1.0\% = 3.3\%$ .

**Table 6.12:** The average execution time (in seconds) of mesh partition, migration and total execution on the 133M mesh of the AAA model on 2,048 processors on NERSC Hopper through the local partitioning strategy with ParMetis.

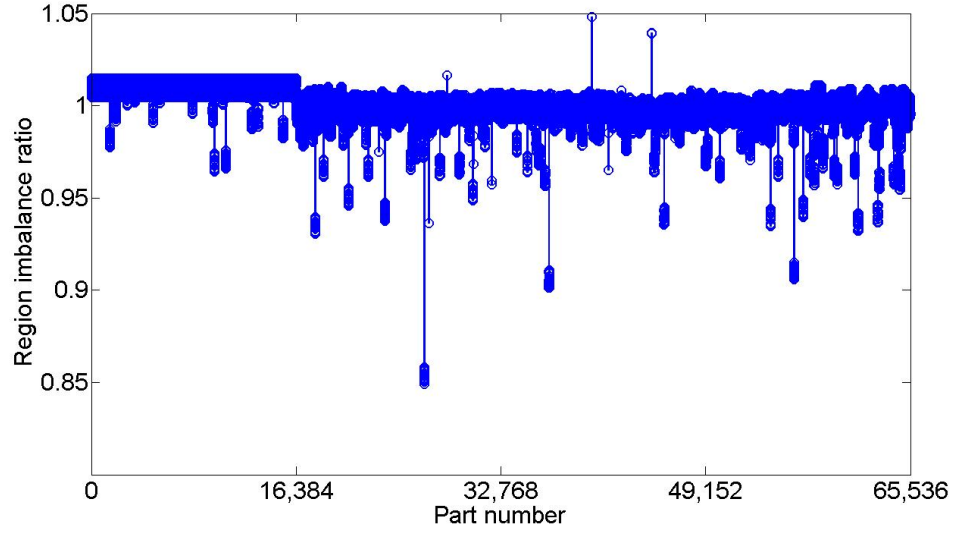
Total#parts	#RgnsToMove	Partition(sec)	Migration(sec)	Total(sec)
4,096	66,987,061	3.53	4.61	8.14
8,192	100,483,219	3.49	6.92	10.41
16,384	117,224,100	3.51	8.24	11.75
32,768	125,600,912	3.55	9.34	12.89
65,536	129,786,818	3.57	10.72	14.29
131,072	131,879,721	3.59	13.99	17.58

**Table 6.13:** The average number of regions on each part of various partitions (with region imbalance in percentage) on the 133M mesh of the AAA model on 2,048 processors on NERSC Hopper through the local partitioning strategy with ParMetis.

Total#parts	Local#parts	Ave.#Rgns	Rgn. imb.
4,096	2	32,708	2.0
8,192	4	16,354	2.0
16,384	8	8,177	2.0
32,768	16	4,088	3.9
65,536	32	2,044	4.8
131,072	64	1,022	4.9

The average execution time (in seconds) of mesh partition (*Partition* in the 3rd column), mesh migration (*Migration* in the 4th column) and total execution (*Total* in the 5th column) is collected in Table 6.12, along with the total number of regions to migrate (*RgnsToMove*) during the migration stage. As the total number of parts increases, the total execution time as well as the time for mesh migration increases, since the total number of regions to migrate increases. The time for mesh partition does not increase much, since the local partitioning only considers the local information on each process.

Table 6.13 lists the average number of regions on each part (the 3rd column) in each partition, along with the region imbalance in percentage (the 4th column). As



**Figure 6.8: Region imbalance ratio on each part of a 133M region mesh for a partition with 65,536 parts.**

the total number of parts increases, the region imbalance in the 4,096 up to 16,384 part partitions is exactly 2%, while the region imbalance in the 32,768 up to 131,072 part partitions increases to almost 5%. Figure 6.8 depicts the region imbalance ratio on each part in the 65,536 part partition. In this figure, the region imbalance of most parts is below 3%, i.e. imbalance ratio  $\leq 1.03$ . The region imbalance of only two parts is close to 5% (the two spikes). The local partitioning has been applied with various region imbalance tolerance, such as 1%, 2% or 0.5%, set in the graph partitioner ParMetis, but the spikes in the 32,768 up to 131,072 part partitions still exist.

#### 6.4.2.3 Hypergraph-based local partitioning

The local partitioning with the hypergraph-based partitioner, PHG [92], is applied on the 2,048 part mesh of 1.0% region imbalance to obtain various partitions, containing 4,096 up to 131,072 parts. The imbalance tolerance is set at 1% for each mesh partition.

The average execution time (in seconds) of mesh partition (*Partition* in the 3rd column), mesh migration (*Migration* in the 4th column) and total execution (*Total*

**Table 6.14: The average execution time (in seconds) of mesh partition, migration and total execution on the 133M mesh of the AAA model on 2,048 processors on NERSC Hopper through the local partitioning strategy with PHG.**

Total#parts	#RgnsToMove	Partition(sec)	Migration(sec)	Total(sec)
4,096	66,534,505	2.33	4.66	6.99
8,192	100,190,323	3.13	7.05	10.18
16,384	117,067,747	3.89	8.40	12.29
32,768	125,518,648	4.73	9.76	14.49
65,536	129,745,852	5.55	11.53	17.08
131,072	131,859,752	6.70	15.52	22.22

in the 5th column) to obtain various partitions is collected in Table 6.14, along with the total number of regions to migrate (*RgnsToMove*) during the migration stage. As the total number of parts increases, the total execution time increases. During the migration stage, since the total number of regions to migrate increases from almost half to all the regions in the partition, the time for migration increases. Although the local partitioning only considers the local information on each process, the time for mesh partition increases as the total number of parts increases. This is not the case in the graph-based local partitioning, where the time for mesh partition is around 3.54 seconds. Part of the reason is that the hypergraph partitioning usually produces partitions of higher quality than the graph partitioning [92]. Table 6.15 lists the average number of regions on each part (the 3rd column) in each partition, along with the region imbalance in percentage (the 4th column). The region imbalance in each partition is fixed at 2%.

#### 6.4.2.4 Compare the three methods

This subsection compares the time usage and region imbalance for each partition through three methods: the graph-based global partitioning (*Global ParMetis*), the graph-based local partitioning with the ParMetis (*Local ParMetis*), and the hypergraph-based local partitioning with the PHG (*Local PHG*).

Table 6.16 compares the average total execution time (in seconds) on the 133M



**Table 6.15:** The average number of regions on each part of various partitions (with region imbalance in percentage) on the 133M mesh of the AAA model on 2,048 processors on NERSC Hopper through the local partitioning strategy with PHG.

Total#parts	Local#parts	Ave.#Rgns	Rgn. imb.
4,096	2	32,708	2.0
8,192	4	16,354	2.0
16,384	8	8,177	2.0
32,768	16	4,088	2.0
65,536	32	2,044	2.0
131,072	64	1,022	2.0

**Table 6.16:** Compare the average total execution time (in seconds) of mesh repartition on the 133M mesh of the AAA model on 2,048 processors on NERSC Hopper through three methods: Global partitioning with ParMetis, Local partitioning with ParMetis and Local partitioning with PHG.

Total #parts	Global <i>ParMetis</i>	Local <i>ParMetis</i>	Global/ Local(Par)	Local <i>PHG</i>	Global/ Local(PHG)	Local(PHG)/ Local(Par)
4,096	24.02	8.14	3.0	6.99	3.4	0.9
8,192	25.93	10.41	2.5	10.18	2.5	1.0
16,384	28.29	11.75	2.4	12.29	2.3	1.0
32,768	35.12	12.89	2.7	14.49	2.4	1.1
65,536	N/A	14.29	N/A	17.08	N/A	1.2
131,072	N/A	17.58	N/A	22.22	N/A	1.3

mesh partitioning on 2,048 processors of the three methods: *Global ParMetis* (in the 2nd column), *Local ParMetis* (in the 3rd column), and *Local PHG* (in the 5th column). The 4th column (*Global/Local(Par)*) lists the time ratio of the global partitioning over the graph-based local partitioning. The 6th column (*Global/Local(PHG)*) lists the time ratio of the global partitioning over the hypergraph-based local partitioning. The 7th column (*Local(PHG)/Local(Par)*) lists the time ratio of the hypergraph-based local partitioning over the graph-based local partitioning. It shows that in the 2,048 part mesh case:

- To obtain various partitions of the region imbalance within 3%, the local partitioning (either hypergraph or graph based) is at least 2.3 times faster than the global partitioning.
- As the total number of parts increases, the global partitioning could fail due to the insufficient computer resources. For example, for the partition using 2,048 processors on Hopper system with 1.29GB of memory per core, the repartition from 2,048 parts to 65,536 and 131,072 parts fails in the global partitioning due to the insufficient per core memory, and succeeds in the local partitioning.
- As the total number of parts increases, the time for the hypergraph-based local partitioning increases much faster than the graph-based local partitioning. For example, in the 4,096 and 8,192 part partitions, the hypergraph-based local partitioning takes less time than the graph-based partitioning, while in the 16,384 up to 131,072 part partitions, the hypergraph-based local partitioning takes more time than the graph-based partitioning.

Combined with the cost to generate the 2,048 part partition from 512 part mesh through the global partitioning with different region imbalance tolerance, the local partitioning is still faster than the global partitioning. For example, to create a 32,768 part partition, the local graph-based partitioning is 1.2 times faster than the global graph-based partitioning. The time usage is calculated as follows:

- The global partitioning takes  $79.30 + 35.12 = 114.42$  seconds in total: 79.30 seconds for the global partitioning from 512 to 2,048 parts, and 35.12 seconds for the global partitioning from 2,048 to 32,768 parts.
- The local partitioning takes  $79.58 + 12.89 = 92.47$  seconds in total: 79.58 seconds for the global partitioning from 512 to 2,048 parts, and 12.89 seconds for the local partitioning from 2,048 to 32,768 parts.

Table 6.17 compares the region imbalance (in percentage) of various partitions on the 133M mesh of the AAA model on 2,048 processors on NERSC Hopper through the three methods: *Global ParMetis* (in the 2nd column), *Local ParMetis* (in the 3rd column), and *Local PHG* (in the 4th column). The region imbalance

**Table 6.17: Compare the region imbalance (in percentage) of various partitions on the 133M mesh of the AAA model on 2,048 processors on NERSC Hopper through three methods: Global partitioning with ParMetis, Local partitioning with ParMetis, and Local partitioning with PHG.**

Total#parts	Global(ParMetis)	Local(ParMetis)	Local(PHG)
4,096	2.2	2.0	2.0
8,192	2.4	2.0	2.0
16,384	2.2	2.0	2.0
32,768	2.2	3.9	2.0
65,536	N/A	4.8	2.0
131,072	N/A	4.9	2.0

obtained from the local hypergraph-based partitioning is fixed at 2.0%, and is less than or equal to any other obtained from either the global or local graph-based partitioning. The local graph-based partitioning could create partitions of region imbalance above 3%, as the total number of parts increases.

### 6.4.3 Mesh partitioning with predictive load balancing

The goal of mesh adaptation is to modify the mesh so that the element sizes and distribution provide the desired mesh resolution for nearly an optimum number of elements. To achieve this goal, mesh refinement and coarsening is carried out over various portions of the mesh domain. In this process some parts that have most of the mesh refinement can often exceed the limit of the physical memory of the processors, this may slow down or even kill the process of mesh repartitioning after mesh adaptation. To avoid running out of memory in the whole adaptation process, predictive load balancing [1] can be used to improve the balance of mesh regions in the adapted mesh. However, as long as mesh adaptation is performed on a fixed number of processors, the mesh size could increase and eventually exceed the limit of the physical memory of processors. The capability to have multiple parts per process helps to solve this problem. Combining predictive load balancing with multiple parts per process partitioning can support mesh adaptation more effectively and efficiently in terms of memory usage, this is especially useful on supercomputers

with relatively smaller per core memory, such as BG/L and BG/P system with only 512 MB or 1GB per core memory.

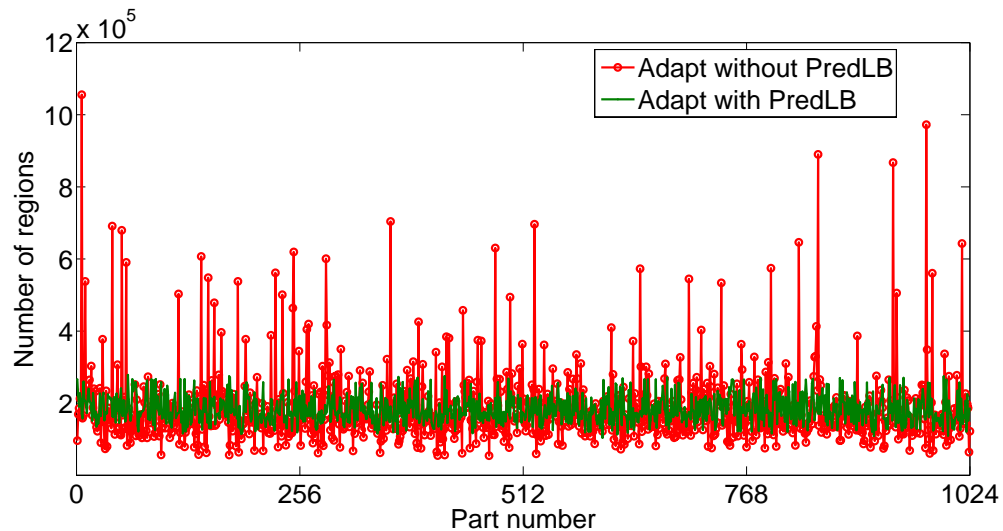


Figure 6.9: The number of regions on each part of the mesh of 187 million regions on 1,024 parts after mesh adaptation with and without *PredLB* on a straight pipe model with air bubbles.

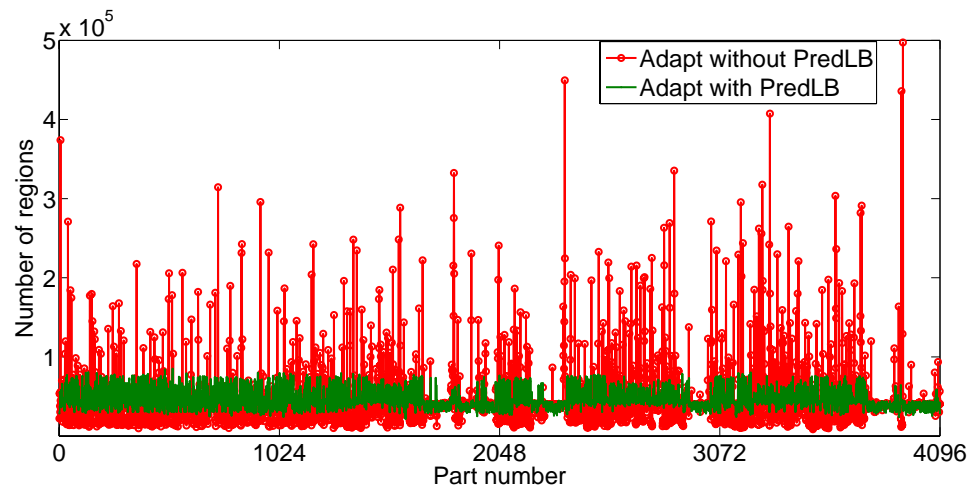


Figure 6.10: The number of regions on each part of the mesh of 187 million regions on 4,096 parts after mesh adaptation with and without *PredLB* on a straight pipe model with air bubbles.

Predictive load balancing is a mesh repartitioning step applied before mesh adaptation. In the predictive load balancing step, mesh repartition is carried out

based on the partition object (element) weight, which is specified to each of the current elements based on a prediction of how many elements will fill the space of that element, and thus the mesh will be nearly balanced after mesh adaptation [1].

An initial mesh on a straight pipe model with air bubbles distributing in the pipe is considered. Figure 3.3a shows the mesh size field which represents the motion of air bubbles in the geometric model. A zoomed bubble in the mesh is colored by the magnitude of size field in Figure 3.3b.

In the first case, the mesh adaptation procedure developed in SCOREC [3, 20] is carried out on the initial mesh on 1,024 parts of 164,728,413 regions, with and without predictive load balancing respectively, on 1,024 processors on NERSC Hopper, using 1.29GB per core memory. The adapted meshes contain about 187 million regions. Figure 6.9 shows the number of regions after adaptation with (green line) and without (red line) using predictive load balancing (*PredLB*) on 1,024 parts. Without *PredLB*, the number of regions on some parts are much higher than others after mesh adaptation. For example, the part of the highest spike (red line) contains 1,055,737 regions after mesh adaptation. The balance of mesh regions in the adapted mesh is improved by using *PredLB* (green line), where the maximum number of regions on a part in the adapted mesh is 278,292.

In the second case, the mesh on 1,024 parts is first repartitioned into 4,096 parts through the global partitioning. Then the mesh adaptation procedure is carried out on the resulting 4,096 part mesh of 164,728,413 regions, with and without predictive load balancing respectively, on 4,096 processors on NERSC Hopper, using 1.29GB per core memory. Figure 6.10 shows the number of regions after adaptation with (green line) and without (red line) using predictive load balancing (*PredLB*) on 4,096 parts. Without *PredLB*, the number of regions on some parts are much higher than others after mesh adaptation. For example, the part of the highest spike (red line) contains 497,257 regions after adaptation, nearly half of that (1,055,737) in the previous case without *PredLB*. By using *PredLB* the balance of mesh regions in the adapted mesh (green line) is improved, where the maximum number of regions on a part is 84,967, which is less than 1/3 of that (278,292) in the previous case with using *PredLB* and also less than 1/10 of that (1,055,737) in the previous

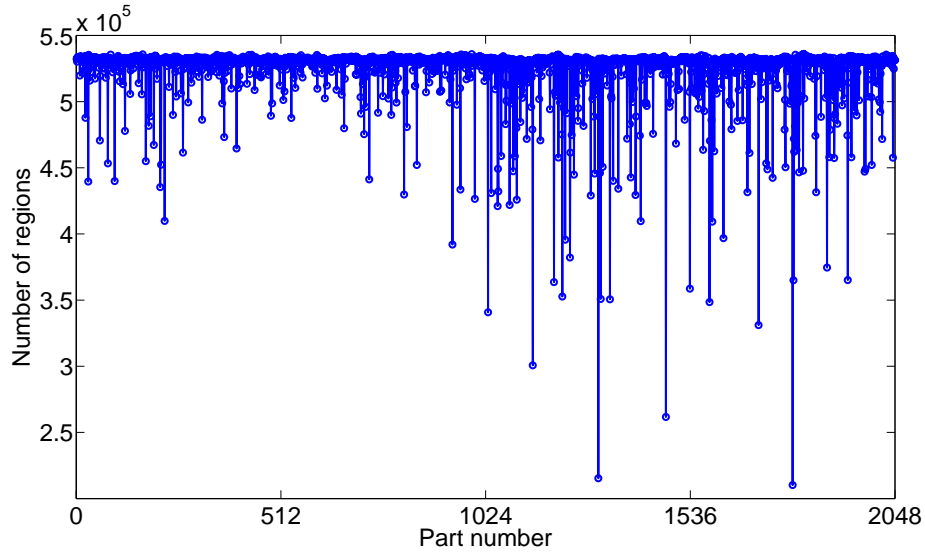
case without using *PredLB*. In other word, by using predictive load balancing and multiple part per process partitioning together, the required memory per part for mesh adaptation can be reduced by more than 90%.

#### 6.4.4 AAA model with billions of elements for large-scale adaptive simulations

The mesh adaptation procedure developed in SCOREC [3, 20] is applied on the well balanced 133M mesh, distributed on 512 parts with a 1.2% region imbalance of the AAA model, to generate meshes of billions of elements through two methods: (i) the original FMDB allowing only one part per process, and (ii) the improved FMDB allowing multiple parts per process. All the tests are carried out on NERSC Hopper. Based on the partitioned meshes of billions of elements, an actual use case of a large-scale adaptive simulation is presented.

In the first case, one uniform refinement is applied on the 133M mesh on 512 processing cores on NERSC Hopper, using 1.29GB memory per core. The test fails and reports the error of running out of memory. This failure is proved by the second step, where the same adaptation process is performed on 512 processing cores using increased 2.58GB memory per core and obtains an adapted mesh of 1,071,787,520 (1B) regions distributed on 512 parts (1.3% region imbalance with 2,093,335 regions per part in average). In the next step, another uniform refinement is further applied on the resulting well-balanced 1B mesh on 512 processing cores, using 2.58GB memory per core, and fails again because of running out of memory.

In the second case, instead of adapting the 133M mesh on 512 processing cores directly, a global partitioning is first carried out on the 133M mesh to obtain a 2,048 part partition (with 4 parts per process) on 512 processing cores using 1.29GB memory per core on NERSC Hopper. Then one uniform refinement is performed on the well-balanced 133M mesh on 2,048 parts (2.3% region imbalance with 65,417 regions per part in average) on 2,048 processing cores using 1.29GB memory per core, and obtains an adapted mesh of 1,071,787,520 (1B) regions (2.4% region imbalance with 523,334 regions per part in average). Figure 6.11 depicts the number of regions on each part of this 1B mesh distributed on 2,048 parts. In the



**Figure 6.11: Number of regions on each part of a 1B region mesh for a partition with 2,048 parts.**

next step, the resulting 1B mesh is further repartitioned from 2,048 to 16,384 parts through the global partitioning on 2,048 processing cores using 1.29GB memory per core. This 16,384-part partitioned mesh could be further adapted to another mesh of 8.57 billion regions (with about 523,334 regions per part in average) on 16,384 processing cores, still using 1.29GB memory per core.

In the first case, since the simulation is constrained to a fixed number of processors (e.g. 512 processing cores) by the original FMDB implementation, the per core memory space becomes the bottleneck of the whole simulation as the mesh size increases. This resulting 1B mesh could not be even fit on any supercomputer with smaller per core memory, such as IBM BG/L and BG/P with only 512MB or 1GB of memory per core. In the second case, the memory is no longer a bottleneck of the whole simulation as the mesh size increases, by using multiple parts per process partitioning.

In the third case, a large partitioned mesh with 8.57 billion (8.57B) regions on the same AAA model with 16,384 parts is first generated from an anisotropic mesh of 133 million regions using the previous steps in the second case. Based on the globally balanced partitioned 16,384 part mesh, various partitions are then created

**Table 6.18: The average number of regions per part of the 8.57B-region mesh on the AAA model, along with the region imbalance (in percentage) [93].**

Total#parts	Local#parts	Local#rgn	Rgn. imb.
32,768	2	261,670	5.1
65,536	4	130,830	5.1
98,304	6	87,222	5.1
131,072	8	65,417	5.1
294,912	18	29,074	5.1

by further splitting each part into multiple parts locally through the hypergraph-based partitioner, PHG [92]. The region imbalance tolerance is set at 3% for each partition. Table 6.18 presents the average number of regions on a part for each partition along with the region imbalance. Due to the local partitioning, the region imbalance in each resulting partition is 5.1%.

Using these partitioned meshes of 8.57B regions on the AAA model listed in Table 6.18, scaling studies of an adaptive dynamic fluid simulation are performed on Intrepid (IBM BG/P) and Kraken (Cray XT5), and a 3D unstructured mesh fluid flow analysis tool, PHASTA, has demonstrated an excellent strong scaling on up to 131,072 processing cores on Intrepid (IBM BG/P) and super-linear speedup on up to 98,304 processing cores on Kraken (Cray XT5), respectively [93].



## CHAPTER 7

### CONCLUSION AND FUTURE WORK

This thesis has presented a set of mesh data management software components, building on the FMDB mesh database, which are used in the adaptive control of unstructured meshes.

To extend FMDB in a manner that FMDB can continue evolving to support a broad range of future application requirements, this thesis extended the generic programming methods in FMDB with the introduction of set, iterator and tagging functionalities. The performance result of the mesh adaptation on air bubble meshes on massively parallel computers demonstrates the efficiency of these reusable, generic components, which achieve code reusability without sacrificing the performance of mesh adaptation, compared to the traditional object-oriented programming.

The generic set component was then extended to support two types of mesh sets: P-sets that require all the entity members staying on a single part together, and NP-sets that contain mesh entities spanning multiple parts. Efficient algorithms for mesh partitioning and mesh migration were developed using P-sets. In mesh migration, all the members in a P-set are required to be migrated as a unit. The support for boundary layer mesh adaptations in parallel clearly demonstrates the effective usage of P-sets.

The capability of mesh matching was developed to deal with applications that require the mesh representation related to specific geometric model entities, particularly applications with periodic boundary conditions, be identical. Chapter 5 introduced the definitions and operations for mesh matching. Mesh partitioning and migration algorithms were developed to maintain matched entity information for matched entities. The support for mesh adaptation demonstrates the capability to deal with matched meshes.

Multiple parts per process partitioning was developed to support changing the number of processors being used during a parallel computation. The partition model and entity migration algorithm were extended to support multiple parts per

process partitioning. Chapter 6 compared the performance results of mesh partitioning through the global and local partitioning strategies, and has demonstrated the capability to support adaptive simulations on meshes of millions or billions of elements on hundreds of thousands of processors.

In the future, the iterator, set and tagging functionalities will support geometric model and field libraries through their well-defined API's. At the same time, for better supporting unstructured mesh applications, more generic components will be designed and developed, such as the *(i)* the *relation* component for relating arbitrary data in different data models when no direct interactions through API's are available, and *(ii)* the *communicator* component for supporting efficient parallel functionalities such as architecture-aware communication and data distribution on massively parallel computers.

More software engineering techniques and generic programming methods can be applied in the software component design for adaptive unstructured mesh simulations. For example, instead of using raw function pointers, the generic iterator component discussed in Chapter 3 can use *Boost.Function* library [94], thus to allow user greater flexibility in the implementation. However, running the third party software libraries on supercomputers needs to consider the portability issue.

In the current multiple parts per process partitioning, each part, either on-process or off-process, is treated as a serial mesh. Thus duplicated part boundary entities can exist on a process. An on-going research development focused on multiple-core per node computer architectures is considering the distributed mesh data structures and algorithms which will eliminate data duplication for part boundaries on a node.

In a large-scale mesh-based adaptive simulation, the volume of mesh data created by the simulation can be overwhelming, such as the mesh files written from a multiple parts per process partitioning. An on-going research development is to remove the file import/export procedures and to realize fileless connections between the steps in the adaptive cycle of a simulation.

## REFERENCES

- [1] M. Zhou, T. Xie, S. Seol, M. S. Shephard, O. Sahni, and K. E. Jansen, “Tools to support mesh adaptation on massively parallel computers,” *Eng. Comput.*, pp. 1–15, 2011.
- [2] X. Li, M. S. Shephard, and M. W. Beall, “Accounting for curved domains in mesh adaptation,” *Int. J. Numer. Meth. Eng.*, vol. 58, no. 2, pp. 247–276, 2003.
- [3] X. Li, M. S. Shephard, and M. W. Beall, “3D anisotropic mesh adaptation by mesh modification,” *Comput. Method. Appl M.*, vol. 194, pp. 4915–4950, Nov. 2005.
- [4] O. Sahni, K. E. Jansen, M. S. Shephard, C. A. Taylor, and M. W. Beall, “Adaptive boundary layer meshing for viscous flow simulations,” *Eng. Comput.*, vol. 24, no. 3, pp. 267–285, 2008.
- [5] O. Sahni, J. Müller, K. E. Jansen, M. S. Shephard, and C. Taylor, “Efficient anisotropic adaptive discretization of the cardiovascular system,” *Comput. Method. Appl M.*, vol. 195, no. 41-43, pp. 5634–5655, 2006.
- [6] M. S. Shephard, K. E. Jansen, O. Sahni, and L. A. Diachin, “Parallel adaptive simulations on unstructured meshes,” *J. Phys.: Conf. Ser.*, vol. 78, p. 012053, Jul. 2007.
- [7] O. Sahni, *Automated adaptive viscous flow simulations*. Ph.D. dissertation, Dept. Mech. Eng., Rensselaer Polytechnic Inst., Troy, NY, 2007.
- [8] M. Zhou, *Petascale adaptive computational fluid dynamics*. Ph.D. dissertation, Dept. Mech. Eng., Rensselaer Polytechnic Inst., Troy, NY, 2009.
- [9] Department of Energy’s Scientific Discovery through Advanced Computing (SciDAC), “Interoperable Technologies for Advanced Petascale Simulations (ITAPS).” [Online]. Available: <http://www.itaps.org/>, [Accessed Apr. 5, 2012], Dec. 2010.
- [10] Department of Energy’s Scientific Discovery through Advanced Computing (SciDAC), “Framework, Algorithms, and Scalable Technologies for Mathematics (FASTMath).” [Online]. Available: <http://www.fastmath-scidac.org/toolset.html/>, [Accessed Apr. 5, 2012].
- [11] K. K. Chand, L. F. Diachin, X. Li, C. Ollivier-Gooch, E. S. Seol, M. S. Shephard, T. Tautges, and H. Trease, “Toward interoperable mesh, geometry

- and field components for PDE simulation development,” *Eng. Comput.*, vol. 24, pp. 165–182, Nov. 2007.
- [12] C. Ollivier-Gooch, L. Diachin, M. S. Shephard, T. Tautges, J. Kraftcheck, V. Leung, X. Luo, and M. Miller, “An interoperable, data-structure-neutral component for mesh query and manipulation,” *ACM Trans. Math. Softw.*, vol. 37, pp. 29:1–29:28, Sep. 2010.
  - [13] K. Devine, L. Diachin, J. Kraftcheck, K. E. Jansen, V. Leung, X. Luo, M. Miller, C. Ollivier-Gooch, A. Ovcharenko, O. Sahni, M. S. Shephard, T. Tautges, T. Xie, and M. Zhou, “Interoperable mesh components for large-scale, distributed-memory simulations,” *J. Phys.: Conf. Ser.*, vol. 180, p. 012011, Jul. 2009.
  - [14] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, “Zoltan data management services for parallel dynamic applications,” *Comput. Sci. Eng.*, vol. 4, no. 2, pp. 90–97, 2002.
  - [15] E. S. Seol, *FMDB: Flexible distributed Mesh DataBase for parallel automated adaptive analysis*. Ph.D. dissertation, Dept. Comput. Sci., Rensselaer Polytechnic Inst., Troy, NY, 2005.
  - [16] E. S. Seol and M. S. Shephard, “Efficient distributed mesh data structure for parallel automated adaptive analysis,” *Eng. Comput.*, vol. 22, pp. 197–213, Nov. 2006.
  - [17] M. H. Austern, *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Reading, Massachusetts: Addison-Wesley, 1 ed., Oct. 1999.
  - [18] D. R. Musser and A. A. Stepanov, “Generic programming,” in *ISSAC’88 Symbolic Algebraic Comput.: Int. Symp.* (P. Gianni, ed.), vol. 358, (Rome, Italy), pp. 13–25, Springer-Verlag, 1988.
  - [19] D. R. Musser and A. A. Stepanov, “Algorithm-oriented generic libraries,” *Softw. Pract. Exper.*, vol. 24, no. 7, pp. 623–642, 1994.
  - [20] F. Alauzet, X. Li, E. S. Seol, and M. S. Shephard, “Parallel anisotropic 3D mesh adaptation by mesh modification,” *Eng. Comput.*, vol. 21, pp. 247–258, Jan. 2006.
  - [21] H. de Cougny and M. Shephard, “Parallel refinement and coarsening of tetrahedral meshes,” *Int. J. Numer. Meth. Eng.*, vol. 46, no. 7, pp. 1101–1125, 1999.
  - [22] M. S. Shephard, M. W. Beall, R. M. O’Bara, and B. E. Webster, “Toward simulation-based design,” *Finite Elem. Anal. Des.*, vol. 40, pp. 1575–1598, Jul. 2004.

- [23] M. W. Beall and M. S. Shephard, "An object-oriented framework for reliable numerical simulations," *Eng. Comput.*, vol. 15, pp. 61–72, 1999.
- [24] M. Mortenson, *Geometric Modeling*. Wiley, 2nd ed., Jan. 1997.
- [25] M. W. Beall, J. Walsh, and M. S. Shephard, "Accessing CAD geometry for mesh generation," in *12th Int. Meshing Roundtable*, pp. 33–42, 2003. Sandia National Laboratories, SAND-2003-3030P.
- [26] T. J. Tautges, "CGM: A Geometry Interface for Mesh Generation, Analysis and Other Applications," *Eng. Comput.*, vol. 17, pp. 299–314, 2001.
- [27] K. J. Weiler, "The radial-edge structure: a topological representation for non-manifold geometric boundary representations," in *Geometric Modeling for CAD applications* (M. J. Wozney, H. W. McLaughlin, and J. L. Encarnacao, eds.), pp. 3–36, North Holland: Elsevier Science Publishers B.V., 1988.
- [28] K. J. Weiler, *Topological structures for geometric modeling*. Ph.D. dissertation, Dept. Comput. Sci., Rensselaer Polytechnic Inst., Troy, NY, 1986.
- [29] M. S. Shephard and M. K. Georges, "Reliability of automatic 3D mesh generation," *Comput. Method. Appl. M.*, vol. 101, pp. 443–462, Dec. 1992.
- [30] T. J. Tautges, R. Meyers, K. Merkley, C. Stimpson, and C. Ernst, "MOAB: A Mesh-Oriented DataBase," Tech. Rep., Sandia, 2004.
- [31] R. M. O'Bara, M. W. Beall, and M. S. Shephard, "Attribute management system for engineering analysis," *Eng. Comput.*, vol. 18, no. 4, pp. 339–351, 2002.
- [32] R. M. O'Bara, M. W. Beall, and M. S. Shephard, "Analysis model visualization and graphical analysis attribute specification system," *Finite Elem. Anal. Des.*, vol. 19, pp. 325–348, May 1995.
- [33] M. S. Shephard, "The specification of physical attribute information for engineering analysis," *Eng. Comput.*, vol. 4, pp. 145–155, Sep. 1988.
- [34] Simmetrix Inc., "The simulation modeling suite." [Online]. Available: <http://www.simmetrix.com/>, [Accessed Apr. 5, 2012].
- [35] M. W. Beall and M. S. Shephard, "A general topology-based mesh data structure," *Int. J. Numer. Meth. Eng.*, vol. 40, no. 9, pp. 1573–1596, 1997.
- [36] W. Celes, G. H. Paulino, and R. Espinha, "A compact adjacency-based topological data structure for finite element mesh representation," *Int. J. Numer. Meth. Eng.*, vol. 64, pp. 1529–1556, Nov. 2005.
- [37] R. V. Garimella, "Mesh data structure selection for mesh generation and FEA applications," *Int. J. Numer. Meth. Eng.*, vol. 55, pp. 451–478, Oct. 2002.

- [38] R. V. Garimella, “MSTK: a flexible infrastructure library for developing mesh-based application,” in *Proc. 13th Int. Meshing Roundtable*, (Williamsburg, VA), 2004.
- [39] J.-F. Remacle and M. S. Shephard, “An algorithm oriented mesh database,” *Int. J. Numer. Meth. Eng.*, vol. 58, pp. 349–374, Sep. 2003.
- [40] M. S. Shephard and E. S. Seol, “Flexible distributed mesh data structure for parallel adaptive analysis,” in *Advanced Computational Infrastructures for Parallel and Distributed Adaptive Applications*, John Wiley & Sons, 2007.
- [41] M. S. Shephard, S. Dey, and J. E. Flaherty, “A straightforward structure to construct shape functions for variable p-order meshes,” *Comput. Method. Appl. M.*, vol. 147, pp. 209–233, Aug. 1997.
- [42] Interoperable Technologies for Advanced Petascale Simulations (ITAPS) Center, “The itaps imesh interface documentation.” [Online]. Available: <http://www.itaps.org/software/iMesh.html/index.html/>, [Accessed Apr. 5, 2012], Dec. 2010.
- [43] M. S. Shephard, “Meshing environment for geometry-based analysis,” *Int. J. Numer. Meth. Eng.*, vol. 47, pp. 169–190, Jan. 2000.
- [44] J. E. Flaherty, R. M. Loy, C. Özturan, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz, “Parallel structures and dynamic load balancing for adaptive finite element computation,” *Appl. Numer. Math.*, vol. 26, pp. 241–263, 1996.
- [45] C. Özturan, *Distributed environment and load balancing for adaptive unstructured meshes*. Ph.D. dissertation, Dept. Comput. Sci., Rensselaer Polytechnic Inst., Troy, NY, 1995.
- [46] J. D. Teresco, *A hierarchical partition model for parallel adaptive finite element computation*. Ph.D. dissertation, Dept. Comput. Sci., Rensselaer Polytechnic Inst., Troy, NY, 2000.
- [47] J. D. Teresco, M. W. Beall, J. E. Flaherty, and M. S. Shephard, “A hierarchical partition model for adaptive finite element computation,” *Comput. Method. Appl. M.*, vol. 184, pp. 269–285, Apr. 2000.
- [48] B. S. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey, “libMesh: a C++ library for parallel adaptive mesh refinement/coarsening simulations,” *Eng. Comput.*, vol. 22, no. 3–4, pp. 237–254, 2006.
- [49] L. Oliker, R. Biswas, and H. N. Gabow, “Parallel tetrahedral mesh adaptation with dynamic load balancing,” *Parallel Comput.*, vol. 26, pp. 1583–1608, 1999.

- [50] C. Özturan, H. L. deCougny, M. S. Shephard, and J. E. Flaherty, “Parallel adaptive mesh refinement and redistribution on distributed memory computers,” *Comput. Method. Appl. M.*, vol. 119, pp. 123–137, Nov. 1994.
- [51] J.-F. Remacle, O. Klaas, J. E. Flaherty, and M. S. Shephard, “Parallel algorithm oriented mesh database,” *Eng. Comput.*, vol. 18, pp. 274–284, Oct. 2002.
- [52] A. Ovcharenko, D. Ibanez, F. Delalandre, O. Sahni, K. E. Jansen, C. D. Carothers, and M. S. Shephard, “Neighborhood communication paradigm to increase scalability in large-scale dynamic scientific applications,” *Parallel Comput.*, vol. 38, no. 3, pp. 140–156, 2012.
- [53] J. D. Teresco, K. D. Devine, and J. E. Flaherty, “Partitioning and dynamic load balancing for the numerical solution of partial differential equations,” in *Numerical Solution of Partial Differential Equations on Parallel Computers* (A. M. Bruaset and A. Tveito, eds.), vol. 51 of *Lecture Notes in Computational Science and Engineering*, pp. 55–88, Springer Berlin Heidelberg, 2006.
- [54] M. Zhou, O. Sahni, K. D. Devine, M. S. Shephard, and K. E. Jansen, “Controlling unstructured mesh partitions for massively parallel simulations,” *SIAM J. Sci. Comput.*, vol. 32, pp. 3201–3227, 2010.
- [55] M. J. Berger and S. H. Bokhari, “A partitioning strategy for nonuniform problems on multiprocessors,” *IEEE Trans. Comput.*, vol. C-36, pp. 570–580, May 1987.
- [56] A. Patra and J. T. Oden, “Problem decomposition for adaptive hp finite element methods,” *Comput. Syst. Eng.*, vol. 6, pp. 97–109, Apr. 1995.
- [57] M. Warren and J. Salmon, “A parallel hashed oct-tree n-body algorithm,” in *Proc. 1993 ACM/IEEE Conf. Supercomput.*, Supercomput. ’93, (New York, NY, USA), pp. 12–21, ACM, 1993.
- [58] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdag, R. T. Heaphy, and L. A. Riesen, “A repartitioning hypergraph model for dynamic load balancing,” *J. Parallel. Distr. Com.*, vol. 69, pp. 711–724, Aug. 2009.
- [59] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek, “Parallel hypergraph partitioning for scientific computing,” in *Proc. 20th Int. Conf. Parallel Distrib. Process.*, IPDPS’06, (Washington, DC, USA), pp. 124–124, IEEE Computer Society, 2006.
- [60] G. Karypis, K. Schloegel, and V. Kumar., “ParMETIS: parallel graph partitioning and sparse matrix ordering library,” Tech. Rep., Dept. Comput. Sci. Eng., Univ. Minnesota, Twin Cities, 1997.

- [61] K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio, "New challenges in dynamic load balancing," *Appl. Numer. Math.*, vol. 52, pp. 133–152, Feb. 2005.
- [62] H. D. Simon, "Partitioning of unstructured problems for parallel processing," *Comput. Syst. Eng.*, vol. 2, no. 2-3, pp. 135–148, 1991.
- [63] U. V. Catalyurek and C. Aykanat, "Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, pp. 673–693, 1999.
- [64] G. Cybenko, "Dynamic Load Balancing for Distributed Memory Multiprocessors," *J. Parallel. Distr. Com.*, vol. 7, pp. 279–301, 1989.
- [65] B. Hendrickson and K. Devine, "Dynamic load balancing in computational mechanics," *Comput. Method. Appl M.*, vol. 184, pp. 485–500, 2000.
- [66] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 1st ed., Feb. 2001.
- [67] D. R. Musser, G. J. Derge, and A. Saini, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Boston, MA: Addison-Wesley, 2nd ed., 2001.
- [68] B. Stroustrup, *The C++ Programming Language*. Reading, MA: Addison-Wesley, 3rd ed., 1997.
- [69] J. G. Siek and A. Lumsdaine, "The Matrix Template Library: generic components for high-performance scientific computing," *Comput. Sci. Eng.*, vol. 1, no. 6, pp. 70–71, 1999.
- [70] L.-Q. Lee and A. Lumsdaine, "The Generic Message Passing framework," in *Proc. 2003 Int. Parallel Distrib. Process. Symp.*, p. 10 pp., Apr. 2003.
- [71] G. Berti, "A generic toolbox for the grid craftsman," in *Proc. 17th GAMM Seminar Construction Grid Generation Algorithms*, 2001.
- [72] G. Berti, "GrAL-the grid algorithms library," *Future Gener. Comp Sy.*, vol. 22, no. 1-2, pp. 110–122, 2006.
- [73] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr, "On the design of CGAL a computational geometry algorithms library," *Softw. Pract. Exper.*, vol. 30, pp. 1167–1202, Sep. 2000.
- [74] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander, "A generic grid interface for parallel and adaptive scientific computing. Part II: implementation and tests in DUNE," *Computing*, vol. 82, pp. 121–138, Jul. 2008.



- [75] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander, “A generic grid interface for parallel and adaptive scientific computing. part i: abstract framework,” *Computing*, vol. 82, pp. 103–119, Jul. 2008.
- [76] F. Cirak and J. C. Cummings, “Generic programming techniques for parallelizing and extending procedural finite element programs,” *Eng. Comput.*, vol. 24, pp. 1–16, Jan. 2008.
- [77] Silicon Graphics International (SGI), “Standard Template Library programmer’s guide.” [Online]. Available: <http://www.sgi.com/tech/stl/>, [Accessed Apr. 5, 2012], 2011.
- [78] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1 ed., Nov. 1994.
- [79] Scientific Computation Research Center (SCOREC) at Rensselaer Polytechnic Inst., “Simulation model and data management components.” [Online]. Available: <https://www.scorec.rpi.edu/software.php/>, [Accessed Apr. 5, 2012].
- [80] Argonne Leadership Computing Facility, Argonne National Laboratory, “Intrepid / challenger / surveyor guide.” [Online]. Available: <http://www.alcf.anl.gov/resource-guides/intrepid-and-surveyor-guide/>, [Accessed Apr. 5, 2012].
- [81] K. Hrbacek and T. Jech, *Introduction to Set Theory, Second Edition, Revised and Expanded*. Marcel Dekker, Inc., 1984.
- [82] K. Kuratowski, *Introduction to Set Theory and Topology*. PWN - Polish Scientific Publishers, 2nd ed., 1972.
- [83] Scientific Computation Research Center (SCOREC), “Flexible distributed Mesh DataBase user’s guide.” [Online]. Available: <http://scorec.rpi.edu/FMDB/documentation.html/>, [Accessed Apr. 5, 2012], 2012.
- [84] Scientific Computation Research Center (SCOREC) at Rensselaer Polytechnic Inst., “Scientific Computation Research Center software, list of computers.” [Online]. Available: [https://www.scorec.rpi.edu/wiki/List\\_of\\_Computers/](https://www.scorec.rpi.edu/wiki/List_of_Computers/), [Accessed Apr. 5, 2012], Dec. 2011.
- [85] A. Ovcharenko, K. Chitale, O. Sahni, K. E. Jansen, and M. S. Shephard, “Parallel anisotropic mesh adaptation with boundary layers,” *Int. J. Numer. Meth. Eng.*, 2012. unpublished.
- [86] A. K. Karanam, K. E. Jansen, and C. H. Whiting, “Geometry based pre-processor for parallel fluid dynamic simulations using a hierarchical basis,” *Eng. Comput.*, vol. 24, pp. 17–26, Jun. 2007.

- [87] C. Dobrzynski, M. Melchior, L. Delannay, and J.-F. Remacle, “A mesh adaptation procedure for periodic domains,” *Int. J. Numer. Meth. Eng.*, vol. 86, no. 12, pp. 1396–1412, 2011.
- [88] X. Luo and B. A., “Towards to periodic boundary condition on geometric model,” Tech. Rep., Rensselaer Polytechnic Inst., 2006. Scientific Computation Research Center (SCOREC).
- [89] Rensselaer Polytechnic Inst. and IBM and NY State, “Computational Center for Nanotechnology Innovations (CCNI).” [Online]. Available: <http://www.rpi.edu/research/ccni/>, [Accessed Apr. 5, 2012], 2011.
- [90] National Energy Research Scientific Computing (NERSC) Center, “Hopper.” [Online]. Available: <http://www.nersc.gov/users/computational-systems/hopper/>, [Accessed May 5, 2012], May 2012.
- [91] Karypis Lab at Dept. Comput. Sci. Eng. at Univ. Minnesota, “ParMETIS - parallel graph partitioning and fill-reducing matrix ordering.” [Online]. Available: <http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview/>, [Accessed Apr. 5, 2012], 2011.
- [92] Sandia National Laboratories, “Zoltan: parallel partitioning, load balancing and data-management services.” [Online]. Available: <http://www.cs.sandia.gov/Zoltan/>, [Accessed Apr. 5, 2012], 2010.
- [93] M. Zhou, O. Sahni, T. Xie, M. S. Shephard, and K. E. Jansen, “Unstructured mesh partition improvement for implicit finite element at extreme scale,” *J. Supercomput.*, vol. 59, pp. 1218–1228, 2012.
- [94] The Boost community, “The Boost C++ libraries.” [Online]. Available: <http://www.boost.org/>, [Accessed Apr. 5, 2012], 2007.