FMDB: FLEXIBLE DISTRIBUTED MESH DATABASE FOR PARALLEL AUTOMATED ADAPTIVE ANALYSIS

By

Eunyoung Seegyoung Seol

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:

_____
Mark S. Shephard, Thesis Adviser

_____
David R. Musser, Thesis Adviser

_____
Boleslaw K. Szymanski, Member

_____
Kenneth E. Jansen, Member

Rensselaer Polytechnic Institute
Troy, New York

July 2005
(For Graduation August 2005)

FMDB: FLEXIBLE DISTRIBUTED MESH DATABASE FOR PARALLEL AUTOMATED ADAPTIVE ANALYSIS

By

Eunyoung Seegyoung Seol

An Abstract of 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

The original of the complete thesis is on file
in the Rensselaer Polytechnic Institute Library

Examining Committee:

Mark S. Shephard, Thesis Adviser

David R. Musser, Thesis Adviser

Boleslaw K. Szymanski, Member

Kenneth E. Jansen, Member

Rensselaer Polytechnic Institute
Troy, New York

July 2005
(For Graduation August 2005)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

ACKNOWLEDGMENT

First, I would like to give special thank to my mentor, Dr. Mark S. Shephard, for his support and insightful guidance throughout my doctoral research work. He has lightened my tough way to accomplish the degree with computational science & engineering which was entirely new to me before I met him. I am grateful to Dr. David R. Musser for his invaluable technical advice and courses. I have learned a lot from him about advanced programming techniques and the importance of programming with concepts and philosophy. I thank Dr. Boleslaw K. Szymanski and Dr. Kenneth E. Janson for their time and effort to serve on my committee and reviewing this dissertation. I know I have been very fortunate to work closely with these prominent figures during my stay at Rensselaer. I am truly grateful to all my committee for their brilliance, help, support, and encouragement which led to my professional as well as personal development. Without their guidance and patience, this thesis would not be possible.

I express my gratitude to the great SCOREC people, Dr. Ottmar Klaas, Dr. Jean-François Remacle, Dr. Xiangrong Li, Dr. Nicolas Chevaugeon, Dr. Frédéric Alauzet, Dr. Andrew C. Bauer, Jie Wan, Xiaojuan Luo, Mohan Nuggehally, Onkar Osahni and Christophe Dupre, Dinesh Godavarty, Dr. Luzhong Yin, whom I had the pleasure of discussing with, learning from, and sharing friendship and joy each and every day. I thank Dr. Xiangrong Li for many fruitful discussions. I also thank Dr. Frédéric Alauzet for providing me parallel DG simulations.

Most of all, I thank my respectable parents, Ilsoon Lee and Insoo Seol, for letting me stand up on their shoulders. For their whole lives, they have shown personally the answer to what is true love and heartily support. I also thank my parents-in-law, Hwasook Lee and Soonjung Kwon, and my brothers, Dongchun and Dongjae Seol for their encouragement and understanding. Last but greatest, I thank my 3 great men. My wonderful husband, Dr. Yongchai Kwon, has been so supportive with his heartily understanding that a married woman can have a dream and furthermore can make the dream come true. My lovely, but sometimes naughty, two little men, Danny and Austin, have been so much patient for their greedy mommy since their birth. I must admit that my husband, Danny and Austin have given up so many ordinary things compared to other men and kids who have home-staying wife and mother. I would not be where I am today without love, support and understanding of all my family members.

This work is supported by the US Department of Energy's Scientific Discovery through Advanced Computing (SciDAC) program as part of the Terascale Simulation Tools and Technology (TSTT) center. The author greatly appreciates the financial support from this agency.

ABSTRACT

A mesh is piece-wise decomposition of the space/time domain where used by numerical simulation procedures. The data structure of the mesh strongly influences the overall performance of the simulations since it is an infrastructure executing underneath providing all needed mesh-based operations. From a fact that the flexibility of a mesh data structure comes from the levels of mesh entities and adjacencies present, and by the needs of a distributed mesh data structure operates in a scalable manner, this thesis focused on the development of a Flexible distributed Mesh DataBase (FMDB) capable of shaping its representation based on the specific needs of the application that efficiently supports parallel adaptive analysis in a parallel computing environment.

In order to properly maintain the needed representation even with mesh modification, the mesh entity creation/deletion operators are declared as function objects, initially undetermined. Once the needed representation is provided, they are dynamically set to the proper operators. The needed representation can be provided to the mesh database either by the user explicitly or by running the Dynamic Mesh Usage Monitor, which monitors mesh usage of the application and provides an appropriate representation. For the purpose of supporting distributed meshes on parallel computers, a partition model has been developed. The partition model is located between the partitioned mesh and the geometric model to represent mesh partitioning and support the mesh-based inter-partition operations.

Performance results of the FMDB well demonstrate its efficiency and scalability. Compared to the one-level fixed representation, a decrease in storage obtained with flexible reduced representations is $6-79\%$. The migration procedure with reduced representations outperforms full representations as the number of entities to migrate or the number of partitions increases. The FMDB is embedded in SCOREC simulation packages effectively supporting parallel automated adaptive analyses.

CHAPTER 1
INTRODUCTION

1.1 Background and Motivation

Computer-aided design and simulation have been an important tool for industry and scientific research for a wide range of problems. Three dimensional modeler systems, for example Parasolid^{TM} [28], ACIS^{TM} [91], Pro/Engineer^{TM} [68], have matured and can provide complex geometry definitions from a single machinery part to the whole product assembly. Simulation systems perform various analysis governed by partial differential equations that mathematically describe the physical phenomenon of interest [37, 103].

The finite element method is a powerful tool for solving partial differential equations on complex geometry domains [37, 103]. It is a double discretization process in which the geometric domain is discretized into a set of piecewise components and the equations to be solved are discretized over these individual piecewise components [86]. In the first step of the double discretization, a mesh is a geometric domain decomposition over which the simulation is to be run. Figure 1.1 illustrates a sample of a geometric model and a mesh of it.

The general topology-based spatial mesh representation consists of 0 to 3D topological entities and connectivity between them (adjacencies). A mesh data structure is a toolbox that provides the information required by the applications that create and/or use the mesh data. The data structure of the mesh strongly influences the overall performance of the simulations.

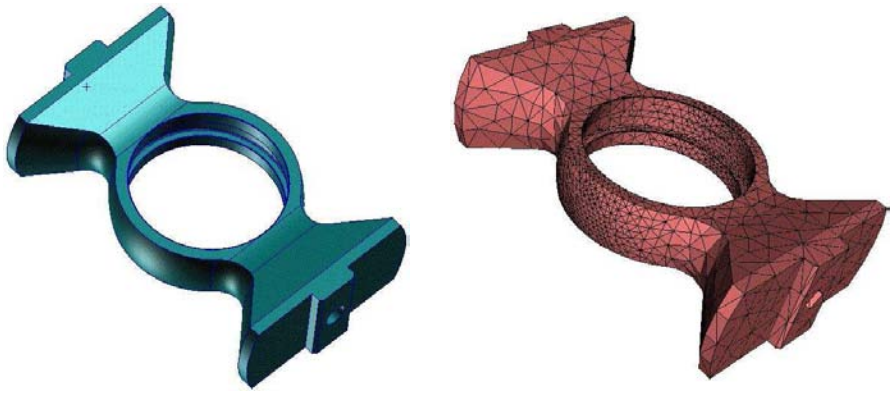


Figure 1.1: Bearing: geometric model and mesh

From the fact that the particular mesh-based application has its own needs of the level of entities and adjacencies, the effective mesh data structure is one that provides mesh entities and adjacencies needed by the mesh-based application. Clearly, the design of good mesh data structure is highly dependent on the application for which it is used. Generally, there are 3 approaches in designing a mesh data structure.

- Ad-hoc mesh data structure shaped to the specific application [10, 16, 36, 39, 52, 53, 54, 61, 69, 71, 90].
- Fixed general mesh representation [6, 13, 20, 66, 83, 89, 94, 100].
- Flexible mesh representation that is capable of shaping its representation dynamically based on the needs of the application [32, 75, 76, 92].

It is not possible to design a single mesh data structure that is the most compact and the most efficient and meets the needs of all applications. Therefore the usual approach was to use a mesh data structure that was needed for the specific application (the first approach). Some specific data structures have been published for the specific parts of numerical analysis, such as mesh generation [10, 65, 54], mesh adaptation [16, 65, 39, 61, 71] and the solution process [36], however a mesh data structure specific to the application lacks extendibility and reusability. As the needs of applications became more complex and the users wanted easy software extendibility for new applications, the general mesh data structure that works with reasonable efficiency in most operations of the applications was taken (the second approach). But the general mesh data structure has disadvantage of inefficiency in some cases as well as typically requiring more storage. This inevitably leads to the consideration that mesh data structure must be flexible enough to easily switch between various representations for different phase of an application and build the custom mesh representation based on a meaningful criteria for achieving a good compromise between the storage and computation costs, optimized for the particular application (the third approach). A flexible mesh data structure is one that is capable of supporting representation options based on the needs of the application [32, 75, 92].

As for the first trial of a flexible mesh data structure, the Algorithm Oriented Mesh Database (AOMD) [75] has been proposed and extended to support distributed meshes [76], however it was not fully implemented for the support of all mesh needs. The AOMD did not fully address algorithmic complexity, efficiency or software engineering practices. The goal of the work presented in this thesis was the design, algorithmic analysis

and effective implementation of a complete flexible distributed mesh data management system including efficient mesh manipulation algorithms that satisfy the specific needs of applications on distributed domain as well as providing the best efficiency both in memory requirements and computational cost. Particularly, given:

- any mesh with manifold/non-manifold[1] domain;
- any user-requested or application-needed mesh representation option;

develop a distributed mesh data structure that is able to shape the mesh representation dynamically based on the mesh representation needs for the purpose of efficiently supporting parallel automated adaptive simulations, called Flexible distributed Mesh DataBase (FMDB).

1.2 Organization

Chapter 2 introduces general topology-based mesh data structures, compares various mesh representation options and presents a historic review of mesh data structures. Chapter 3 describes the design of the flexibility in the FMDB that can shape its structure dynamically based on the user requested representation. To shape the FMDB, the needed mesh representation should be provided for each application. Since it is not always possible to know the needed representation in advance, the Dynamic Mesh Usage Monitor (DMUM) was developed to inform the software of the representational requests made. The needed representation can be provided either by the user explicitly or by the output of the DMUM. Chapter 4 introduces the design of distributed meshes and provides definitions, properties and algorithms of a partition model. The partition model is an intermediary model located between the mesh and the geometric model to support mesh partitioning and mesh-level parallel operations through inter-partition communication links. Chapter 5 presents the mesh migration procedure with the flexible mesh representations to extend flexibility to distributed meshes. Chapter 6 discusses the software aspects of the FMDB in terms of programming elements utilized and the design of each component of the system, and Chapter 7 provides performance results. Chapter 8 presents applications of the FMDB, and demonstrates scalability of it with parallel mesh adaptation. Chapter 9 concludes this thesis by summarizing the contributions and discussing the future work.

[1] For definition and examples, see references [58, 101]

For the readers interested in various mesh representation options in detail, Appendices A through D follow with the mesh algorithms of four representation options and their computational efficiency.

1.3 Nomenclature

V	the model, $V \in \{G,\ P,\ M\}$ where G signifies the geometric model, P signifies the partition model, and M signifies 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. Shorthand for $V\{V^d\}_i$, $d = 0$ for a vertex, $d = 1$ for an edge, $d = 2$ for a face, and $d = 3$ for a region. In topology, edges, faces, and regions are bounded by the lower order entities.
$\{\partial(V_i^d)\}$	a set of entities on the boundary of V_i^d.
$\{V_i^d\{V^q\}\}$	the set of entities of dimension q in model V that are adjacent to V_i^d.
$V_i^d\{V^q\}_i$	the i^{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.
$\mathscr{P}[M_i^d]$	a set of partition id(s) where entity M_i^d exists.
N_i	the number of entities of dimension i in the mesh. Shorthand for $\lvert\{M\{M^i\}\}\rvert$. $N_0,\ N_1,\ N_2,\ N_3$ are, respectively, the number of vertices, edges, faces, regions in the mesh.
$\mathcal{R}$	mesh representation matrix.

Examples

$\{M\{M^2\}\}$	the set of all the faces in the mesh.
$\{M_3^1\{M^3\}\}$	a set of mesh regions adjacent to mesh edge M_3^1.
$M_1^3\{M^1\}_2$	the 2^{nd} edge adjacent to mesh region M_1^3.

CHAPTER 2
RELATED INVESTIGATIONS

In a geometry-based analysis environment, mesh data structures house the discretization of the domain, a mesh, and provide the mesh-level services to applications. This chapter presents the basics of a mesh data structure including functional requirements, the analysis of various mesh representation options, and a historic review.

2.1 Overview of Geometry-based Analysis Environment

The structures used to support the problem definition, the discretization of the model and their interactions are central to mesh-based analysis methods like finite element and finite volumes. The geometry-based analysis environment consists of four parts: *the geometric model* which houses the topological and shape description of the domain of the problem, *attributes* describing the rest of information needed to define and solve the problem, *the mesh* which describes the discretized representation of the domain used by the analysis method, and *fields* which describe the distribution of solution tensors over the mesh entities [5, 86, 89]. Figure 2.1 represents the general interactions between the four components.

2.1.1 Geometric model

The most common geometric representation is a boundary representation. A general representation of general non-manifold domains is the Radial Edge Data Structure [101, 102]. Non-manifold models are common in engineering analyses. Simply speaking, non-manifold models consist of general combinations of solids, surfaces, and wires. Figure 2.2

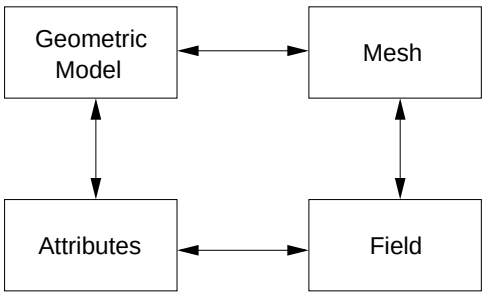


Figure 2.1: The relationship between components of the geometry-based analysis environment [5]

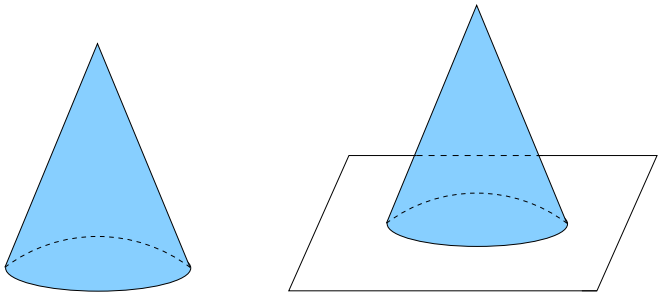


Figure 2.2: Example of (left) manifold and (right) non-manifold models

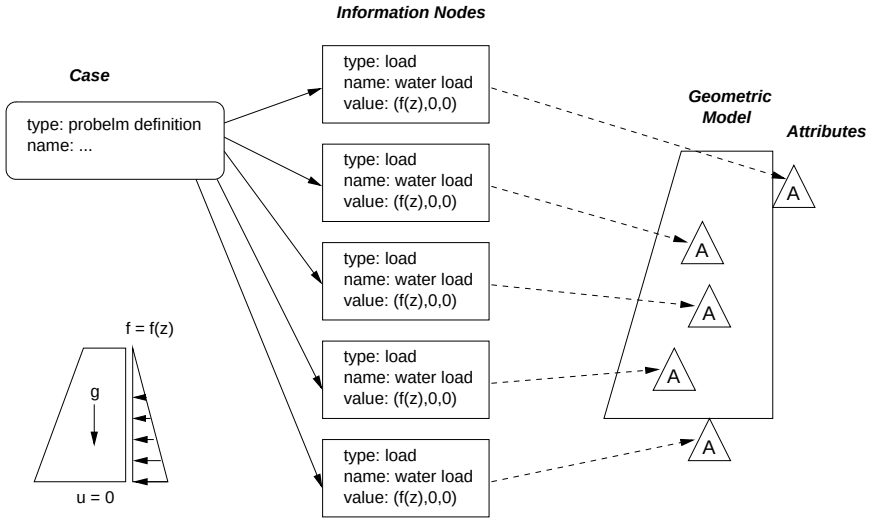


Figure 2.3: Example geometry-based problem definition [5]

illustrates examples of manifold and non-manifold model.

In the boundary representation, the model is a hierarchy of topological entities called regions, shells, faces, loops, edges, vertices, and in case of non-manifold models, use entities for vertices, edges, loops, and faces. The data structure implementing the geometric model supports operations to find the various model entities that make up a model, information about which model entities are adjacent to a given entity, operations relating to perform geometric shape queries, and queries about what attributes are associated with model entities.

2.1.2 Attribute

In addition to geometric model, the definition of a problem requires other information that describes material properties, loads and boundary conditions, etc. These are described in terms of tensor-valued attributes and may vary in both space and time.

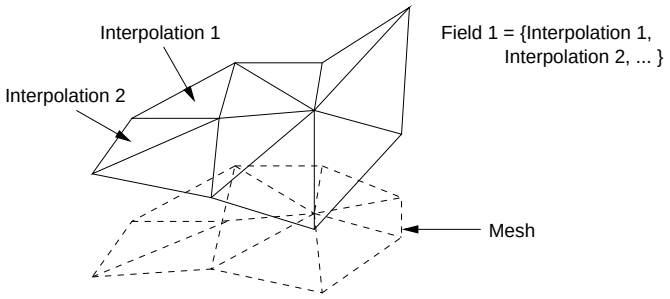


Figure 2.4: Representation of a field defined over a mesh [5]

Attributes are meaningful only when applied to a geometric model entity.

Figure 2.3 illustrates an example of a problem definition. The problem being modeled is a dam subjected to loads due to gravity and due to the water behind the dam. There is a set of attribute information nodes that are all under the attribute case for the problem definition. When this case is associated with the geometric model, attributes are created and attached to the individual model entities on which they act [5, 89]. The attributes are indicated by triangles with A's inside of them.

2.1.3 Mesh

A mesh is a geometric discretization of a domain. With restrictions on the mesh entity topology [6], a mesh is represented with a hierarchy of regions, faces, edges and vertices. Each mesh entity maintain a relation, called geometric classification [6, 85], to the model entity that it was created to partially represent. Geometric classification allows an understanding of which attributes (e.g. boundary conditions or material properties) are related to the mesh entities and the how the solution relates back to the original problem description, and is critical in mesh generation and adaptation [5, 6, 85, 86]. The detailed discussions on the mesh is presented in §2.2.

2.1.4 Field

A field describes the variation of solution tensors over the mesh entities discretizing one or more entities in a geometric model. The spatial variation of the field is defined in terms of mesh level distribution functions [5, 86]. Figure 2.4 demonstrates the concept of a field written in terms of C^0 interpolating distribution functions.

2.2 General Topology-Based Mesh Data Structure

The mesh consists of a collection of mesh entities of controlled size, shape, and distribution. The relationships of the entities defining the mesh are well described by topological adjacencies, which form a graph of the mesh. A critical capability needed by automated, adaptive geometry-based analysis procedures is to manipulate the mesh of the analysis domain. A mesh data structure is a toolbox that provides the mesh-level services to the applications that create/use the mesh data. The differing needs of the applications dictate that the database be able to answer to the needed queries about the mesh. The three functional requirements of a general topology-based mesh data structure are: topological entities, geometric classification, and adjacencies between entities [6].

2.2.1 Topological entities

Topology provides an unambiguous, shape-independent abstraction of the mesh. With reasonable restrictions on the topology, a mesh is represented with only the basic 0 to d dimensional topological entities, where d is the dimension of the domain of the interest. The full set of mesh entities in 3D is $\{\{\mathrm{M}\{M^0\}\}, \{\mathrm{M}\{M^1\}\}, \{\mathrm{M}\{M^2\}\}, \{\mathrm{M}\{M^3\}\}\}$, where $\{\mathrm{M}\{M^d\}\}$, $d = 0, 1, 2, 3$, are, respectively, the set of vertices, edges, faces, and regions. Mesh edges, faces, and regions are bounded by the lower order mesh entities.

Restrictions on the topology of a mesh are:

- Regions and faces have no interior holes.
- Each entity of order d in a mesh, M_i^d, may use a particular entity of lower order, p, M_j^p, $p < d$, 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. (Note, based on mesh entity classification, it is possible to relax this restriction in the case of equal order classification [6])

The first restriction means that regions may be represented by one shell of faces that bounds them, and faces may be represented by one loop of edges that bounds them. The second restriction allows the orientation of an entity to be defined in terms of its boundary entities without introduction of use entities. The third restriction means that an interior entity is uniquely specified by its bounding entities.

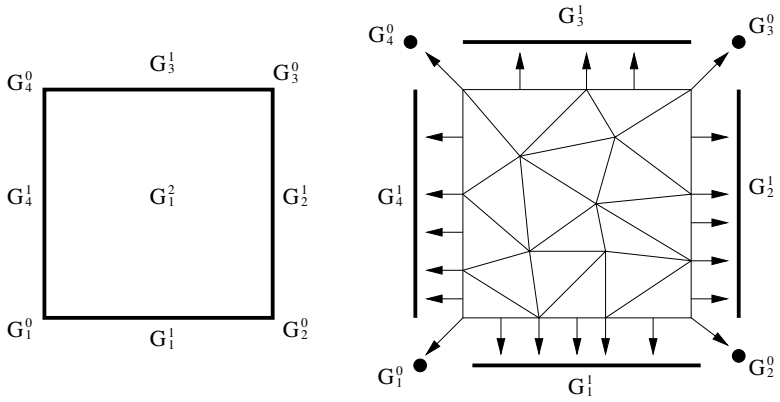


Figure 2.5: Example of simple model(left) and mesh(right) showing their association via geometric classification [89]

2.2.2 Geometric classification

The linkage of the mesh to the geometric model is critical for mesh generation and adaptation procedures since it allows the specification of analysis attributes in terms of the original geometric model, the proper approximation of the geometry during mesh adaptation and supports direct links to the geometric shape information of the original domain need to improve geometric approximation and useful in p-version element integration [5, 6, 85, 86].

The unique association of a mesh entity of dimension d_i, $M_i^{d_i}$, to the geometric model entity of dimension d_j, $G_j^{d_j}$, $d_i \leq d_j$, on which it lies is termed geometric classification, and is denoted $M_i^{d_i} \sqsubset G_j^{d_j}$, where the classification symbol, $\sqsubset$, indicates that the left hand entity, or a set of entities, is classified on the right hand entity. In Figure 2.5, a mesh of simple square model with entities labeled is shown with arrows indicating the classification of the mesh entities onto the model entities. All of the interior mesh faces, mesh edges, and mesh vertices are classified on the model face G_1^2.

2.2.3 Adjacencies

Adjacencies describe how 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 on the closure of the entity for a downward adjacency ($d > q$), or for which the entity is part of the closure for an upward adjacency ($d < q$). For denoting specific downward first-order adjacent entity, $M_i^d\{M^q\}_j$, the ordering conventions can be used to enforce the order. Figure 2.6, 2.7, and 2.8 illustrate a common canonical order of bounding entities. Figure 2.9 is an adjacency graph that depicts 12 first-order adjacencies possible in the

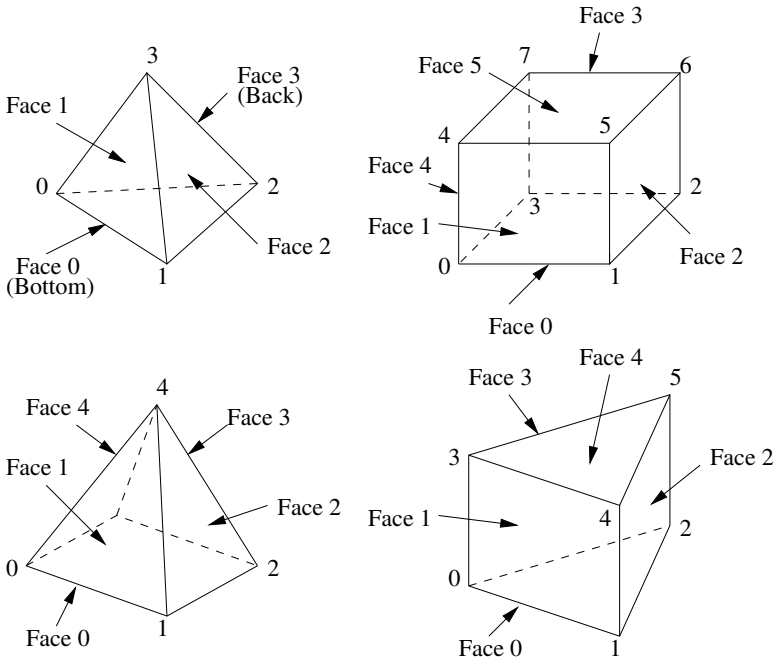


Figure 2.6: Vertex and face order on a region [89]

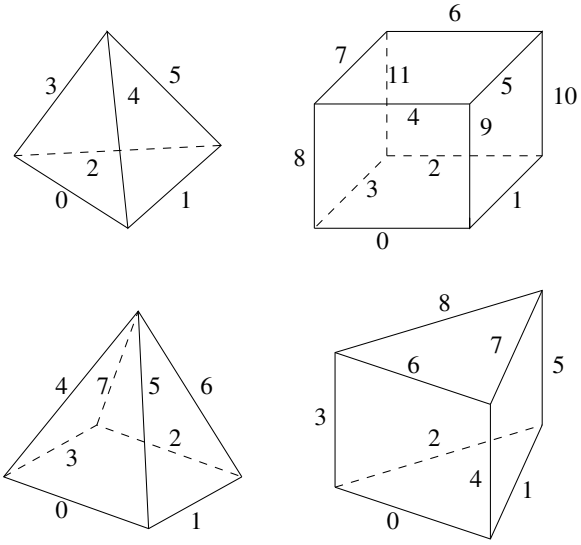


Figure 2.7: Edge order on a region [89]

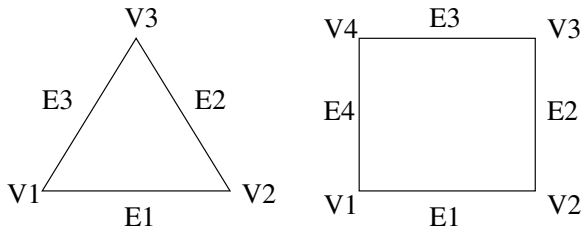


Figure 2.8: Edge order on a face [89]

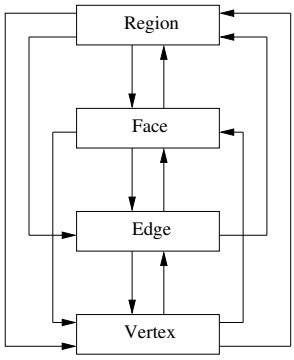


Figure 2.9: 12 adjacencies possible in the mesh representation [31]

mesh data structure where a solid box and a solid arrow denote, respectively, explicitly stored level of entities and explicitly stored adjacencies from outgoing level to incoming level.

For an entity of dimension d, second-order adjacencies describe all the mesh entities of dimension q that share any adjacent entities of dimension b, where $d \neq b$ and $b \neq q$. Second-order adjacencies can be derived from first-order adjacencies.

Examples of adjacency requests include: for a given face, the regions on either side of the face (first-order upward); the vertices bounding the face (first-order downward); and the faces that share any vertex with a given face (second-order).

2.3 Mesh Representation Options

Depending on the levels of entities and adjacencies explicitly stored in the representation, there are many options in the design of the mesh data structure.

2.3.1 Criteria

The mesh representation can be categorized with two criteria, resulting in 4 different groups.

- *Full vs. 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 vs. incomplete*

  Mesh data structures can be differentiated based on the cost of adjacency retrieval.

- Data structures for which all adjacencies can be obtained by one of instant access, construction and traversal in O(1) time.
- Data structures for which all adjacencies can be obtained, but one or more adjacency requires time which is a function of mesh size.
- Data structures of which adjacencies cannot be obtainable.

Completeness of adjacency indicates the ability of a mesh representation to provide any type of adjacencies requested without involving an operation dependent on the mesh size such as the global mesh search or mesh traversal. Regardless of full or reduced, if all adjacency information is obtainable in O(1) time (the first circle), the representation is complete, otherwise it is incomplete.

The *general* topology-based mesh data structures must satisfy completeness of adjacencies to support adaptive analysis efficiently. It doesn't necessarily mean that all d level entities and adjacencies need be explicitly stored in the representation. There are many representation options in the design of general topology-based mesh data structure.

2.3.2 Minimum sufficient representation

This work is built on the hypotheses of the AOMD. The hypotheses used in the AOMD are:

1. Any mesh entity of dimension higher than zero can be described by a set of mesh entities of any lower dimension.
2. Two entities are the same if they have the same vertices.
3. Predefined order of downward adjacencies are used.

The first hypothesis means that entities are represented using at least one set of entities of any lower dimension. Regions may be defined with faces, edges or vertices, faces may be defined with edges or vertices, and edges are defined with vertices. From this, it is always possible to obtain mesh entities' vertices. Vertices are atomic entities, thus, to be able to differentiate vertices, unique id's are assigned to vertices and the vertex id is used in the entity equality operator. The second hypothesis means that two mesh entities are comparable for equality based on vertex id's even though their representations are different. For example, two regions, one defined by edges and the other defined by

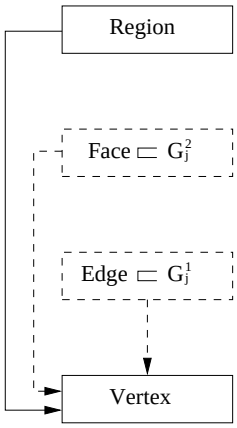


Figure 2.10: Adjacency graph of the MSR

faces, are always comparable in the AOMD. The third hypothesis, the predefined ordering of $\{M^d\{M^p\}\}$, $p < d$, (Figure 2.6, 2.7, and 2.8), enforces uniqueness in restoring non-existing mesh entities and adjacencies.

Based on the hypotheses, the AOMD identified a minimum set of mesh entities and adjacencies needed to be able to construct all entities with their 12 adjacencies and geometric classification without losing any information about the mesh, called *the minimum sufficient representation* (MSR). The MSR consists of all mesh entities equally classified on the equal dimension model entity and all vertices, that is $\{\mathrm{M}\{M^0\}\}$; $\{\mathrm{M}\{M^1\}\}$ classified on model edges; $\{\mathrm{M}\{M^2\}\}$ classified on model faces; and $\{\mathrm{M}\{M^3\}\}$ classified on model regions. All the vertices are present in the representation but, only ones classified on model vertices require classification information. For the remaining vertices, their geometric classification can be unknown.

With the MSR, mesh entities of dimension greater than *0* are defined with vertices since all faces and edges are not available for entity definition. Adjacency $\{M_i^2\{M^0\}\}$ and $\{M_i^1\{M^0\}\}$ are maintained only for equally classified faces and edges. Figure 2.10 depicts the adjacency graph of the MSR. In the adjacency graph, a dotted box denotes that among entities of the level, only equally classified ones are explicitly stored, and a dotted arrow denotes that adjacencies from an outgoing level to an incoming level are maintained only for the stored entities. In Figure 2.10, the boxes for faces and edges, and the lines for adjacency $\{M^2\{M^0\}\}$ and $\{M^1\{M^0\}\}$ are dotted since non-equally classified faces and edges are omitted and adjacencies $\{M^2\{M^0\}\}$ and $\{M^1\{M^0\}\}$ are maintained only for the existing faces and edges.

Table 2.1: Four categories of mesh representations

	full	reduced
complete	(a), (b), (c)	(e)
incomplete	(d)	(f), MSR

2.3.3 Examples of mesh representation options

The representation given in Figure 2.9 is *the greedy representation* where all d levels of entities and all possible 12 adjacencies are stored. Figure 2.11 illustrates adjacency graphs of six additional 3D mesh representation options.

Representation (a) is *the circular adjacency representation* that stores downward adjacencies from each entity to the entity one dimension lower and stores adjacencies from the vertices up to the highest-order entities that are using them (in a 3D manifold mesh, this would be the mesh regions) [6]. The adjacency graph of the circular adjacency representation has been simplified to be a manifold mesh. In case of non-manifold, vertices can point to mesh faces and mesh edges, respectively, classified on model faces and model edges. For simplicity of explanation, we discuss the simpler version. Representation (b) is *the one-level adjacency representation* that maintains adjacencies between entities one dimension apart [6]. Representation (e) is *the complete minimum sufficient representation* that stores the minimum sufficient representation plus upward adjacencies from vertices to their bounding entities of level > 0 [13]. Representation (f) is the classic mesh connectivity structure that describes the mesh only in terms of elements and nodes, and also has been used for several finite element applications [6]. Although the classic mesh data structure is sufficient for the fixed finite element analysis codes, it is incomplete and is therefore inadequate for a full range of procedures of an adaptive analysis due to lack of general topological information and information relating the mesh back to the original geometric model.

Mesh representation can be grouped in four categories based on their properties between full vs. reduced and complete vs. incomplete. In Figure 2.11, (a) to (c) 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, (d) is full and incomplete since it requires mesh level global search or traversal to get proper adjacencies, (e) is reduced and complete, and (f) and the MSR are reduced and incomplete. (See Table 2.1)

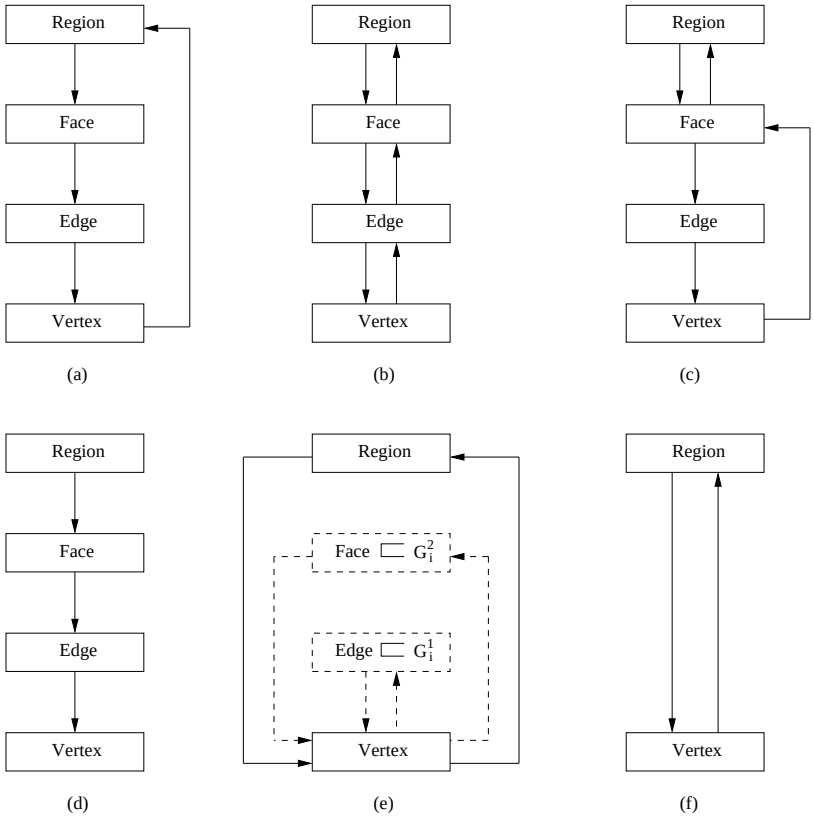


Figure 2.11: Example of 3D mesh representations

2.4 Analysis of Mesh Representation Options

Important factors in designing a mesh data structure are *storage* and *computational efficiency*, which are mainly dominated by the entities and adjacencies present in the mesh representation. The analysis of mesh data structures of various representations suggests how the mesh representation option and intelligent mesh algorithms are important to achieve efficiency with mesh applications.

In [6], Beall and Shephard presented three different implementation options in terms of mesh representation and compared efficiency of each option based on storage requirement and computational efficiency. Garimella [31] also presented the comparison among different mesh representations, from six complete to four reduced ones based on storage requirement and computational efficiency of adjacency retrieval. The analysis presented in this section uses a tool devised for evaluating efficiency of mesh data structures which is more extensive and precise in a fact that the analysis is based on 22 primary mesh operators and object-oriented programming paradigm [24, 82] where each entity's information is stored in the entity as internal members.

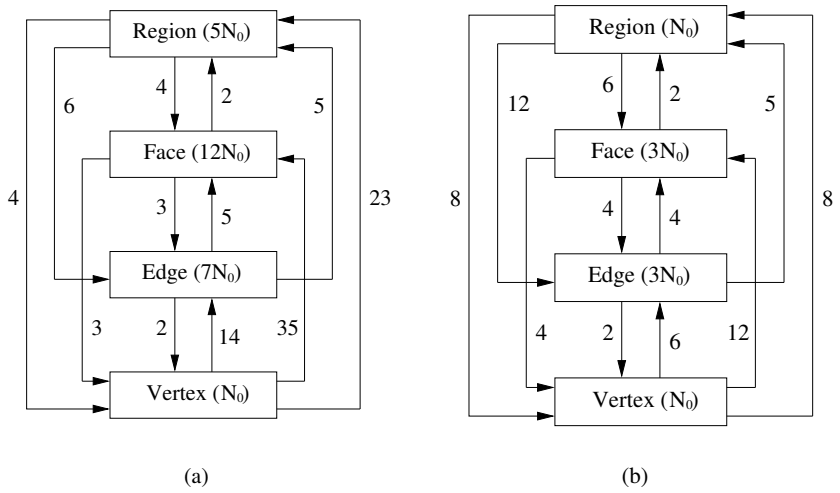


Figure 2.12: Average number of entities and adjacencies [6, 31]: (a) tetrahedral mesh (b) hexahedral mesh

For the purpose of analyzing various mesh representation options, it is necessary to know the average number of entities and adjacencies in the mesh. Figure 2.12 presents the relative number of mesh entities of different dimension to the number of vertices and the average size of 12 adjacency sets, respectively, of tetrahedral and hexahedral meshes. In the figures, the number in parentheses in any box represents the number of entities of that type in a mesh expressed in terms of the number of vertices in the mesh, N_0. Also the number by the each edge connecting two different types of mesh entities (representing $\{M_i^d\{M^q\}\}$), is the average $|\{M_i^d\{M^q\}\}|$ in the mesh.

For quantifying performance of a mesh data structure, two measurements, storage cost and computational cost in terms of the number of atomic operation steps are used.

2.4.1 Storage cost

The memory usage of a mesh data structure is computed with the following equation [31, 75].

$$\begin{aligned} Storage(M) &= \sum_{d=0}^{3}\sum_{i=1}^{N_d} Storage(M_i^d) \\ &= \sum_{d=0}^{3}\sum_{i=1}^{N_d}(S_{ent} + \text{the number of adjacencies stored at } M_i^d \times S_{adj}) \\ &= \sum_{d=0}^{3}\sum_{i=1}^{N_d}(S_{ent} + \sum_{q=0}^{3,q\neq d} \mid \{M_i^d\{M^q\}\} \mid \times S_{adj}) \end{aligned} \tag{2.1}$$

where N_i is the number of mesh entities of dimension i, $i = 0, 1, 2, 3$, S_{ent} is the amount of memory each mesh entity uses (in the FMDB, at least, this includes identifier, dimension, geometrical classification), S_{adj} is the amount of memory each adjacency uses (it is excluded from the memory usage of the mesh entity itself), and, $|\{M_i^d\{M^q\}\}|$ is the number of adjacent entities $\{M_i^d\{M^q\}\}$ explicitly stored in the mesh data structure

For example, for a tetrahedral mesh that has a region which is bounded by 4 vertices and bi-directional adjacencies between vertices and a region ($\{M_i^0\{M^3\}\}$, $\{M_i^3\{M^0\}\}$), the memory requirement is:

$$\begin{aligned} Storage(M) &= (S_{ent} + 4S_{adj}) \ /* \text{ storage for a region } */ \\ &\quad + 4(S_{ent} + S_{adj}) \ /* \ 4 \times \text{ storage for a vertices } */ \\ &= 5S_{ent} + 8S_{adj} \end{aligned} \tag{2.2}$$

Based on the mesh statistics in Figure 2.12 and Equation 2.1, the storage requirements for various mesh representations are computed as the following:

- Greedy adjacency representation

$$\begin{aligned} Storage(M) &= N_0(S_{ent} + (14 + 35 + 23) \times S_{adj}) + N_1(S_{ent} + (2 + 5 + 5) \times S_{adj}) \\ &\quad + N_2(S_{ent} + (3 + 3 + 2) \times S_{adj}) + N_3(S_{ent} + (4 + 6 + 4) \times S_{adj}) \\ &= (N_0 + N_1 + N_2 + N_3) \times S_{ent} + (92N_0 + 12N_1 + 8N_2 + 14N_3) \times S_{adj} \\ &= (N_0 + 7N_0 + 12N_0 + 5N_0) \times S_{ent} \\ &\quad + (92N_0 + 12 \cdot 7N_0 + 8 \cdot 12N_0 + 14 \cdot 5N_0) \times S_{adj} \\ &= 25N_0S_{ent} + 342N_0S_{adj} \end{aligned} \tag{2.3}$$

- Circular adjacency representation

$$\begin{aligned} Storage(M) &= N_0(S_{ent} + 23S_{adj}) + N_1(S_{ent} + 2S_{adj}) + N_2(S_{ent} + 3S_{adj}) + N_3(S_{ent} \\ &\quad + 4S_{adj}) \\ &= (N_0 + N_1 + N_2 + N_3) \times S_{ent} + (23N_0 + 2N_1 + 3N_2 + 4N_3) \times S_{adj} \\ &= (N_0 + 7N_0 + 12N_0 + 5N_0) \times S_{ent} \\ &\quad + (23N_0 + 2 \cdot 7N_0 + 3 \cdot 12N_0 + 4 \cdot 5N_0) \times S_{adj} \\ &= 25N_0S_{ent} + 93N_0S_{adj} \end{aligned} \tag{2.4}$$

- One-level adjacency representation

$$\begin{aligned} Storage(M) &= N_0(S_{ent} + 14S_{adj}) + N_1(S_{ent} + (2+5) \times S_{adj}) \\ &\quad + N_2(S_{ent} + (3+2) \times S_{adj}) + N_3(S_{ent} + 4S_{adj}) \\ &= (N_0 + N_1 + N_2 + N_3) \times S_{ent} + (14N_0 + 7N_1 + 6N_2 + 4N_3) \times S_{adj} \\ &= (N_0 + 7N_0 + 12N_0 + 5N_0) \times S_{ent} \\ &\quad + (14N_0 + 7 \cdot 7N_0 + 6 \cdot 12N_0 + 4 \cdot 5N_0) \times S_{adj} \\ &= 25N_0S_{ent} + 155N_0S_{adj} \end{aligned} \tag{2.5}$$

- Reduced representations

  The exact amount of storage requirement varies depending on the mesh due to the number of boundary edges and faces vary.

Consider one million tetrahedral mesh and S_{ent} and S_{adj} are, respectively, 70 and 4 bytes. The mesh consists of 55,600 faces classified on model faces among total 2,083,475 faces and 1,435 edges classified on model edges among 1,264,707 edges, and 193,932 vertices.

$$\begin{aligned} Storage(M) &= N_0 \cdot S_{ent} + N_1(S_{ent} + 2 \times S_{adj}) \\ &\quad + N_2(S_{ent} + 3 \times S_{adj}) + N_3(S_{ent} + 4 \times S_{adj}) \\ &= (N_0 + N_1 + N_2 + N_3) \times S_{ent} + (2N_1 + 3N_2 + 4N_3) \times S_{adj} \\ &= (193,932 + 1,435 + 55,600 + 1,000,000) \times S_{ent} \\ &\quad + (2,870 + 166,800 + 4,000,000) \times S_{adj} \\ &= 1,250,967 \cdot S_{ent} + 4,169,670 \cdot S_{adj} \\ &= 104.24 \text{ MB} \end{aligned} \tag{2.6}$$

$$\begin{aligned} Storage(M) &= N_0(S_{ent} + 23 \times S_{adj}) + N_1(S_{ent} + 2 \times 2 \times S_{adj}) \\ &\quad + N_2(S_{ent} + 2 \times 3 \times S_{adj}) + N_3(S_{ent} + 4 \times S_{adj}) \\ &= (N_0 + N_1 + N_2 + N_3) \times S_{ent} + (23N_0 + 4N_1 + 6N_2 + 4N_3) \times S_{adj} \\ &= (193,932 + 1,435 + 55,600 + 1,000,000) \times S_{ent} \\ &\quad + (4,460,436 + 5,740 + 333,600 + 4,000,000) \times S_{adj} \\ &= 1,250,967 \cdot S_{ent} + 8,799,776 \cdot S_{adj} \\ &= 122.75 \text{ MB} \end{aligned} \tag{2.7}$$

Table 2.2: Storage requirement for 1 million tetrahedral mesh (MB)

representation	memory
MSR	96.4
complete MSR	116.4
one level	420.7

Table 2.3: Storage decrease for 1 million tetrahedral mesh (%)

representation	by equation	by experiment
MSR	75.29	77.08
complete MSR	70.90	72.32

$$\begin{aligned} Storage(M) &= N_0 \cdot (S_{ent} + 14 \times S_{adj}) + N_1(S_{ent} + 7 \times S_{adj}) \\ &\quad + N_2(S_{ent} + 5 \times S_{adj}) + N_3(S_{ent} + 4 \times S_{adj}) \\ &= (N_0 + N_1 + N_2 + N_3) \times S_{ent} + (14N_0 + 7N_1 + 5N_2 + 4N_3) \times S_{adj} \\ &= (193,932 + 1,264,707 + 2,083,475 + 1,000,000) \times S_{ent} \\ &\quad + (14 \cdot 193,932 + 7 \cdot 1,264,707 + 5 \cdot 2,083,475 + 4 \cdot 1,000,000) \times S_{adj} \\ &= 4,542,114 \cdot S_{ent} + 25,985,372 \cdot S_{adj} \\ &= 421.89 \text{ MB} \end{aligned} \tag{2.8}$$

For the example mesh, the storage requirement for the MSR and the complete MSR is, respectively, 104.24 MB (Equation 2.6) and 122.75 MB (Equation 2.7) by computation while the storage with one-level adjacency representation is 421.89 MB (Equation 2.8). Table 2.2 shows the actual storage requirements for the example mesh.

Decrease in memory consumption using flexible representations compared to the fixed one-level mesh representation can be computed with Equation 2.9. Table 2.3 shows the decrease of memory cost of the MSR and the complete MSR compared to the one-level representation.

$$\textit{Storage decrease from flexibility} = 100 - (\frac{\text{memory for user-requested rep.}}{\text{memory for one-level rep.}}) \times 100 \ (\%) \tag{2.9}$$

2.4.2 Computational cost

Computational efficiency of mesh data structure refers to the execution time for manipulating mesh entities including create/delete entities and obtaining mesh entities'

Table 2.4: Run time of basic statements

Operator	Explanation	# step
CREATE A	create an object A	1
PUT(*container*, A)	write data A to *container*	1
PUT_UNIQ(*container*, A)	write data A uniquely to *container* (size n)	$\lg n$+2
GET(A)	retrieve data A from storage	1
IF (*cond*)	condition	1
$A \leftarrow B$	assignment	1
FIND(*container*, A)	search data A from *container* (size n)	$\lg n$
RETURN	return to the caller	1
MARK(M_i^d, *tag_id*, *data*)	attach *data* to M_i^d with *tag_id*	2
UNMARK(M_i^d, *tag_id*)	remove *data* attached to M_i^d with *tag_id*	2
MARKED(M_i^d, *tag_id*)	return *true* M_i^d has attached data with *tag_id*	2

adjacency relations, which varies for different mesh representations.

In analyzing the various mesh operators in terms of computational efficiency, we use *run-time* of an algorithm, denoted as *Time*, which is the number of primitive operations or steps executed as a unit of measuring the time costs. For convenience, we define the notion of *step* a constant amount of time required to execute each line of pseudo code in the algorithm, which is as machine independent as possible [19].

Table 2.4 presents the run time cost in the number of steps of basic statements used in mesh operators. According to the complexity of generic search algorithms in the STL [84], we assume the run time of search algorithm, $FIND$, to be $\lg n$ for the container of size n. PUT_UNIQ writes data into a container uniquely involving search ($\lg n$), condition check(1), and write(1), thus its run time is $\lg n$+2. Note the statement $RETURN$ performs at most once throughout a mesh operator. The statements $MARK$, $UNMARK$, and $MARKED$ are used for associating arbitrary data to the entity with arbitrary user-defined unique tag id. They also can be used in marking/unmarking entity for specific purpose. In the analysis, the run time for changing attached data to an entity or checking if the data is attached to the entity is taken as 2; 1 for accessing space reserved for tag id and 1 for either changing value or getting it.

The run time can be provided in an asymptotic notation with respect to the size of mesh implying the run time of operator as a function of mesh size, which should be prohibited for the *efficient* data structure.

The 22 basic mesh operators that are frequently used in support of automated adaptive analysis are the following.

Data : x, y, z coordinates
Result: Create a vertex V
begin
CREATE a vertex M_i^0; /* *1 step* */
PUT({M{M^0}}, M_i^0); /* *1 step* */
coordinates ← (x,y,z); /* *1 step* */
RETURN M_i^0; /* *1 step* */
end

/* Time= *1+1+1+1 = 4* */

Algorithm 2.1: M_createV(x, y, z)

1. V_edges(M_i^0): for a given vertex M_i^0, return $\{M_i^0\{M^1\}\}$.
2. V_faces(M_i^0): for a given vertex M_i^0, return $\{M_i^0\{M^2\}\}$.
3. V_regions(M_i^0): for a given vertex M_i^0, return $\{M_i^0\{M^3\}\}$.
4. E_vertices(M_i^1): for a given edge M_i^1, return $\{M_i^1\{M^0\}\}$.
5. E_faces(M_i^1): for a given edge M_i^1, return $\{M_i^1\{M^2\}\}$.
6. E_regions(M_i^1): for a given edge M_i^1, return $\{M_i^1\{M^3\}\}$.
7. F_vertices(M_i^2): for a given face M_i^2, return $\{M_i^2\{M^0\}\}$.
8. F_edges(M_i^2): for a given face M_i^2, return $\{M_i^2\{M^1\}\}$.
9. F_regions(M_i^2): for a given face M_i^2, return $\{M_i^2\{M^3\}\}$.
10. R_vertices(M_i^3): for a given region M_i^3, return $\{M_i^3\{M^0\}\}$.
11. R_edges(M_i^3): for a given region M_i^3, return $\{M_i^3\{M^1\}\}$.
12. R_faces(M_i^3): for a given region M_i^3, return $\{M_i^3\{M^2\}\}$.
13. E_exist(M_i^0, M_j^0): return an edge bounded by M_i^0 and M_j^0 if exists.
14. F_exist(M_i^0, M_j^0, M_k^0): return a face bounded by M_i^0, M_j^0, and M_k^0 if exists.
15. F_exist(M_i^1, M_j^1, M_k^1): return a face bounded by M_i^1, M_j^1, and M_k^1 if exists.
16. R_exist(M_i^0, M_j^0, M_k^0, M_l^0): return a region bounded by M_i^0, M_j^0, M_k^0 and M_l^0 if exists.
17. M_createV(x, y, z): create a vertex with given coordinates x, y, z. (Algorithm 2.1).
18. M_createE(M_i^0, M_j^0): create an edge bounded by given 2 vertices.
19. M_createF(M_i^0, M_j^0, M_k^0): create a face bounded by given 3 vertices.
20. M_createF(M_i^1, M_j^1, M_k^1): create a face bounded by given 3 edges.

Table 2.5: Run time of 22 mesh operators of four complete representations

operator	greedy	circular	one level	complete MSR
V_edges	14	584 (B.1)	14	14,362 (D.1)
V_faces	35	1871 (B.2)	548 (C.1)	14,291 (D.2)
V_regions	23	23	1020 (C.3)	23
E_vertices	2	2	2	2
E_faces	5	941 (B.3)	5	49,372 (D.3)
E_regions	5	455 (B.4)	53 (C.5)	27,142 (D.4)
F_vertices	3	10 (B.5)	10 (B.5)	3
F_edges	3	3	3	291 (D.5)
F_regions	2	235 (B.6)	2	40,784 (D.6)
R_vertices	4	17 (B.7)	17 (B.7)	4
R_edges	6	9 (B.8)	9 (B.8)	582 (D.7)
R_faces	4	4	4	584 (D.8)
E_exist(M_i^0,M_j^0)	71 (A.1)	641 (B.9)	71 (A.1)	71 (A.1)
F_exist(M_i^0,M_j^0,M_k^0)	106 (A.2)	1647 (B.10)	140 (C.7)	106 (A.2)
F_exist(M_i^1,M_j^1,M_k^1)	47 (A.3)	982 (B.11)	47 (A.3)	117 (D.9)
R_exist(M_i^0,M_j^0,M_k^0,M_l^0)	125 (A.4)	766 (B.12)	544 (C.8)	125 (A.4)
M_createV	4	4	4	4
M_createE	16 (A.5)	5 (B.13)	16 (A.5)	6 (A.5)
M_createF(M_i^0,M_j^0,M_k^0)	256 (A.6)	1929 (B.14)	231 (C.9)	27 (D.10)
M_createF(M_i^1,M_j^1,M_k^1)	50 (A.7)	9 (B.15)	21 (C.10)	33 (D.11)
M_createR(M_i^0,M_j^0,M_k^0,M_l^0)	703 (A.8)	6627 (B.16)	579 (C.11)	33 (D.12)
M_createR(M_i^2,M_j^2,M_k^2,M_l^2)	201 (A.9)	7 (B.17)	19 (C.12)	46 (D.13)
TOTAL	1,685	16,771	3,359	148,008

21. M_createR(M_i^0, M_j^0, M_k^0, M_k^0): create a region bounded by given 4 vertices.

22. M_createR(M_i^2, M_j^2, M_k^2, M_k^2): create a region bounded by given 4 faces.

The run time of vertex creation operator, *M_createV* (Algorithm 2.1), is 4 regardless of the mesh representation options. For the comparison purpose, we analyze various mesh representation options in terms of the run time of 22 mesh operators in steps only with a tetrahedral mesh. The run time of 22 mesh operators in 4 complete mesh representations are summarized in Table 2.5.

If M_i^d stores n entities of dimension q in $\{M_i^d\{M^q\}\}$ explicitly, the run time of operator inquiring $\{M_i^d\{M^q\}\}$ is n steps. Thus, for the adjacency operator on directly stored adjacencies, their run time is obviously the number of the adjacencies stored. For example, *E_regions*(M_i^1) in complete mesh representation takes 23 steps due to direct storage access. For the operators of which algorithm is not straightforward, their algorithms are described in Appendix A (greedy representation), B (circular representation),

C (one-level representation) and D (complete MSR). In case of the complete MSR, the cost of adjacency operators requesting interior levels (for example, V_edges, V_faces, E_faces, F_edges, R_edges, and R_faces) is computed based on implicit entities not stored in the representation.

2.5 Historic Review

The particular mesh-based application has its own need of the level of entities and adjacencies. It is not possible to design a single mesh data structure that is the most compact and the most efficient, and meets the needs of all applications. Clearly, the design of good mesh data structure is highly dependent on the application for which it is used. The usual approach in designing a mesh data structure is to tailor the mesh data structure to the specific application [10, 16, 36, 39, 52, 53, 54, 61, 69, 71, 90]. For example. the mesh data structure of references [52] and [53] stores only a minimal representation consisting of elements (regions in 3D) and nodes (or vertices). In a 3D mesh, GRUMMP [65] stores vertices, faces, and regions. Reference [9] stores vertices, edges and regions.

However, a mesh data structure tailored to the specific application gains little acceptance to be used in most mesh-based applications due to lack of extendibility and reusability. The *general* topology-based mesh data structure works with reasonable efficiency in most operations of the applications [6, 13, 20, 66, 83, 89, 94, 100].

Waltz [100] presented a general mesh data structure with an assumption that the mesh consists of a set of nodes, elements (regions in 3D), and a set of nodes on the boundary of the domain. The other mesh information is described as derived data structures. In a 3D mesh, for each element, the data structure stores a set of faces bounding the element and a set of the neighboring elements adjacent to any of the faces of the element. An empty set of neighboring elements is used to recognize a face on the boundary on the domain.

Celes et al. [13] took an approach that has modified reduced representation to provide memory efficiency while ensuring consistency of implicit entities by having oriented topological entities. The oriented topological entities represent the use of face, edge, or a vertex by an element without imposing any additional storage by representing them implicitly. It presented a compact general mesh data structure with reduced but complete representation with augmented data structures and additional computations.

Based on the work in [5, 6, 20, 66, 94], MeshSim^{TM} [89] were presented to effi-

ciently support tools and techniques for geometry-based analysis. MeshSim^{TM} was extended to PMeshSim^{TM} to be operable on parallel computers. For the memory efficiency, MeshSim^{TM} and PMeshSim^{TM} allow the user to choose between full vs. reduced representation through the different mesh data stored in the mesh file. The mesh without curving can be loaded with the reduced representation from the mesh file, which is generated with the reduced representation.

Although the general mesh data structure works with reasonable efficiency in most operations of the applications, the analysis of mesh representation options given in §2.4 clearly demonstrates a few caveats of the general mesh database with fixed mesh representations such as a prohibitive cost of storage space in case of full representations and ensuring the efficiency and consistency of implicit entities in case of reduced representations. In case of the applications that need only elements and nodes, for example, Lagrangian finite elements, Hierarchic finite elements, Discontinuous Monomials finite elements or Infinite finite elements, smoothing, etc., maintaining complete representations results in wasting resources [31, 75]. This inevitably leads to the consideration that the mesh data structure must be flexible enough to switch between various representations for different phase of an application and build the custom mesh representation based on a meaningful criteria for achieving efficiency both in the storage and computational cost by being optimized for a particular application [32, 75, 92].

The AOMD [75] supports the flexibility in the mesh representation stored by building the requested representation on the fly using a mesh operator that creates or deletes entities and adjacencies of a specific level through mesh traversal and search, which is obviously not efficient.

MOAB [92] also provides the flexibility in the mesh representation, but in a different manner from the AOMD. The flexible mesh representation of MOAB is controlled not by the mesh database, but by the mesh entities and adjacencies stored in the mesh file.

Based on investigation of the efficiency of mesh representation options in reference [31], Garimella presented a flexible mesh infrastructure named MSTK [32]. MSTK allows the user to switch the representation only from a set of mesh representation options available to choose. It also provides flexibility in specifying geometric classification information, allowing the user to specify if the geometric classification is stored or not.

Even though all of AOMD, MOAB and MSTK are classified into flexible mesh databases, the definition of *flexibility* is different from each other. In terms of AOMD,

a flexible mesh database is the one that is able to change its mesh data stored in the representation dynamically based on the user's request. In case of MOAB, the flexibility means an ability to load (resp. save) the mesh with different mesh representations from (resp. to) the mesh file. The flexibility of mesh representations with MSTK is limited to a predefined set of mesh representations available in the MSTK. The FMDB defines *a flexible mesh database* as the one that is able to shape its mesh data structure, both of *data and operators*, dynamically based on the user's representation needs to maintain the mesh data correct even with mesh modification, for example, entity creation/deletion and migration between processors in parallel.

A review of distributed mesh data structures, such as libMesh [52], PMDB [20, 66], Selwood [83], etc., is presented in §4.1.

CHAPTER 3
FLEXIBLE MESH DATA STRUCTURE

This chapter discusses the design of the flexibility in the FMDB which enables the mesh database to shape its structure based on the representational needs.

The issues to be considered for the flexible mesh data structure are:

- how to describe the mesh representation needs,
- how to provide the user-requested representation at a minimum cost,
- how to keep the user-requested representation correct even with mesh modification such as mesh entity creation/deletion and mesh migration in parallel, and
- how to proceed when the user doesn't know the necessary mesh representation for the application.

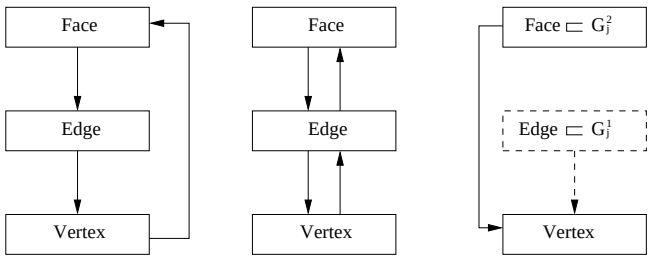
To represent mesh representation in need, we devised mesh representation matrix, shortly called MRM (§3.1). The MRM provided by the user is then optimized to provide the needed representation at a minimum cost (§3.2). In order to keep the user-requested representation correct even with mesh modification, the mesh entity creation/deletion operators are declared as function pointers, and initially undetermined (§3.2). Once the user-requested representation is provided, the mesh entity creation/deletion operators are set to the proper ones dynamically to maintain the needed representation. For the cases when the needed mesh representation is not known in advance, the Dynamic Mesh Usage Monitor (DMUM) was developed (§3.3). DMUM collects mesh usage statistics in terms of the levels of entities and adjacencies needed by the application and provides the information for use in setting the appropriate representation in the FMDB.

3.1 Mesh Representation Matrix

The user requested mesh representation is provided to the mesh database in the form of a 4×4 matrix, called Mesh Representation Matrix (MRM). The matrix used in reference [75] to describe a mesh representation has been extended to be able to represent the equally classified mesh entities and adjacencies available only for the stored entities.

Definition 5.1 *Mesh Representation Matrix (MRM)*

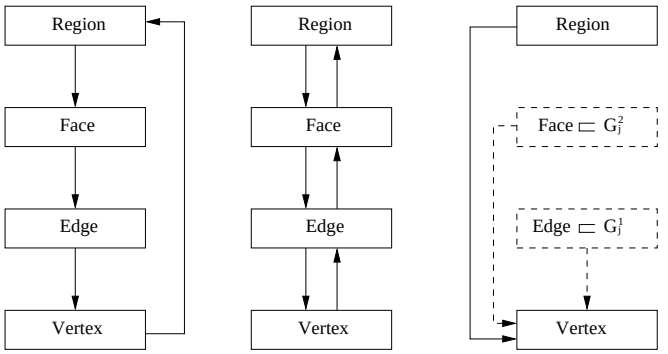
A 4×4 matrix $\mathcal{R}$ of which diagonal element $\mathcal{R}_{i,i}$ is equal to 1 if mesh entities of



$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 & 0 \\ - & - & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

circular one level MSR

Figure 3.1: MRM's of 2D mesh representation



$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 & 0 \\ - & - & 0 & 0 \\ - & 0 & - & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

circular one level MSR

Figure 3.2: MRM's of 3D mesh representation

dimension i are present in the representation, is equal to $-$ if only entities of the same order as the geometric model entities they are classified on are stored, and is equal to 0 if not stored. Non-diagonal element $\mathcal{R}_{i,j}$ of $\mathcal{R}$ is equal to 1 if $\mathcal{R}_{i,i} = \mathcal{R}_{j,j} = 1$ and $\{M^i\{M^j\}\}$ is present, is equal to $-$ if $\{M^i\{M^j\}\}$ is present only for stored $\{M\{M^i\}\}$ and $\{M\{M^j\}\}$, and is equal to 0 if the adjacency is not stored at all. $i \neq j$ and $0 \leq i, j \leq 3$.

Figure 3.1 and 3.2 give, respectively, 2D and 3D MRM's of the 3 representations; circular, one-level and minimum sufficient representation. In the adjacency graph of 3D

MSR given in Figure 3.2, $\mathcal{R}_{0,0}$, $\mathcal{R}_{3,3}$, and $\mathcal{R}_{3,0}$ are 1 since all the vertices, regions and adjacency $\{M^3\{M^0\}\}$ are present in the representation. $\mathcal{R}_{1,1}$ and $\mathcal{R}_{2,2}$ are $-$ due to only edges classified on model edges and faces classified on model faces are present. $\mathcal{R}_{1,0}$ and $\mathcal{R}_{2,0}$ are $-$ since the downward adjacencies $\{M^1\{M^0\}\}$ and $\{M^2\{M^0\}\}$ are stored only for present edges and faces. The remaining $\mathcal{R}_{i,j}$, $i \neq j$, is 0.

3.2 Design of a Flexible Mesh Data Structure

Based on the three hypotheses of the AOMD (§2.3), this section presents a new method to design a flexible mesh data structure that efficiently supports adaptive analysis. The requirements of the flexible mesh data structure are:

- The user-requested representation should be properly maintained even with mesh modification such as entity creation/deletion and mesh migration.
- Restoration of implicit entities should produce valid entities in every aspect such as geometrical classification and vertex ordering.
- Any mesh operators, except mesh loading/exporting and query to unrequested adjacencies, should be effective without involving global mesh level search or traversal to ensure efficiency and scalability in parallel.

To meet the requirements, the mesh database is designed to shape its data structure dynamically by setting mesh modification operators to the proper ones that keep the requested representation correct. Shaping mesh data structure is performed in three steps:

3.2.1 Step 1: Union the user-requested representation with the minimum sufficient representation

First, union of the user-requested representation with the MSR is performed since the MSR is the minimal representation to be stored in the mesh with no information loss. For two MRM's, R^1 and R^2, the union operation is performed on each pair of $R^1_{i,j}$ and $R^2_{i,j}$, $i,j = 0,1,2,3$, where union of $R^1_{i,j}$ and $R^2_{i,j}$ returns the maximum of $R^1_{i,j}$ and $R^2_{i,j}$, $1 > - > 0$.

Figure 3.3 depicts examples of 3D MRM union. By union, $\mathcal{R}^a$, $\mathcal{R}^d$, and $\mathcal{R}^g$ are, respectively, modified to $\mathcal{R}^b$, $\mathcal{R}^e$ and $\mathcal{R}^h$. In case of $\mathcal{R}^d$, $\mathcal{R}^d_{1,1}$ and $\mathcal{R}^d_{1,0}$ are set to $-$ to store edges classified on model edges with their bounding vertices. $\mathcal{R}^d_{3,0}$ and $\mathcal{R}^d_{2,0}$ are, respectively, set to 1 and $-$ since regions and faces are defined in terms of vertices in the

$$
\begin{array}{lll}
 & \text{user representation} & \text{after union} \\
Case\ 1: & \mathcal{R}^a = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} & \mathcal{R}^b = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & - & 0 & 0 \\ - & 0 & - & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \\
Case\ 2: & \mathcal{R}^d = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} & \mathcal{R}^e = \begin{bmatrix} 1 & 0 & 1 & 1 \\ - & - & 0 & 0 \\ - & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix} \\
Case\ 3: & \mathcal{R}^g = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} & \mathcal{R}^h = \begin{bmatrix} 1 & 0 & 0 & 0 \\ - & 1 & 1 & 0 \\ - & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}
\end{array}
$$

Figure 3.3: Example of 3D MRM union

MSR. In case of $\mathcal{R}^g$, $\mathcal{R}^g_{0,0}$ and $\mathcal{R}^g_{3,3}$ are set to 1 to store vertices and regions. $\mathcal{R}^g_{1,0}$ and $\mathcal{R}^g_{2,0}$ are set to $-$ and $\mathcal{R}^g_{3,0}$ is set to 1 to store adjacent vertices of edges, faces and regions.

3.2.2 Step 2: Optimize the representation

The second step optimizes the MRM in order to provide the optimal representation that satisfies the user requested representation at the minimum memory cost. Optimizing the MRM is different depending on the dimension of a mesh. Optimization is performed in multiple sub-steps as the following:

2.1 *Correct MRM*

The first sub-step corrects the MRM to be consistent in terms of entity existence and adjacency request. If $\mathcal{R}_{i,j} = 1$ but any of $\mathcal{R}_{i,i}$ and $\mathcal{R}_{j,j}$ is not 1, $\mathcal{R}_{i,j}$ is corrected to $-$. If $\mathcal{R}_{i,j} = -$ and both $\mathcal{R}_{i,i}$ and $\mathcal{R}_{j,j}$ are 1, $\mathcal{R}_{i,j}$ is corrected to 1.

2.2 *Determine the level of bounding entities*

By the hypotheses of the AOMD, a face can be created by vertices or edges, and a region can be created with vertices, edges or faces. However, to maintain the needed adjacencies efficiently, it is desirable to determine the level of bounding entities for face and region definition, and create face and region only with pre-determined level of entities. For example, for a representation that requires adjacencies $\{M^2\{M^3\}\}$ and $\{M^3\{M^2\}\}$, creating a region with faces is more effective than creating a region with vertices in terms of updating adjacencies between regions and faces. Thus

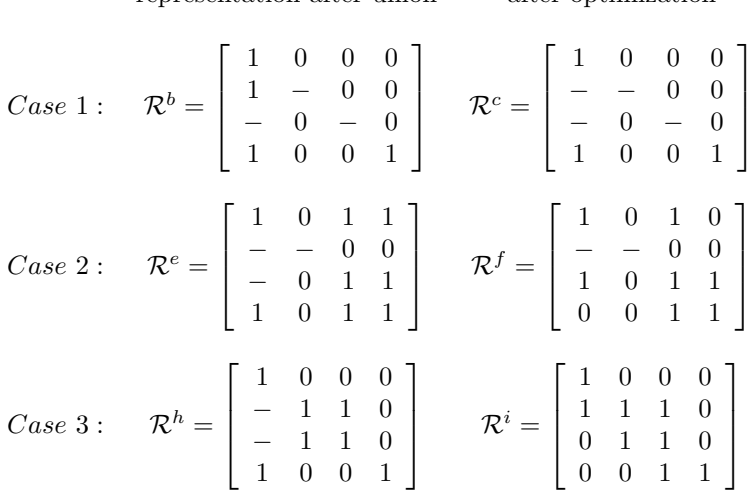


Figure 3.4: Example of 3D MRM optimization

the second step determines the level of bounding entities in face/region creation to expedite the adjacency update.

Note restricting the lower level of entities for face/region creation doesn't necessarily mean that creating face/region with other lower level of entities is not supported. It does mean creating a face/region with a non-preferred level of entities will involve more effort to update desired adjacencies.

2.3 *Suppress unnecessary adjacencies*

The third step removes unnecessary adjacencies which are effectively obtainable by local traversal to save the storage. For instance, consider $\mathcal{R}_{1,2}$, $\mathcal{R}_{1,3}$ and $\mathcal{R}_{2,3}$ are equal to 1. Then $\mathcal{R}_{1,3}$ is suppressed to 0 since $\{M^1\{M^3\}\}$ can be effectively obtained by traversing $\{M^1\{M^2\}\}$ and $\{M^2\{M^3\}\}$. This step can be turned off by the user in case that the user doesn't want local traversal for specific adjacency queries.

Figure 3.4 depicts examples of 3D MRM optimization. By optimization, $\mathcal{R}^b_{1,0}$ is corrected to $-$ since $\mathcal{R}^b_{1,1}$ is not 1. $\mathcal{R}^e_{2,0}$ is corrected to 1 since both $\mathcal{R}^e_{0,0}$ and $\mathcal{R}^e_{2,2}$ are 1. $\mathcal{R}^e_{0,3}$ and $\mathcal{R}^e_{3,0}$ are set to 0 since they are obtainable, respectively, by $\{M^0\{M^2\}\{M^3\}\}$ and $\{M^3\{M^2\}\{M^0\}\}$. In case of $\mathcal{R}^h$, first, $\mathcal{R}^h_{1,0}$ and $\mathcal{R}^h_{2,0}$ are corrected to 1 since all $\mathcal{R}^h_{i,i}$, $i = 0, 1, 2$, are 1. Then, $\mathcal{R}^h_{2,0}$ and $\mathcal{R}^h_{3,0}$ are set to 0, and $\mathcal{R}^h_{3,2}$ is set to 1. Regions and faces with $\mathcal{R}^c$ are determined to create with vertices. Regions with $\mathcal{R}^f$ and $\mathcal{R}^i$ are determined to create with faces. Faces with $\mathcal{R}^f$ (resp. $\mathcal{R}^i$) are determined to create with vertices (resp. edges).

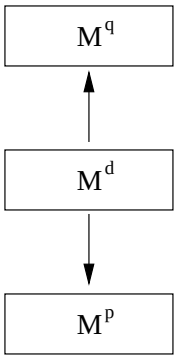


Figure 3.5: Example of adjacency needs

3.2.3 Step 3: Shape mesh data structure via setting mesh operators

This step shapes the mesh data structure based on the mesh representation.

To keep the user-requested adjacency even with mesh modification efficient and correct, the needed adjacencies should be updated on the fly at the moment when the mesh entities are created or deleted. For example, suppose an application that requests adjacency $\{M^0\{M^2\}\}$. In order to keep $\{M^0\{M^2\}\}$, face creation must be followed by adding M_i^2 into $\{M_i^0\{M^2\}\}$, and face deletion must be followed by deleting M_i^2 from $\{M_i^0\{M^2\}\}$, for each $M_i^0 \in \{\partial(M_i^2)\}$. Obviously, by doing this, the representation is kept correct even with mesh modification.

In the new approach, we shape the mesh data structure by setting the mesh operators that create or delete the mesh entities to the proper ones in order to preserve the user-requested representation. Unlike the previous method taken in reference [75], shaping the representations using dynamic setting of mesh operators doesn't involve any mesh size operation such as search and traversal, and maintains a valid representation under all mesh level operations. The operators which involve mesh entity creation/deletion during operation must work differently depending on the representation. These operators are mesh entity creation, mesh entity deletion, mesh load, and mesh migration. The mesh migration procedure with flexible mesh representations will be covered in Chapter 5.

Consider the adjacency request given in Figure 3.5 where $\{M^d\{M^p\}\}$ and $\{M^d\{M^q\}\}$ are requested, $p < d < q$. The following are the rules for determining mesh entity creation/deletion operators which are declared as function pointers:

1. when M_i^d is created, $\{M_i^d\{M^p\}\}$ is stored for each $M_i^p \in \{\partial(M_i^d)\}$.

2. when M_i^q is created, $\{M_i^d\{M^q\}\}$ is stored for each $M_i^d \in \{\partial(M_i^q)\}$.

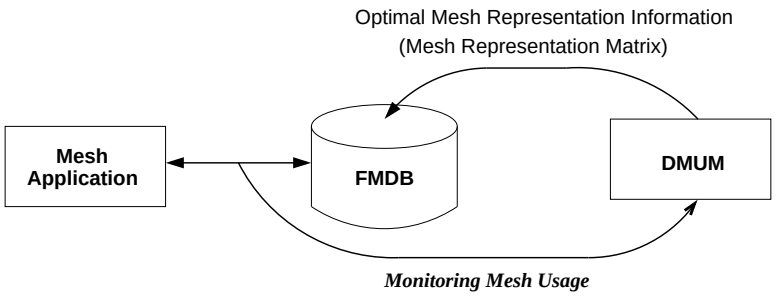


Figure 3.6: The relationship between mesh applications, FMDB and DMUM

3. when M_i^d is deleted, $\{M_i^d\{M^p\}\}$ doesn't need to be explicitly updated.

4. when M_i^q is deleted, $\{M_i^d\{M^q\}\}$ is updated for each $M_i^d \in \{\partial(M_i^q)\}$ to remove M_i^q.

Rule 1 means that when M_i^d is created, M_i^p is added to the downward adjacency $\{M_i^d\{M^p\}\}$ for each $M_i^p \in \{\partial(V_i^d)\}$. Rule 2 means that when M_i^q is created, M_i^q is added to the upward adjacency $\{M_i^d\{M^q\}\}$ for each $M_i^d \in \{\partial(M_i^q)\}$. In the object-oriented paradigm where a mesh entity stores its adjacency information as the member data of the entity [5, 24, 82], the downward adjacency $\{M_i^d\{M^p\}\}$ is removed automatically when M_i^d is deleted. Thus, Rule 3 means that when M_i^d is deleted, the downward adjacencies of M_i^d don't need to be removed explicitly. However, when M_i^q is deleted, M_i^q is not deleted from the upward adjacency of $\{M_i^d\{M^q\}\}$ stored in M_i^d for each $M_i^d \in \{\partial(M_i^q)\}$. Rule 4 means, when M_i^q is removed, M_i^q should be removed explicitly from all the stored upward adjacency $\{M_i^d\{M^q\}\}$ for each $M_i^d \in \{\partial(M_i^q)\}$.

3.3 The Dynamic Mesh Usage Monitor (DMUM)

It is not always possible to know the user requested representation of the application in advance. Sometimes, the exact needs of the mesh representation is not known without running the application. For instance, when the application manipulates the mesh through the DOE SciDAC [81] Terascale Simulation Tools and Technologies (TSTT)'s mesh interface [95], which enables the application to use several different mesh databases via language-interoperability provided by SIDL-Babel [14], it's not possible to provide the needed representation to the mesh database. For those cases, the Dynamic Mesh Usage Monitor (DMUM) was developed to provide the needed mesh representation of the specific application to the FMDB. The needed mesh representation can be provided either by the user explicitly or the output of the DMUM.

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

(a) 2D Rayleigh Taylor instabilities

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

(b) 2D riemann

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

(c) 3D riemann

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

(d) 3D sedov explosion

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

(e) 3D mesh adaptation

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

(f) 3D error estimation

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

(g) 2D coordinate-based load balancing

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

(h) 3D graph-based load balancing

Figure 3.7: Example of MRM's generated by DMUM

Figure 3.6 illustrates the relationship between a mesh application, the FMDB and the DMUM. The DMUM monitors mesh usage of the application in terms of the levels of entities and adjacencies used and produces the needed representation in the form of the MRM. The output of the DMUM is used by the FMDB to shape its data structure to the optimal that meets the specific needs of the application.

The operation of the DMUM is performed with the following main functions:

- `startMonitoring`(M): This function starts the dynamic monitoring. The monitoring module starts to collect the information about the levels of mesh entities and adjacencies manipulated by the application.
- `pauseMonitoring`(M): This function pauses monitoring. With `resumeMonitoring`, it enables the partial monitoring.
- `resumeMonitoring`(M): This function resumes monitoring.
- `stopMonitoring`(M): This function stops the dynamic monitoring.

Figure 3.7 shows examples of mesh representation matrix generated by the DMUM with various DG [30, 74, 77, 78] applications (a-d), 3D mesh adaptation [48, 49, 50, 51] (e), 3D mesh error estimation with ZZ-SPR [104] (f) and the mesh load balancing [20] procedure with manifold model (g, h). The 3D mesh adaptation procedure requires all d levels of entities with 12 adjacencies.

CHAPTER 4
PARALLEL MESH DATA STRUCTURE: A PARTITION MODEL

This chapter presents the historic review of distributed mesh data structures published to data and discusses the design of distributed mesh data structure in the FMDB with an assumption of full, complete mesh representation. Finally, the algorithm for migrating mesh entities between partitions is presented.

4.1 Historic Review

A distributed mesh data structure is an infrastructure executing underneath providing all parallel mesh-based operations needed to support parallel adaptive analysis. An efficient and scalable distributed mesh data structure is mandatory to achieve performance since it strongly influences the overall performance of adaptive mesh-based simulations. In addition to the general mesh-based operations [6], such as mesh entity creation/deletion, adjacency and geometric classification, iterators, arbitrary attachable data to mesh entities, etc., the distributed mesh data structure must support (*i*) efficient communication between entities duplicated over multiple processors, (*ii*) migration of mesh entities between processors, and (*iii*) dynamic load balancing.

Papers have been published on the issues of parallel adaptive analysis including parallel mesh generation [22, 23, 44, 45, 65, 79, 96], dynamic mesh load balancing techniques [17, 20, 25, 94, 99], and data structure and algorithms for parallel structured [4, 42, 57, 69] or unstructured mesh adaptation [21, 52, 65, 64, 66, 70, 76, 83].

Parashar and Browne presented a distributed mesh data structure for parallel non-conforming *h*-refinement called DAGH (Distributed Adaptive Grid Hierarchy) [69]. DAGH represents a mesh with grid hierarchy. In case of a distributed grid, inter-grid operations are performed locally on each processor without involving any communication or synchronization due to the mesh refinement is non-conforming. The mesh load balancing is performed by varying granularity of the DAGH blocks.

LibMesh [52] is a distributed mesh data structure developed at the university of Texas in order to support parallel finite element simulations with refinement. It opted the classic element-node data structure supporting only *h*- uniform refinement and serial mesh partitioning for initial distribution.

Reference [9] presented a distributed mesh data structure to support parallel adap-

tive numerical computation based on refinement and coarsening [64]. A mesh data consists of vertices, edges and regions with a linked list data structure and maintains the shared processor lists for entities on partition boundaries through the message passing. Global identifiers are assigned to every entity, thus, all data structure are updated to contain consistent global information during adaptation. The provided the owning processor of shared entities which is randomly selected and the dynamic mesh load balancing with ParMETIS [40].

In reference [83], Selwood and Berzins presented a general distributed mesh data structure that supports parallel mesh refinement and de-refinement. It represents a mesh with all d level mesh entities and adjacencies, and provides dynamic load balancing with the Zoltan [80] library. In order to be aware of the part of the mesh which is distributed on other processors, the pointers to the adjacent tetrahedral that are on other processors are kept for each processor.

Reference [20, 66] presented a general distributed mesh data structure called PMDB (Parallel Mesh DataBase), which was capable of supporting parallel adaptive simulations. In PMDB, the data related to mesh partitioning were kept at the mesh entity level and the inter-processor links were managed by doubly-linked structures. These structures provided query routines such as processor adjacency, lists of entities on partition boundaries, and update operators such as insertion and deletion of these entities. An owning partition update rule which lets the processor owning a shared entity on partition boundary to collect and inform the updated links to the processors holding these entities was presented. The owning processor of an entity on the partition boundary was determined to the processor with minimum processor id. In reference [94], PMDB was enhanced with addition of RPM (Rensselaer Partition Model) that represents heterogeneous processor and network of workstations, or some combination of these for the purpose of improving performance by accounting for resources of parallel computers.

Most of distributed mesh data structures published to date are data structure shaped to specific mesh applications [9, 52, 65, 69], support only part of adaptive analysis such as refinement step [4, 52, 57, 69, 70], or is able to handle only manifold meshes [9, 65, 52, 57, 69, 70, 76, 83, 89, 94]. The development of the *general* distributed mesh data structure to efficiently support parallel adaptive analysis procedures including the solvers and the adaptation procedures is not trivial due to data structure complexity, the nature of the mesh with general non-manifold models, the consistantly evolving nature in the mesh as

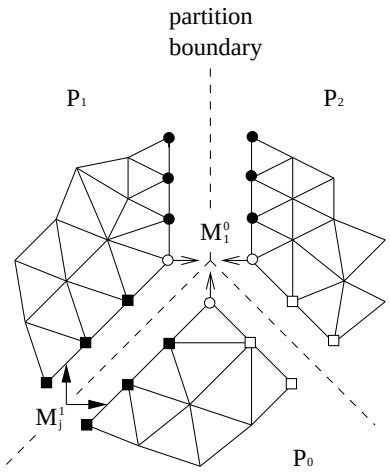


Figure 4.1: Distributed mesh on three partitions P_0, P_1 and P_2 [76]

it is adapted, and the needs for dynamic load balancing.

4.2 Distributed Mesh Representation

4.2.1 Definitions and properties

A distributed mesh is a mesh divided into partitions for distribution over a set of processors for specific reasons, for example, parallel computation.

Definition 4.1 *Partition*

A partition P_i consists of the set of mesh entities assigned to i^{th} processor. For each partition, a unique partition id can be given.

Each partition will be treated as a serial mesh with the addition of mesh partition boundaries to describe groups of mesh entities that are on inter-partition boundaries. Mesh entities on partition boundaries are duplicated on all partitions on which they are used in adjacency relations. Mesh entities not on the partition boundary exist on only one partition. Figure 4.1 depicts a mesh that is distributed on 3 partitions. Vertex M_1^0 is common to three partitions and on each partition, several mesh edges like M_j^1 are common to two partitions. The dashed lines are *partition boundaries* that consist of mesh vertices and edges duplicated on multiple partitions.

In order to simply denote the partition(s) that a mesh entity resides, we define an operator $\mathscr{P}$.

Definition 4.2 *Residence partition operator* $\mathscr{P}[M_i^d]$

An operator that returns a set of partition id(s) where M_i^d exists.

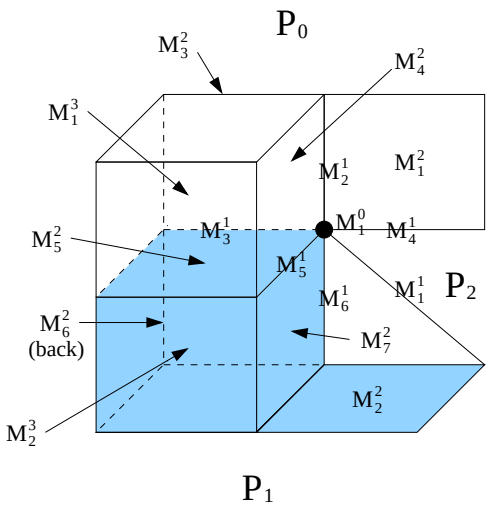


Figure 4.2: Example 3D mesh distributed on 3 partitions

Definition 4.3 *Residence partition equation of* M_i^d

If $\{M_i^d\{M^q\}\} = \emptyset$, $d < q$, $\mathscr{P}[M_i^d] = \{p\}$ where p is the id of a partition where M_i^d exists. Otherwise, $\mathscr{P}[M_i^d] = \cup\ \mathscr{P}[M_j^q \mid M_i^d \in \{\partial(M_j^q)\}]$.

For any entity M_i^d not on the boundary of any higher order mesh entities and on partition p, $\mathscr{P}[M_i^d]$ returns $\{p\}$ since when the entity is not on the boundary of any other mesh entities of higher order, its residence partition is determined simply to be the partition where it resides. If entity M_i^d is on the boundary of other higher order mesh entities, M_i^d is duplicated on multiple partitions depending on the residence partitions of its bounding entities since M_i^d exists wherever a mesh entity it bounds exists. Therefore, the residence partition(s) of M_i^d is the union of residence partitions of all entities that it bounds. For a mesh topology where the entities of order $d > 0$ are bounded by entities of order $d-1$, $\mathscr{P}[M_i^d]$ is determined to be {p} if $\{M_i^d\{M_k^{d+1}\}\} = \emptyset$. Otherwise, $\mathscr{P}[M_i^d]$ is $\cup\ \mathscr{P}[M_k^{d+1} \mid M_i^d \in \{\partial(M_k^{d+1})\}]$. For instance, for the 3D mesh depicted in Figure 4.2, where M_1^3 and M_1^2 are on P_0, M_2^3 and M_2^2 are on P_1 and M_1^1 is on P_2, residence partition ids of M_1^0 are $\{P_0, P_1, P_2\}$ since the union of residence partitions of its bounding edges, $\{M_1^1, M_2^1, M_3^1, M_4^1, M_5^1, M_6^1\}$, are $\{P_0, P_1, P_2\}$.

To migrate mesh entities to other partitions, the destination partition id's of mesh entities must be specified before moving the mesh entities. The resident partition equation implies that once the destination partition id of a M_i^d that is not on the boundary of any other mesh entities is set, the other entities needed to migrate are determined by the entities it bounds. Thus, a mesh entity that is not on the boundary of any higher order

mesh entities is the basic unit to assign the destination partition id in the mesh migration procedure.

Definition 4.4 *Partition object*

The basic unit to which a destination partition id is assigned. The full set of partition objects is the set of mesh entities that are not part of the boundary of any higher order mesh entities. In a 3D mesh, 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.

In case of a manifold model, partition objects are all mesh regions in 3D and all mesh faces in 2D. In case of a non-manifold model, the careful lookup for entities not being bounded is required over the entities of one specific dimension. For example, partition objects of the mesh in Figure 4.2 are M_1^1, M_1^2, M_2^2, M_1^3, and M_2^3.

4.2.2 Functional requirements of distributed meshes

Functional requirements of the mesh data structure for supporting mesh operations on distributed meshes are:

- *Communication links*: Mesh entities on the partition boundaries (shortly, partition boundary entities) must be aware of where they are duplicated.

  Definition 4.5 *Remote partition*

  Non-self partition[2] where a mesh entity is duplicated.

  Definition 4.6 *Remote copy*

  The memory location of a mesh entity duplicated on remote partition p.

  In parallel adaptive analysis, the mesh and its partitioning can change thousands of time during the simulation. Therefore, at the mesh functionality level, an efficient mechanism to update the mesh partitioning and keep the links between partitions updated are mandatory to achieve scalability.

- *Entity ownership*: For entities on partition boundaries, it is beneficial to assign a specific copy as the owner of the others and let the owner be in charge of communication or computation between the copies. There are 2 common strategies in determining the owning partition of partition boundary entities.

[2] A partition which is not the current local partition

Figure 4.3: Hierarchy of domain decomposition: geometry model, partition model, and distributed mesh on 4 processors

- *Static entity ownership*: The owning partition of a partition boundary entity is always fixed to P_i regardless of mesh partitioning [66, 94]. It has been observed that the static entity ownership produces mesh load imbalance in adaptive analysis.
- *Dynamic entity ownership*: The owning partition of the partition boundary entity is dynamically specified, for example, reference [9] assigns a random number.

For the dynamic entity ownership, there can be several options in determining owning processor of mesh entities. With the FMDB, the entity ownership is determined based on the rule of *the poor-to-rich ownership*, which assigns the poorest partition to the owner of entity, where the poorest partition is the partition that has the least number of partition objects among residence partitions of the entity. With this scheme, mesh load balance is kept during adaptive mesh simulations since the local mesh migration procedure performed during mesh adaptation to gain the necessary entities for a specific mesh modification operator [1, 21] always migrates entities to poor partitions improving the overall performance of the parallel simulation.

4.3 A Partition Model

To meet the goals and functionalities of distributed meshes, a partition model has been developed between the mesh and the geometric model. As illustrated in Figure 4.3, the partition model can be viewed as a part of hierarchical domain decomposition. Its purpose is to represent mesh partitioning in topology and support mesh-level parallel operations through inter-partition boundary links with ease.

The specific implementation is the parallel extension of the FMDB, such that standard FMDB entities and adjacencies are used on processors only with the addition of the partition entity information needed to support all operations across multiple processors.

4.3.1 Definitions

The partition model introduces a set of topological entities that represents the collections of mesh entities based on their location with respect to the partitioning. Grouping mesh entities to define a partition entity can be done with multiple criteria based on the level of functionalities and needs of distributed meshes.

At a minimum, *residence partition* must be a criterion to be able to support the inter-partition communications. *Connectivity* between entities is also desirable for a criterion to support operations quickly and can be used optionally. Two mesh entities are *connected* if they are on the same partition and reachable via adjacency operations. The connectivity is expensive but useful in representing separate chunks in a partition. It enables diagnoses of the quality of mesh partitioning immediately at the partition model level. In our implementation, for the efficiency purpose, only residence partition is used for the criterion. Definition 4.7 defines the partition model entity based on the residence partition criterion.

Definition 4.7 *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 $\mathscr{P}$. Each partition model entity is uniquely determined by $\mathscr{P}$.

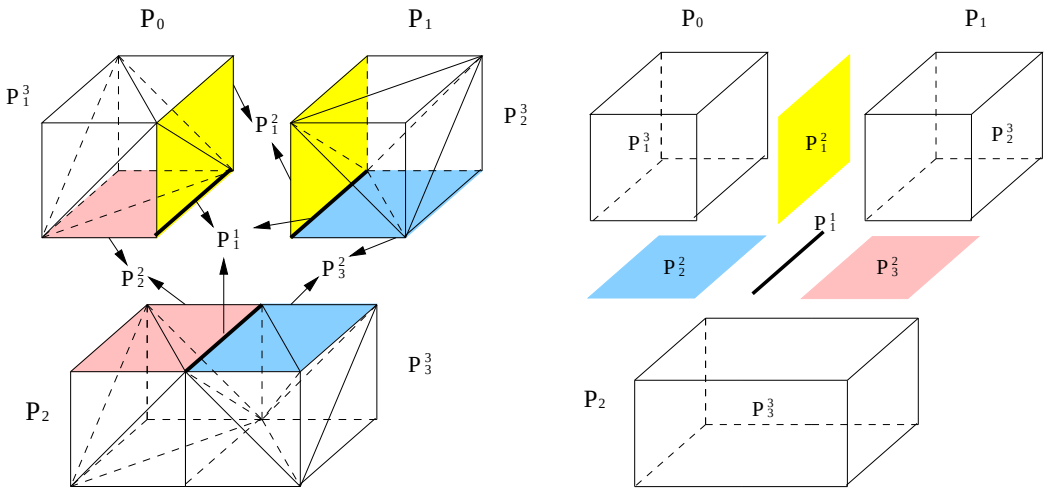
Each partition model entity stores dimension, id, residence partition(s), and the owning partition. From a mesh entity level, by keeping proper relation to the partition model entity, all needed services to represent mesh partitioning and support inter-partition communications are easily supported.

Definition 4.8 *Partition classification*

The unique association of mesh topological entities 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 $M_i^{d_i} \sqsubset P_j^{d_j}$.

Definition 4.9 *Reverse partition classification*

For each partition entity, the set of equal order mesh entities classified on that entity



(a) distributed mesh (b) partition model

Figure 4.4: Distributed mesh and its association with the partition model via partition classifications

defines the reverse partition classification for the partition model entity. The reverse partition classification is denoted as $\mathrm{RC}(P_j^d) = \{M_i^d \mid M_i^d \sqsubset P_j^d\}$.

Figure 4.4 illustrates a 3D distributed mesh with mesh entities labeled with arrows indicating the partition classification of the entities onto the partition model entities and its associated partition model. The mesh vertices and edges on the thick black lines are classified on partition edge P_1^1. The mesh vertices, edges and faces on the shaded planes are classified on the partition faces pointed with each arrow. The remaining mesh entities are non-partition boundary entities, therefore they are classified on the partition regions. Note the reverse classification returns only the same order mesh entities. The reverse partition classification of P_1^1 returns mesh edges located on the thick black lines, and the reverse partition classification of partition face P_i^2 returns mesh faces on the shaded planes.

4.3.2 Building a partition model

When the partition model entities are uniquely defined with the two criteria of residence partition and connectivity between entities, the following rules govern the creation of a corresponding partition model and specify the partition classification of mesh entities:

1. *High-to-low mesh entity traversal*: The partition classification is set from high order to low order entity (resident partition equation).

2. *Inheritance-first*: If $M_i^d \in \{\partial(M_j^q)\}$ and $\mathscr{P}[M_i^d] = \mathscr{P}[M_j^q]$, M_i^d inherits the partition classification from M_j^q as a subset of the partitions it is on.

3. *Connectivity-second*: If M_i^d and M_j^d are connected and $\mathscr{P}[M_i^d] = \mathscr{P}[M_j^d]$, M_i^d and M_j^d are classified on the same partition entity.

4. *Partition entity creation-last*: If neither of rule 2 nor 3 applies for M_i^d, a new partition entity P_j^d is created.

```
Data  : a distributed mesh M
Result: build a partition model and set its association with M
begin
  /* high-to-low order mesh entity traversal */
  for d ← 3 to 0 do
    for each M_i^d ∈ {M{M^d}} do
      /* inheritance-first */
      for each M_j^{d+1} ∈ {M_i^d{M^{d+1}}} do
        if P[M_i^d] = P[M_j^{d+1}]
          set partition classification of M_i^d to P_k^q where M_i^d ⊏ P_k^q;
          GOTO A;
        endif
      endfor
      /* connectivity-second */
      for each M_j^d connected with M_i^d do
        if P[M_i^d] = P[M_j^d]
          set partition classification of M_i^d to P_k^q where M_j^d ⊏ P_k^q;
          GOTO A;
        endif
      endfor
      /* new partition entity creation-last */
      create a new partition entity P_k^d;
      set partition classification of M_i^d to P_k^d;
      A:
    endfor
  endfor
end
```

Algorithm 4.1: M_buildPModel(M)

Rule 2 means if the residence partitions of M_i^d is identical to those of its bounding entity of higher order, M_j^q, it is classified on the partition entity that M_j^q is classified on. For example, in Figure 4.4(a), any mesh faces, edges and vertices that are not on the shaded planes nor on the thick black lines are classified on the partition region by inheriting partition classification from the regions it bounds. Note multiple inheritance

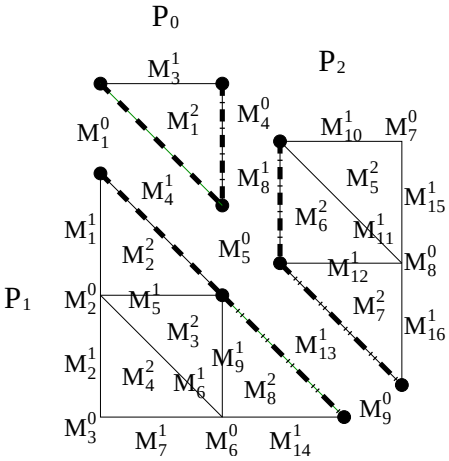
produces unique partition classification. For instance, internal mesh faces on partition P_0 which are not on the shaded planes can inherit partition classification from any of its bounding regions. However, the derived partition classification will always be P_1^3 regardless of the region it was derived from. Rule 3 is applied when Rule 2 is not satisfied. Rule 3 means if residence partitions of M_i^d and M_j^d are the same and they are connected, M_i^d is classified on the same partition entity where M_j^d classified on. When neither Rule 2 nor Rule 3 is satisfied, Rule 4 applies, thus the new partition entity of dimension d is created for the partition classification of entity M_i^d.

Based on the rules, Algorithm 4.1 is the pseudo code that builds a partition model from a distributed mesh. For each mesh entity, M_i^d, where d is decreasing from 3 to 0, it checks if there is any one-level upward adjacent entity, M_j^{d+1}, of which residence partitions are the same as those of M_i^d. If such a M_j^{d+1} exists, it sets the partition classification of M_i^d to that of M_j^{d+1} and proceeds to the next entity. Otherwise, it checks if there is an entity M_j^d which is reachable from M_i^d via adjacencies and of which residence partitions are identical to those of M_i^d. If such a M_j^d is found, it sets the partition classification of M_i^d to that of M_j^d and proceeds to the next entity. Otherwise, it creates a partition model entity, P_k^d, with residence partitions of M_i^d and set the partition classification of M_i^d to P_k^d.

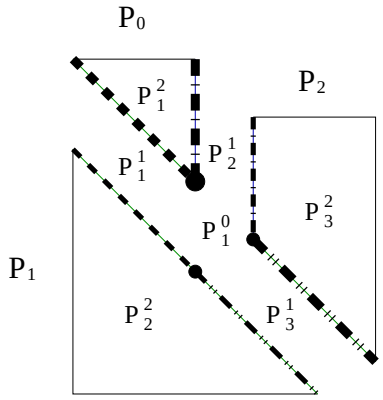
4.4 Algorithm of Mesh Migration with Full Complete Representations

The mesh migration procedure migrates mesh entities from partition to partition. It is performed frequently in parallel adaptive analysis to re-gain mesh load balance, to obtain the mesh entities needed for mesh modification operators or to distribute a mesh into partitions. An efficient mesh migration algorithm with minimum resources (memory and time) and parallel operations designed to maintain the mesh load balance throughout the computation are important factors for high performance in parallel adaptive mesh-based simulations. The mesh migration algorithm described in this section is based on full complete mesh representations. It will be modified to work with the flexible mesh data structure in Chapter 5.

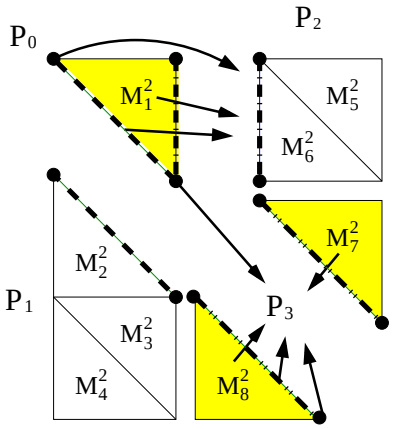
Figure 4.5(a) and (b) illustrate the 2D partitioned mesh and its associated partition model to be used as for the example of mesh migration throughout this section. In Figure 4.5(a), the partition classification of entities on the partition boundaries is denoted with the lines of the same pattern in Figure 4.5(b). For instance, M_1^0 and M_4^1 are classified



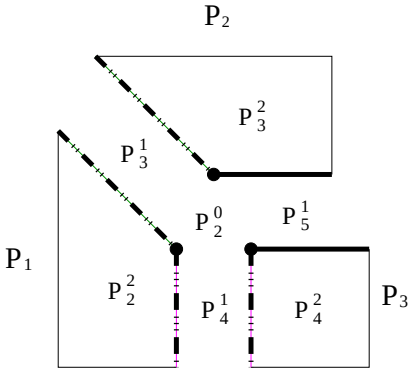
(a) initial mesh



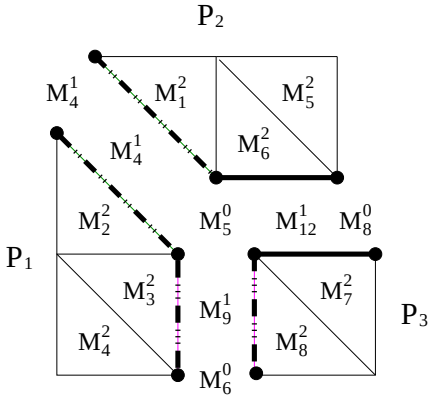
(b) partition model of (a)



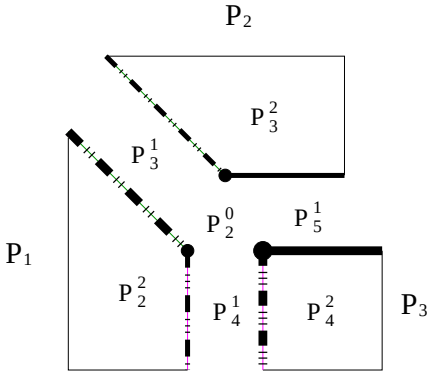
(c) mesh during migration



(d) partition model updated based on the new partition topology



(e) final mesh



(e) final partition model with ownership

Figure 4.5: Example of 2D mesh migration

on P_1^1, and depicted with the dashed lines as P_1^1. In Figure 4.5(b). the owning partition of a partition model edge (resp. vertex) is illustrated with thickness (resp. size) of lines (resp. circles). For example, the owning partition of partition vertex P_1^0 is P_0 since P_0 has the least number of partition objects among 3 residence partitions of P_1^0. Therefore P_1^0 on P_0 is depicted with a bigger-sized circle than P_1^0 on P_1 or P_2 implying that P_0 is the owning partition of P_1^0.

The input of the mesh migration procedure is a list of partition objects to migrate and their destination partition ids, called, for simplicity, $POsToMove$. For the example mesh in Figure 4.5(a), we assume that the input of the mesh migration procedure is $<(M_1^2,2), (M_7^2,3), (M_8^2,3)>$; M_1^2 will migrate to P_2 and M_7^2 and M_8^2 will migrate to P_3. Partition P_3 is currently empty.

The overall procedure for mesh migration is the following:

1. Given the $POsToMove$, collect entities to process and clear partitioning data ($\mathscr{P}$ and partition classification) of them.
2. Determine residence partition(s).
3. Update the partition classification and collect entities to remove.
4. Exchange entities and update remote copies.
5. Remove unnecessary entities.
6. Update the owning partition of partition model entities.

Given list of partition objects to migrate, the first procedure collects the entities of which partitioning-related data (i.e. $\mathscr{P}$, the partition classification, and remote copies) will be updated after migration. After computing $\mathscr{P}$ of the entities, the partition classification is updated to reflect a new updated partition model. The entities to remove from the local partition are determined and collected based on $\mathscr{P}$. After migrating needed entities to their destination partitions, remote copy information of the partition boundary entities are updated and the entities collected to remove are deleted from the local partition. Finally, the owning partition of partition model entities is updated based on the new partitioning topology. Algorithm 4.2 is the pseudo code of the mesh migration procedure.

```
Data  : M, POsToMove
Result: migrate partition objects in POsToMove
begin
    /* STEP 1: collect entities to process and clear partitioning data. See
    §4.4.1 */
    for each M_i^d ∈ POsToMove do
        insert M_i^d into entitiesToUpdate[d];
        reset partition classification and 𝒫;
        for each M_j^q ∈ {∂(M_i^d)} do
            insert M_j^q into entitiesToUpdate[q];
            reset partition classification and 𝒫;
        endfor
    endfor
    /* STEP 2: determine residence partition. See §4.4.2 */
    M_setResidencePartition(POsToMove, entitiesToUpdate[q]);
    /* STEP 3: update partition classification and collect entities to remove.
    See §4.4.3 */
    for d ← 3 to 0 do
        for each M_i^d ∈ entitiesToUpdate[d] do
            determine partition classification;
            if P_local ∉ 𝒫[M_i^d]
                insert M_i^d into entitiesToRemove[d];
            endif
        endfor
    endfor
    /* STEP 4: exchange entities. See §4.4.4 */
    for d ← 0 to 3 do
        M_exchangeEnts(entitiesToUpdate[d]);
    endfor
    /* STEP 5: remove unnecessary entities. See §4.4.5 */
    for d ← 3 to 0 do
        for each M_i^d ∈ entitiesToRemove[d] do
            if M_i^d is on partition boundary
                remove copies of M_i^d on other partitions;
            endif
            remove M_i^d;
        endfor
    endfor
    /* STEP 6: update ownership. See §4.4.6 */
    for each P_i^d in P do
        owning partition of P_i^d ← the poorest partition among 𝒫[P_i^d];
    endfor
end
```

Algorithm 4.2: M_migrate(M, $POsToMove$)

Table 4.1: Contents of vector *entitiesToUpdate* after Step 1

	P_0	P_1	P_2
$entititesToUpdate[0]$	M_1^0, M_4^0, M_5^0	M_1^0, M_5^0, M_6^0, M_9^0	M_4^0, M_5^0, M_8^0, M_9^0
$entititesToUpdate[1]$	M_3^1, M_4^1, M_8^1	M_4^1, M_9^1, M_{13}^1, M_{14}^1	M_8^1, M_{12}^1, M_{13}^1, M_{16}^1
$entititesToUpdate[2]$	M_1^2	M_8^2	M_7^2

4.4.1 Step 1: Preparation

Step 1 collects mesh entities of which internal data needs to be updated based on the new partitioning topology. Internal data to be updated include the partition classification, $\mathscr{P}$, remote partitions and remote copies. The entities collected are:

- $M_i^d \in POsToMove$.
- For each M_i^d, all downward entities $M_j^q \in \{\partial(M_i^d)\}$, $q < d$, and their remote copies on remote partitions.

The mesh migration algorithm works only on those entities. The remaining entities are not affected by migration, thus must not be considered nor touched. The entities collected for the update are maintained in vector *entitiesToUpdate* where *entitiesToUpdate*$[i]$ contains the entities of dimension i, $i = 0, 1, 2, 3$. With a single program multiple data paradigm [67] in parallel, each partition maintains the separate *entitiesToUpdate*$[i]$ with different contents.

For the example 2D mesh given in Figure 4.5(a), the contents of *entitiesToUpdate* is given in Table 4.1. Only entities listed in Table 4.1 will be affected by migration in terms of their location and partitioning-related internal data. *entitiesToUpdate*[2] contains the mesh faces to be migrated from each partition. *entitiesToUpdate*[1] contains the mesh edges which bound any mesh face in *entitiesToUpdate*[2] and their remote copies. *entitiesToUpdate*[0] contains the mesh vertices that bound any mesh edge in *entitiesToUpdate*[1] and their remote copies. The partition classification and $\mathscr{P}$ of entities in *entitiesToUpdate* are cleared for further update.

4.4.2 Step 2: Determine residence partition

Step 2 determines $\mathscr{P}$ of the entities in *entitiesToUpdate*. The pseudo-code is given in Algorithm 4.3. In Step 2.1, according to the resident partition equation, for each pair of partition object and its destination partition id p in $POsToMove$, p is added to $\mathscr{P}$ of the partition object and its all downward entities. Note a non-partition object entity

```
Data  : M, POsToMove
Result: determine 𝒫 of entities in entitiesToUpdate
begin
    /* STEP 2.1: set 𝒫 of entities in entitiesToUpdate through downward
    adjacency of partition objects in POsToMove */
    for each pair (M_i^d, p) ∈ POsToMove do
        𝒫[M_i^d] ← {p};
        for each M_j^q ∈ {∂(M_i^d)} do
            𝒫[M_j^q] ← 𝒫[M_j^q] ∪ {p};
        endfor
    endfor
    /* STEP 2.2: determine if an entity will exist on the local partition after
    migration */
    for d ← 0 to 2 do
        for each M_i^d ∈ entitiesToUpdate[d] do
            if M_i^d is a partition object
                continue; // proceed to the next M_i^d
            endif
            for each M_j^q, M_i^d ∈ {∂(M_j^q)} do
                if M_j^q will stay on P_local
                    𝒫[M_i^d] ← 𝒫[M_i^d] ∪ {P_local};
                    break; // exit for loop
                endif
            endfor
        endfor
    endfor
    /* STEP 2.3: unify 𝒫 of partition boundary entities */
    Do one round of communication to exchange 𝒫 of partition boundary
    entities in entitiesToUpdate;
end
```

Algorithm 4.3: M_setResidencePartition(*POsToMove*, *entitiesToUpdate*)

in *entitiesToUpdate* must exist on the current local partition even after migration if it is on the boundary of other partition object which will not be migrated. For notational simplicity, we denote a local partition of an entity where the entity is currently located P_{local}. Step 2.2 determines if a non-partition object entity will exist on P_{local} after migration by checking if there's any adjacent partition object to stay on P_{local}. One round of communication is performed in Step 2.3 to exchange $\mathscr{P}$ of the partition boundary entities to unify them between remote copies. Table 4.2 gives $\mathscr{P}$ of M_1^0 and M_4^1 on each partition by steps.

Table 4.2: Residence partition(s) of M_1^0 and M_4^1 by steps

step	$\mathscr{P}[M_1^0]@P_0$	$\mathscr{P}[M_1^0]@P_1$	$\mathscr{P}[M_1^0]@P_2$	$\mathscr{P}[M_4^1]@P_0$	$\mathscr{P}[M_4^1]@P_1$
before 2.1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
after 2.1	{2}	$\emptyset$	{3}	{2}	$\emptyset$
after 2.2	{2}	{1}	{3}	{2}	{1}
after 2.3	{1, 2, 3}	{1, 2, 3}	{1, 2, 3}	{1, 2}	{1, 2}

Table 4.3: Contents of vector *entitiesToRemove* after Step 3

	P_0	P_1	P_2
entititesToRemove[0]	M_1^0, M_4^0, M_5^0	M_9^0	M_9^0
entititesToRemove[1]	M_3^1, M_4^1, M_8^1	M_{13}^1, M_{14}^1	M_{13}^1, M_{16}^1
entititesToRemove[2]	M_1^2	M_8^2	M_7^2

4.4.3 Step 3: Determine partition classification and entities to remove

Based on $\mathscr{P}$ computed in Step 2, Step 3 determines the partition classification of the entities in *entitiesToUpdate*. Based on the new partitioning topology after migration, a new partition model is constructed while the partition classification is updated. Figure 4.5(d) is the partition model updated based on the new partition topology.

Based on $\mathscr{P}$, the entities to remove from the local partition are collected and stored in vector *entitiesToRemove*, where *entitiesToRemove*[i] stores entities of dimension i, $i = 0, 1, 2, 3$. An entity is determined to remove from its local partition if $\mathscr{P}$ of the entity doesn't contain the local partition id, P_{local}. Table 4.3 gives the contents of *entitiesToRemove* of each partition.

4.4.4 Step 4: Exchange entities and update remote copies

Since an entity of dimension > 0 is bounded by lower dimension entities, mesh entities are exchanged from low to high dimension. Step 4 exchanges entities from dimension 0 to 3, and creates entities on the destination partitions. Note that the partition classification and $\mathscr{P}$ of the entities were already updated in Step 2 and Step 3 according to the new partitioning topology. However, the remote copies are not up-to-date. The remote copies of the entities created on the destination partitions are updated in this step after exchanging entities. The remote copies of the entities removed from the local partitions are updated in Step 5. Algorithm 4.4 is the pseudo-code that exchanges the entities contained in *entitiesToUpdate*[d].

- Step 4.1 sends the messages to destination partitions to create new mesh entities. Suppose entity M_i^d duplicated on several partitions and need to be migrated to

Data : $entitiesToUpdate[d]$
Result: create entities on the destination partitions and update remote copies
begin
  /* *STEP 4.1: send a message to the destination partitions* */
  for *each* $M_i^d \in entitiesToUpdate[d]$ **do**
    if $P_{local} \neq$ *minimum partition id where* M_i^d *exists*
      continue;
    endif
    for *each partition id* $P_i \in \mathscr{P}[M_i^d]$ **do**
      if M_i^d *exists on partition* P_i *(i.e.* M_i^d *has remote copy of* P_i*)*
        continue;
      endif
      send message A (address of M_i^d on P_{local}, information of M_i^d) to P_i;
    endfor
  endfor
  /* *STEP 4.2: create a new entity and send the new entity information to the broadcaster* */
  while P_i *receives message A (address of* M_i^d *on* P_{bc}*, information of* M_i^d*) from* P_{bc} **do**
    create M_i^d with the information of M_i^d;
    if $M_i^d \neq$ *partition object*
      send message B (address of M_i^d on P_{bc}, address of M_i^d created) to P_{bc};
    endif
  end
  /* *STEP 4.3: the broadcaster sends the new entity information* */
  while P_{bc} *receives message B (address of* M_i^d *on* P_{bc}*, address of* M_i^d *on* P_i*) from* P_i **do**
    $M_i^d \leftarrow$ entity located in the address of M_i^d on P_{bc};
    for *each remote copy of* M_i^d *on remote partition* P_{remote} **do**
      send message C (address of M_i^d on P_{remote}, address of M_i^d on P_i, P_i) to P_{remote};
    endfor
    M_i^d saves the address of M_i^d on P_i as for the remote copy on P_i;
  end
  /* *STEP 4.4: update remote copy information* */
  while P_{remote} *receives message C (address of* M_i^d *on* P_{remote}*, address of* M_i^d *on* P_i*,* P_i*) from* P_{bc} **do**
    $M_i^d \leftarrow$ entity located in the address of M_i^d 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 4.4: M_exchangeEnts($entitiesToUpdate[d]$)

P_i. In order to reduce the communications between partitions, only one partition sends the message to P_i to create M_i^d. The partition to send the message to create M_i^d is the partition of the minimum partition id among residence partitions of M_i^d. The partition that sends messages to create a new entity is called the *broadcaster*, denoted as P_{bc}. The broadcaster is in charge of creating as well as updating of M_i^d over all partitions. For instance, among 3 copies of vertex M_5^0 in Figure 4.5(a), P_0 will be the *broadcaster* of M_5^0 since its partition id is the minimum among $\mathscr{P}[M_5^0]$. The arrows in Figure 4.5(c) show the broadcasters of the entities to migrate. In addition, before sending a message to P_i, M_i^d is checked if it already exists on P_i and ignored if exists. The purpose of sending only non-existing mesh entity is to avoid mesh-level global search on the destination partition before entity creation.

For each M_i^d to migrate, P_{bc} of M_i^d sends a message composed of the address of M_i^d on P_{bc} and the information of M_i^d necessary for entity creation, which consists of the following:

- unique vertex id (if vertex)
- entity shape information
- required entity adjacencies
- geometric classification information
- residence partition(s) for setting partition classification
- remote copy information

For instance, to create M_5^0 on P_3, P_0 sends a message composed of the address of M_5^0 on P_0 and information of M_5^0 including its $\mathscr{P}$ (i.e., P_1, P_2, and P_3) and remote copy information of M_5^0 stored on P_0 (i.e. the address of M_5^0 on P_2 and the address of M_5^0 on P_3).

- For the message received on P_i from P_{bc} (sent in Step 4.1), a new entity M_i^d is created on P_i. If the new entity M_i^d created is not a partition object, its address should be sent to back to the sender (M_i^d on P_{bc}) for the update of communication links. The message to be sent back to P_{bc} is composed of the address of M_i^d on P_{bc} and the address of new M_i^d created on P_i. For example, after M_5^0 is created on P_3, the message composed of the address of M_5^0 on P_0 and the address of M_5^0 on P_3 is sent back to P_0.

- In Step 4.3, the message received on P_{bc} from P_i (sent in Step 4.2) are sent to the remote copies of M_i^d on P_{remote} and the address of M_i^d on P_i is saved as the remote copy of M_i^d. The messages sent are received in Step 4.4 and used to save the address of M_i^d on P_i on all the remaining remote partitions of M_i^d. For instance, M_5^0 on P_0 sends the message composed of the address of M_5^0 on P_3 and P_3 to M_5^0 on P_1 and M_5^0 on P_2.

- For the message received on P_{remote} from P_{bc} (sent in Step 4.3), Step 4.4 updates the remote copy of M_i^d on P_{remote} to include the address of M_i^d on P_i. For instance, when M_5^0's on P_1 and P_2 receive the message composed of the address of M_5^0 on P_3 and P_3, they add it to their remote copy.

4.4.5 Step 5: Remove unnecessary entities

Step 5 removes unnecessary mesh entities collected in Step 3 which will be no longer used on the local partition. If the mesh entity to remove is on the partition boundary, it also must be removed from other partitions where it is kept as for remote copies through one round of communication. As for the opposite direction of entity creation, entities are removed from high to low dimension.

4.4.6 Step 6: Update entity ownership

Step 6 updates the owning partition of the partition model entities based on the poor-to-rich partition ownership rule. The partition model given in the right of Figure 4.5(e) is the final partition model with ownership.

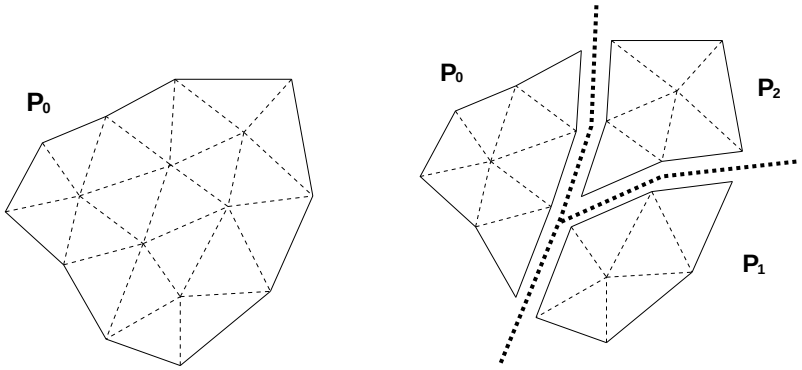
CHAPTER 5
FLEXIBLE DISTRIBUTED MESH DATA STRUCTURE

This chapter describes the extension of the flexible mesh data structure (Chapter 3) to distributed meshes (Chapter 4) to support flexible mesh representations in parallel. The algorithm of the mesh migration procedure with flexible mesh representations is discussed.

5.1 Representational Requirements of Flexible Distributed Mesh Data Structure

To support flexible mesh representations with distributed meshes, the mesh migration procedure must migrate the needed mesh entities regardless of mesh representation options while keeping requested mesh representation correct and updating the partition model and communication links based on new mesh partitioning. Figure 5.1(a) is an example 2D mesh with the minimum sufficient representation where all interior edges are reduced. The reduced edges are denoted with the dotted lines. Figure 5.1(b) is the partitioned mesh over 3 partitions with the MSR, where the only interior edges not on the partition boundaries are reduced. After migration, the interior edges on the partition boundaries must be restored in order to represent partitioning topology and support communication links between partitions.

To support mesh migration regardless of mesh representation options, an important question is what is a minimum set of entities and adjacencies necessary for migration. By the analysis of the mesh migration procedure in §4.4, the representational requirements



(a) serial mesh (b) partitioned mesh

Figure 5.1: Example 2D mesh with the MSR

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ - & - & 0 & 0 \\ - & 0 & - & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & - & - & 0 \\ - & - & 0 & 0 \\ - & 0 & - & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

(a) input MRM (b) 1^{st} MRM adjustment for parallel

Figure 5.2: 3D MRM adjustment (1 of 2)

for flexible distributed meshes are the following:

- For each partition object M_i^d, downward adjacent entities $\in \{\partial(M_i^d)\}$.
- For each downward adjacent entity of M_i^d, M_j^p,
  - other partition objects adjacent to M_j^p, and
  - remote copies.

Other partition objects adjacent to M_j^p are necessary in setting $\mathscr{P}$ of M_j^p to check if it will be existing on the local partition even after migration (See Algorithm 4.3). The representational requirements must be satisfied regardless of representation options to perform migration. In case that the user-requested representation doesn't satisfy the requirements, the representation is adjusted to meet the representational requirements to support mesh migration.

5.1.1 Efficient interior entity restoration

To provide communication links between entities on the partition boundaries and represent partitioning topology, non-existing internal mesh entities must be resurrected if they are located on the partition boundaries after migration. For a reduced representation, checking existence of downward entities in entity restoration can be efficiently done in O(1) time by maintaining $\{M^0\{M^d\}\}$ for each reduced level d. Therefore, to support efficient downward entity restoration, the first MRM adjustment is to modify the MRM to maintain $\{M^0\{M^d\}\}$ for each reduced level d. For instance, for the 3D user-requested representation given in Figure 5.2(a) which is the MSR, $\mathcal{R}_{0,1}$ and $\mathcal{R}_{0,2}$ are set to $-$ as seen in Figure 5.2(b). By maintaining the upward adjacencies $\{M^0\{M^1\}\}$ and $\{M^0\{M^2\}\}$ for existing edges and faces, obtaining $\{M_i^3\{M^1\}\}$ and $\{M_i^3\{M^2\}\}$ is done in a constant time either by local searching or restoration. See Algorithm D.7 and D.8 for the analysis of operators that return adjacent edges/faces of a region with a reduced representation.

```
Data  : M, M_i^d, p, where d > p
Result: construct downward adjacency {M_i^d{M^p}}
begin
    down_template_sz ← predefined size of |{M_i^d{M^p}}|;
    /* repeat |{M_i^d{M^p}}| times */
    for var ← 0 to down_template_sz-1 do
        M_i^0 ··· M_k^0 ← vertices to be in {M_var^p{M^0}} based on ordering
        templates; /* see Appendix D */
        /* check existence of edge/face (Algorithm A.1 and A.2) */
        M_var^p ← E_exist or F_exist(M_i^0 ··· M_k^0); /* O(1) */
        if M_var^p = NULL /* M_var^p is not found in M */
            /* create an interior edge/face (Algorithm A.5/D.10) */
            M_j^p ← M_createE or M_createF(M_i^0 ··· M_k^0); /* O(1) */
            M_i^d{M^p}_var ← M_var^p;
            set geo. classification of M_j^p to G_i^{d_1} where M_i^d ⊏ G_i^{d_1};
        endif
    endfor
end

/* Time = |{M_i^d{M^p}}| · O(1) = O(1)*/
```

Algorithm 5.1: createDownAdjacency_URR(M, M_i^d, p)

The Algorithm 5.1 is the pseudo code that restores downward adjacent entities of M_i^d, if necessary. The operator *E_exist* (resp. *F_exist*) returns an edge (resp. a face) which is defined by the input list of entities by local traversal of existing adjacencies, $\{M_0\{M_1\}\}$ (resp. $\{M_0\{M_2\}\}$) in this case. It returns $NULL$ if no edge or a face exists for the given entity list. *createDownAdjacency_URR* creates a downward interior entity if $NULL$ is returned by *E_exist* or *F_exist*. The list of vertices that define the interior downward entities are computed through the ordering templates illustrated in Figure 2.6 – 2.8.

5.1.2 Neighboring partition objects

In mesh migration using a complete representation, checking if an entity will exist on the current partition after migration is done via checking if there is any upward adjacent partition object that is maintained in the local partition. If any upward adjacent partition object remains in the local partition after migration, the current partition id, P_{local}, must be added into $\mathscr{P}$ of the entity.

With flexible mesh representations, especially in case where upward adjacency to

the level of partition objects is not available, to determine if an entity will exist on the current partition after migration or not, the alternatives are either:

1. during migration, creates the upward adjacency, or
2. while creating partition object M_i^d, store adjacency $\{M_i^0\{M_i^d\}\}$ for each $M_i^0 \in \{\partial(M_i^d)\}$.

Data : M_i^d, q where $d < q$
Result: construct upward adjacency $\{M_i^d\{M^q\}\}$ using $\{M^q\{M^d\}\}$
begin
  if $\{M_i^q\{M^d\}\} = \emptyset$
    return;
  endif
  / repeat $|\{M\{M^q\}\}|$ times */*
  for *each* $M_j^q \in \{M\{M^q\}\}$ **do**
    if $M_i^d \in \{M_j^q\{M^d\}\}$ *and* $M_j^q \notin \{M_i^d\{M^q\}\}$
      $\{M_i^d\{M^q\}\} \leftarrow \{M_i^d\{M^q\}\} \cup M_j^q$;
    endif
  endfor
end

/ Time = $|\{M\{M^q\}\}|$·O(1) = O($|\{M\{M^q\}\}|$) */*

Algorithm 5.2: createUpAdjacency_URR(M_i^d, q)

The first option restores upward adjacency to the level of partition objects when necessary. Algorithm 5.2 is pseudo code that creates upward adjacency $\{M_i^d\{M^q\}\}$, $d < q$, from scratch. It involves global mesh-level traversal of level q. Thus this method is not appropriate for the effective migration procedure.

The second option maintains upward adjacency $\{M_i^0\{M_i^d\}\}$ for each vertex M_i^0 on the boundary of partition object M_i^d. The neighboring partition objects of M_i^d is a set of partition objects M_j^{dj} that is bounded by M_j^p where $M_j^p \in \{\partial(M_i^d)\}$. Upward adjacency $\{M_i^0\{M_i^d\}\}$ for each $M_i^0 \in \{\partial(M_i^d)\}$ enable obtaining neighboring partition objects in a constant time. Based on the resident partition equation, for each $M_j^p \in \{\partial(M_i^d)\}$, if the neighboring partition objects of M_i^d is available, existence of M_j^p on the local partition after migration can be checked using downward adjacency of the neighboring partition objects.

Consider the user-requested representations in Figure 5.3(a). $\{M^0\{M^1\}\}$ and $\{M^0\{M^2\}\}$ are already set to $-$ by the first step of MRM adjustment (Figure 5.3(b)). By $\{M^0\{M^1\}\}$

$$\begin{bmatrix} 1 & - & - & 0 \\ - & - & 0 & 0 \\ - & 0 & - & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & - & - & 1 \\ - & - & 0 & 0 \\ - & 0 & - & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

(b) MRM after 1^{st} adjustment (c) MRM after 2^{nd} adjustment

Figure 5.3: 3D MRM adjustment (2 of 2)

input MRM

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ - & - & 0 & 0 \\ - & 0 & - & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ - & 0 & - & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ - & - & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

$\Downarrow \qquad \Downarrow \qquad \Downarrow \qquad \Downarrow$

adjusted MRM

$$\begin{bmatrix} 1 & - & - & 1 \\ - & - & 0 & 0 \\ - & 0 & - & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & - & 0 \\ 1 & 1 & 0 & 1 \\ - & 0 & - & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & - & 0 & 1 \\ - & - & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Figure 5.4: Example of 3D MRM adjustment for parallel

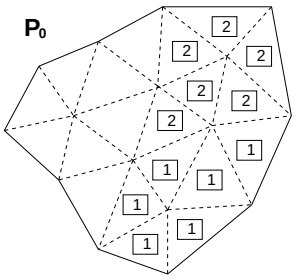
and $\{M^0\{M^2\}\}$, the neighboring partition objects of level 1 and 2 are obtainable. The second step of MRM adjustment set $\{M^0\{M^3\}\}$ to 1 in order to support neighboring partition objects of level 3 as seen in Figure 5.3(c). The penalty of this option is storing unrequested adjacency information. However, these adjacencies are necessary to avoid the needed, mesh-size dependent operations.

Figure 5.4 gives examples of MRM adjustment in parallel. For simplicity, we assume that the mesh is 3D and manifold models. The representations in the upper row are the mesh representations requested by the users and the representations in the lower row are the ones after MRM adjustment for parallel. Due to MRM adjustment, in parallel, the memory cost can increase compared that of the original user-requested representation. Memory increase due to the extra entities and adjacencies maintained for parallel is analyzed in §7.1.

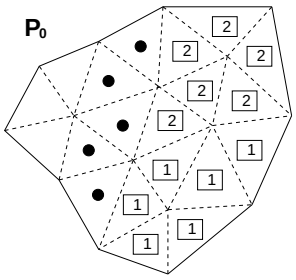
5.2 Algorithm of Mesh Migration with Flexible Representations

In this section, the mesh migration procedure *M_migrate* developed based on use of complete mesh representations is extended to work with any mesh representation options. The overall procedure for the mesh migration is the following: given $POsToMove$,

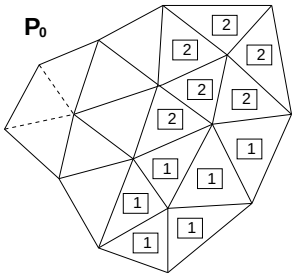
1. Collect neighboring partition objects.



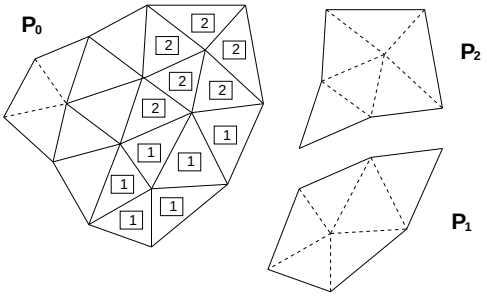
(a) mark destination pid



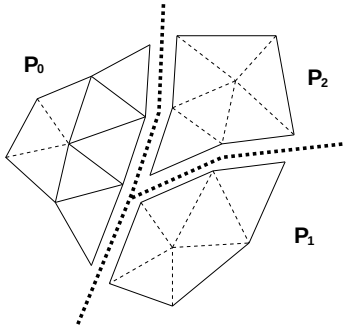
(b) get neighboring partition objects



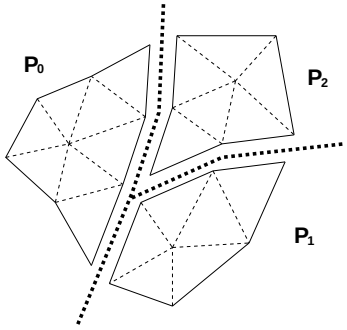
(c) restore internal entities



(d) migrate entities to dest. partition



(e) delete entities migrated



(f) delete internal entities

Figure 5.5: Steps of 2D mesh migration with the MSR

2. Restore needed downward interior entities.

3. Collect entities to be updated with migration and clear partitioning data ($\mathscr{P}$ and partition classification) of them.

4. Determine residence partition.

5. Update partition classification and collect entities to remove.

6. Exchange entities and update remote copies.

7. Remove unnecessary entities.

8. Update ownership of partition model entities.

9. Remove unnecessary interior entities and adjacencies.

Figure 5.5 depicts the 2D mesh migration procedure with a reduced representation by steps. For the given list of partition objects to migrate, $POsToMove$, (Figure 5.5(a)), first it collects a set of partition objects which are adjacent to any partition object in $POsToMove$ and store them in a separate container named $neighborPOs$ (Figure 5.5(b)). Second, for partition objects in $POsToMove$ or $neighborPOs$, restore their interior entities and associated downward adjacencies (Figure 5.5(c)). Based on downward adjacency restored, collect entities to be updated by migration in terms of their partitioning information such as $\mathscr{P}$, partition classification and remote copies, and save them in a container named $entitiesToUpdate$ for further manipulation. Using downward adjacencies and neighboring partition objects information, $\mathscr{P}$ and partition classification of entities in $entitiesToUpdate$ are updated. Based on $\mathscr{P}$ updated, the entities to remove from the local partition after migration are determined among the entities in $entitiesToUpdate$. After migrating only *necessary* entities to the destination partitions, remote copies of the entities on the partition boundaries are updated to refresh the communication links (Figure 5.5(d)). The entities collected to remove are deleted from the local partition (Figure 5.5(e)). Finally, the interior entities and adjacencies restored in the second step are removed to keep the original requested mesh representation (Figure 5.5(f)).

Algorithm 5.3 is pseudo code that migrates partition objects with flexible mesh representations.

5.2.1 Step A: Collect neighboring partition objects

For the given list of partition objects to migrate, $POsToMigrate$, Step A collects neighboring partition objects of them, which will be used in Step 2 to determine $\mathscr{P}$ of entities. Neighboring partition objects collected are stored in a container named $neighborPOs$. One round of communication is performed to gather neighboring partition objects on remote partitions.

5.2.2 Step B: Restore downward entities

In Step B, iterating over $POsToMigrate$ and $neighborPOs$, $M\_buildAdj\_URR$ restores non-existing downward interior entities of each partition object (Algorithm 5.4). $M\_adjacencyCost(M, i, j)$ is used to determine the call of $createDownAdjacency\_URR$.

Data : M, $POsToMigrate$
Result: migrate partition objects in $POsToMigrate$
begin
/* *STEP A: collect neighboring partition objects. See §5.2.1* */
For each partition object in $POsToMigrate$, collect neighboring partition objects and store them in $neighborPOs$;
/* *STEP B: restore downward entities. See §5.2.2* */
M_buildAdj_URR(M,$POsToMigrate$,$neighborPOs$);
/* *STEP 1: collect entities to process and clear partitioning data. See §5.2.3* */
Run STEP 1 in Algorithm 4.2;
/* *STEP 2: determine residence partition. See §5.2.4* */
M_setResidencePartition_URR($POsToMigrate$, $neighborPOs$);
/* *STEP 3: update p. classification and collect entities to remove. See §5.2.5* */
Run STEP 3 in Algorithm 4.2;
/* *STEP 4: exchange entities. See §5.2.6* */
for $d \leftarrow$ *0 to 3* **do**
M_exchangeEnts_URR($entitiesToUpdate[d]$);
endfor
/* *STEP 5: remove unnecessary entities. See §5.2.7* */
Run STEP 5 in Algorithm 4.2;
/* *STEP 6: update ownership. See §5.2.8* */
Run STEP 6 in Algorithm 4.2;
/* *STEP C: remove unnecessary interior entities and adjacencies. See §5.2.9* */
M_destoryAdj_URR(M, $entitiesToUpdate$, $neighborPOs$);
end

Algorithm 5.3: M_migrate_URR(M, $POsToMigrate$)

Data : M, $POsToMigrate$, $neighborPOs$
Result: for each partition object in $POsToMigrate$ or $neighborPOs$, restore downward entities
begin
for *each* $M_i^d \in POsToMigrate$ *or* $neighborPOs$ **do**
if *M_adjacencyCost(M, d, d − 1)* $\neq$ *Immediate or Local*
createDownAdjacency_URR(M, M_i^d, $d-1$);
endif
if d *= 3 and M_adjacencyCost(M, d, 1)* $\neq$ *Immediate or Local*
for *each* $M_j^2 \in \{M_i^d\{M^2\}\}$ **do**
createDownAdjacency_URR(M, M_j^2, 1);
endfor
endif
endfor
end

Algorithm 5.4: M_buildAdj_URR(M, $POsToMigrate$, $neighborPOs$)

createDownAdjacency_URR returns the cost of adjacency query $\{M^i\{M^j\}\}$ of which value is one of $\{Immediate, Local, Global, Unavailable\}$.

For each $M_i^d \in POsToMigrate$ or $neighborPOs$, if $M\_adjacencyCost(M, d, d-1)$ is neither $Immediate$ nor $Local$, interior entities of level $d-1$ are created using $createDownAdjacency\_URR(M, M_i^d, d-1)$. If M_i^d is a region and $M\_adjacencyCost(3, 1)$ is neither $Immediate$ nor $Local$, for each $M_j^2 \in \{\partial(M_i^d)\}$, $createDownAdjacency\_URR(M, M_j^2, 1)$ is called to create interior edges of M_i^d. $createDownAdjacency\_URR(M, M_i^1, 0)$ is not necessary to call since $\mathcal{R}_{1,0}$ is stored by default.

5.2.3 Step 1: Preparation

Using downward entities restored in Step B, Step 1 collects entities to be updated with migration, stores them in list vector *entitiesToUpdate* and resets partition classification and $\mathscr{P}$ of those entities. See §4.4.1.

Data : M, $POsToMigrate$, $entitiesToUpdate$, $neighborPOs$
Result: determine $\mathscr{P}$ of entities in $entitiesToUpdate$
begin
  /* *STEP 2.1: set $\mathscr{P}$ of entities in entitiesToUpdate through downward adjacency of partition objects in POsToMigrate* */
  for *each pair* $(M_i^d, p) \in POsToMove$ **do**
    $\mathscr{P}[M_i^d] \leftarrow \{p\}$;
    for *each* $M_j^q \in \{\partial(M_i^d)\}$ **do**
      $\mathscr{P}[M_j^q] \leftarrow \mathscr{P}[M_j^q] \cup \{p\}$;
    endfor
  endfor
  /* *STEP 2.2: determine if an entity will exist on the local partition after migration* */
  for *each* $M_i^d \in neighborPOs$ **do**
    for *each* $M_j^q \in \{\partial(M_i^d)\}$ **do**
      $\mathscr{P}[M_j^q] \leftarrow \mathscr{P}[M_j^q] \cup \{P_{local}\}$;
    endfor
  endfor
  /* *STEP 2.3: unify $\mathscr{P}$ of partition boundary entities* */
  Do one round of communication to exchange $\mathscr{P}$ of partition boundary entities in $entitiesToUpdate$;
end

Algorithm 5.5: M_setResidencePartition_URR($POsToMigrate$, $entitiesToUpdate$, $neighborPOs$)

5.2.4 Step 2: Determine residence partition

Step 2 determines $\mathscr{P}$ of entities collected in *entitiesToUpdate* (Algorithm 5.5). In Step 2.1, according to the resident partition equation, for each partition object M_i^d to migrate to partition p, $\mathscr{P}[M_i^d]$ is set to p, and p is added into $\mathscr{P}[M_j^q\ ]$, where $M_j^q \in \{\partial(M_i^d)\}$. For non-partition object entity M_j^q, their $\mathscr{P}$ must include local partition id, P_{local}, if it will exist on the local partition even after migration. Step 2.2 determines if M_j^q will exist or not on the local partition after migration based on downward adjacency of neighboring partition objects. For partition boundary entities in *entitiesToUpdate*, Step 2.3 performs one round of communication to unify $\mathscr{P}$ of them.

5.2.5 Step 3: Determine partition classification and entities to remove

For each entity in *entitiesToUpdate*, Step 3 determines partition classification and determines if it will be removed from the local partition. See § 4.4.3 for the details.

```
Data  : entitiesToUpdate[d]
Result: create entities on the destination partitions and update remote copies
begin
    /* STEP 4.1: send a message to the destination partitions */
    for each M_i^d ∈ entitiesToUpdate[d] do
        if P_local ≠ minimum partition id where M_i^d exists
            continue;
        endif
        if R_{d,d} ≠ 1
            if M_i^d will not be on p.boundaries or not equally classified
                continue;
            endif
        endif
        for each partition id P_i ∈ P[M_i^d] do
            if M_i^d exists on partition P_i (i.e. M_i^d has remote copy of P_i)
                continue;
            endif
            send message A (address of M_i^d on P_local, information of M_i^d) to
            P_i;
        endfor
    endfor
    Run STEP 4.2 to 4.4 in Algorithm 4.4;
end
```

Algorithm 5.6: M_exchangeEnts_URR(*entitiesToUpdate*[*d*])

```
Data  : M, entitiesToUpdate, neighborPOs
Result: remove unnecessary interior entities and downward adjacencies
begin
    /* STEP C.1: collect entities to process by dimension */
    for each M_i^d ∈ neighborPOs do
        store M_i^d in a list vector entitiesToProcess[d];
        for each M_i^q ∈ {∂(M_i^d)} do
            store M_i^q in a list vector entitiesToProcess[q];
        endfor
    endfor
    /* STEP C.2: remove {M^3{M^2}} if unnecessary */
    if M_adjacencyCost(M, 3, 2) ≠ Immediate or Local
        for each M_i^3 ∈ entitiesToProcess[3] do
            {M_i^3{M^2}} ← ∅;
        endfor
    endif
    /* STEP C.3: remove interior faces and {M^2{M^1}} if unnecessary */
    if M_adjacencyCost(M, 2, 1) ≠ Immediate or Local
        for each M_i^2 ∈ entitiesToUpdate[2] and entitiesToProcess[2] do
            if R_{2,2} ≠ 1
                if M_i^2 is not on partition boundary or not equally classified
                    M_deleteFace(M, M_i^2);
                endif
            else
                {M_i^2{M^1}} ← ∅;
            endif
        endfor
    endif
    /* STEP C.4: remove interior edges if unnecessary */
    if R_{1,1} ≠ 1
        for each M_i^1 ∈ entitiesToUpdate[1] and entitiesToProcess[1] do
            if  M_i^1 is not on partition boundary or not equally classified
                M_deleteEdge(M, M_i^1);
            endif
        endfor
    endif
end
```

Algorithm 5.7: M_destroyAdj_URR(M, *entitiesToUpdate*, *neighborPOs*)

$$\mathcal{R}^a = \begin{bmatrix} 1 & - & 0 & 0 \\ - & - & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \quad \mathcal{R}^b = \begin{bmatrix} 1 & - & - & 1 \\ - & - & 0 & 0 \\ - & 0 & - & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

Figure 5.6: Example MRM for 3D manifold model

5.2.6 Step 4: Exchange entities and update remote copies

Step 4 exchanges mesh entities from dimension 0 to 3 to create mesh entities on destination partitions. Algorithm 4.4 has been slightly modified to Algorithm 5.6 in order to work with any mesh representation options. Differences from Algorithm 4.4 are the following:

- The dimension of the entities used to create(define) faces and regions are determined based on the MRM. Consider region M_i^3 to migrate from P_i to P_j and the two MRM's, $\mathcal{R}^a$ ad $\mathcal{R}^b$, given in Figure 5.6. In case of $\mathcal{R}^a$, the message sent from P_i to P_j to create M_i^3 contains the number of faces that bound M_i^3 and a list of local copies of the faces on P_j. In case of $\mathcal{R}^b$, the message sent from P_i to P_j to create M_i^3 contains the number of vertices that bound M_i^3 and a list of local copies of the vertices on P_j.

- Not all interior mesh entities are migrated to the destination partitions. They are migrated only when necessary. In other words, interior entities are migrated to destination partitions only when they will be on the partition boundaries in new mesh partitioning topology after migration.

Figure 5.5(c) is an intermediary mesh after Step 4 where mesh faces marked for migration are created on destination partitions with reduced interior edges. On the destination partitions, the interior edges on partition boundaries were created to provide communication links. The faces migrated to the destination partitions are not deleted from the original partitions yet.

5.2.7 Step 5: Remove unnecessary entities

Step 5 removes unnecessary mesh entities collected in Step 3, which are not used on the local partition any more. See §4.4.5 for more information. Figure 5.5(d) is an intermediary mesh after Step 5, where mesh faces migrated to the destination partitions and their unnecessary adjacent edges and vertices are removed from partition P_0. Note the interior entities of neighboring partition objects restored in Step B still exist on partition P_0.

5.2.8 Step 6: Update entity ownership

Step 6 updates ownership of partition model entities. See §4.4.6.

5.2.9 Step C: Restore mesh representation

Step C restores the mesh representation modified to have interior entities and associated downward adjacencies in Step B to the original modified MRM. Note that some interior entities were already deleted in Step 5 while removing partition objects migrated. Also note that when partition objects are created on the destination partitions in Step 4, their representation is consistent with the modified MRM. The entities to be considered to remove or update in this step include neighboring partition objects and their downward entities, and entities in *entitiesToUpdate* not removed in Step 5. Algorithm 5.7 is pseudo-code of Step C performing 4 sub-steps.

[C.1] In addition to *entitiesToUpdate*, the entities to be considered to remove or update are collected from *neighborPOs*, and stored in a list vector named *entitiesToProcess* to expedite the process. *entitiesToProcess*[i] contains the entities of dimension i, $i = 0, 1, 2, 3$.

[C.2] For each region M_i^3 in *entitiesToProcess*[3], if *M_adjacencyCost*(M, 3, 2) is not *Immediate* nor *Local*, $\{M_i^3\{M^2\}\}$ is removed.

[C.3] *M_adjacencyCost*(M, 2, 1) is not *Immediate* nor *Local* in two cases: $\mathcal{R}_{2,2} \neq 1$ or $\mathcal{R}_{2,2} = 1$. If $\mathcal{R}_{2,2} \neq 1$, iterating over *entitiesToUpdate*[2] and *entitiesToProcess*[2], a face is removed from the database if the face is not on the partition boundary or not classified on model face. In the second case, only the adjacency $\{M^2\{M^1\}\}$ is removed for each face in *entitiesToUpdate*[2] or *entitiesToProcess*[2].

[C.4] If $\mathcal{R}_{1,1} \neq 1$, iterating over *entitiesToUpdate*[1] and *entitiesToProcess*[1], an edge is removed from the database if the edge is not on the partition boundary or not classified on model edge.

5.3 Summary

The following are the comparisons of the migration procedures, *M_migrate_URR* in Algorithm 5.3 (Steps A, B, 1 to 6, C) and *M_migrate* in Algorithm 4.2 (Steps 1 to 6):

- In Step A, *M_migrate_URR* collects neighboring partition objects to support computation of $\mathscr{P}$ without upward adjacencies.
- In Step B, *M_migrate_URR* restores downward entities and associated downward adjacencies of partition objects to migrate or neighboring.

- Step 1 is identical.
- In Step 2, $M\_migrate$ determines the existence of entities on the local partition after migration based on the existence of adjacent partition objects not to be migrated. $M\_migrate\_URR$ uses downward adjacency of neighboring partition objects on that determination.
- Step 3 is identical.
- In Step 4, $M\_migrate\_URR$ doesn't create interior entities on destination partitions if they are not on partition boundaries. In the message sent to the destination partitions to create an entity, the dimension of bounding entities for entity creation is explicitly specified.
- Step 5 is identical.
- Step 6 is identical.
- In Step C, $M\_migrate\_URR$ restores the representation to the modified MRM by removing unnecessary downward entities and adjacencies restored in Step B.

If a step involves inter-partition communications, the cost of the step is affected by the complexity of the partitioning topology. We assume that as the more entities are on the partition boundaries or the number of partitions increases, the partitioning topology gets more complicated. Obviously, as the partitioning topology gets more complex, the more cost is required for communications.

Partitioning topology refers to either of one before migration or one after migration, or both. For instance, the communications in Step A is influenced by $before$ partitioning topology. Whereas, the communications in Step 6 is influenced by $after$ partitioning topology. The communications in Step 4 is affected by both of $before$ and $after$ partitioning topology since it migrates entities based on $after$ partitioning topology and updates remote copies of entities based on $before$ partitioning topology. Note in Step 4 of $M\_migrate\_URR$, as the number of interior entities not on partition boundaries increases, the cost of the step decreases due to the less number of entity exchanges.

Conclusively, the computational cost of each migration procedures is the following:

$$\text{Time}(M\_migrate) = f\,(\#\ \text{POs to migrate}^{1}, \text{before/after partitioning topology}) \quad (5.1)$$

$$\begin{aligned}\text{Time}(M\_migrate\_URR) = f\,(&\#\text{ POs to migrate}^1, \#\text{ neighboring POs}^1,\\ &\text{before/after partitioning topology},\\ &\#\text{ non-existing interior entities}^1,\\ &\#\text{ interior entities to be exchanged}^1)\end{aligned} \tag{5.2}$$

Equation 5.1 means that the run time for $M\_migrate$ is a function of the number of partition objects and the complexity of the partitioning topology. The migration time increases with increases in the number of partition objects to migrate and the entities on partition boundaries increase. Equation 5.2 means that the run time of $M\_migrate\_URR$ increases as the number of partition objects to migrate, the number of neighboring partition objects, the number of interior entities to restore, the complexity of partitioning topology and the number of interior entities to be exchanged.

It has been noted that Step 4 spends most of the migration time among all steps both in $M\_migrate$ and $M\_migrate\_URR$ due to communication for entity exchange is most costly. In case of $M\_migrate\_URR$, the total migration time varies substantially depending on mesh representation options and partitioning topology due to the varying number of entity exchanges in Step 4. Performance results in §7.3 demonstrates that $M\_migrate\_URR$ with reduced representations tends to outperform $M\_migrate$ with the one-level adjacency representation as the mesh size and the number of partitions increase.

CHAPTER 6
IMPLEMENTATION

One important aspect of building a mesh database is its software design and implementation. The FMDB is implemented with C++ and provides an API (Application Programming Interface) for C/C++ and FORTRAN. Several advanced C++ programming elements such as the STL (Standard Template Library), functors, templates, singletons, and generic programming are used for the purpose of achieving reusability of the software [2, 24, 33, 59, 60, 62, 63, 82, 84, 98]. MPI (Message Passing Interface) [3, 38, 43, 67] and Autopack [55] are used for efficient parallel communications between processors. The Zoltan library [80] is used to make partition assignment during dynamic load balancing. In this chapter, the design and implementation of the FMDB including its parallel operation are presented. The FMDB is open source available at *http://www.scorec.rpi.edu/FMDB*.

6.1 Design/Implementation of Classes

This section discusses the design and implementation of classes for mesh, entity, partition model, and partition model entity.

6.1.1 Mesh

Figure 6.1 illustrates the relationship between the geometric model, the partition model, the mesh and their entities using the Unified Modeling Language notation [11, 12]. In the UML, boxes and lines between boxes denote, respectively, classes and relationships. *Inheritance* is indicated by a line ending in an arrow pointing from the derived class to the base class. *Aggregation*, that is, the where one object is a collection of other objects, is shown as a line connected to the "collection" object by a diamond. A general *association* is indicated by a simple line from one class box to another. The association may have a name. Each end of the association may have a role name that describes what the class at that end of the association does in that association. Also each end of the association may have a multiplicity indicator which indicates how many objects are involved in that association. A multiplicity of unknown number is indicated by an $*$.

A geometric model, *gModel*, is a collection of geometric model entities (*gEntity*). A partition model, *pModel*, is a collection of partition model entities (*pEntity*), and it is constructed from a set of mesh entities assigned to partitions. A mesh, *mMesh*, is a

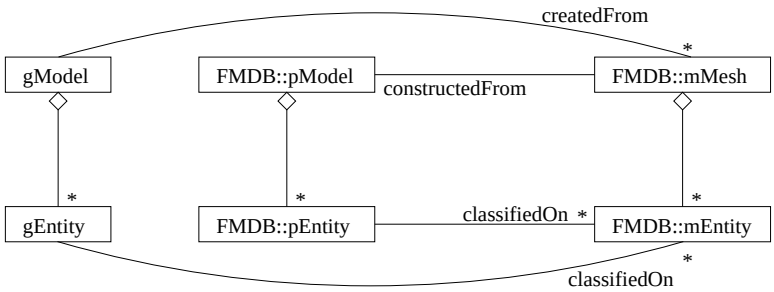


Figure 6.1: Class diagram of mMesh

collection of mesh entities (*mEntity*), and it is created from *gModel* using a mesh generation procedure. The mesh maintains its current classification against a geometric model, *gModel*, and a partition model, *pModel*.

```
class mMesh  {
protected:
   gModel* theGeoModel;  // geometric model
   pModel* thePtnModel; // partition model
   // container of all mesh entities by dimension
   mMeshEntityContainer allEntities;
public:
   typedef mMeshEntityContainer::iter iterall;  // iterators on all entities
   mMesh(gModel* model);  // constructor
   virtual ~mMesh();      // virtual destructor
   virtual void add(mEntity*);  // add a mesh entity to entity container
   mVertex* createVertex(double*, gEntity*); // create a vertex
   mEdge* createEdge(mVertex*, mVertex*, gEntity*);  // create an edge
   // create a new face
   mFace* createFace(mVertex**, int num_vertices, gEntity*);
   mFace* createFace(mEdge**, int num_edges, gEntity*);
   // create a new region
   mRegion* createRegion(mVertex**, int num_vertices, gEntity*);
   mRegion* createRegion(mFace**, int num_faces, gEntity*);
   ...
   virtual void DEL(mEntity*);  // entity deletion
   virtual mIterator beginall(int what) const; // iterator on all entities
   virtual mIterator endall(int what) const;
   // iterator on entities of dimension what classified on which
   virtual mIterator beginclas(int what, int which, int onwhat) const;
```

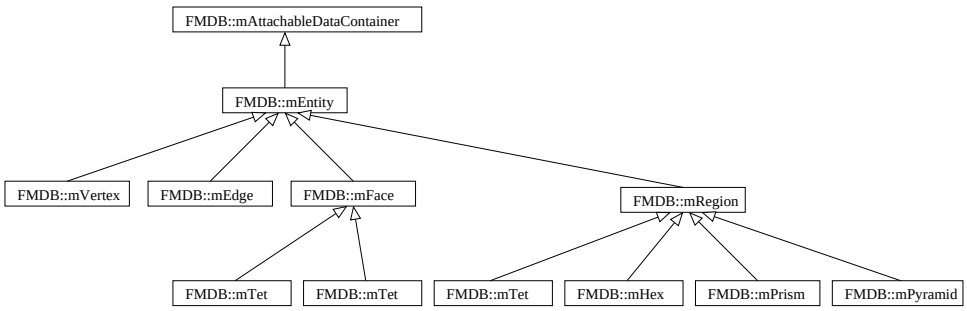


Figure 6.2: Class diagram of mEntity

```
    virtual mIterator endclas(int what, int which, int onwhat) const;
    // dimension of the mesh (0,1,2 or 3).
    int getDim() const;
    ...
};
```

An *mMesh* object acts as a container for mesh entities. Class *mMeshEntityContainer* is a class defined for a container of mesh entities using an STL *hash_table*. *mMesh* provides mesh entity creation functions for various entity topologies. There are a number of member functions to allow accessing either all of the entities in the mesh or the entities classified on certain model entities. The concept of iterator similar to STL iterators is used in the design of access to the entities in the mesh. Class *mIterator* provides an iterator to the mesh which is a separate object which, once obtained, allows one to advance to the next item sequentially. This separation of the functionality of a container and the access to the contents of the container allows multiple independent traversals of the information in the container.

6.1.2 Mesh entity

The class diagram of the mesh entity classes is illustrated in Figure 6.2. The functionality of arbitrary attachable data to mesh entities is provided by inheriting *mEntity* from class *mAttachableContainer*. Class *mAdjacencyContainer* is a class for a container to store upward and downward adjacent entities for each mesh entity. This is an internal of the implementation that is only accessible through STL-like iterators, *mAdjacencyContainer::iter begin()* and *mAdjacencyContainer::iter end()*, via *mEntity* interface functions. Four classes for mesh region (*mRegion*), mesh face (*mFace*), mesh edge (*mEdge*) and mesh vertex (*mVertex*) are inherited from the base class *mEntity*.

The mesh entity class is defined as the following:

```
class mEntity : public mAttachableDataContainer  {
 public:
    // bounding partition ids operator
    typedef map<int, mEntity*> remoteCopyMap;
    typedef map<int, mEntity*>::iterator RCIter;
    // virtual destructor
    virtual ~mEntity();
    // classification
    void setGClassification(gEntity*);   // set geo. classification
    gEntity* getGClassification() const; // get geo. classification
    void setPClassification(pEntity*);  // set partition classification
    pEntity* getPClassification() const;// get partition classification
    // remote copy operator
    mEntity* getRemoteCopy(int pid);
    void addRemoteCopy(int, mEntity*);
    void deleteRemoteCopy(int);
    virtual RCIter rcBegin()  { return theRemoteCopies.begin(); }
    virtual RCIter rcEnd()  { return theRemoteCopies.end(); };
    // adjacencies
    void add(mEntity* m);  // add entity m into adjacency list
    void del(mEntity* m);  // delete entity m from adjacency list
    mEntity* find(mEntity* m) const;  //search entity m in adjacency list
    mAdjacencyContainer::iter begin(int what);  // iterator on adjacent entities
    mAdjacencyContainer::iter end(int what);
    // Get the ith adjacent entity of level what
    inline mEntity* get(int what, int ith) const;
    ...
protected:
   int id;     // entity identifier
   // four adjacency sets. 0 is for vertices,
   // 1 for edges, 2 for faces and 3 for regions
   mAdjacencyContainer* theAdjacencies[4];
   gEntity* theClassification;  // geometric classification
   mEntity();  // constructor without parameters
   pEntity* thePClassification; // partition classification
   // container for multiple pair of remote partition and remote copy
```

```
    remoteCopyMap theRemoteCopies;
    ...
};
```

Each mesh entity stores geometric classification and partition model classification as data members. When an entity is located on the partition boundaries, its remote copies and remote partitions are maintained. A set of pairs of remote copy and remote partition of each mesh entity is stored in the STL *map* since remote copy and remote partition are one-to-one mapped. Iteration over remote copies is provided through STL-like iterators, *RCIter rcBegin()* and *RCIter rcEnd()*.

6.1.3 Partition model

The partition model, *pModel*, acts as a container for partition model entities. All partition model entities are stored in an STL *set* and iterated through STL-like iterators, *PEIter peBegin()* and *PEIter peEnd()*. It provides member functions such as updating the owner partition of partition model entities and computing partition classification of a mesh entity. Since a partition model must be dynamically updated as mesh partitioning changes, each partition model instance must maintain the relation to the corresponding distributed mesh.

```
class pModel  {
public:
    typedef set<pEntity*, pEntityLessThanKey>::const_iterator PEIter;
    pModel(mMesh*);  // constructor
    ~pModel();  // destructor
    void updateOwnership();  // update ownership of mesh entities
    void addPMEntity(pEntity*)  // add a partition entity
    // return a partition entity where the entity e is be classified on
    pEntity* getPClassification(mEntity* e)
    PEIter peBegin() const;  // partition entity iterator
    PEIter peEnd() const;
    ...
private:
    set<pEntity*, pEntityLessThanKey> allPEntities;
    mMesh* mesh;  // a mesh pointer
};
```

6.1.4 Partition model entity

pEntity is a base class of partition model entities and partition model region (*pRegion*), partition model face (*pFace*), partition model edge (*pEdge*) and partition model vertex (*pVertex*) are derived from *pEntity*. Each partition model entity, *pEntity*, stores id, dimension, the owner partition id, and the set of residence partitions as its member data. From the mesh entity level, by maintaining a relation to the partition model entity (the partition classification), all needed information in terms of partitioning, such as the owner partition and residence partition(s), are obtained with ease. Residence partitions of a partition entity are stored in an STL *set* since they are unique and iterated through STL-like iterators, *RPIter rpBegin()* and *RPIter rpEnd()*.

```
class pEntity  {
public:
   typedef set<int>::iterator RPIter;
   pEntity(int id, set<int>& bps, int dim);  // constructor
   ~pEntity();  // destructor
   RPIter rpBegin()  { return BPs.begin(); }  // residence partition iterator
   RPIter rpEnd()    { return BPs.end(); }
   ...
private:
  int id;
  int dimension;     // dimension
  int owner;         // owner partition
  set<int> RPs;      // set of residence partitions
};
```

6.2 Flexible Mesh Data Structure

In the implementation, mesh entity creation/deletion operators are declared as function pointers, and they are undetermined initially. The following are type definitions of mesh entity creation/deletion operators. The arguments of an entity creation operator are the mesh, the number of lower order bounding entities, the list of lower order bounding entities, and the geometric model entity to be classified on. The entity creation operator returns the entity created. The arguments of an entity deletion operator are the mesh and the mesh entity to be deleted.

```
// mesh edge creation:
```

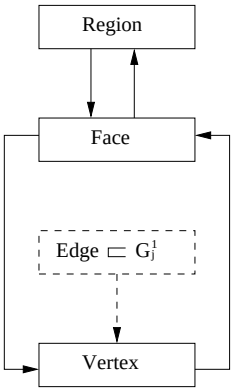


Figure 6.3: Example of user-requested representation

```
typedef mEdge* (*t_createE_FP)(mMesh*, int, mEntity**, gEntity*);
t_createE_FP createE_FP;
// mesh face creation:
typedef mFace* (*t_createF_FP)(mMesh*, int, mEntity**, gEntity*);
t_createF_FP createF_FP;
// mesh region creation}:
typedef mRegion* (*t_createR_FP)(mMesh*, int, mEntity**, gEntity*);
t_createR_FP createR_FP;
// mesh edge deletion:
typedef void (*t_deleteE_FP)(mMesh*, mEntity*);
t_deleteE_FP deleteE_FP;
// mesh face deletion:
typedef void (*t_deleteF_FP)(mMesh*, mEntity*);
t_deleteF_FP deleteF_FP;
// mesh region deletion:
typedef void (*t_deleteR_FP)(mMesh*, mEntity*);
t_deleteR_FP deleteR_FP;
```

Once the user-requested representation is provided, entity creation/deletion operators are dynamically set to the proper ones based on the rules of the design of mesh entity creation/deletion discussed in §3.2.3. Consider the user-requested representation depicted in Figure 6.3. For the given representation, mesh modification operators are shaped as the following:

```
// mesh edge creation: for each M_i^0 ∈ {∂(M_i^1)}, the edge creation operator
// adds M_i^0 to {M_i^1{M^0}}
createE_FP = createE_EV;  // createE_EV updates {M^1{M^0}}
```

```
// mesh face creation: for each M_i^0 ∈ {∂(M_i^2)}, the face creation operator
// adds M_i^0 to {M_i^2{M^0}} and adds the created face into {M_i^0{M^2}}
createF_FP = createF_VF;  // createF_VF updates {M^2{M^0}} and {M^0{M^2}}
// mesh region creation: for each M_i^2 ∈ {∂(M_i^3)}, the region creation operator
// adds M_i^2 to {M_i^3{M^0}} and adds the created region into {M_i^2{M^3}}
createR_FP = createR_FR;  // createR_FR updates {M^3{M^2}} and {M^2{M^3}}
// mesh edge deletion: the edge deletion operator updates no adjacencies.
deleteE_FP = deleteE;  // deleteE updates no adjacencies
// mesh face deletion: for each M_i^0 ∈ {∂(M_i^2)}, the face deletion operator
// deletes M_i^2 from {M_i^2{M^0}}
deleteF_FP = deleteF_VF;  // deleteF_VF updates {M^0{M^2}}
// mesh region deletion: for each M_i^2 ∈ {∂(M_i^3)}, the region deletion operator
// deletes M_i^3 from {M_i^3{M^2}}
deleteR_FP = deleteR_FR;  // deleteR_FR updates {M^2{M^3}}
```

By rule 1, the edge creation operator is set to the one that updates $\{M_i^1\{M^0\}\}$. By rule 1 and 2, the face (resp. region) creation operator is set to the one that updates $\{M_i^2\{M^0\}\}$ and $\{M_i^0\{M^2\}\}$(resp. $\{M_i^3\{M^2\}\}$ and $\{M_i^2\{M^3\}\}$). By rule 3, the edge deletion operator is set to the one that merely deletes the edge. By rule 4, the face (resp. region) deletion operator is set to the one that deletes the face (resp. region) from mesh database and removes the face (resp. region) from $\{M_i^0\{M^2\}\}$ (resp. $\{M_i^2\{M^3\}\}$) for each bounding vertex (resp. face) M_i^0 of it.

6.3 Parallel Functionalities

6.3.1 Parallel services

The parallel utility class, *ParUtil*, supports various services for parallel programming. Its main purpose is to hide the details of parallelization and let the user program do parallel operations without knowing details of parallel components. Class *ParUtil* is a singleton (i.e., it is based on the Singleton pattern [2, 33]) so only one single instance, *ParUtil::Instance()*, can exist overall and be accessible globally. The main goal in the design of distributed meshes is to have a serial mesh be a distributed mesh on a single processor. All parallel utility functions are also available in serial. For instance, in the serial case, *ParUtil::Instance()->size()* and *ParUtil::Instance()->rank()* return, respec-

tively, *1* and *0*, which denote the number of partitions and the partition id. The class definition of *ParUtil* is the following:

```
class ParUtil  {
public:
   static ParUtil* Instance();  // return the only instance
   void init(int &argc, char **&argv);   // initialize MPI and Autopack
   void Barrier(int, const char*);  // synchronize
   void Finalize();  // close communications
   double wTime() const;  // compute wall clock time
   // prints a message, the same format as printf
   void Msg(MessageLevel lev, char* fmt, ...);
   inline int rank()  { return myrank; }  // gets the partition id
   inline int size()  { return mysize; }  // gets the number of partitions
   inline int master(){ return myrank==0; } // return 1 if the master partition
   ...
private:
   static ParUtil* instance;
   int myrank;
   int mysize;
   MPI_Comm seq_local_comm;
   ParUtil();   // constructor
   ~ParUtil();  // destructor
   ...
};
```

6.3.2 Efficient communications: Autopack

Since communication is costly for distributed memory systems, it is important to group small pieces of messages and send all out in one inter-processor communication. The message packing library Autopack [55] is used for the purpose of reducing the number of message fragments exchanged between partitions.

Algorithm 6.1 gives the general non-blocking pseudo codes embedded in parallel mesh-based algorithms to minimize communications: beginning by allocating a local buffer on each processor (line 1), for mesh entities on partition boundaries, the messages for remote copies to be sent to remote partitions are collected in the buffer (line 2-4). When all desired messages have been processed, the messages collected in local buffer are sent to remote partitions(line 5). Then the remote partitions process what they received (line

6).

begin
1. initialize a buffer at local processor by *AP_send_begin*()
2. loop over desired mesh entities on each partition:
3. if the entity to be processed is on a partition boundary
4. pack messages and send them to the local buffer by *AP_send*()
5. send all messages in buffer using *AP_send_end*()
6. receive and process all received data

end

Algorithm 6.1: Pseudo-code of communications between partitions using *Autopack*

The following is the template of a program used for communications between remote copies of partition boundary entities using Autopack.

```
#include "autopack.h"

// send phase
int* sendcounts = new int[ParUtil::Instance()->size()];
for (int i = 0; i < ParUtil::Instance()->size(); ++i) sendcounts[i] = 0;
 for each entity on the partition boundary   {
    for each remote copy of the entity   {
       void* buf = AP_alloc(..., remote partition id, );
       fill the buffer ;
       AP_send(buf);
       ++sendcounts[remote partition id];
   }
}
// receive phase
AP_check_sends(AP_NOFLAGS);
AP_reduce_nsends(sendcounts);
int count, message = 0;
while (!AP_recv_count(&count) || message<count)  {
   void* msg;
   int rc = AP_recv(..., &msg, ...);
   if (rc)  {
      ++message;
      process msg received ;
      AP_free(msg);
```

```
    }
  }
  AP_check_sends(AP_WAITALL);
  delete[] sendcounts;
```

The C integer array *sendcounts* is a counter for the number of messages sent to each partition. After initializing *sendcounts*, for each remote copy of each entity, *AP_alloc* allocates memory for a message. After filling the message buffer to send, *AP_send* sends the message to the remote partition where the remote copy exists. *AP_recv* receives the message and the appropriate operation is performed on the remote copy. *AP_recv_count* keeps track of the number of messages received ensuring the number of messages sent and received are identical. Packing many small messages into larger messages is hidden from the user.

6.3.3 Generic data communicator

Communications between remote copies of partition boundary entities are performed frequently during parallel adaptive simulations. The typical pattern of communications between remote copies of a mesh entity on the partition boundary is the following:

1. For each mesh entity on the partition boundaries:
   (a) For each remote copy of the mesh entity on remote partitions:
      i. Fill the message buffer to send to the remote copy on the remote partition. One message per remote copy is filled. This allows sending different messages depending on the destination partition.
      ii. Send the messages to the remote copy of the mesh entity.
      iii. The remote copy of the entity receives the message from the *sender* partition and processes the received data.

The only difference between each communication is the data to be sent, the tag of message (an MPI term, it is like the address on a mail message) and how the remote copy will process the data received. To avoid coding of the communications over and over with the same pattern and different messages and/or operations, a generic communicator callback class, *pmDataExchanger*, and a generic data exchange operator, *genericDataExchanger*, have been developed.

```
class pmDataExchanger  {
public :
   virtual int tag() const = 0;  // get a message tag
   // send a message to the remote copy of a mesh entity e
   // on the remote partition pid
   virtual void* alloc_and_fill_buffer(mEntity* e,
                                  int pid, mEntity*, int tag) = 0;
   // receive data from partition pid
   virtual void receiveData(int pid, void* buf) = 0;
};
template <class Iterator>
void genericDataExchanger(const Iterator &beg, const Iterator &end,
                          pmDataExchanger& de)  {
   mEntity* ent;
   int* sendcounts = new int[ParUtil::Instance()->size()];
   for (int i = 0; i < ParUtil::Instance()->size(); ++i) sendcounts[i] = 0;
   for (Iterator it=beg; it != end ; ++it)  {
      ent = *it;
      if (ent->getNumRemoteCopies() == 0)  continue;
      for (mEntity::RCIter rcIter = ent->rcBegin(); rcIter != ent->rcEnd();
                           ++rcIter)  {
         void* buf = de.alloc_and_fill_buffer(ent,
            (*rcIter).first, (*rcIter).second, de.tag());
         if (buf)  {
            AP_send(buf);
            ++sendcounts[(*rcIter).first];
         }
      }
   }
   AP_check_sends(AP_NOFLAGS);
   AP_reduce_nsends(sendcounts);
   int count, message = 0;
   while (!AP_recv_count(&count) || message<count)  {
      void* msg;
      int from, tag, size, rc;
      rc = AP_recv(MPI_ANY_SOURCE, de.tag(), AP_BLOCKING|AP_DROPOUT,
                   &msg, &size, &from, &tag);
      if (rc)  {
```

```
            ++message;
            de.receiveData(from, msg);
            AP_free(msg);
         }
      }
      AP_check_sends(AP_WAITALL);
   }
```

Class *pmDataExchanger* is an abstract base class since it defines pure virtual member functions; such functions must be given definitions in a derived class. Users must specify data to be filled in the message buffer, message tag and the operation to be performed when the data is received to the remote copy. Using a specialized instance of *pmDataExchanger*, operator *genericDataExchanger* exchanges data between remote copies of entities provided through templated parameter *Iterator*. A typical round of communications looks like the following:

```
class myExchanger: public pmDataExchanger {...};

vector<mEntity*> entsOnPtnBdry;
fill entsOnPtnBdry;
myExchanger myCallback;
genericDataExchanger(entsOnPtnBdry.begin(), entsOnPtnBdry.end(), myCallback);
```

The following is the example of one round of communications that exchange residence partitions of the partition boundary entities in order to unify them between copies. This is called in Step 2.3 of the mesh migration procedures.

```
class rpsExchanger: public pmDataExchanger  {
public :
   virtual int tag() const  { return 2222; }
   virtual void* alloc_and_fill_buffer(mEntity* e, int pid, mEntity* rc, int);
   virtual void receiveData(int pid, void* buf);
};
void* rpsExchanger::alloc_and_fill_buffer
      (mEntity* ent, int pid, mEntity* remoteEnt, int d_tag)  {
   // ent->RPs is an STL vector that stores residence partitions of
   // the entity temporarily for a computation purpose
   int nbRPs = ent->RPs.size();  // number of residence partitions
   char* buf = (char*) AP_alloc(pid, d_tag,
```

```
                        (sizeof(mEntity*)+(nbRPs+1)*sizeof(int)));
   memcpy(buf, &remoteEnt, sizeof(mEntity*));
   int* resPids = (int*)malloc((nbRPs+1)*sizeof(int));
   resPids[0] = nbRPs;
   int count = 0;
   for (vector<int>::iterator rpIter = ent->RPs.begin();
                              rpIter != ent->RPs.end(); ++rpIter)
      resPids[++count] = *rpIter;
   memcpy(&buf[sizeof(mEntity*)], resPids, (nbRPs+1)*sizeof(int));
   free(resPids);
   return buf;
}
void rpsExchanger::receiveData(int senderPid, void* buf)  {
   char* mybuf = (char*) buf;
   mEntity* e;
   memcpy(&e,mybuf,sizeof(mEntity*));
   int nbRPs;
   memcpy(&nbRPs, &mybuf[sizeof(mEntity*)], sizeof(int));
   int* resPids = (int*)malloc((nbRPs+1)*sizeof(int));
   memcpy(resPids, &mybuf[sizeof(mEntity*)], (nbRPs+1)*sizeof(int));
   for (int i = 1; i <= nbRPs; ++i)
      e->RPs.push_back(resPids[i]);
   free(resPids);
}
int M_migrate(mMesh* mesh, list<pair<mEntity*, int> >& POsToMove)  {
   ...
   vector<mEntity*> entitiesOnPtnBdry;
   // M_getEntsOnPtnBdry returns entities on the partition boundaries
   M_getEntsOnPtnBdry(mesh, entitiesOnPtnBdry);
   rpsExchanger aCallback;  // declare an instance of rpsExchanger
   genericDataExchanger(entitiesOnCB.begin(), entitiesOnCB.end(), aCallback);
   ...
}
```

Class *rpsExchanger* is derived from *pmDataExchanger* and specialized to exchange resident partitions as the following:

- The *sender* partition fills the message buffer with residence partitions of an entity.

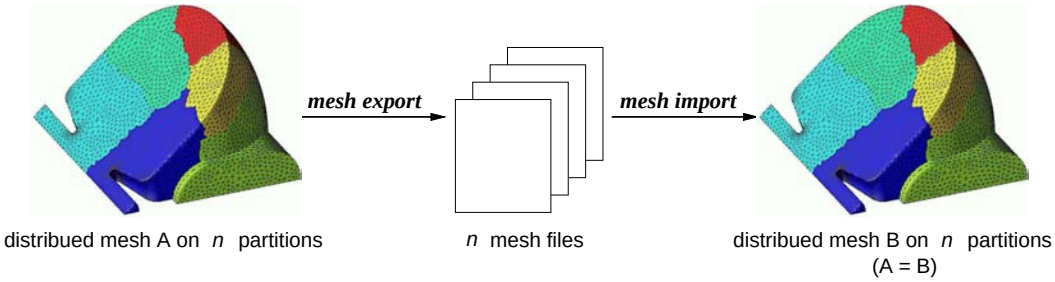


Figure 6.4: Parallel mesh I/O

- The message buffers are tagged with integer 2222 (an arbitrary value).
- The *receiver* partition adds the received residence partition(s) into the residence partition(s) of the remote copy.

Operator *M_getEntsOnPtnBdry* returns a set of partition boundary entities in a given STL vector container. To perform communication, first, an instance of *rpsExchanger*, named *aCallback* in the example, is declared. Next, operator *genericDataExchanger* is called with 2 input iterator arguments, *begin()* and *end()* to the vector container filled by *M_getEntsOnPtnBdry*, and the specialized instance of *pmDataExchanger*, *aCallback*.

6.3.4 Parallel mesh I/O

Figure 6.4 illustrates the parallel mesh I/O. The parallel mesh import/export procedures let the user export the distributed mesh into mesh files and recover the mesh later from the files. The parallel mesh export operator writes a distributed mesh on n partitions into n mesh files and the parallel mesh import operator reads the n mesh files and constructs the identical distributed mesh on n partitions as before the export. The information kept in the mesh file to recover the distributed mesh from the file includes:

- the partition model information, and
- for each entity on the partition boundaries, the partition model classification information.

Note that with parallel mesh I/O, mesh entities' partition classification should be kept the same and a list of remote partition ids are determined from the partition classification of mesh entities. Therefore, in exporting a distributed mesh, mesh entity's partition classification should be stored for the future recovery. However, the remote copy on the remote partition cannot be set until mesh distribution is finished since the remote copy

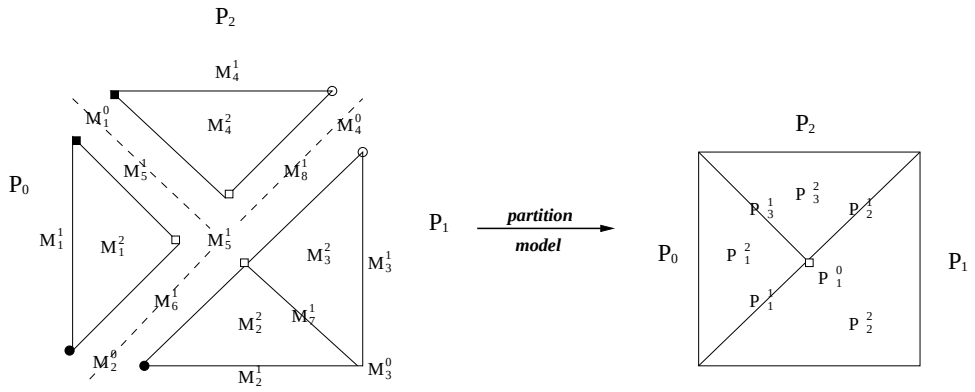


Figure 6.5: Simple 2D distributed mesh on 3 partitions and its partition model

is a memory address (location) of the mesh entity on a remote partition. Thus, for the purpose of parallel mesh I/O, the following must be considered.

1. How to store and recover the partition classification of mesh entity
2. How to set a remote copy on a remote partition of mesh entity *efficiently*, especially without involving mesh entity search of which time complexity is $O(n \lg n)$ where n is the number of mesh entities in the mesh of the entity level.

6.3.4.1 Parallel mesh exporting: storing partition model

Because the partition model should be kept the same before and after the parallel mesh I/O, the partition model of the mesh should be stored in the file along with the mesh. For this purpose, three pieces of information are needed: partition id, partition entity information, and the reverse partition classification of partition entity (or the partition classification of mesh entity). We use the partition classification of mesh entities instead of reverse partition classification. For a simple example mesh in Figure 6.5, Figure 6.6 illustrates the contents of the mesh file dumped from partition P_1.

Under an assumption of one partition per processor taken in the FMDB, a pair of the partition id and processor id doesn't need to be kept the same before and after parallel mesh I/O.

6.3.4.2 Parallel mesh importing: recovering partition model

When a distributed mesh is recovered from files, partition entities are created first, and then, mesh entities are created with given partition classification in the file. By doing this for all mesh entities, parallel classification and the list of remote partition ids

```
// PART 1
// partition id
          1

// PART 2
// partition entity id    residence partition id    dimension    owner partition id
          1                       0, 1                1              0
          2                       1, 2                1              2
          4                       0, 1, 2             0              0

// PART 3
// mesh entity              partition entity id
          ...                       ...
```

Figure 6.6: Mesh file dumped from partition P_1

partition entity	residence partitions	owner partition
P_1^1	P_0, P_1	P_0
P_2^1	P_1, P_2	P_2
P_1^0	P_0, P_1, P_2	P_0

Table 6.1: Partition model entities recovered on partition P_1 from mesh file

are properly set. For the sample mesh file given in Figure 6.6, Table 6.1 and 6.2 show, respectively, the partition entities and mesh entities recovered from the file on partition P_1. Now, the remaining task is to set remote copies of partition boundary entities to recover partition boundary links.

6.3.4.3 Parallel mesh importing: recovering partition boundary links

This step requires one round of communication to exchange the memory locations of the duplicated copies of mesh entities. The most expensive part would be, when a

mesh entity	partition classification	remote partition id	remote copy
M_2^2	P_2^2	-	-
M_3^2	P_2^2	-	-
M_2^1	P_2^2	-	-
M_3^1	P_2^2	-	-
M_6^1	P_1^1	P_0	-
M_7^1	P_2^2	-	-
M_8^1	P_2^1	P_2	-
M_2^0	P_1^1	P_0	-
M_3^0	P_2^2	-	-
M_4^0	P_2^1	P_2	-
M_5^0	P_1^0	P_0	-
		P_2	-

Table 6.2: Mesh entities recovered on partition P_1 from mesh file

remote copy is received from other partitions, finding the corresponding mesh entity to set the remote copy from the received data. The time complexity of mesh entity search is $O(N_i \lg N_i)$ where N_i is the number of mesh entities in the mesh of level i. To be efficient, this search operation must be avoided. The algorithms for setting remote copies efficiently is the following:

1. Before exporting the mesh entities to the file, unique ids are assigned to all of the partition boundary entities. The unique id assignments start from 0 since the array indices in C/C++ start from 0. Entity's owner partition is used in this step. For each partition boundary entity, if the current partition is the owner of the entity, the unique id is assigned to the entity and the unique id is propagated to all copies of the entity by this partition. After one round of communication is finished to assign unique ids, the unique ids are dumped to files along with mesh entities.

2. In loading mesh entities, if an entity is given with unique id i, the entity is saved in the i^{th} element of a vector. Therefore, each partition boundary entity can be accessed directly without searching.

3. To set remote copy(s) of a partition boundary entity, for each entity on partition boundaries, send a message (P_{local}, address of the entity on P_{local}, j) to each remote partition, where P_{local} is the current partition id.

4. When the message (P_i, address of the entity on P_i, unique_id) is received from another partition, the corresponding mesh entity is the j^{th} element of the array vector filled in step 2. For that entity, the address of the entity on P_i is saved as for a remote copy of the entity on P_i.

6.4 Dynamic Mesh Load Balancing

6.4.1 Design of the load balancing procedure

Figure 6.7 shows the main components of parallel adaptive analysis. An adaptive analysis typically starts with a coarse mesh and a low-order numerical solution of a problem and based on an estimate of the local discretization error either refines the mesh ($h-$refinement), increases the order of numerical solution ($p-$refinement), moves the mesh ($r-$refinement), or does combinations of $h-$, $p-$ and $r-$refinements to improve the quality of the solution. In order to perform each component in parallel, the computation and mesh data need to be partitioned in such a way that the load balance is achieved in

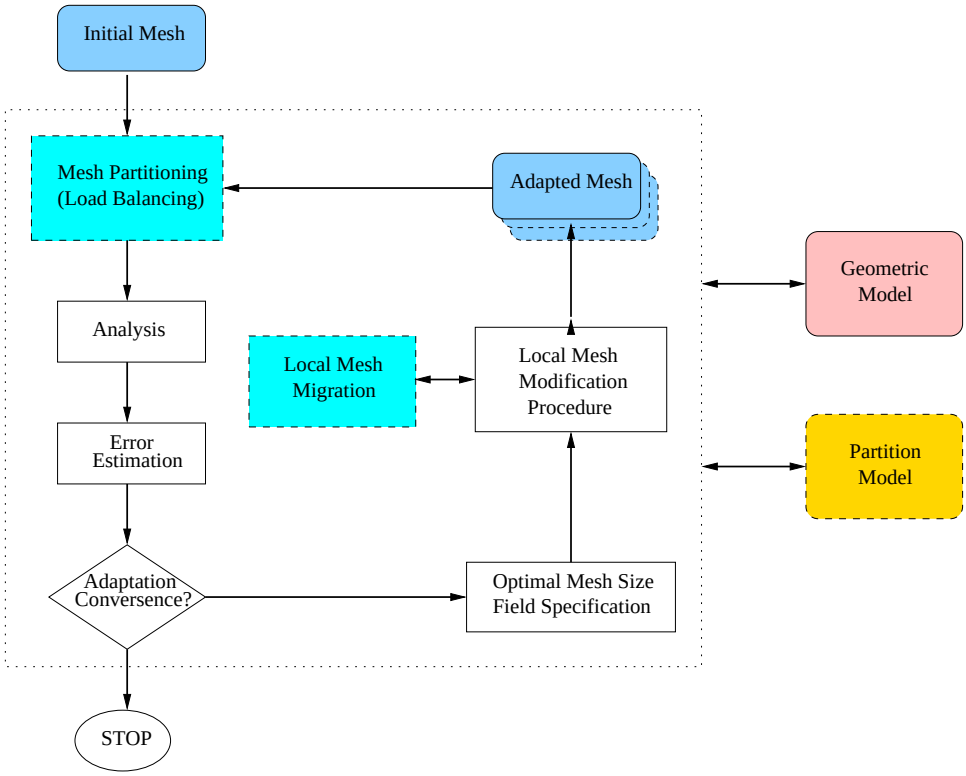


Figure 6.7: Key steps of parallel automated adaptive analysis

each partition and the size of the inter-partition boundaries is kept minimal. In addition, having adapted the distributed mesh, the processors may become imbalanced in which case either a parallel re-partitioning or dynamic load balancing phase needs to be applied. These two basic requirements are necessary to maximize utilization of all processors by minimizing idling of processors due to load imbalance and inter-processor communication.

The Zoltan library [80] is a collection of data management services for parallel, unstructured, adaptive, and dynamic applications. It includes a suite of parallel algorithms for dynamically partitioning problems over sets of processors. The FMDB interfaces with the Zoltan to obtain redistribution information of the mesh. The load balancing procedure computes the input to the Zoltan which is a representation of the distributed mesh, usually a weighted graph. With the distribution information from Zoltan, the re-partitioning or initial partitioning step is completed by calling the mesh migration procedure that moves the appropriate entities from one partition to another. Figure 6.8 illustrates an example of 2D mesh load balancing. In the left, the partition objects (all mesh faces in this case) are tagged with their destination partition ids. The final balanced mesh is given on the right.

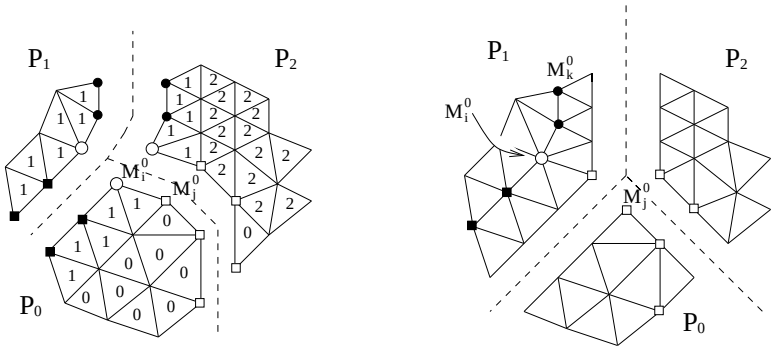


Figure 6.8: Example of 2D mesh load balancing: (left) partition objects are tagged with their destination pids (right) mesh after load balancing

6.4.2 User interface: Zoltan callbacks

The load balance procedure utilizes a weighted graph or coordinates of partition objects to define a model for computational load. The load of a given processor P_i is defined as the number of elements in its partition multiplied by a weight that can be determined based on, for example, the elements computational demands. If no weight information is provided, the FMDB sets all weights to 1.0 by default so that the load is simply proportional to the number of elements. The aim of load balancing is to balance loads between processors while minimizing inter-processor communications, i.e., size of partition boundaries. The user can also provide weights for partition boundaries in order to take into account differing communication costs. Finally, arbitrary data attached to mesh entities can be migrated as well as mesh entities. The interface for load balancing is the following:

```
class pmLBCallbacks  {
public :
   // from a given graph, we retrieve a partition information
   virtual void partition(FMDB_distributed_graph &theGraph,
                          int *partitionInfo) = 0;
   // get user data of size "size" attached to a mesh entity for migration
   virtual void * getUserData (mEntity*, int dest_pid, int &size) = 0;
   // delete or free user attached data
   virtual void deleteUserData (void*) = 0;
   // receive user data. mEntity* is now the mesh entity on the
   // remote partition
   virtual void receiveUserData (mEntity*, int pid, int tag, void *buf) = 0;
};
```

```
class pmZoltanCallbacks: public pmLBCallbacks  {
public :
   // Zoltan algorithms available
   typedef enum Algorithm {LDiffusion, GDiffusion, Remap, MLRemap, Random,
                           Octree, Serial};
   // Constructor takes the algorithm as input
   pmZoltanCallbacks(Algorithm algo = Remap);
   // this function interfaces with Zoltan
   virtual void partition(FMDB_distributed_graph &theGraph,
                          int *partitionVector);
   // change the algorithm
   void setAlgorithm (const Algorithm &algo);
private:
   // save the algorithm
   Algorithm theAlgorithm;
};
```

Class *pmLBCallbacks* is purely virtual. It serves as a base class of the Zoltan callbacks class, *pmZoltanCallbacks*, to provide functions to retrieve partition information from the Zoltan in a form of an integer array of which the i^{th} item denotes the destination partition of i^{th} partition object in the mesh and pack/unpack arbitrary attached data to mesh entities for communication. Class *pmZoltanCallbacks* is a derived class from *pmLBCallbacks* with addition of a function that enables one to choose one specific load balancing algorithm among various partitioning services of the Zoltan.

A typical set of instructions for doing a dynamic load balancing with the FMDB is to declare a derived class from *pmZoltanCallbacks*, choose a Zoltan algorithm, fill in the behaviors for packing/unpacking attached data to mesh entities, set weights of entities. Reference [80] provides more detailed discussions on the partitioning algorithms provided by Zoltan. The following is a simple user-defined Zoltan callback class for initial serial mesh partitioning:

```
class myCB: public pmZoltanCallbacks  {
public :
   myCB() : pmZoltanCallbacks(pmZoltanCallbacks::Serial){}
   virtual void * getUserData (mEntity* e, int dest_proc, int &size)
   { return; }
   virtual void receiveUserData (mEntity* e, int pid, int tag, void *buf)
   { return; }
```

```
   virtual void deleteUserData (void *buf)  { free(buf); }
};

// define a mesh object
mMesh* theMesh = new mMesh();
// load a serial mesh from a mesh file
M_load(theMesh, mesh_file);
// declare Zoltan callback object
myCB myCB_object;
// call load balance procedure
M_loadbalance(theMesh, myCB_object);
```

The load balance procedure, *M_loadbalance*, takes two inputs: a distributed mesh and an object of user-defined Zoltan callback type. For simplicity, attached data or weights were not considered in the user-defined Zoltan callback type, *myCB*.

CHAPTER 7
PERFORMANCE RESULTS

This chapter presents performance results of the FMDB (serial and parallel). Throughout this thesis, all the experiments were performed on an IBM HPC cluster machine with 32 compute nodes, each with two Intel Xeon 2.0GHz processors and 2GB memory, IBM x335 master node with 2GB memory and two Intel Xeon 2.0GHz processors, and Myrinet-2000 interconnect with peak performance 53.3 Gflops running Linux.

Some of tetrahedral meshes used in the performance tests are listed in Table 7.1. For test purposes, some of the mesh entities are deleted to have the exact number of regions. The number given in parentheses represents the number of entities equally classified on model entities and the percentage represents the ratio of those entities among all entities of the level.

7.1 Storage Efficiency with Flexibility

Figure 7.1 shows four mesh representations used to measure the memory cost of FMDB, the minimum sufficient representation (MSR), representation with no interior faces which consists of regions, edges, vertices and faces on the boundary (denoted as REV), representation with no interior edges which consists of a full set of regions, faces,

Table 7.1: Example meshes used in FMDB performance tests

name name	# regions	# faces (# faces $\sqsubset G_i^2$, %)	# edges (# edges $\sqsubset G_i^1$, %)	# vertices
1K	1000	2,332 (477, 20.45%)	1,659 (116, 6.99%)	335
5K	5000	10,858 (1,378, 12.69%)	7,068 (198, 2.80%)	1,237
10K	10,000	21,242 (1,461, 6.87%)	13,419 (198, 1.47%)	2,266
50K	50000	103,913 (5,060, 4.86%)	63,945 (356, 0.55%)	10,414
100K	100,000	208,853 (12,641, 6.05%)	127,992 (2,009, 1.57%)	19,761
500K	500,000	1,023,105 (25,458, 2.48%)	614,570 (804, 0.13%)	94,526
1M	1,000,000	2,083,475 (55,600, 2.67%)	1,264,707 (1,435, 0.11%)	193,932

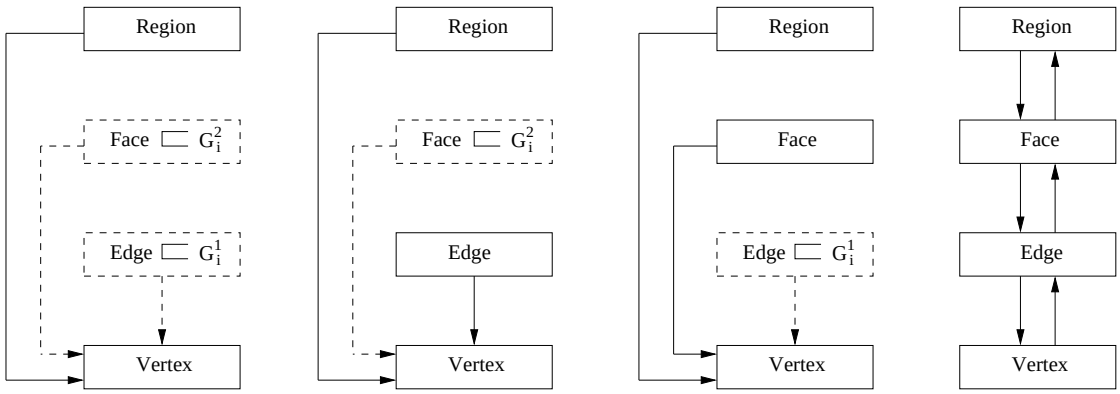


Figure 7.1: Mesh representation used in performance tests: (a) minimum sufficient representation (b) reduced faces (c) reduced edges (d) one level

Table 7.2: Storage cost for 3D meshes (MB)

	$\mathcal{R}^1$ (MSR)	$\mathcal{R}^2$ (REV)	$\mathcal{R}^3$ (RFV)	$\mathcal{R}^4$ (one level)
MRM	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ - & - & 0 & 0 \\ - & 0 & - & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ - & 0 & - & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ - & - & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$
1K	1.85	1.99	2.05	2.27
5K	2.42	3.04	3.41	4.58
10K	3.03	4.21	5.00	7.21
50K	7.82	13.18	17.66	28.64
100K	14.26	24.86	33.75	55.75
500K	59.19	110.73	158.17	264.73
1M	96.45	177.50	244.57	420.71
1.2M	117.76	223.79	318.91	536.94
2.3M	214.51	405.11	585.57	975.02

vertices and edges on the boundary (denoted as RFV), and one-level representation.

Table 7.2 shows the storage cost of 4 mesh representation options in mega bites (MB); $\mathcal{R}^1$, $\mathcal{R}^2$, $\mathcal{R}^3$, and $\mathcal{R}^4$ are, respectively, MRM for the minimum sufficient representation (MSR), representation with no interior faces (REV), representation with no interior edges (RFV), and one-level representation. Memory usage is measured with resident set size (RSS). In a virtual memory system, a process' resident set is that part of a process' address space which is currently in main memory [72] obtained through "*ps -ly -C*" command and includes that of geometric model. Note that even with the same number of

Table 7.3: Relative storage cost for 3D meshes

MRM	$\mathcal{R}^1$ (MSR)	$\mathcal{R}^2$ (REV)	$\mathcal{R}^3$ (RFV)	$\mathcal{R}^4$ (one level)
1K	.81 (.82)	.87 (.89)	.90 (.94)	1
5K	.52 (.55)	.66 (.71)	.74 (.84)	1
10K	.42 (.45)	.58 (.64)	.69 (.82)	1
50K	.27 (.31)	.46 (.53)	.61 (.77)	1
100K	.25 (.30)	.44 (.52)	.60 (.77)	1
500K	.22 (.27)	.41 (.49)	.59 (.77)	1
1M	.22 (.27)	.42 (.49)	.58 (.74)	1
1.2M	.21 (.27)	.41 (.49)	.59 (.77)	1
2.3M	.22 (.26)	.41 (.49)	.60 (.76)	1

region meshes, depending on the number of faces and edges in the representation, the memory usage varies substantially.

Table 7.3 shows the relative storage cost of 4 mesh representation options with an assumption of storage cost 1 with the one-level adjacency representation. For reduced representations, $\mathcal{R}^1$, $\mathcal{R}^2$ and $\mathcal{R}^3$, relative storage for distributed meshes, which were measured with adjusted MRM's, are given in parentheses.

The reduction in memory consumption with the flexible mesh representation options varies depending on the level of reduced entities and adjacencies, and the ratio of equally classified entities. For the example meshes, a decrease in storage is $19-79\%$ with $\mathcal{R}^1$, $13-59\%$ with $\mathcal{R}^2$, $10-42\%$ with $\mathcal{R}^3$. In the parallel case, a decrease in storage is $18-74\%$ with $\mathcal{R}^1$, $11-51\%$ with $\mathcal{R}^2$, and $6-26\%$ with $\mathcal{R}^3$. It has been observed that memory savings with reduced representations increase as the mesh size gets greater.

The memory savings with the flexible mesh representation options shows a clear benefit of running mesh-based applications with customized mesh data structures that fit to the needs of the application in terms of the mesh representation. The potential improvement from the flexible mesh representation depends to a large extent on the mesh representation. If the application requires all d levels of mesh entities and most of adjacencies, very little will be gained by using the flexible mesh representation.

7.2 Overhead of Function Pointers

In order to support the flexible mesh data structure, the mesh creation/deletion operators are declared not as functions, but as function pointers. We conducted experiments to evaluate the overhead generated by function pointers. We ran several applications of mesh adaptation [49, 50, 51] and discontinuous Galerkin methods [30, 74] using fixed and flexible mesh data structures, both with the one-level representation.

The overhead is measured by subtracting the run time with the fixed mesh data structure from that of the flexible mesh data structure of which mesh creation/deletion operators declared as the function pointers are set to the ones for one-level adjacencies. It has been observed that a difference in execution time between two mesh data structures, fixed and flexible, is $0-5\%$ depending on the frequency of calls to entity creation/deletion operator, which are declared with function pointers. In most cases, the overhead with function pointers was less than 1%.

If the application is mesh modification intensive, the overhead with flexible mesh data structure increases since it is strongly affected by the frequency of calls to entity creation/deletion operator. In cases where the application needs complete representation and modification intensive, it's desirable to run the application with the fixed mesh data structure to avoid the overhead from the flexibility due to function pointers. The FMDB lets the user choose between the fixed mesh data structure with one-level adjacency and the flexible mesh data structure via a flag.

7.3 Efficiency of Mesh Migration

We've performed the mesh migration procedures with the 1000K mesh to measure the run time of it. Algorithm 7.1 is a test program used to determine $POsToMove$. Given distributed mesh M, the number of partition objects to migrate per partition, n, and the number of partitions, p, it picks n partition objects and their destination partition id's using a random number generator and performs mesh migration. The total number of partition objects migrated over p partitions, N, is $n \times p$.

The run time of the migration procedure obtained with a randomly, intentionally generated mesh partitioning may be less meaningful, thus not appropriate to demonstrate the efficiency of the distributed mesh data structure developed. However it may be helpful in discussing how the migration procedure is affected by mesh representation options and various combinations of n and p. Demonstration of the efficiency of the distributed mesh data structure with the real adaptive simulations is presented in §8.1.

As the number of partitions participating in migration increases, partitioning topology becomes more complicated since more entities will be on partition boundaries and/or have more remote copies. Thus, the test is to migrate a fixed total number of partition objects with various numbers of partitions in order to see how the migration procedures are affected as the number of partition objects decreases and partitioning topologies com-

```
Data  : M, n, p
Result: migrate n × p partition objects
begin
    counter ← 0;
    for each partition object M_i^d in M do
        pid ← rand() % p;
        if pid = the local partition id
            continue; // proceed to the next partition object
        endif
        POsToMove.push_back(M_i^d, pid);
        ++counter;
        if counter = n
            break; // exit for loop
        endif
    endfor
    migrate partition objects collected in POsToMove;
end
```

Algorithm 7.1: Test program of the mesh migration procedure

plicated. Obviously, with the fixed total number of partition objects, as the number of partitions increases, the number of partition objects migrated per partition decreases.

Table 7.4 shows the run time to migrate total $100,000 - 500,000$ partition objects over $2 - 48$ partitions. The migration time is collected for 4 mesh representation options given in Table 7.2. Migration with reduced representations, $\mathcal{R}^1$ (MSR), $\mathcal{R}^2$ (REV, reduced interior faces), and $\mathcal{R}^3$ (RFV, reduced interior edges) is performed with $M\_migrate\_URR$. Migration with $\mathcal{R}^4$ (one-level representation) is performed with $M\_migrate$. In comparison between the four mesh representation options, the fastest is denoted with bold.

Comparing total migration time of $M\_migrate$ and $M\_migrate\_URR$, $M\_migrate$ with $\mathcal{R}^4$ is much faster than $M\_migrate\_URR$ when n and p are relatively small. However as n and p increase, $M\_migrate\_URR$ tends to outperform $M\_migrate$ due to the less number of entity exchanges. For the cases where the number of total partition object migrated is 400,000 or 500,000, $M\_migrate\_URR$ with $\mathcal{R}^1$ performed the best.

Table 7.4: Run time of migrating total N partition objects (sec)

		# partitions							
N	MRM	2	4	8	16	24	32	40	48
100,000	$\mathcal{R}^1$	36.28	21.24	17.35	14.88	17.08	**19.45**	**25.05**	29.13
	$\mathcal{R}^2$	28.26	19.26	16.21	13.18	17.00	19.51	25.45	29.65
	$\mathcal{R}^3$	21.81	15.28	14.16	12.11	**16.53**	20.01	26.98	**25.35**
	$\mathcal{R}^4$	**16.76**	**13.81**	**12.98**	**11.77**	17.05	21.46	27.60	26.20
200,000	$\mathcal{R}^1$	26.82	20.05	18.96	16.65	28.81	**35.46**	43.00	**31.69**
	$\mathcal{R}^2$	24.31	20.18	18.78	17.12	**26.23**	36.28	**42.15**	32.13
	$\mathcal{R}^3$	24.01	19.96	**17.91**	17.23	27.01	40.31	43.18	34.26
	$\mathcal{R}^4$	**22.79**	**19.64**	18.38	**16.01**	26.61	42.32	44.29	34.95
300,000	$\mathcal{R}^1$	36.38	**39.46**	**39.74**	**39.95**	**52.96**	66.73	69.71	**70.34**
	$\mathcal{R}^2$	35.21	40.98	40.18	39.99	53.21	**65.01**	69.23	72.31
	$\mathcal{R}^3$	**33.12**	40.26	40.00	40.21	53.96	66.98	**66.12**	78.12
	$\mathcal{R}^4$	33.47	45.61	41.93	41.82	53.74	65.35	67.13	81.18
400,000	$\mathcal{R}^1$	**45.98**	**43.97**	**45.60**	**43.10**	**61.36**	**69.28**	**72.23**	**77.32**
	$\mathcal{R}^2$	48.23	51.25	50.26	45.35	65.36	72.28	75.95	79.23
	$\mathcal{R}^3$	51.26	50.88	49.23	48.23	65.26	80.55	84.23	86.28
	$\mathcal{R}^4$	54.36	60.04	53.16	52.02	72.05	78.13	81.26	88.36
500,000	$\mathcal{R}^1$	**46.27**	**47.53**	**50.56**	**51.87**	**63.24**	**75.12**	**75.36**	**79.23**
	$\mathcal{R}^2$	51.48	56.23	55.23	52.81	65.53	76.26	75.98	80.23
	$\mathcal{R}^3$	55.64	63.54	73.54	75.34	70.34	76.34	80.34	89.34
	$\mathcal{R}^4$	64.30	75.69	81.19	76.38	82.82	85.78	90.26	93.41

CHAPTER 8
APPLICATIONS

This chapter presents the use for the FMDB (serial and parallel) in a set of mesh-based applications including its performance data.

8.1 Parallel Anisotropic 3D Mesh Adaptation

Anisotropic mesh modification [48, 49, 50, 51] provides a general mesh adaptation procedure that applies local mesh modification operations to yield a mesh of elements matching the required sizes and shapes. The mesh adaptation procedure is governed by a discrete anisotropic metric field specified at each mesh vertex of the current mesh [48, 51]. The procedure consists of the four interacted high-level components of refinement, coarsening, swapping [51], and projecting new vertices created onto curved model boundaries onto the boundaries [50].

The serial mesh modification procedure has been extended to work with the distributed meshes in parallel [1, 51].

8.1.1 Parallelizing mesh modification procedures

The key technical issues in parallelizing mesh modification procedures deal with: (*i*) evaluating[3] and executing mesh modifications on the partition boundaries and (*ii*) an effective message packing and load balancing.

Applying mesh modification operation on the partition boundary requires inter-partition communication for its evaluation and may break partition boundary links during its execution. Therefore, a technique is required to prevent the operation to break the links or follow the operation with a process to repair the broken links. Since inter-partition communications are costly, it is important for message packing to reduce the number of such communications. Thus, small pieces of messages from many mesh modifications are packed together and sent out in as few communications as possible. The load balancing is critical for the parallel procedure to achieve speed-up. In addition, performing local mesh modification operations on partition boundaries between several partitions requires parallelization of each local mesh modification operator, in particular, repartitioning the

[3]Predicting the validity or quality of the result mesh.

mesh via the mesh migration procedure to treat mesh entities on or near the partition boundaries.

The distributed mesh data structure of the FMDB provides all needed functionalities for supporting parallel mesh adaptation such as dynamic mesh load balancing, individual local mesh migration for coarsening, projecting and swapping, easy-to-customize templated communication, etc.. The mesh migration procedures combined with *the poor-to-rich ownership* maintain the mesh load balance during processes, reducing the frequency of calls to the mesh load balancing procedure. The metric field of each mesh vertex is implemented with attachable data to the mesh vertices. While migrating mesh vertices, the metric field attached to the vertex must be exchanged properly to proceed the adaptation. As discussed in §6.4.2, the Zoltan callback of the FMDB supports the functionality to customize packing/unpacking any arbitrary attached data to mesh entities in the migration procedure. Detailed discussions on parallelizing each mesh modification component are presented in reference [1].

8.1.2 Experiments

The parallel mesh adaptation procedure has been tested against a wide range of models under either analytical or adaptively defined mesh size field definitions [48]. Some results of the parallel mesh adaptation are presented to demonstrate the scalability of the distributed mesh data structure developed. The scalability of a parallel program running on p processors is defined as the *speedup*.

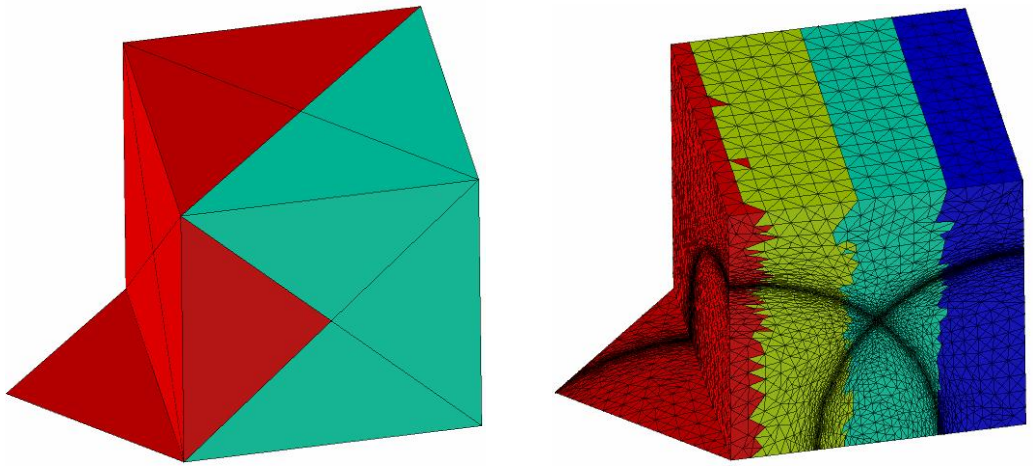
$$speedup = \frac{\text{run time on 1 processor}}{\text{run time on } p \text{ processors}} \tag{8.1}$$

The relative speedup is the speedup against the program on $\frac{p}{2}$ processors.

$$relative\ speedup = \frac{\text{run time on } \frac{p}{2} \text{ processors}}{\text{run time on } p \text{ processors}} \tag{8.2}$$

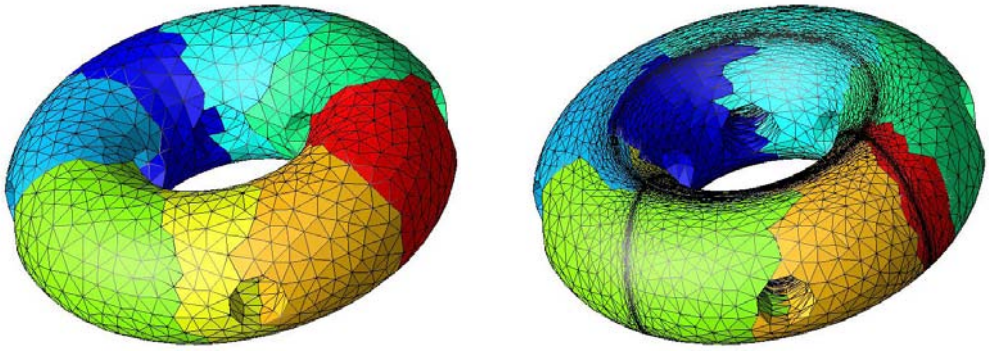
Figure 8.1 shows a uniform initial non-manifold mesh of a $1 \times 1 \times 1$ cubic and triangular surface domain and the adapted mesh with two spherical mesh size fields on 4 processors. Different color represents different partitions.

The geometry of the mesh in Figure 8.2 is a torus with four circular holes. The initial mesh is 20,067 tetrahedron. The adapted, approximately 2 million tetrahedron mesh with two spherical shocks is given with the speedup.



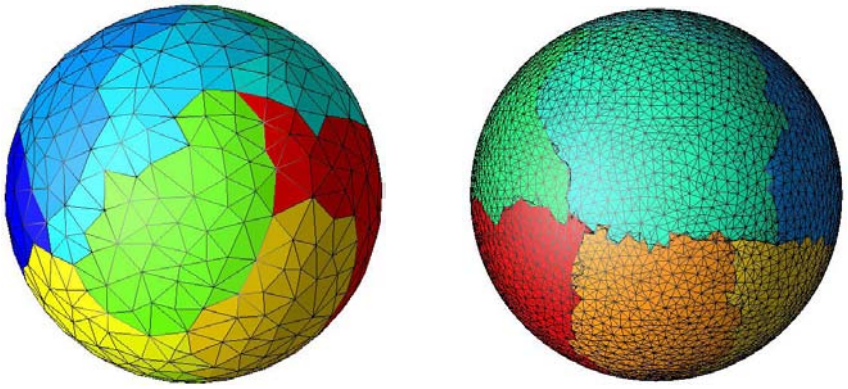
# proc	2	4	8	16
speedup	2.23	3.37	5.48	8.40
rel. speedup	2.23	1.50	1.62	1.53

Figure 8.1: Parallel mesh adaptation I: (left) initial 36 tet mesh, (right) adapted approx. 1 million tet mesh



# proc	2	4	8	16	32
speedup	1.52	2.47	4.18	8.28	18.71
rel. speedup	1.52	1.57	1.76	1.93	2.26

Figure 8.2: Parallel mesh adaptation II: (left) initial 20,067 tet mesh, (right) adapted approx. 2 million tet mesh



# proc	12	24	48
rel. speedup	-	1.86	1.87

Figure 8.3: Parallel mesh adaptation III: (left) initial 6,213 tet mesh, (right) adapted approx. 7 million tet mesh

Figure 8.3 gives the initial coarse mesh with sphere geometry and adapted mesh with the analytical size field. The final mesh is of approximately 7 million tetrahedron.

8.2 Adaptive Loop for Accelerator Design

The Stanford Linear Accelerator Center (SLAC)'s eigenmode solver Omega3P [47] is being used in the design of next generation linear accelerators. Recently, Omega3P has been integrated with adaptive mesh control [48, 49, 50, 51] to improve the accuracy and convergence of wall loss (or quality factor) calculations in accelerating cavities. The simulation procedure consists of interfacing Omega3P to automatic mesh generator, general mesh modification, and error estimator components to form an adaptive loop as depicted in Figure 8.4.

The accelerator geometries are defined as ACIS^{TM} solid models [91] and physical parameters required by the simulation are associated with geometric model entities. Using functional interfaces between geometric model and meshing techniques [8], the automatic mesh generation tools of Simmetrix^{TM} [89] generates an initial mesh. After Omega3P calculates the solution fields, the error indication procedure determines a new mesh size field and the mesh modification procedures modify the mesh to generate a new mesh for the next execution of Omega3P. This iterative procedure repeats until the desired accuracy is reached.

The FMDB is used as for a mesh database to support these processes including mesh adaptation. As seen in Figure 3.7, all d levels of mesh entities with 12 first-order

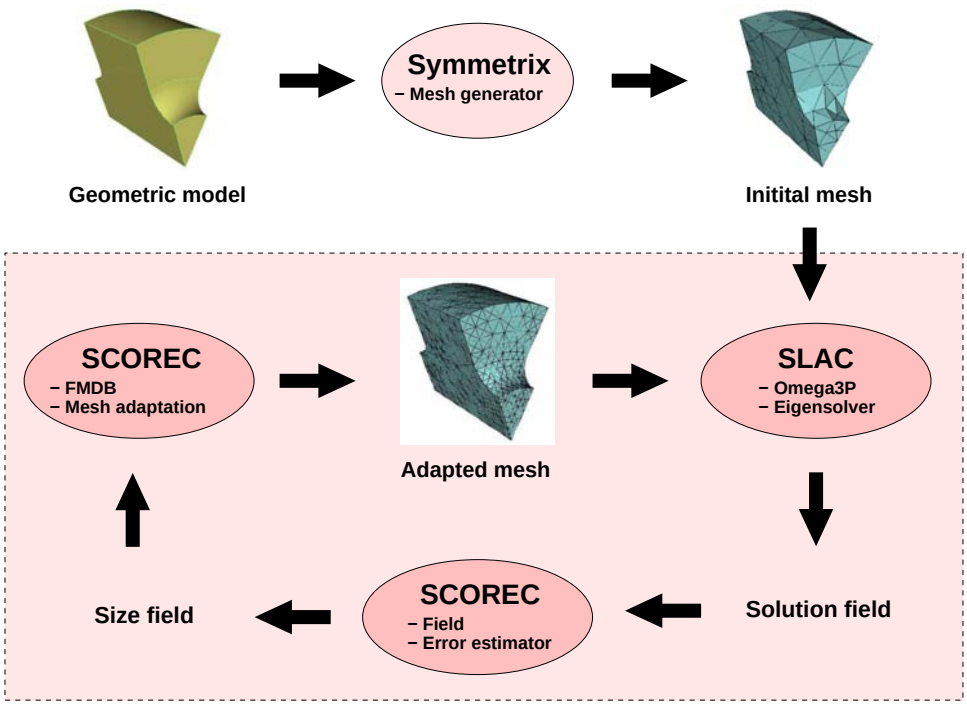


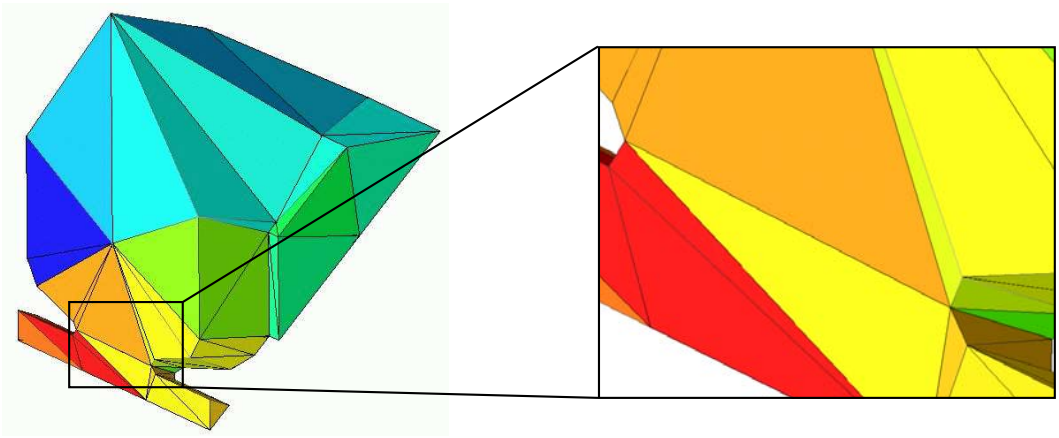
Figure 8.4: Framework of adaptive loop for accelerator design

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

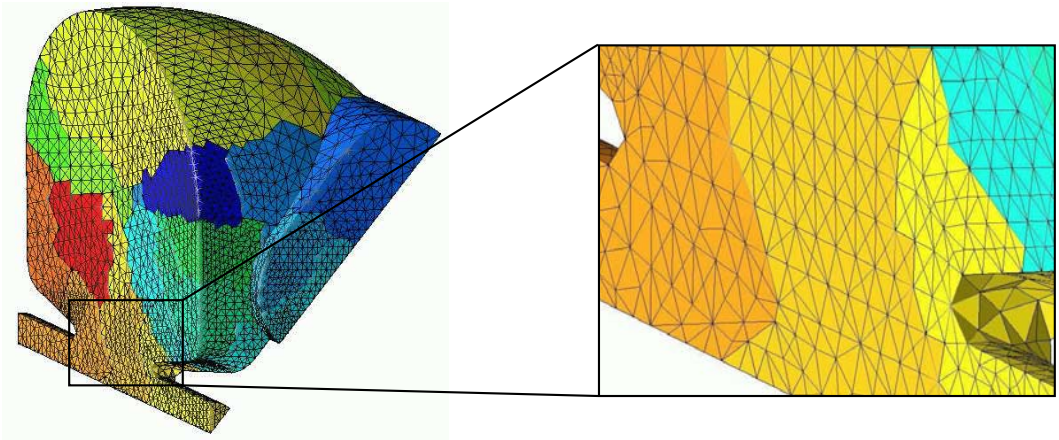
Figure 8.5: MRM's for SLAC adaptive loop: (left) requested (right) optimized

adjacencies are necessary to run the mesh adaptation procedure (note this is currently what is being used, and it would be better being considered to improve). The MRM given in the left of Figure 8.5 represents the mesh representation requested by the adaptive loop. To compromise between the memory and computational efficiency, the representation provided in the adapted loop is the one-level adjacency representation with the addition of $\{M^3\{M^0\}\}$ as given in the right of Figure 8.5. The last step of MRM optimization that suppresses unnecessary adjacencies which are obtainable by local traversal (See §3.2.2) was turned off since vertices of a region is frequently used by the adaptation procedure, thus storing them explicitly is more beneficial in time.

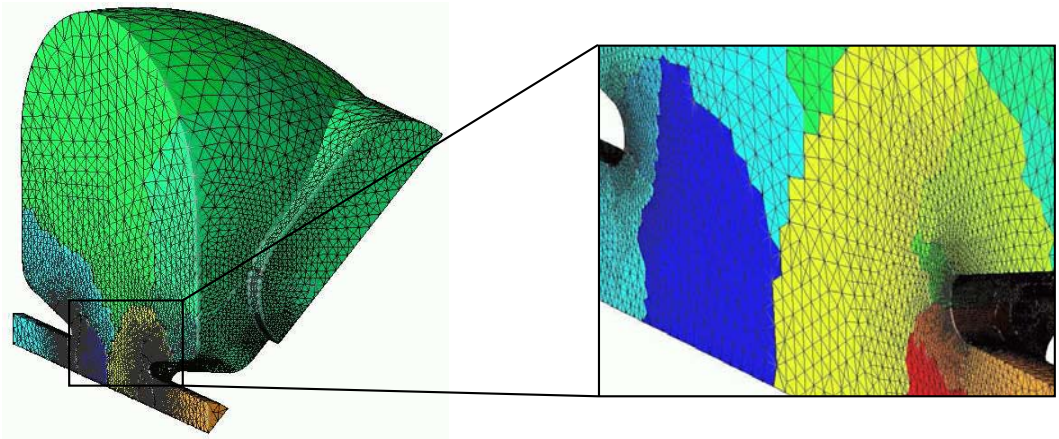
The parallel adaptive procedure has been applied to Trispal model and RFQ model. In these examples, the size fields were intentionally set to generate big meshes to demonstrate the scalability of the FMDB. The speedups given are just for the parallel mesh adaptation portion of the process.



(a) initial mesh



(b) after the 2^{nd} loop



(c) after the 8^{th} loop

# proc	20	40
rel. speedup	-	1.81

Figure 8.6: Parallel adaptive loop for SLAC I: (a) initial coarse Trispal mesh (65 tets), (b) adapted mesh after the second adaptive loop (approx. 1 million tet), (c) the final mesh converged to the solutions after the eighth adaptive loop. (approx. 12 million tets)

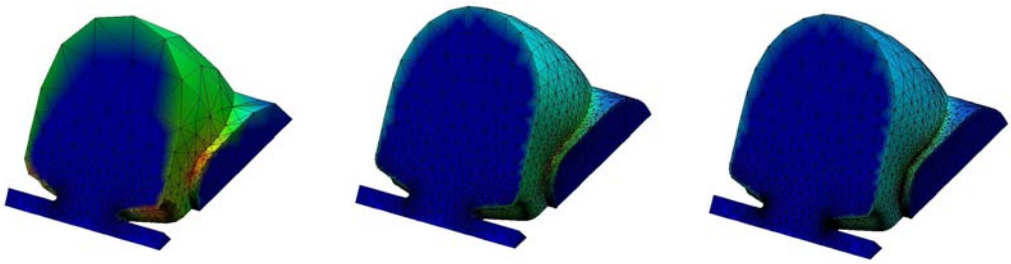
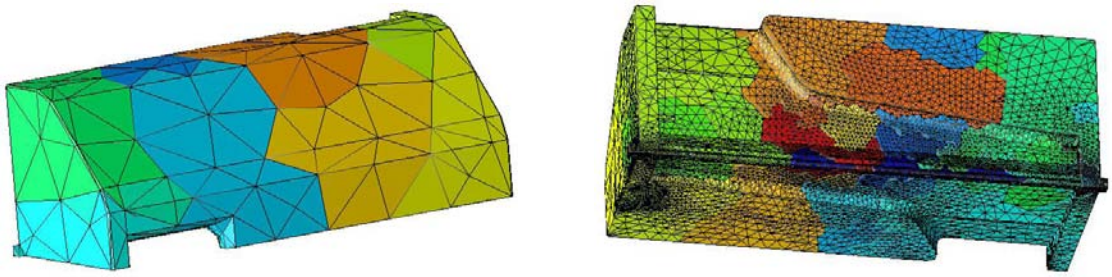
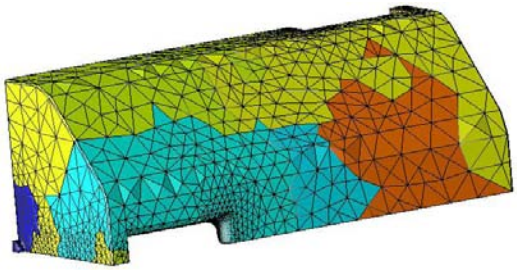


Figure 8.7: Mesh and wall-loss distribution for 3 adaptive steps

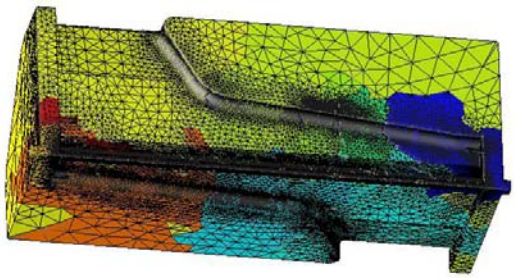


(a) initial (front)

(b) 1^{st} loop (back)



(c) 2^{nd} loop (front)



(d) 2^{nd} loop (back)

# proc	28	56
rel. speedup	-	1.97

Figure 8.8: Parallel adaptive loop for SLAC II: (a) initial coarse RFQ mesh (1,595 tet), (b) adapted mesh from the first adaptive loop (approx. 1 million tet), (c) front view of adapted mesh from the second adaptive loop (approx. 24 million tet), (d) back view of (c)

Figure 8.6 shows the Trispal meshes during the parallel adaptive loop, (a) gives the initial mesh composed of 65 tetrahedron, (b) is the adapted mesh after the second adaptive loop, which is of approximately 1 million tetrahedron, (c) is the adapted, approximately 12 million tetrahedral, mesh after the eighth adaptive loop, which converged to the solution fields.

Figure 8.7 shows the Trispal mesh and wall loss distribution on the cavity surface for three adaptive steps with an increasingly denser mesh in the area of high field concentration

Table 8.1: Relative storage cost with DG applications

	one level	flexible
2D Rayleigh Taylor	1	.93
3D Rayleigh Taylor	1	.72
2D riemann	1	.82
3D riemann	1	.66
3D sedov explosion	1	1

(from left to right).

Figure 8.8 gives the RFQ meshes during the parallel adaptive loop, (a) gives the initial coarse mesh of 1,595 tetrahedron, (b) is the adapted mesh after the first adaptive loop, which is approximately 1 million tetrahedron, (c) and (d) are the front and back view of the adapted mesh after the second adaptive loop, which contains about 24 million tetrahedron.

In both of Trispal and RFQ cavities, it has been observed that the parallel adaptive procedure reliably produces the results with the desired accuracy and quality factors.

8.3 Parallel Discontinuous Galerkin Method

The Discontinuous Galerkin (DG) method was initially introduced by Reed and Hill in 1973 [73] as a technique to solve neutron transport problems. It is now being used to solve ordinary differential equations, hyperbolic, parabolic and elliptic partial differential equations [78].

The SCOREC DG codes has been developed for solving complex phenomena featuring sharp moving fronts such as transient inviscid flows (e.g. Rayleigh-Talor instabilities [26, 35], or blast wave propagations [30, 74, 77, 78]). The FMDB has been used for a mesh database supporting conforming/non-conforming adaptive mesh refinement calculations in the DG codes. The DG codes also were parallelized to be able to run on parallel computers supported by the FMDB.

Table 8.1 gives the memory savings with various DG applications via flexible mesh representation options. There was no memory savings with 3D sedov explosion since it performs the mesh adaptation procedure which requires all d levels of entities with 12 adjacencies.

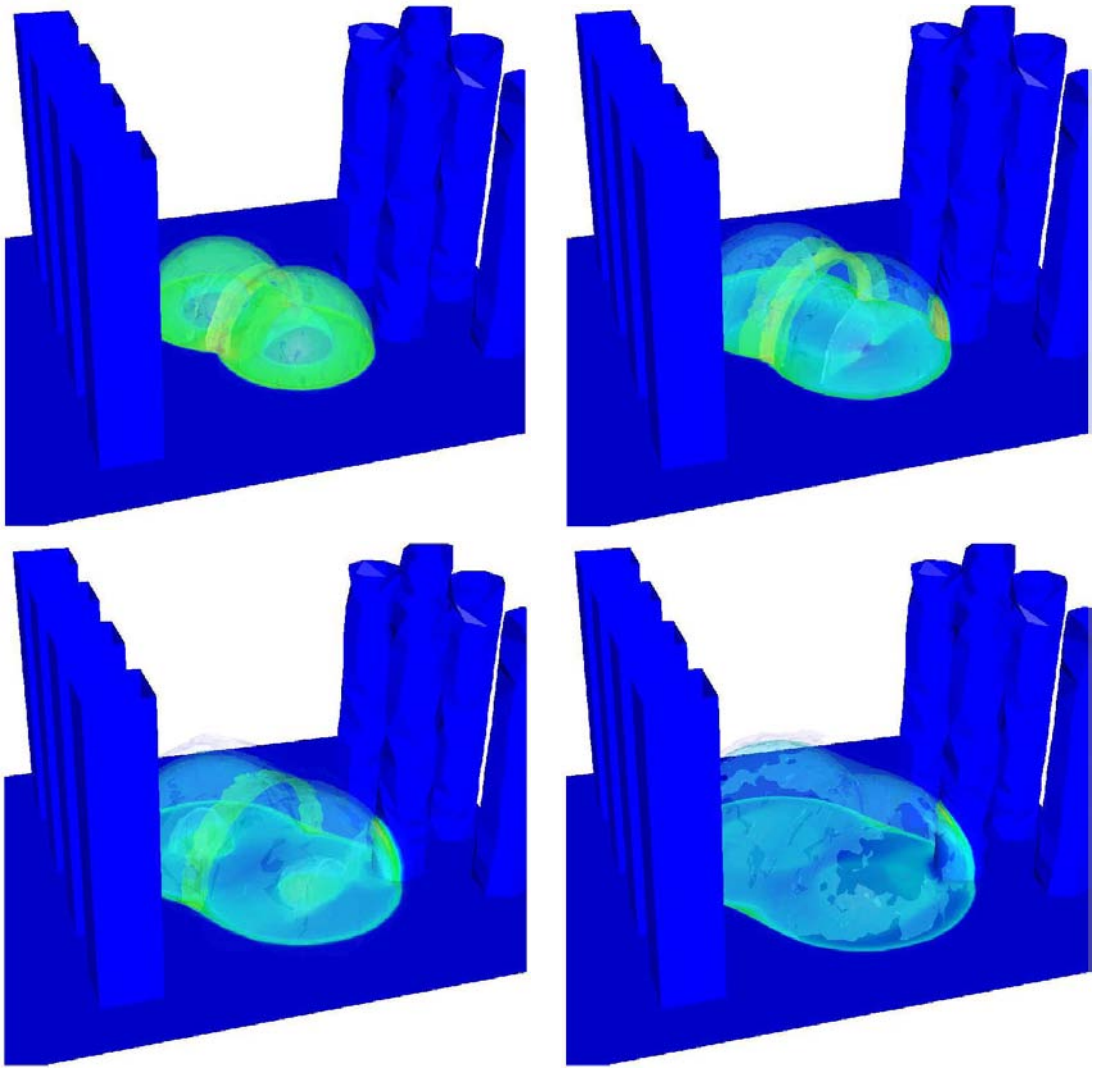


Figure 8.9: Isosurface of pressure evolution in the domain at time 0.04**,** 0.06**,** 0.08 **and** 0.1 **(from left to right and top to bottom)**

8.3.1 A double sedov explosion simulation

The parallel DG with adaptation applied to a double sedov explosion simulation, modeled by the Euler equations, in a 3D geometry that simulates the evolution of two spherical blasts in a homogeneous medium. This problem is a purely hydro dynamical test and it involves strong shocks.

In the example, a domain with a non-dimensional size of $[-1,1] \times [-1,1] \times [0,1]$ containing several obstacles is considered. The domain is filled with an homogeneous medium. To initialize the simulation, two quantities of energy equal to 1 are considered into two small hemisphere regions with a radius 0.15 and for centers $(-0.1, 0.2, 0)$ and

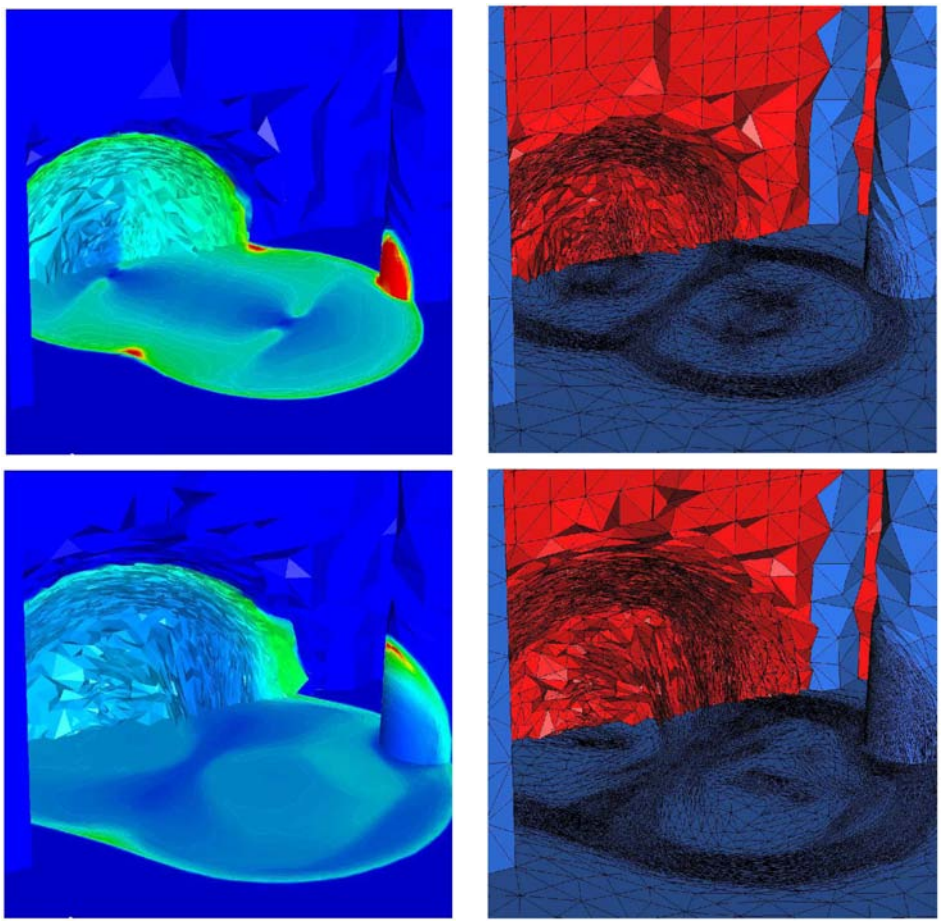


Figure 8.10: (left) Isopressure distribution at time 0.06 **and** 0.1**, (right) the associated anisotropic adapted meshes**

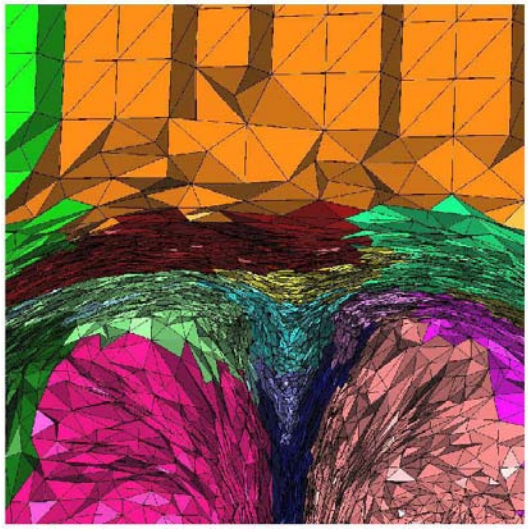


Figure 8.11: Final adapted partitioned mesh at time 0.1

$(0.1, 0.1, 0)$.

The shock waves propagation (pressure isosurface) is illustrated in Figure 8.9 at various time steps. Figure 8.10 shows the anisotropic adapted mesh and the corresponding isopressure distributions at time 0.06 and 0.1. The load balancing procedure was performed to keep even mesh distribution for each mesh partition as the mesh size evolves slowly along the adaptive computation. Figure 8.11 gives the final mesh that contains $638,677$ tetrahedron.

CHAPTER 9
CLOSING REMARKS

The aim of this thesis was to develop an efficient flexible distributed mesh data structure including parallel algorithms which are scalable for parallel automated adaptive mesh-based simulations. This chapter summarizes the contributions made in achieving the goal, and suggests directions for future work.

9.1 Research Contributions

The essential requirement for the efficient mesh data structure is the design and implementation of algorithms that manipulate the mesh and mesh entities effectively both in serial and parallel. The core of the flexible distributed mesh data structures includes the functions to construct the user-requested representations and the mesh migration procedures. All mesh entity level operations were developed to execute in O(1) time. This includes operators affected by representation such as entity creation, deletion and migration.

Data structures for distributed meshes were designed based on the hierarchical domain decomposition, providing a partition model as intermediate domain decomposition between the geometric model and the mesh. For that purpose, the definitions and properties of the partition model and relations between the distributed mesh and the partition model were identified. From the support of partitioning at the partition model level as well as optimal order algorithms to construct and use it, local mesh entity migration and dynamic load balancing are supported effectively. The distributed mesh environment was designed to handle 2D triangular and 3D tetrahedral meshes on non-manifold geometries.

The entity's owner partition rule lets the owning partition of either a duplicated entity on a partition boundary or an internal entity to the partition boundary be the partition which has the least number of entities among all partitions where the entity exists. In supporting mesh modification with entities on or near the partition boundaries, the poor-to-rich owner partition rule migrates entities to the partitions which have less load, enhancing the overall performance of parallel mesh adaptation by keeping mesh load balance.

The FMDB is supported by the DOE SciDAC [81] program through the TSTT center [95] that is developing interoperable and interchangeable meshing and discretization

software for scientific computation. The FMDB was developed to be TSTT compliant and represent core functionalities of the TSTT meshing tools. It is a TSTT component developed to fully support the extensive demands of adaptive mesh-based analyses and mesh operations on parallel computers.

The FMDB is embedded in SCOREC simulation packages effectively supporting parallel automated adaptive analyses such as parallel adaptive loop for SLAC and parallel discontinuous Galerkin methods.

9.2 Future Directions

There are many extensions that can be made to the FMDB.

- Currently, only triangular and tetrahedral elements are supported. The data structures for flexibility and migration algorithms are general and can support other types of elements such as quadrilateral or hexahedral elements with minor modifications.

- There are 2 categories in Zoltan's parallel data management algorithms: coordinate and graph-based. Currently, the load balancing procedure works only with graph-based algorithms, especially ParMETIS [40]. In the load balancing procedure, the part that interfaces with Zoltan is being modified to work with extensive set of load balancing libraries included in Zoltan, either coordinate or graph-based.

- It is also worthwhile to explore other types of links which point to remote copies of duplicated entities on remote partitions. Currently, full links, as described in §4.2.2, are used. With full links, the remote copy update involves all of the remote copies of the duplicated entities. Further optimizations in terms of memory and communication volume reduction in remote copy updates are possible through the use of compressed (minimal) links.

- In order to support more effective load balancing on heterogeneous clusters which expose a growing heterogeneity in processing and communication capabilities, the FMDB is being combined with the Dynamic Resource Utilization Module (DRUM) [29]. The DRUM supports hierarchical and architecture-dependent load balancing in conjunction with the Zoltan toolkit. By using the DRUM underneath, any of the applications that use Zoltan can better tailor partitions to a given architecture.

- Even through the FMDB was developed to be TSTT compliant, its implementation is at the initial stage. Extensive unit testing and development of applications that

access the FMDB through the TSTT Mesh API are necessary to ensure the FMDB is an interoperable/interchangeable component in the mesh-based analysis environment. The current working plan includes running Mesquite [95] the mesh quality improvement library and SCOREC mesh adaptation procedure [48, 49, 50, 51] through the TSTT Mesh API of the FMDB.

- The interest when solving a partial differential equation (PDE) on a parallel computer is obtaining a solution with a prescribed error tolerance in minimal time rather than in speedup [66, 87]. Hence the success of parallel PDE solver should be measured in terms of error attained and the time it takes. The experiments for the success with parallel PDE solver were not covered in this thesis, thus this will need to be done in the near future.

- Currently, the mesh I/O is performed through text formatted files. It would be better to have binary formatted file I/O for better performance. The most commonly used library for binary formatted data access is the Network Common Data Form (NetCDF) [97]. The NetCDF is an interface for array-oriented data access and a library that provides an implementation of the interface. The NetCDF library also defines a machine-independent format for representing scientific data such as meshes. Together, the interface, library, and format support the creation, access, and sharing of scientific data.

- Parallel mesh I/O discussed in §6.3.4 assumed that the partition model is kept the same before and after the partitioned mesh is recovered from the mesh files. However, for more flexibility, it's possible to have different partition models before and after, for example a different number of partitions. Further investigation will be needed to realize this functionality.

LITERATURE CITED

[1] Alauzet F, Li X, Seol ES, Shephard MS (2005) Parallel anisotropic 3D mesh adaptation by mesh modification *In preparation to submit to J. Computing and Information Science.*

[2] Alexandrescu A (2001) Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley.

[3] Argonne National Laboratory (2005) The Message Passing Interface (MPI) standard library. http://www-unix.mcs.anl.gov/mpi.

[4] Balsara DS, Norton CD (2001) Highly parallel structured adaptive mesh refinement using parallel language-based approaches *Parallel Computing* 27:37-70.

[5] Beall MW (1999) An object-oriented framework for the reliable automated solution of problems in mathematical physics. PhD Dissertation Mechanical Engineering Dept. Rensselaer Polytechnic Institute Troy NY.

[6] Beall MW, Shephard MS (1997) A general topology-based mesh data structure *Int. J. Numer. Meth. Engng* 40:1573-1596.

[7] Beall MW, Shephard MS (1999) An object-oriented framework for the reliable numerical simulations *Engineering with Computers* 20(3):210-221.

[8] Beall MW, Walsh J, Shephard MS (2004) A comparison of techniques for geometry access related to mesh generation *Engineering with Computers* 20(3):210-221.

[9] Biswas R, Oliker L (1994) A new procedure for dynamic adaptation of three-dimensional unstructured grids *Appl. Numer. Math* 13:437-452.

[10] Bonet J, Peraire J (1991) An alternating digital tree (ADT) algorithm for 3D geometric and intersection problems *Int. J. Numer. Meth. Engng* 31:1-17.

[11] Booch G (1994) Object-Oriented Analysis and Design with Applications. The Benjamin/Cummings Publishing Company Inc. Redwood City, CA.

[12] Booch G, Jacobson I, Rumbaugh J (1999) Unified Modeling Language for Object-Oriented Development Documentation Set Version 0.91 Addendum, Rational Software Corp. Santa Clara, CA.

[13] Celes W, Paulino GH, Espinha R (2005) A compact adjacency-based topological data structure for finite element mesh representation *Int. J. Numer. Meth. Engng* (in press).

[14] Center for component technologies for terascale simulation science (2005) SIDL/Babel user's guide. http://www.llnl.gov/CASC/components/babel.html.

[15] Chand K, Diachin LF, Fix B, Ollivier-Gooch C, Seol ES, Shephard MS, Tauges T (2005) Toward interoperable mesh, geometry, and field components for PDE simulation development *Submitted to Engineering with Computers.*

[16] Chellamuthu KC, Ida N (1994) Algorithms and data structures for 2D and 3D adaptive finite element mesh refinement *Finite Elements in Analysis and Design* 17(3)205-229.

[17] Chen J, Taylor VE (2000) ParaPART: parallel mesh partitioning tool for distributed systems *Concurrency: Pract. Exper* 12:111-123.

[18] Cockburn B, Karniadakis G, Shu CW (2000) Discontinuous galerkin methods *Vol. 11 of Lecture Notes in Computational Science and Engineering* Berlin.

[19] Cormen TH, Leiserson CE, Rivest RL, Stein C (2001) Introduction to Algorithm 2^{nd} Ed.. MIT Press.

[20] de Cougny HL, Devine KD, Flaherty JE, Loy RM, Özturan C, Shephard MS (1995) Load balancing for the parallel solution of partial differential equations *Appl. Numer. Math* 16:157-182.

[21] de Cougny HL, Shephard MS (1999) Parallel refinement and coarsening of tetrahedral meshes *Int. J. Numer. Meth. Engng* 46:1101-1125.

[22] de Cougny HL, Shephard MS (1999) Parallel unstructured grid generation *CRC Handbook of Grid Generation* Thompson JF, Soni Bk, Wetherill NP, Eds. CRC Press Inc. Boca Raton p24.1-24.18.

[23] de Cougny HL, Shephard MS, Özturan C (1996) Parallel three-dimensional mesh generation on distributed memory (MIMD) computers *Engineering with Computers* 12(2):94-106.

[24] Deitel & Deitel (2001) C++ How To Program 2^{nd} Ed.. Prentice Hall.

[25] Diekmann R, Preis R, Schlimbach F, Walshaw C (2000) Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM *Parallel Computing* 26:1555-1581.

[26] Drazin PG, Reid WH (1982) Hydrodynamic Stability. Cambridge University Press.

[27] DUNE - A unified framework for scientific computing (2005) http://dune.uni-hd.de.

[28] Eletronic Data Systems Corp. (1994) Parasolid V6 Functional Description. Maryland Heights MO 63043.

[29] Faik J, Flaherty JE, Gervasio LG, Teresco JD, Devine KD, Boman EG (2005) A model for resource-aware load balancing on heterogeneous clusters. http://www.cs.williams.edu/drum.

[30] Flaherty JE, Krivodonova L, Remacle JF, Shephard MS (2002) Aspects of discontinuous Galerkin methods for hyperbolic conservation laws *Comp. Meth. Appl. Mech. Engng* 38:889-908.

[31] Galimella RV (2002) Mesh data structure for mesh generation and FEA applications *Int. J. Numer. Meth. Engng* 55:451-478.

[32] Galimella RV (2004) MSTK - A flexible infrastructure library for developing mesh based applications *13th International Meshing Roundtable, Sandia National Laboratories* p203-212. Available from http://www.andrew.cmu.edu/user/sowen/topics/data.html.

[33] Gamma E, Johnson R, Helm R, Vlissides JM (1994) Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

[34] Ge L, Lee LQ, Li Z, Ng C, Ko K, Luo Y, Shephard MS (2004) Adaptive mesh refinement for high accuracy wall loss determination in accelerating cavity design *Eleventh Biennial IEEE Conference on Eletromagnetic Field Computation, Seoul, Korea.*

[35] Glimm J, Grove J, Li X, Oh W, Tan DC (1988) The dynamics of bubble growth for Rayleigh-Taylor unstable interfaces *Physics of Fluids* 31:447-465.

[36] Hawken DM, Townsend P, Webster MF (1992) The use of dynamic structures in finite element applications *Int. J. Numer. Meth. Engng* 33:1795-1811.

[37] Hughes TJR (2000) The Finite Element Method: Linear Static and Dynamic Finite Element Analysis. Dover Publication Inc..

[38] Joint Institute for Computational Science (1997) Beginner's Guide to MPI Message Passing Interface. University of Tennessee.

[39] Kallinderis Y, Kavouklis C (2004) A dynamic adaptation scheme for general 3-D hybrid meshes *Comp. Meth. Appl. Mech. Engng* (in press).

[40] Karypis G, Schloegel K, Kumar V (1998) ParMETIS: Parallel Graph Partitioning & Sparse Matrix Ordering Library V. 2.0. University of Minnesota, Computer Science Dept. Army HPC Research Center Minneapolis MN.

[41] Koenig A, Moo BE (2001) Accelerated C++. Addison-Wesley.

[42] Kohn SR, Baden SB (2001) Parallel software abstractions for structured adaptive mesh methods *J. of Parallel and Distributed Computing* 61(6):713-736.

[43] LAM/MPI Parallel Computing (2005) http://www.lam-mpi.org.

[44] Lämmer L, Burghardt M (2000) Parallel generation of triangular and quadrilateral meshes *Advances in Engineering Software* 31(12):929-936.

[45] Larwood BG, Weatherill NP, Hassan O, Morgan K (2003) Domain decomposition approach for parallel unstructured mesh generation *Int. J. Numer. Meth. Engng* 58(2)2:177-188.

[46] Le Saint P (1975) Sur la résolution des systèmes hyperboliques du premier order par la méthode des éléments finis. PhD Dissertation Université Pierre et Marie Curie.

[47] Lee LQ, et al. (2004) Solving large sparse linear systems in end-to-end accelerator structure simulations. SLAC-PUB-10320 January.

[48] Li X (2003) Mesh modification procedures for general 3D non-manifold domains. PhD Dissertation Mechanical Engineering Dept. Rensselaer Polytechnic Institute Troy NY.

[49] Li X, Remacle JF, Nicolas C, Shephard MS (2004) Anisotropic mesh gradation control 13^{th} *International Meshing Roundtable, Williamsburg VA*.

[50] Li X, Shephard MS, Beall MW (2002) Accounting for curved domains in mesh adaptation *Int. J. Numer. Meth. Engng* 58:247-276.

[51] Li X, Shephard MS, Beall MW (2003) 3D Anisotropic mesh adaptation by mesh modifications *Submitted to Comp. Meth. Appl. Mech. Engng.*

[52] libMesh: Parallel data structures for finite element computations (2005) http://www.cfdlab.ae.utexas.edu.

[53] Lo SH, Wang WX (2002) An algorithm for the intersection of quadrilateral surfaces by tracing of neighbours *Comp. Meth. Appl. Mech. Engng* 192:2319-2338.

[54] Löhner R (1988) Some useful data structures for the generation of unstructured grids *Comm. appl. numer. methods* 4:123-135.

[55] Loy R (2000) Autopack User Manual. Science Division Argonne National Laboratory.

[56] Luo X, Shephard MS, Remacle JF, O'Bara RM, Beall MW, Szabó BA, Actis R (2002) p-version mesh generation issues 11^{th} *International Meshing Roundtable, Sandia National Laboratories* p343-354.

[57] MacNeice P, Olson KM, Mobarry C, Fainchtein RD, Packer C (2000) PARAMESH: A parallel adaptive mesh refinement community toolkit *Computer Physics Communications* 126(3):330-354.

[58] Mäntylä M (1988) An Introduction to Solid Modeling. Computer Science Press Rockville Maryland.

[59] Meyers S (2000) Effective C++ 2^{nd} Ed.. Addition-Wesley.

[60] Meyers S (2001) Effective STL. Addison-Wesley.

[61] Mohamed SA (1997) Automatic mesh refinement and data structure for multigrid finite elements techniques *Computers & Stuctures* 65(6):958-993.

[62] Musser DR (2005) Generic programming. http://www.cs.rpi.edu/∼musser/gp.

[63] Musser DR, Derge GJ, Saini A (2001) STL Tutorial and Reference Guide 2^{nd} Ed.. Addison-Wesley.

[64] Oliker L, Biswas R, Gabow HN (2000) Parallel tetrahedral mesh adaptation with dynamic load balancing *Parallel Computing* 26:1583-1608.

[65] Ollivier-Gooch C (2005) GRUMMP: Generation and Refinement of Unstructured, Mixed-element Meshes in Parallel. http://tetra.mech.ubc.ca/GRUMMP.

[66] Özturan C, de Cougny HL, Shephard MS, Flaherty JE (1994) Parallel adaptive mesh refinement and redistribution on distributed memory computers *Comp. Meth. Appl. Mech. Engng* 119:123-127.

[67] Pacheco PS (1997) Parallel Programming with MPI. Morgan Kaufmann Publisher.

[68] Parametric Technologies Corp. (1997) Pro/TOOLKIT reference manual 128 Technology Drive. Waltham MA.

[69] Parashar M, Browne JC (2005) DAGH: Data Management for Parallel Adaptive Mesh Refinement Techniques. http://www.caip.rutgers.edu/∼parashar/DAGH.

[70] Park Y, Kwon O (2005) A parallel unstructured dynamic mesh adaptation algorithm for 3-D unsteady flows *Int. J. Numer. Meth. Fluids* 48:671-690.

[71] Provatas N, Goldenfeld N, Dantzig J (1998) Adaptive mesh refinement computation of solidification microstructures using dynamic data structures *J. Computational Physics* 148:265-290.

[72] Ravenbrook (2005) The memory management reference. http://www.memorymanagement.org.

[73] Reed W, Hill T (1973) Triangular mesh methods for the neutron transport equation. Tech. Report LA-UR-73-479 Los Alamos Scientific Laboratory.

[74] Remacle JF, Flaherty JE, Shephard MS (2003) An adaptive discontinuous Galerkin technique with an orthogonal basis applied compressible flow problems *SIAM Review* 45(1):53-72.

[75] Remacle JF, Karamete BK, Shephard MS (2003) Algorithm oriented mesh database *Int. J. Numer. Meth. Engng* 58:349-374.

[76] Remacle JF, Klaas O, Flaherty JE, Shephard MS (2002) A parallel algorithm oriented mesh database *Engineering with Computers* 18:274-284.

[77] Remacle JF, Li X, Shephard MS, Flaherty JE (2005) Anisotropic adaptive simlation of transient flows using discontinuous Galerkin methods *Int. J. Numer. Meth. Engng* 62:899-923.

[78] Remacle JF, Pinchedez K, Flaherty JE, Shephard MS (2002) An efficient local time stepping-discontinuous Galerkin scheme for adaptive transient computations *Submitted to Comp. Meth. Appl. Mech. Engng.*

[79] Said R, Weatherill NP, Morgan K, Verhoeven NA (1999) Distributed parallel Delaunay mesh generation *Comp. Meth. Appl. Mech. Engng* 177:109-125.

[80] Sandia National Labratories (2005) Zoltan: data-management services for parallel applications. http://www.cs.sandia.gov/Zoltan.

[81] SciDAC: Scientific Discovery through Advanced Computing (2005) http://www.scidac.org.

[82] Scott ML (2000) Programming Language Pragmatics. Morgan Kaufmann Publisher.

[83] Selwood PM, Berzins M (1999) Parallel unstructured tetrahedral mesh adaptation: algorithms, implementation and scalability *Concurrency: Pract. Exper* V11(14) 863-884.

[84] Sgi Inc. (2005) Standard Template Library. http://www.sgi.com/tech/stl/stl_index.html.

[85] Shephard MS (2000) Meshing environment for geometry-based analysis *Int. J. Numer. Meth. Engng* 47:169-190.

[86] Shephard MS, Fischer P, Chand KK, Flaherty JE (2003) Simulation information structures. http://www.tstt-scidac.org.

[87] Shephard MS, Flaherty JE, Bottasso CL, de Cougny HL, Özturan C, Simone ML (1997) Parallel automated adaptive analysis *Parallel Computing* 23:1327-1347.

[88] Shephard MS, Seol ES, Wan J, Bauer AC (2004) Component-based adaptive mesh control procedures *Conference on Analysis, Modeling and Computation on PDE and Multiphase Flow* Stony Brook NY.

[89] Simmetrix Inc. (2005) Simulation modeling suite. http://www.simmetrix.com.

[90] Soetebier I, Birthelmer H, Sahm J, Luckas V (2004) Managing large progressive meshes *Computers & Graphics* 28:691-701.

[91] Spatial Technology Inc. (2005) http://www.spatial.com/components/acis.

[92] Tautges T (2004) MOAM-SD: integrated structured and unstructured mesh representation *Engineering with Computers* 20:286-293. http://www.sass1693.sandia.gov/cubit.

[93] Tautges T, Ernst C, Merkley K, Meyers R, Stimpson C (2005) Mesh Oriented datABase (MOAB) http://cubit.sandia.gov/MOAB.

[94] Teresco JD, Beall MW, Flaherty JE, Shephard MS (2000) A hierarchical partition model for adaptive finite element computations *Comput. Methods Appl. Mech. Engrg* 184:269-285.

[95] The SciDAC Terascale Simulation Tools and Technology (TSTT) center (2005) http://www.tstt-scidac.org.

[96] Topping BHV, Cheng B (1999) Parallel and distributed adaptive quadrilateral mesh generation *Computers & Structures* 73:519-536.

[97] Unidata Program Center (2005) Network Common Data Form (NetCDF) http://my.unidata.ucar.edu/content/software/netcdf/index.html.

[98] Vandevoorde D, Josuttis NM (2003) C++ Templates. Addison-Wesley.

[99] Walshaw C, Cross M (2000) Parallel optimisation algorithms for multilevel mesh partitioning *Parallel Computing* 26(12):1635-1660.

[100] Waltz J (2002) Derived data structure algorithms for unstructured finite element meshes *Int. J. Numer. Meth. Engng* 54:945-963.

[101] Weiler KJ (1988) The radial-edge structure: a topological representation for non-manifold geometric boundary representations *Geometric Modeling for CAD Applications* p3-36.

[102] Weiler KJ (1986) Topological Structures for Geometric Modeling. PhD Dissertation Mechanical Engineering Dept. Rensselaer Polytechnic Institute Troy NY.

[103] Zienkiewicz OC (1997) The Finite Element Method 3rd Ed.. McGraw-Hill.

[104] Zienkiewicz OC, Zhu JZ (1992) The superconvergent patch recovery and a posteriori error estimates. Part 1: The Recovery Technique *Int. J. Numer. Meth. Engng* 33:1331-1364.

APPENDIX A
ALGORITHMS OF MESH OPERATORS WITH GREEDY ADJACENCY

Data : M_i^0, M_j^0
Result: M_i^1 bounded by M_i^0, M_j^0
begin
  GET($\{M_i^0\{M^1\}\}$); /* *14 steps* */
  for *each* $M_i^1 \in \{M_i^0\{M^1\}\}$ /* *repeat 14 times* */ **do**
    GET($\{M_i^1\{M^0\}\}$); /* *2 steps* */
    if *FIND*($\{M_i^1\{M^0\}\}$, M_j^0) = *true* /* $\lg 2$+*1 steps* */
      RETURN M_i^1; /* *1 step* */
    endif
  endfor
  RETURN $NULL$;
end

/* Time = *14+14·(2+2)+1 = 71* */

Algorithm A.1: Greedy adjacency: E_exist(M_i^0, M_j^0)

Data : M_i^0, M_j^0, M_k^0
Result: M_i^2 bounded by M_i^0, M_j^0, M_k^0
begin
  $M_i^1 \leftarrow$ E_exist(M_i^0, M_j^0); /* *71+1 steps* */
  GET($\{M_i^1\{M^2\}\}$); /* *5 steps* */
  for *each* $M_i^2 \in \{M_i^1\{M^2\}\}$ /* *repeat 5 times* */ **do**
    GET($\{M_i^2\{M^0\}\}$); /* *3 steps* */
    if *FIND*($\{M_i^2\{M^0\}\}$, M_k^0) = *true* /* $\lg 3$+*1 steps* */
      RETURN M_i^2; /* *1 step* */
    endif
  endfor
  RETURN $NULL$;
end

/* Time = *72+5+5(3+*($\lg 3$+*1))+1* $\approx$ *106* */

Algorithm A.2: Greedy adjacency: F_exist(M_i^0, M_j^0, M_k^0)

Data : M_i^1, M_j^1, M_k^1

Result: M_i^2 bounded by M_i^1, M_j^1, M_k^1

begin
  GET($\{M_i^1\{M^2\}\}$); /* *5 steps* */
  for *each* $M_i^2 \in \{M_i^1\{M^2\}\}$ /* *repeat 5 times* */ **do**
    GET($\{M_i^2\{M^1\}\}$); /* *3 steps* */
    if *FIND*($\{M_i^2\{M^1\}\}$, M_j^1) = *FIND*($\{M_i^2\{M^1\}\}$, M_k^1) = *true* /* *2*($\lg 3$ + *1) steps* */
      RETURN M_i^2; /* *1 step* */
    endif
  endfor
  RETURN $NULL$;
end

/* Time = *5+5(3+2*($\lg 3$*+1))+1* $\approx$ *47* */

Algorithm A.3: Greedy adjacency: F_exist(M_i^1, M_j^1, M_k^1)

Data : M_i^0, M_j^0, M_k^0, M_l^0

Result: M_i^3 bounded by M_i^0, M_j^0, M_k^0, M_l^0

begin
  $M_i^2 \leftarrow$ F_exist(M_i^0, M_j^0, M_k^0); /* *107+1 steps* */
  GET($\{M_i^2\{M^3\}\}$); /* *2 steps* */
  for *each* $M_i^3 \in \{M_i^2\{M^3\}\}$ /* *repeat 2 times* */ **do**
    GET($\{M_i^3\{M^0\}\}$); /* *4 steps* */
    if *FIND*($\{M_i^3\{M^0\}\}$, M_l^0) = *true* /* $\lg 4$*+1 steps* */
      RETURN M_i^3; /* *1 step* */
    endif
  endfor
  RETURN $NULL$;
end

/* Time = *108+2+2(4+(2+1))+1 = 125* */

Algorithm A.4: Greedy adjacency: R_exist(M_i^0, M_j^0, M_k^0, M_l^0)

Data : M_i^0, M_j^0

Result: create and return M_i^1 bounded by M_i^0 and M_j^0

begin

CREATE an edge M_i^1; */* 1 step */*

PUT({M{M^1}}, M_i^1); */* 1 step */*

/ update link from edge to vertex */*

$M_i^1\{M^0\}_0 \leftarrow M_i^0$; */* 1 step */*

$M_i^1\{M^0\}_1 \leftarrow M_j^0$; */* 1 step */*

/ update link from vertex to edge */*

PUT_UNIQ({$M_i^0\{M^1\}$}, M_i^1); */* $\lg 14$+2 steps */*

PUT_UNIQ({$M_j^0\{M^1\}$}, M_i^1); */* $\lg 14$+2 steps */*

RETURN M_i^1; */* 1 step */*

end

/ Time = 1+1+1+1+2($\lg 14$+2)+1 $\approx$ 16 */*

Algorithm A.5: Greedy adjacency: M_createE(M_i^0, M_j^0)

Data : M_i^0, M_j^0, M_k^0

Result: create M_i^2 bounded by M_i^0, M_j^0, M_k^0

begin

CREATE a face M_i^2; */* 1 step */*

PUT({M{M^2}}, M_i^2); */* 1 step */*

/ update link from face to vertex and vice versa */*

$M_i^2\{M^0\}_0 \leftarrow M_i^0$; */* 1 step */*

$M_i^2\{M^0\}_1 \leftarrow M_j^0$; */* 1 step */*

$M_i^2\{M^0\}_2 \leftarrow M_k^0$; */* 1 step */*

PUT_UNIQ({$M_i^0\{M^2\}$}, M_i^2); */* $\lg 35$+2 steps */*

PUT_UNIQ({$M_j^0\{M^2\}$}, M_i^2); */* $\lg 35$+2 steps */*

PUT_UNIQ({$M_k^0\{M^2\}$}, M_i^2); */* $\lg 35$+2 steps */*

/ update link from face to edge - 71+1 steps for each */*

$M_i^2\{M^1\}_0 \leftarrow$ E_exist(M_i^0, M_j^0);

$M_i^2\{M^1\}_1 \leftarrow$ E_exist(M_j^0, M_k^0);

$M_i^2\{M^1\}_2 \leftarrow$ E_exist(M_i^0, M_k^0);

/ update link from edge to face - $\lg 5$+2 steps for each */*

PUT_UNIQ({$M_i^2\{M^1\}_0\{M^2\}$}, M_i^2);

PUT_UNIQ({$M_i^2\{M^1\}_0\{M^2\}$}, M_i^2);

PUT_UNIQ({$M_i^2\{M^1\}_0\{M^2\}$}, M_i^2);

RETURN M_i^2; */* 1 step */*

end

/ Time = 1+1+3(1+$\lg 35$+2)+3(71+1)+3($\lg 5$+2)+1 $\approx$ 256 */*

Algorithm A.6: Greedy adjacency: M_createF(M_i^0, M_j^0, M_k^0)

Data : M_i^1, M_j^1, M_k^1

Result: create M_i^2 bounded by M_i^1, M_j^1, M_k^1

begin

CREATE a face M_i^2; */* 1 step */*

PUT({M{M^2}}, M_i^2); */* 1 step */*

/ update link from face to edge and face to vertex*/*

$M_i^2\{M^1\}_0 \leftarrow M_i^1$; */* 1 step */*

$M_i^2\{M^0\}_0 \leftarrow M_i^1\{M^0\}_0$; */* 2 (GET and ASSIGN) steps */*

$M_i^2\{M^0\}_1 \leftarrow M_i^1\{M^0\}_1$; */* 2 steps */*

if *dir[0]=1 /* 1 step */*

  if *FIND*($\{M_j^1\{M^0\}\}$, M_i^0) = *true /** $\lg 2$*+1 steps */*

    $M_i^2\{M^1\}_1 \leftarrow M_k^1$; */* 1 step */*

    $M_i^2\{M^1\}_2 \leftarrow M_j^1$; */* 1 step */*

  else

    $M_i^2\{M^1\}_1 \leftarrow M_j^1$;

    $M_i^2\{M^1\}_2 \leftarrow M_k^1$;

  endif

else

  if *FIND*($\{M_j^1\{M^0\}\}$, M_i^0) = *true*

    $M_i^2\{M^1\}_1 \leftarrow M_j^1$;

    $M_i^2\{M^1\}_2 \leftarrow M_k^1$;

  else

    $M_i^2\{M^1\}_1 \leftarrow M_k^1$;

    $M_i^2\{M^1\}_2 \leftarrow M_j^1$;

  endif

endif

$M_i^2\{M^0\}_2 \leftarrow M_i^2\{M^1\}_2\{M^0\}_1$; */* 3 (2 GET's 1 ASSIGN) steps */*

/ update link from vertex to face and edge to face */*

for var $\leftarrow$ *0 to 2 /* repeat 3 times */* **do**

  PUT_UNIQ($\{M_i^2\{M^0\}_{var}\{M^2\}\}$, M_i^2); */* $\lg 35$+2 steps */*

  PUT_UNIQ($\{M_i^2\{M^1\}_{var}\{M^2\}\}$, M_i^2); */* $\lg 5$+2 steps */*

endfor

RETURN M_i^2; */* 1 step */*;

end

*/** Time = *1+1+1+2+2+1(if dir)+2(if FIND)+1+1+3+3(*$\lg 35$*+2+*$\lg 5$*+2)+1* $\approx$ *50 */*

Algorithm A.7: Greedy adjacency: M_createF(M_i^1, M_j^1, M_k^1, *dir*[3])

sData : M_i^0, M_j^0, M_k^0, M_l^0

Result: create M_i^3 bounded by M_i^0, M_j^0, M_k^0, M_l^0

begin

CREATE a region M_i^3; /* *1 step* */

PUT($\{\mathrm{M}\{M^3\}\}$, M_i^3); /* *1 step* */

/* *update link from vertex to region and vice versa* */

for var ← *i to l /* repeat 4 times */* **do**

PUT_UNIQ($\{M_{var}^0\{M^3\}\}, M_{var}^0$); /* $\lg 23$*+2* */

$\{M_i^3\{M^0\}\} \leftarrow M_{var}^0$; /* *1 step* */

endfor

/* *update* $\{M_i^3\{M^1\}\}$ *- (71+1) steps for each* */

$M_i^3\{M^1\}_0 \leftarrow$ E_exist(M_i^0,M_j^0);

$M_i^3\{M^1\}_1 \leftarrow$ E_exist(M_j^0,M_k^0);

$M_i^3\{M^1\}_2 \leftarrow$ E_exist(M_i^0,M_k^0);

$M_i^3\{M^1\}_3 \leftarrow$ E_exist(M_i^0,M_l^0);

$M_i^3\{M^1\}_4 \leftarrow$ E_exist(M_j^0,M_l^0);

$M_i^3\{M^1\}_5 \leftarrow$ E_exist(M_k^0,M_l^0);

/* *update* $\{M_i^3\{M^2\}\}$ *- (3+47+1) steps for each* */

$M_i^3\{M^2\}_0 \leftarrow$ F_exist($M_i^3\{M^1\}_0$, $M_i^3\{M^1\}_1$, $M_i^3\{M^1\}_2$));

$M_i^3\{M^2\}_2 \leftarrow$ F_exist($M_i^3\{M^1\}_0$, $M_i^3\{M^1\}_1$, $M_i^3\{M^1\}_3$));

$M_i^3\{M^2\}_1 \leftarrow$ F_exist($M_i^3\{M^1\}_1$, $M_i^3\{M^1\}_5$, $M_i^3\{M^1\}_4$));

$M_i^3\{M^2\}_3 \leftarrow$ F_exist($M_i^3\{M^1\}_0$, $M_i^3\{M^1\}_2$, $M_i^3\{M^1\}_3$));

/* *update link from edge to region -* $\lg 5$*+2 steps for each* */

for var ← *0 to 5 /* repeat 6 times */* **do**

PUT_UNIQ($\{M_i^3\{M^1\}_{var}\{M^3\}\}$, M_i^3);

endfor

/* *update link from face to region -* $\lg 2$*+2 steps for each* */

for var ← *0 to 3 /* repeat 5 times */* **do**

PUT_UNIQ($\{M_i^3\{M^2\}_{var}\{M^3\}\}$, M_i^3);

endfor

RETURN M_i^3; /* *1 step* */

end

/* Time =
1+1+4($\lg 23$*+2+1)+6(71+1)+4(3+47+1)+6(*$\lg 5$*+2)+4(*$\lg 2$*+2)+1* $\approx$ *703*
*/

Algorithm A.8: Greedy adjacency: M_createR(M_i^0, M_j^0, M_k^0, M_l^0)

Data : M_i^2, M_j^2, M_k^2, M_l^2

Result: create M_i^3 bounded by M_i^2, M_j^2, M_k^2, M_l^2

begin

CREATE a region M_i^3; */* 1 step */*

PUT({M{M^3}}, M_i^3); */* 1 step */*

/ update link from face to region and vice versa */*

for var ← *i to l /* repeat 4 times */* **do**

PUT_UNIQ({M_{var}^2{M^3}}, M_{var}^2); */* $\lg 2$+2 steps */*

{M_i^3{M^2}} ← M_{var}^2; */* 1 step */*

endfor

/ update {M_i^3{M^1}} - (14+1) steps for each */*

$M_i^3\{M^1\}_0$ ← F_commonEdge(M_i^2,M_j^2); */* See Algorithm A.12 */*

$M_i^3\{M^1\}_1$ ← F_commonEdge(M_i^2,M_k^2);

$M_i^3\{M^1\}_2$ ← F_commonEdge(M_i^2,M_l^2);

$M_i^3\{M^1\}_3$ ← F_commonEdge(M_j^2,M_l^2);

$M_i^3\{M^1\}_4$ ← F_commonEdge(M_j^2,M_k^2);

$M_i^3\{M^1\}_5$ ← F_commonEdge(M_k^2,M_l^2);

/ update {M_i^3{M^0}} - (2+9+1) steps for each */*

$M_i^3\{M^0\}_0$ ← E_commonVertex($M_i^3\{M^1\}_0$, $M_i^3\{M^1\}_2$); */* See Algorithm A.10 */*

$M_i^3\{M^0\}_1$ ← E_commonVertex($M_i^3\{M^1\}_0$, $M_i^3\{M^1\}_1$);

$M_i^3\{M^0\}_2$ ← E_commonVertex($M_i^3\{M^1\}_1$, $M_i^3\{M^1\}_2$);

$M_i^3\{M^0\}_3$ ← E_commonVertex($M_i^3\{M^1\}_3$, $M_i^3\{M^1\}_4$);

/ update link from edge to region -* $\lg 5$*+2 steps for each */*

for var ← *0 to 5 /* repeat 6 times */* **do**

PUT_UNIQ({$M_i^3\{M^1\}_{var}${M^3}}, M_i^3);

endfor

/ update link from vertex to region -* $\lg 23$*+2 steps for each */*

for var ← *0 to 3 /* repeat 4 times */* **do**

PUT_UNIQ({$M_i^3\{M^0\}_{var}${M^3}}, M_i^3);

endfor

RETURN M_i^3; */* 1 step */*

end

/ Time =* $1+1+2(\lg 2+2+1)+6{\cdot}15+4{\cdot}12+6(\lg 5+2)+4(\lg 23+2)+1 \approx 201$ **/*

Algorithm A.9: Greedy adjacency: M_createR(M_i^2, M_j^2, M_k^2, M_l^2)

Data : M_i^1, M_j^1

Result: Common vertex of M_i^1, M_j^1

begin
  GET($\{M_i^1\{M^0\}\}$); */* 2 steps */*
  for var $\leftarrow$ *1 to 2* **do**
    if *FIND*($\{M_i^1\{M^0\}\}$, $\{M_j^1\{M^0\}\}_{\text{var}}$) *= true /* 1(GET)+*$\lg 2$*+1 steps */*
      RETURN $\{M_j^1\{M^0\}\}_{var}$; */* 1 step */*
    endif
  endfor
  RETURN $NULL$;
end

*/** Time*= 2+2·3+1 = 9 */*

Algorithm A.10: E_commonVertex(M_i^1, M_j^1)

Data : M_i^2, M_j^2, M_k^2

Result: Common vertex of M_i^2, M_j^2, M_k^2

begin
  GET($\{M_i^2\{M^0\}\}$); */* 3 steps */*
  GET($\{M_j^2\{M^0\}\}$); */* 3 steps */*
  GET($\{M_k^2\{M^0\}\}$); */* 3 steps */*
  for var $\leftarrow$ *0 to 2 /* repeat 3 times */* **do**
    if *FIND*($\{M_j^2\{M^0\}\}$, $\{M_i^2\{M^0\}\}_{\text{var}}$) *= true and FIND*($\{M_j^2\{M^0\}\}$, $\{M_i^2\{M^0\}\}_{\text{var}}$) *= true /* 2(*$\lg 3$*+1) steps */*
      RETURN $\{M_j^2\{M^1\}\}_{var}$; */* 1 step */*
    endif
  endfor
  RETURN $NULL$;
end

*/** Time*= 3+3+3+3(2(*$\lg 3$*+1))+1 ≈ 25 */*

Algorithm A.11: F_commonVertex(M_i^2, M_j^2, M_k^2)

Data : M_i^2, M_j^2

Result: Common edge of M_i^2 and M_j^2

begin

GET($\{M_i^2\{M^1\}\}$); */* 3 steps */*

for var $\leftarrow$ *0 to 2 /* repeat 3 times */* **do**

if *FIND*($\{M_i^2\{M^1\}\}$, $\{M_j^2\{M^1\}\}_{\text{var}}$) *= true /* 1+*$\lg 3$*+1 steps */*

RETURN $\{M_j^2\{M^1\}\}_{var}$; */* 1 step */*

endif

endfor

RETURN $NULL$;

end

*/** Time*= 3+3(1+*$\lg 3$*+1)+1* $\approx$ *14 */*

Algorithm A.12: F_commonEdge(M_i^2, M_j^2)

APPENDIX B
ALGORITHMS OF MESH OPERATORS WITH CIRCULAR ADJACENCY

Data : M_i^0
Result: $\{M_i^0\{M^1\}\}$
begin
  GET($\{M_i^0\{M^3\}\}$); /* *23 steps* */
  for *each* $M_i^3 \in \{M_i^0\{M^3\}\}$ /* *repeat 23 times* */ **do**
    GET($\{M_i^3\{M^0\}\}$); /* *17 steps* */
    if *FIND*($\{M_i^3\{M^0\}\}$, M_i^0) = *true* /* $\lg 4$*+1 steps* */
      if *FIND*($\{M_i^0\{M^1\}\}$, M_i^1) = *false* /* $\lg 14$*+1 steps* */
        PUT($\{M_i^0\{M^1\}\}$, M_i^1); /* *at most 14 steps* */
      endif
    endif
  endfor
end

/* Time = *23+23(17+2+1+*$\lg 14$*+1)+14* $\approx$ *584* */

Algorithm B.1: Circular adjacency: V_edges(M_i^0)

```
Data  : M_i^0
Result: {M_i^0{M^2}}
begin
    GET({M_i^0{M^3}}); /* 23 steps */
    for each M_i^3 ∈ {M_i^0{M^3}} /* repeat 23 times */ do
        GET({M_i^3{M^2}}); /* 4 steps */
        for each M_i^2 ∈ {M_i^3{M^2}} /* repeat 4 times */ do
            GET({M_i^2{M^0}}); /* 10 steps */
            if FIND({M_i^2{M^0}}, M_i^0) = true /* lg 3+1 steps */
                if FIND({M_i^0{M^2}}, M_i^2) = false /* lg 35+1 steps */
                    PUT({M_i^0{M^2}}, M_i^2); /* at most 35 steps */
                endif
            endif
        endfor
    endfor
end

/* Time = 23+23(4+4(10+lg 3+1+lg 35+1))+35 ≈ 1871 */
```

Algorithm B.2: Circular adjacency: V_faces(M_i^0)

```
Data  : M_i^1
Result: {M_i^1{M^2}}
begin
    M_i^0 ← M_i^1{M^0}_0; /* 2 steps */
    GET({M_i^0{M^3}}); /* 23 steps */
    for each M_i^3 ∈ {M_i^0{M^3}} /* repeat 23 times */ do
        GET({M_i^3{M^2}}) /* 4 steps */
        for each M_i^2 ∈ {M_i^3{M^2}} /* repeat 4 times */ do
            GET({M_i^2{M^1}}); /* 3 steps */
            if FIND({M_i^2{M^1}}, M_i^1) = true and FIND({M_i^1{M^2}}, M_i^2) =
            false /* lg 6+1+lg 5+1 steps */
                PUT({M_i^1{M^2}}, M_i^2); /* at most 5 steps */
            endif
        endfor
    endfor
end

/* Time = 2+23+23(4+4(3+lg 3+1+lg 5+1))+5 ≈ 941 */
```

Algorithm B.3: Circular adjacency: E_faces(M_i^0)

Data : M_i^1
Result: $\{M_i^1\{M^3\}\}$
begin
  $M_i^0 \leftarrow M_i^1\{M^0\}_0$; /* *first vertex of* M_i^1*- 2 steps* */
  GET($\{M_i^0\{M^3\}\}$); /* *23 steps* */
  for *each* $M_i^3 \in \{M_i^0\{M^3\}\}$ /* *repeat 23 times* */ **do**
    GET($\{M_i^3\{M^1\}\}$); /* *9 steps* */
    if *FIND*($\{M_i^3\{M^1\}\}$, M_i^1) = *true and FIND*($\{M_i^1\{M^3\}\}$, M_i^3) = *false* /* $\lg 6$*+1+*$\lg 5$*+1 steps* */
      PUT($\{M_i^1\{M^3\}\}$, M_i^2); /* *at most 5 steps* */
    endif
  endfor
end

/* Time = *2+23+23(9+*$\lg 6$*+1+*$\lg 5$*+1)+5* $\approx$ *455* */

Algorithm B.4: Circular adjacency: E_regions(M_i^0)

Data : M_i^2
Result: $\{M_i^2\{M^0\}\}$
begin
  $M_i^1 \leftarrow M_i^2\{M^1\}_0$; /* *2(GET+ASSIGN) steps* */
  $M_k^1 \leftarrow M_i^2\{M^1\}_2$; /* *2 steps* */
  $M_i^2\{M^0\}_0 \leftarrow M_i^1\{M^0\}_0$; /* *2 steps* */
  $M_i^2\{M^0\}_1 \leftarrow M_i^1\{M^0\}_1$; /* *2 steps* */
  $M_i^2\{M^0\}_2 \leftarrow M_k^1\{M^0\}_1$; /* *2 steps* */
end

/* Time = *2+2+2+2+2 = 10* */

Algorithm B.5: Circular adjacency: F_vertices(M_i^2)

Data : M_i^2
Result: $\{M_i^2\{M^3\}\}$
begin
  /* *first vertex of* $M_i^2\{M^1\}_0$- *3(2 GET's+1 ASSIGN) steps* */
  $M_i^0 \leftarrow M_i^2\{M^1\}_0\{M^0\}_0$;
  GET($\{M_i^0\{M^3\}\}$); /* *23 steps* */
  for *each* $M_i^3 \in \{M_i^0\{M^3\}\}$ /* *repeat 23 times* */ **do**
    GET($\{M_i^3\{M^2\}\}$); /* *4 steps* */
    if *FIND*($\{M_i^3\{M^2\}\}$, M_i^2) = *true* /* $\lg 4$+*1 steps* */
      if *FIND*($\{M_i^2\{M^3\}\}$, M_i^3) = *false* /* $\lg 2$+*1 steps* */
        PUT($\{M_i^2\{M^3\}\}$, M_i^3); /* *at most 2 steps* */
      endif
    endif
  endfor
end

/* Time = *3+23+23(4+2+1+1+1)+2 = 235* */

Algorithm B.6: Circular adjacency: F_regions(M_i^2)

Data : M_i^3
Result: $\{M_i^3\{M^0\}\}$
begin
  /* *first edge of first face of region - 3(2 GET's+1 ASSIGN) steps* */
  $M_i^1 \leftarrow M_i^3\{M^2\}_0\{M^1\}_0$;
  /* *second edge of first face of region - 3 steps* */
  $M_j^1 \leftarrow M_i^3\{M^2\}_0\{M^1\}_1$;
  /* *second edge of second face of region - 3 steps* */
  $M_k^1 \leftarrow M_i^3\{M^2\}_1\{M^1\}_1$;
  $M_i^3\{M^0\}_0 \leftarrow M_i^1\{M^0\}_0$; /* *2 (GET+ASSIGN) steps* */
  $M_i^3\{M^0\}_1 \leftarrow M_i^1\{M^0\}_1$; /* *2 steps* */
  $M_i^3\{M^0\}_2 \leftarrow M_j^1\{M^0\}_1$; /* *2 steps* */
  $M_i^3\{M^0\}_3 \leftarrow M_k^1\{M^0\}_1$; /* *2 steps* */
end

/* Time = *3·3+4·2 = 17* */

Algorithm B.7: Circular adjacency: R_vertices(M_i^0)

Data : M_i^3
Result: $\{M_i^3\{M^1\}\}$
begin
  /* *3 (2 GET's+1 ASSIGN) steps for each*/
  $M_i^3\{M^1\}_0 \leftarrow M_i^3\{M^2\}_0\{M^1\}_0$;
  $M_i^3\{M^1\}_1 \leftarrow M_i^3\{M^2\}_0\{M^1\}_1$;
  $M_i^3\{M^1\}_2 \leftarrow M_i^3\{M^2\}_1\{M^1\}_1$;
end

/* Time = *3·3 = 9* */

Algorithm B.8: Circular adjacency: R_edges(M_i^0)

Data : M_i^0, M_j^0
Result: M_i^1 bounded by M_i^0, M_j^0
begin
  GET($\{M_i^0\{M^1\}\}$); /* *584 steps* */
  for *each* $M_i^1 \in \{M_i^0\{M^1\}\}$ **do**
    GET($\{M_i^1\{M^0\}\}$); /* *2 steps* */
    if *FIND*($\{M_i^1\{M^0\}\}$, M_j^0) = *true* /* $\lg 2$*+1 steps* */
      RETURN M_i^1; /* *1 step* */
    endif
  endfor
  RETURN $NULL$;
end

/* Time = *584+14·(2+2)+1 = 641* */

Algorithm B.9: Circular adjacency: E_exist(M_i^0, M_j^0)

Data : M_i^0, M_j^0, M_k^0

Result: M_i^2 bounded by M_i^0, M_j^0, M_k^0

begin
  $M_i^1 \leftarrow$ E_exist(M_i^0, M_j^0); /* *641+1 steps* */
  GET($\{M_i^1\{M^2\}\}$); /* *941 steps* */
  for *each* $M_i^2 \in \{M_i^1\{M^2\}\}$ /* *repeat 5 times* */ **do**
    GET($\{M_i^2\{M^0\}\}$); /* *10 steps* */
    if *FIND*($\{M_i^2\{M^0\}\}$, M_k^0) = *true* /* $\lg 3$ *+1 steps* */
      RETURN M_i^2; /* *1 step* */
    endif
  endfor
  RETURN *NULL*;
end

/* Time = *642+941+5(10+*$\lg 3$*+1)+1* $\approx$ *1647* */

Algorithm B.10: Circular adjacency: F_exist(M_i^0, M_j^0, M_k^0)

Data : M_i^0, M_j^0, M_k^0

Result: M_i^2 bounded by M_i^2, M_j^2, M_k^2

begin
  GET($\{M_i^1\{M^2\}\}$); /* *941 steps* */
  for *each* $M_i^2 \in \{M_i^1\{M^2\}\}$ /* *repeat 5 times* */ **do**
    GET($\{M_i^2\{M^1\}\}$); /* *3 steps* */
    if *FIND*($\{M_i^2\{M^1\}\}$, M_j^1) = *FIND*($\{M_i^2\{M^1\}\}$, M_k^1)= *true* /* *2(*$\lg 3$ *+1) steps* */
      RETURN M_i^2; /* *1 step* */
    endif
  endfor
  RETURN *NULL*;
end

/* Time = *941+5(3+2(*$\lg 3$*+1))+1* $\approx$ *982* */

Algorithm B.11: Circular adjacency: F_exist(M_i^2, M_j^2, M_k^2)

Data : M_i^0, M_j^0, M_k^0, M_l^0

Result: M_i^3 bounded by M_i^0, M_j^0, M_k^0, M_l^0

begin

GET($\{M_i^0\{M^3\}\}$); */* 23 steps */*

for *each* $M_i^3 \in \{M_i^0\{M^3\}\}$ */* repeat 23 times */* **do**

GET($\{M_i^3\{M^0\}\}$); */* 17 steps */*

if *FIND*($\{M_i^3\{M^0\}\}$, M_j^0) = *FIND*($\{M_i^3\{M^0\}\}$, M_k^0) = *FIND*($\{M_i^3\{M^0\}\}$, M_l^0) = *true /* 3(*$\lg 17$*+1) steps */*

RETURN M_i^3; */* 1 step */*

endif

endfor

RETURN $NULL$;

end

*/** Time = *23+23(17+3(*$\lg 17$*+1))+1* $\approx$ *766 */*

Algorithm B.12: Circular adjacency: R_exist(M_i^0, M_j^0, M_k^0, M_l^0)

Data : M_i^0, M_j^0

Result: create M_i^1 bounded by M_i^0, M_j^0

begin

CREATE an edge M_i^1; */* 1 step */*

PUT($\{\mathrm{M}\{M^1\}\}$, M_i^1); */* 1 step */*

/ update link from edge to vertex */*

$M_i^1\{M^0\}_0 \leftarrow M_i^0$; */* 1 step */*

$M_i^1\{M^0\}_1 \leftarrow M_j^0$; */* 1 step */*

RETURN M_i^1; */* 1 step */*

end

*/** Time = *1+1+1+1+1 = 5 */*

Algorithm B.13: Circular adjacency: M_createE(M_i^0, M_j^0)

Data : M_i^0, M_j^0, M_k^0

Result: create M_i^2 bounded by M_i^0, M_j^0, M_k^0

begin

CREATE a face M_i^2; */* 1 step */*

PUT({M{M^2}}, M_i^2); */* 1 step */*

/ update link from face to edge - 641+1 steps for each */*

$M_i^2\{M^1\}_0 \leftarrow$ E_exist(M_i^0, M_j^0);

$M_i^2\{M^1\}_1 \leftarrow$ E_exist(M_j^0, M_k^0);

$M_i^2\{M^1\}_2 \leftarrow$ E_exist(M_i^0, M_k^0);

RETURN M_i^2; */* 1 step */*

end

/ Time = 1+1+3·642+1 = 1929 */*

Algorithm B.14: Circular adjacency: M_createF(M_i^0, M_j^0, M_k^0)

Data : M_i^1, M_j^1, M_k^1

Result: create M_i^2 bounded by M_i^1, M_j^1, M_k^1

begin
CREATE a face M_i^2; */* 1 step */*
PUT({M{M^2}}, M_i^2); */* 1 step */*
/ update link from face to edge*/*
$M_i^2\{M^1\}_0 \leftarrow M_i^1$; */* 1 step */*
if *dir[0] = 1 /* 1 step */*
if *FIND*($\{M_j^1\{M^0\}\}$, M_i^0) *= true /* 2 step */*
$M_i^2\{M^1\}_1 \leftarrow M_k^1$; */* 1 step */*
$M_i^2\{M^1\}_2 \leftarrow M_j^1$; */* 1 step */*
else
$M_i^2\{M^1\}_1 \leftarrow M_j^1$;
$M_i^2\{M^1\}_2 \leftarrow M_k^1$;
endif
else
if *FIND*($\{M_j^1\{M^0\}\}$, M_i^0) *= true*
$M_i^2\{M^1\}_1 \leftarrow M_j^1$;
$M_i^2\{M^1\}_2 \leftarrow M_k^1$;
else
$M_i^2\{M^1\}_1 \leftarrow M_k^1$;
$M_i^2\{M^1\}_2 \leftarrow M_j^1$;
endif
endif
RETURN M_i^2; */* 1 step */*
end

/ Time = 1+1+1+1(if dir)+2(if FIND)+2+1 = 9 */*

Algorithm B.15: Circular adjacency: M_createF(M_i^1, M_j^1, M_k^1, dir[3])

Data : M_i^0, M_j^0, M_k^0, M_l^0

Result: create M_i^3 bounded by M_i^0, M_j^0, M_k^0, M_l^0

begin

CREATE a region M_i^3; */* 1 step */*

PUT({M{M^3}}, M_i^3); */* 1 step */*

/ update link from vertex to region */*

for var $\leftarrow$ *i to l /* repeat 4 times */* **do**

PUT_UNIQ({M_{var}^0{M^3}}, M_{var}^0); */* $\lg 23$+2 steps */*

endfor

/ update* {M_i^3{M^2}} *- 3+1647+1 steps for each */*

$M_i^3\{M^2\}_0 \leftarrow$ F_exist($M_i^3\{M^0\}_0$, $M_i^3\{M^0\}_1$, $M_i^3\{M^1\}_2$));

$M_i^3\{M^2\}_2 \leftarrow$ F_exist($M_i^3\{M^0\}_0$, $M_i^3\{M^0\}_1$, $M_i^3\{M^1\}_3$));

$M_i^3\{M^2\}_1 \leftarrow$ F_exist($M_i^3\{M^0\}_1$, $M_i^3\{M^0\}_2$, $M_i^3\{M^1\}_3$));

$M_i^3\{M^2\}_3 \leftarrow$ F_exist($M_i^3\{M^0\}_0$, $M_i^3\{M^0\}_2$, $M_i^3\{M^1\}_3$));

RETURN M_i^3; */* 1 step */*

end

/ Time = 1+1+4(*$\lg 23$*+2)+4·1651+1* $\approx$ *6627 */*

Algorithm B.16: Circular adjacency: M_createR(M_i^0, M_j^0, M_k^0, M_l^0)

Data : M_i^2, M_j^2, M_k^2, M_l^2

Result: create M_i^3 bounded by M_i^2, M_j^2, M_k^2, M_l^2

begin

CREATE a region M_i^3; */* 1 step */*

PUT({M{M^3}}, M_i^3); */* 1 step */*

/ update* {M_i^3{M^2}} *- 1 step for each */*

$M_i^3\{M^2\}_0 \leftarrow M_i^2$;

$M_i^3\{M^2\}_1 \leftarrow M_j^2$;

$M_i^3\{M^2\}_2 \leftarrow M_k^2$;

$M_i^3\{M^2\}_3 \leftarrow M_l^2$;

RETURN M_i^3; */* 1 step */*

end

/ Time = 1+1+4·1+1 = 7 */*

Algorithm B.17: Circular adjacency: M_createR(M_i^2, M_j^2, M_k^2, M_l^2)

APPENDIX C
ALGORITHMS OF MESH OPERATORS WITH ONE-LEVEL ADJACENCY

Data : M_i^0
Result: $\{M_i^0\{M^2\}\}$
begin
  GET($\{M_i^0\{M^1\}\}$); /* *14 steps* */
  for *each* $M_i^1 \in \{M_i^0\{M^1\}\}$ /* *repeat 14 times* */ **do**
    GET($\{M_i^1\{M^2\}\}$); /* *5 steps* */
    for *each* $M_i^2 \in \{M_i^1\{M^2\}\}$ /* *repeat 5 times* */ **do**
      if *FIND*($\{M_i^0\{M^2\}\}$, M_i^2) = *false* /* $\lg 35$+*1 steps* */
        PUT($\{M_i^0\{M^2\}\}$, M_i^2); /* *at most 35 steps* */
      endif
    endfor
  endfor
end

/* Time = *14+14(5+5(*$\lg 35$*+1))+35* $\approx$ *548* */

Algorithm C.1: One-Level adjacency: V_faces(M_i^0)

```
Data  : M_i^0
Result: {M_i^0{M^2}}
begin
    GET({M_i^0{M^1}}); /* 14 steps */
    for each M_i^1 ∈ {M_i^0{M^1}} /* repeat 14 times */ do
        GET({M_i^1{M^2}}); /* 5 steps */
        for each M_i^2 ∈ {M_i^1{M^2}} /* repeat 5 times */ do
            if MARKED(M_i^2) = false /* 3 steps */
                PUT({M_i^0{M^2}}, M_i^2); /* at most 35 steps */
                MARK(M_i^2); /* at most 35·2 steps */
            endif
        endfor
    endfor
    for each M_i^2 ∈ {M_i^0{M^2}} /* repeat 35 times */ do
        UNMARK(M_i^2); /* 2 steps */
    endfor
end

/* Time = 14+14(5+5·3))+35+70+70 = 469 */
```

Algorithm C.2: One-Level adjacency: V_faces(M_i^0) with MARK

```
Data  : M_i^0
Result: {M_i^0{M^2}}
begin
    GET({M_i^0{M^1}}); /* 14 steps */
    for each M_i^1 ∈ {M_i^0{M^1}} /* repeat 14 times */ do
        GET({M_i^1{M^2}}); /* 5 steps */
        for each M_i^2 ∈ {M_i^1{M^2}} /* repeat 5 times */ do
            GET({M_i^2{M^3}}); /* 2 steps */
            for each M_i^3 ∈ {M_i^2{M^3}} /* repeat 2 times */ do
                if FIND({M_i^0{M^3}}, M_i^3) = false /* lg 23+1 steps */
                    PUT({M_i^0{M^3}}, M_i^2); /* at most 23 steps */
                endif
            endfor
        endfor
    endfor
end

/* Time = 14+14(5+5(2+2(lg 23+1)))+23 ≈ 1020 */
```

Algorithm C.3: One-Level adjacency: V_region(M_i^0)

```
Data  : M_i^0
Result: {M_i^0{M^2}}
begin
    GET({M_i^0{M^1}}); /* 14 steps */
    for each M_i^1 ∈ {M_i^0{M^1}} /* repeat 14 times */ do
        GET({M_i^1{M^2}}); /* 5 steps */
        for each M_i^2 ∈ {M_i^1{M^2}} /* repeat 5 times */ do
            GET({M_i^2{M^3}}); /* 2 steps */
            if MARKED(M_i^3) = false /* 3 steps */
                PUT({M_i^0{M^3}}, M_i^2); /* at most 23 steps */
                MARK(M_i^3); /* at most 23·2 steps */
            endif
        endfor
    endfor
    for each M_i^3 ∈ {M_i^0{M^3}} /* repeat 23 times */ do
        UNMARK(M_i^3); /* 2 steps */
    endfor
end

/* Time = 14+14(5+5(2+3))+23+23·2+23·2 = 549 */
```

Algorithm C.4: One-Level adjacency: V_region(M_i^0) with MARK

```
Data  : M_i^1
Result: {M_i^1{M^3}}
begin
    GET({M_i^1{M^2}}); /* 5 steps */
    for each M_i^2 ∈ {M_i^1{M^2}} /* repeat 5 times */ do
        GET({M_i^2{M^3}}); /* 2 steps */
        for each M_i^3 ∈ {M_i^1{M^2}} /* repeat 2 times */ do
            if FIND({M_i^1{M^3}}, M_i^3) = false /* lg 5+1 steps */
                PUT({M_i^1{M^3}}, M_i^3); /* at most 5 steps */
            endif
        endfor
    endfor
end

/* Time = 5+5(2+2(lg 5+1))+5 ≈ 53 */
```

Algorithm C.5: One-Level adjacency: E_regions(M_i^0)

```
Data  : M_i^1
Result: {M_i^1{M^3}}
begin
    GET({M_i^1{M^2}}); /* 5 steps */
    for each M_i^2 ∈ {M_i^1{M^2}} /* repeat 5 times */ do
        GET({M_i^2{M^3}}); /* 2 steps */
        for each M_i^3 ∈ {M_i^1{M^2}} /* repeat 2 times */ do
            if MARKED(M_i^3) = false /* 3 steps */
                PUT({M_i^1{M^3}}, M_i^2); /* at most 5 steps */
                MARK(M_i^3); /* at most 5·2 steps */
            endif
        endfor
    endfor
    for each M_i^3 ∈ {M_i^1{M^3}} /* repeat 5 times */ do
        UNMARK(M_i^3) /* 2 steps */
    endfor
end

/* Time = 5+5(2+2·3)+5+10+10 = 70 */
```

Algorithm C.6: One-Level adjacency: E_regions(M_i^0) with MARK

```
Data  : M_i^0, M_j^0, M_k^0
Result: M_i^2 bounded by M_i^0, M_j^0, M_k^0
begin
    M_i^1 ← E_exist(M_i^0, M_j^0); /* 71+1 steps */
    GET({M_i^1{M^2}}); /* 5 steps */
    for each M_i^2 ∈ {M_i^1{M^2}} /* repeat 5 times */ do
        GET({M_i^2{M^0}}); /* 10 steps */
        if FIND({M_i^2{M^0}}, M_k^0) = true /* lg 3 +1 steps */
            RETURN M_i^2; /* 1 step */
        endif
    endfor
    RETURN NULL;
end

/* Time = 72+5+5(10+lg 3+1)+1 ≈ 140 */
```

Algorithm C.7: One-Level adjacency: F_exist(M_i^0, M_j^0, M_k^0)

Data : M_i^0, M_j^0, M_k^0, M_l^0

Result: M_i^3 bounded by M_i^0, M_j^0, M_k^0, M_l^0

begin
  $M_i^2 \leftarrow$ F_exist(M_i^0, M_j^0, M_k^0); /* *140+1 steps* */
  GET($\{M_i^2\{M^3\}\}$); /* *2 steps* */
  for *each* $M_i^3 \in \{M_i^2\{M^3\}\}$ /* *repeat 2 times* */ **do**
    GET($\{M_i^3\{M^0\}\}$); /* *17 steps* */
    if *FIND*($\{M_i^3\{M^0\}\}$, M_l^0) = *true* /* $\lg 4$ *+1 steps* */
      RETURN M_i^3; /* *1 step* */
    endif
  endfor
  RETURN *NULL*;
end

/* Time = *141+2+2(17+(2+1))+1 = 544* */

Algorithm C.8: One-Level adjacency: R_exist(M_i^0, M_j^0, M_k^0, M_l^0)

Data : M_i^0, M_j^0, M_k^0

Result: create M_i^2 bounded by M_i^0, M_j^0, M_k^0

begin
  CREATE a face M_i^2; /* *1 step* */
  PUT($\{\mathrm{M}\{M^2\}\}$, M_i^2); /* *1 step* */
  /* *update link from face to edge - 71+1 steps for each* */
  $M_i^2\{M^1\}_0 \leftarrow$ E_exist(M_i^0, M_j^0);
  $M_i^2\{M^1\}_1 \leftarrow$ E_exist(M_j^0, M_k^0);
  $M_i^2\{M^1\}_2 \leftarrow$ E_exist(M_i^0, M_k^0);
  /* *update link from edge to face* */
  for var $\leftarrow$ *0 to 2* /* *repeat 3 times* - $\lg 5$*+2 steps for each* */ **do**
    PUT_UNIQ($\{\{M_i^2\{M^1\}\}_{var}\{M^2\}$, M_i^2);
  endfor
  RETURN M_i^2; /* *1 step* */
end

/* Time = *1+1+3·72+3(*$\lg 5$*+2)+1* $\approx$ *231* */

Algorithm C.9: One-Level adjacency: M_createF(M_i^0, M_j^0, M_k^0)

Data : M_i^1, M_j^1, M_k^1

Result: create M_i^2 bounded by M_i^1, M_j^1, M_k^1

begin

CREATE a face M_i^2; */* 1 step */*

PUT({M{M^2}}, M_i^2); */* 1 step */*

/ update link from face to edge*/*

$M_i^2\{M^1\}_0 \leftarrow M_i^1$; */* 1 step */*

if *dir[0] = 1 /* 1 step */*

if *FIND*($\{M_j^1\{M^0\}\}$, M_i^0) *= true /* 2 steps */*

$M_i^2\{M^1\}_1 \leftarrow M_k^1$; */* 1 step */*

$M_i^2\{M^1\}_2 \leftarrow M_j^1$; */* 1 step */*

else

$M_i^2\{M^1\}_1 \leftarrow M_j^1$;

$M_i^2\{M^1\}_2 \leftarrow M_k^1$;

endif

else

if *FIND*($\{M_j^1\{M^0\}\}$, M_i^0) *= true*

$M_i^2\{M^1\}_1 \leftarrow M_j^1$;

$M_i^2\{M^1\}_2 \leftarrow M_k^1$;

else

$M_i^2\{M^1\}_1 \leftarrow M_k^1$;

$M_i^2\{M^1\}_2 \leftarrow M_j^1$;

endif

endif

/ update link from edge to face */*

for var $\leftarrow$ *0 to 2 /* repeat 3 times* - $\lg 5$*+2 steps for each */* **do**

PUT_UNIQ($\{\{M_i^2\{M^1\}\}_{var}\{M^2\}\}$, M_i^2);

endfor

RETURN M_i^2; */* 1 step */*

end

/ Time = 1+1+1+1(if dir)+2(if FIND)+2+3(*$\lg 5$*+2)+1* $\approx$ *21 */*

Algorithm C.10: One-Level adjacency: M_createF(M_i^1, M_j^1, M_k^1, *dir*[3])

Data : M_i^0, M_j^0, M_k^0, M_l^0

Result: create M_i^3 bounded by M_i^0, M_j^0, M_k^0, M_l^0

begin

CREATE a region M_i^3; */* 1 step */*

PUT({M{M^3}}, M_i^3); */* 1 step */*

/ update {M_i^3{M^2}} - (140+1) steps for each */*

$M_i^3\{M^2\}_0 \leftarrow$ F_exist(M_i^2, M_j^2, M_k^2));

$M_i^3\{M^2\}_2 \leftarrow$ F_exist(M_i^2, M_j^2, M_l^2));

$M_i^3\{M^2\}_1 \leftarrow$ F_exist(M_l^2, M_k^2, M_l^2));

$M_i^3\{M^2\}_3 \leftarrow$ F_exist(M_i^2, M_k^2, M_l^2));

/ update link from face to region - 1 step for each */*

for var $\leftarrow$ *0 to 3 /* repeat 4 times -* $\lg 2$*+2 steps for each */* **do**

PUT_UNIQ({{M_i^3{M^2}}$_{var}${M^3}}, M_i^3)

endfor

RETURN M_i^3; */* 1 step */*

end

/ Time = 1+1+4·141+4(*$\lg 2$*+2)+1 = 579 */*

Algorithm C.11: One-Level adjacency: M_createR(M_i^0, M_j^0, M_k^0, M_l^0)

Data : M_i^2, M_j^2, M_k^2, M_l^2

Result: create M_i^3 bounded by M_i^2, M_j^2, M_k^2, M_l^2

begin

CREATE a region M_i^3; */* 1 step */*

PUT({M{M^3}}, M_i^3); */* 1 step */*

/ update link from face to region and vice versa */*

for var $\leftarrow$ *i to l /* repeat 4 times */* **do**

PUT_UNIQ({M_{var}^2{M^3}}, M_{var}^2); */** $\lg 2$*+2 steps */*

{M_i^3{M^2}} $\leftarrow M_{var}^2$; */* 1 step */*

endfor

RETURN M_i^3; */* 1 step */*

end

/ Time = 1+1+4(*$\lg 2$*+2+1)+1 = 19 */*

Algorithm C.12: One-Level adjacency: M_createR(M_i^2, M_j^2, M_k^2, M_l^2)

APPENDIX D
ALGORITHMS OF MESH OPERATORS WITH COMPLETE MINIMUM SUFFICIENT

For the purpose of computing downward adjacent entity in terms of the ordering illustrated in Figure 2.6 − 2.8, the following data structures are used in the algorithms. Refer to Appendix A for algorithms of $E\_commonVertex$, $F\_commonVertex$, and $F\_commonEdge$. We assume that M_i^2 is a triangle and M_i^3 is a tetrahedron for the analysis purpose.

- `int Fev[3][2] = {{0,1},{1,2},{2,0}};`

  Structure Fev provides the order of $\{M_i^2\{M^1\}\{M^0\}\}$ in terms of $\{M_i^2\{M^0\}\}$. For example, for M_i^2, $M_i^2\{M^0\}_{Fev[j][k]}$ returns the k^{th} vertex of edge $M_i^2\{M^1\}_j$, $j = 0, 1, 2$, $k = 0, 1$. Thus, $M_i^2\{M^1\}_j$ is identical to $E\_exist(M_i^2\{M^0\}_{Fev[j][0]}, M_i^2\{M^0\}_{Fev[j][1]})$ if exists.

- `int Fve[4][2] = {{2,0},{0,1},{1,2}};`

  Structure Fve provides vertex the order of $\{M_i^2\{M^0\}\}$ as an intersection of 2 edges in $\{M_i^2\{M^1\}\}$. For example, for M_i^2, $E\_commonVertex(M_i^3\{M^1\}_{Fve[j][0]}$, $M_i^2\{M^1\}_{Fve[j][1]})$ returns $M_i^3\{M^0\}_j$, $j = 0, 1, 2$.

- `int Rev[6][2] = {{0,1},{1,2},{2,0},{0,3},{1,3},{2,3}};`

  Structure Rev provides the order of $\{M_i^3\{M^1\}\{M^0\}\}$ in terms of $\{M_i^3\{M^0\}\}$. For example, for M_i^3, $M_i^3\{M^0\}_{Rev[j][k]}$ returns the k^{th} vertex of edge $M_i^3\{M^1\}_j$, $0 \leq j \leq 5$, $k = 0, 1$. Thus, $M_i^3\{M^1\}_j$ is identical to $E\_exist(M_i^3\{M^0\}_{Rev[j][0]}, M_i^3\{M^0\}_{Rev[j][1]})$ if exists.

- `int Rfv[4][3] = {{0,1,2},{0,1,3},{1,2,3},{0,2,3}};`

  Structure Rfv provides the order of $\{M_i^3\{M^2\}\{M^0\}\}$ in terms of $\{M_i^3\{M^0\}\}$. For example, for M_i^3, $M_i^3\{M^0\}_{Rfv[j][k]}$ returns the k^{th} vertex of face $M_i^3\{M^2\}_j$, $j = 0, 1, 2, 3$, $k = 0, 1, 2$. Thus, $M_i^3\{M^2\}_j$ is identical to $F\_exist(M_i^3\{M^0\}_{Rfv[j][0]}$, $M_i^3\{M^0\}_{Rfv[j][1]}$, $M_i^3\{M^0\}_{Rfv[j][2]})$ if exists.

- `int Rfe[4][3] = {{0,1,2},{0,4,3},{1,5,4},{2,3,5}};`

  Structure Rfe provides the order of $\{M_i^3\{M^2\}\{M^1\}\}$ in terms of $\{M_i^3\{M^1\}\}$. For

example, for M_i^3, $M_i^3\{M^1\}_{Rfe[j][k]}$ returns the k^{th} edge of face $M_i^3\{M^2\}_j$, $j = 0,1,2,3$, $k = 0,1,2$. Thus, $M_i^3\{M^2\}_j$ is identical to $F\_exist(M_i^3\{M^1\}_{Rfe[j][0]}$, $M_i^3\{M^1\}_{Rfe[j][1]}$, $M_i^3\{M^1\}_{Rfe[j][2]})$ if exists.

- `int Rve[4][2] = {{0,2},{0,1},{1,2},{3,4}};`

  Structure *Rve* provides the order of $\{M_i^3\{M^0\}\}$ as an intersection of 2 edges in $\{M_i^3\{M^1\}\}$. For example, for M_i^3, $E\_commonVertex(M_i^3\{M^1\}_{Rve[j][0]}, M_i^3\{M^1\}_{Rve[j][1]})$ returns $M_i^3\{M^0\}_j$, $j = 0,1,2,3$.

- `int Rvf[4][3] = {{0,1,3},{0,1,2},{0,2,3},{1,2,3}};`

  Structure *Rvf* provides the order of $\{M_i^3\{M^0\}\}$ as an intersection of 3 faces in $\{M_i^3\{M^2\}\}$. For example, for M_i^3, $F\_commonVertex(M_i^3\{M^2\}_{Rvf[j][0]}, M_i^3\{M^2\}_{Rvf[j][1]}$, $M_i^3\{M^2\}_{Rvf[j][2]})$ returns $M_i^3\{M^0\}_j$, $j = 0,1,2,3$.

- `int Ref[6][2] = {{0,1},{0,2},{0,3},{1,3},{1,2},{2,3}};`

  Structure *Rvf* provides the order of $\{M_i^3\{M^1\}\}$ as an intersection of 2 faces in $\{M_i^3\{M^2\}\}$. For example, for M_i^3, $F\_commonEdge(M_i^3\{M^2\}_{Ref[j][0]}, M_i^3\{M^2\}_{Ref[j][1]})$ returns $M_i^3\{M^1\}_j$, $0 \le j \le 5$.

Data : M_i^0
Result: $\{M_i^0\{M^1\}\}$
begin
  GET($\{M_i^0\{M^3\}\}$); */* 23 steps */*
  for *each* $M_i^3 \in \{M_i^0\{M^3\}\}$ */* repeat 23 times */* **do**
    GET($\{M_i^3\{M^1\}\}$); */* 576 steps */*
    for *each* $M_i^1 \in \{M_i^3\{M^1\}\}$ */* repeat 6 times */* **do**
      if *FIND*($\{M_i^1\{M^0\}\}$, M_i^0) = *true* */* $\lg 2+1$ steps */*
        if *FIND*($\{M_i^0\{M^1\}\}$, M_i^1) = *false* */* $\lg 14+1$ steps */*
          PUT($\{M_i^0\{M^1\}\}$, M_i^1); */* at most 14 steps */*
        endif
      endif
    endfor
  endfor
end

/ Time = 23+23(582+6($\lg 2$+1+$\lg 14$+1))+14 $\approx$ 14,362 */*

Algorithm D.1: Complete MSR: V_edges(M_i^0)

```
Data  : M_i^0
Result: {M_i^0{M^2}}
begin
    GET({M_i^0{M^3}}); /* 23 steps */
    for each M_i^3 ∈ {M_i^0{M^3}} /* repeat 23 times */ do
        GET({M_i^3{M^2}}); /* 584 steps */
        for each M_i^2 ∈ {M_i^3{M^2}} /* repeat 4 times */ do
            if FIND({M_i^2{M^0}}, M_i^0) = true /* lg3+1 steps */
                if FIND({M_i^0{M^2}}, M_i^2) = false /* lg35+1 steps */
                    PUT({M_i^0{M^2}}, M_i^2); /* at most 35 steps */
                endif
            endif
        endfor
    endfor
end

/* Time = 23+23(584+4(lg 3+1+lg 35+1))+35 ≈ 14,291 */
```

Algorithm D.2: Complete MSR: V_faces(M_i^0)

```
Data  : M_i^1
Result: {M_i^1{M^2}}
begin
    GET({M_i^1{M^0}}); /* 2 steps */
    for each M_i^1 ∈ {M_i^1{M^2}} /* repeat 2 times */ do
        GET({M_i^0{M^2}}); /* 14,291 steps */
        for each M_i^2 ∈ {M_i^0{M^2}} /* repeat 35 times */ do
            GET({M_i^2{M^1}}); /* 291 steps */
            if FIND({M_i^2{M^1}}, M_i^1) = true /* lg3+1 steps */
                if FIND({M_i^1{M^2}}, M_i^2) = false /* lg5+1 steps */
                    PUT({M_i^1{M^2}}, M_i^2); /* at most 5 steps */
                endif
            endif
        endfor
    endfor
end

/* Time = 2+2(14291+35(291+lg3+1+lg5+1))+5 ≈ 49,372 */
```

Algorithm D.3: Complete MSR: E_faces(M_i^0)

```
Data  : M_i^1
Result: {M_i^1{M^3}}
begin
    GET({M_i^1{M^0}}); /* 2 steps */
    for each M_i^0 ∈ {M_i^1{M^0}} /* repeat 2 times */ do
        GET({M_i^0{M^3}}); /* 23 steps */
        for each M_i^3 ∈ {M_i^0{M^3}} /* repeat 23 times */ do
            GET({M_i^3{M^1}}); /* 582 steps */
            if FIND({M_i^3{M^1}}, M_i^1) = true /* lg6+1 steps */
                if FIND({M_i^1{M^3}}, M_i^2) = false /* lg5+1 steps */
                    PUT({M_i^1{M^3}}, M_i^2); /* at most 5 steps */
                endif
            endif
        endfor
    endfor
end

/* Time = 2+2(23+23(582+lg6+1+lg5+1))+5 ≈ 27,142 */
```

Algorithm D.4: Complete MSR: E_regions(M_i^0)

```
Data  : M_i^2
Result: {M_i^2{M^1}}
begin
    for var ← 0 to 2 /* repeat 3 times */ do
        M_i^0 ← M_i^2{M^0}_{Fev[var][0]}; /* 3 (2 GET's + ASSIGN) steps */
        M_j^0 ← M_i^2{M^0}_{Fev[var][1]}; /* 3 steps */
        M_i^1 ← E_exist(M_i^0, M_j^0); /* 71+1 steps */
        if M_i^1 = NULL /* 1 step */
            M_i^1 ← M_createE(M_i^0, M_j^0); /* 16+1 steps */
        endif
        M_i^2{M^1}_var ← M_i^1; /* 1 step */
    endfor
end

/* Time = 3(3+3+72+1+17+1) = 291 */
```

Algorithm D.5: Complete MSR: F_edges(M_i^2)

```
Data  : M_i^2
Result: {M_i^2{M^3}}
begin
    GET({M_i^2{M^0}}); /* 3 steps */
    for each M_i^0 ∈ {M_i^2{M^0}} /* repeat 3 times */ do
        GET({M_i^0{M^3}}); /* 23 steps */
        for each M_i^3 ∈ {M_i^0{M^3}} /* repeat 23 times */ do
            GET({M_i^3{M^2}}); /* 584 steps */
            if FIND({M_i^3{M^2}}, M_i^2) = true /* lg4+1 steps */
                if FIND({M_i^2{M^3}}, M_i^3) = false /* lg2+1 steps */
                    PUT({M_i^2{M^3}}, M_i^3); /* at most 2 steps */
                endif
            endif
        endfor
    endfor
end

/* Time = 3+3(23+23(584+lg4+1+lg2+1))+2 = 40,784 */
```

Algorithm D.6: Complete MSR: F_regions(M_i^2)

```
Data  : M_i^3
Result: {M_i^3{M^1}}
begin
    for var ← 0 to 5 /* repeat 6 times */ do
        M_i^0 ← M_i^3{M^0}_Rev[var][0]; /* 3 (2 GET's + ASSIGN) steps */
        M_j^0 ← M_i^3{M^0}_Rev[var][1]; /* 3 steps */
        M_i^1 ← E_exist(M_i^0, M_j^0); /* 71+1 steps */
        if M_i^1 = NULL /* 1 step */
            M_i^1 ← M_createE(M_i^0, M_j^0); /* 16+1 steps */
        endif
        M_i^3{M^1}_var ← M_i^1; /* 1 step */
    endfor
end

/* Time = 6(3+3+72+1+17+1) = 582 */
```

Algorithm D.7: Complete MSR: R_edges(M_i^3)

Data : M_i^3
Result: $\{M_i^3\{M^2\}\}$
begin
  for var $\leftarrow$ *0 to 3 /* repeat 4 times */* **do**
    $M_i^0 \leftarrow M_i^3\{M^0\}_{Rfv[var][0]}$; */* 3 (2 GET's + ASSIGN) steps */*
    $M_j^0 \leftarrow M_i^3\{M^0\}_{Rfv[var][1]}$; */* 3 steps */*
    $M_k^0 \leftarrow M_i^3\{M^0\}_{Rfv[var][2]}$; */* 3 steps */*
    $M_i^2 \leftarrow$ F_exist(M_i^0, M_j^0, M_k^0); */* 106+1 steps */*
    if $M_i^2 = NULL$ */* 1 step */*
      $M_i^2 \leftarrow$ M_createF(M_i^0, M_j^0, M_k^0); */* 27+1 steps */*
    endif
    $M_i^3\{M^2\}_{var} \leftarrow M_i^2$; */* 1 step */*
  endfor
end

/ Time = 4(3+3+3+107+1+28+1) = 584 */*

Algorithm D.8: Complete MSR: R_faces(M_i^3)

Data : M_i^1, M_j^1, M_k^1
Result: M_i^2 bounded by M_i^1, M_j^1, M_k^1
begin
  $M_i^0 \leftarrow M_i^1\{M^0\}_0$; */* 2 steps */*
  $M_j^0 \leftarrow M_i^1\{M^0\}_1$; */* 2 steps */*
  if $M_i^0 \neq M_j^1\{M^0\}_0$ *and* $M_j^0 \neq M_j^1\{M^0\}_0$ */* 4 steps */*
    $M_k^0 \leftarrow M_j^1\{M^0\}_0$; */* 2 steps */*
  else
    $M_k^0 \leftarrow M_j^1\{M^0\}_1$; */* 2 steps */*
  endif
  RETURN F_exist(M_i^0, M_i^0, M_i^0); */* 106+1 step */*
end

/ Time = 2+2+4+2+107 = 117 */*

Algorithm D.9: Complete MSR: F_exist(M_i^1, M_j^1, M_k^1)

Data : M_i^0, M_j^0, M_k^0

Result: create M_i^2 bounded by M_i^0, M_j^0, M_k^0

begin

CREATE a face M_i^2; */* 1 step */*

PUT({M{M^2}}, M_i^2); */* 1 step */*

/ update link from face to vertex and vice versa */*

$M_i^2\{M^0\}_0 \leftarrow M_i^0$; */* 1 step */*

$M_i^2\{M^0\}_1 \leftarrow M_j^0$;

$M_i^2\{M^0\}_2 \leftarrow M_k^0$;

PUT_UNIQ({M_i^0{M^2}}, M_i^2); */* $\lg 35$+2 steps */*

PUT_UNIQ({M_j^0{M^2}}, M_i^2);

PUT_UNIQ({M_k^0{M^2}}, M_i^2);

RETURN M_i^2; */* 1 step */*

end

/ Time = 1+1+3(1+*$\lg 35$*+2)+1* $\approx$ *27 */*

Algorithm D.10: Complete MSR: M_createF(M_i^0, M_j^0, M_k^0)

Data : M_i^1, M_j^1, M_k^1

Result: create M_i^2 bounded by M_i^1, M_j^1, M_k^1

begin

CREATE a face M_i^2; */* 1 step */*

PUT({M{M^2}}, M_i^2); */* 1 step */*

/ update link from face to vertex*/*

$M_i^2\{M^0\}_0 \leftarrow M_i^1\{M^0\}_0$; */* 2 (GET and ASSIGN) steps */*

$M_i^2\{M^0\}_1 \leftarrow M_i^1\{M^0\}_1$; */* 2 steps */*

if *dir[0]=1 /* 1 step */*

  if *FIND*($\{M_j^1\{M^0\}\}$, $M_i^2\{M^0\}_0$) *= true /** $\lg 2$*+1 steps */*

    $M_i^2\{M^0\}_2 \leftarrow M_j^1\{M^0\}_1$; */* 2 steps */*

  else

    $M_i^2\{M^0\}_2 \leftarrow M_k^1\{M^0\}_1$;

  endif

else

  if *FIND*($\{M_j^1\{M^0\}\}$, $M_i^2\{M^0\}_0$) *= true*

    $M_i^2\{M^0\}_2 \leftarrow M_k^1\{M^0\}_1$;

  else

    $M_i^2\{M^0\}_2 \leftarrow M_j^1\{M^0\}_1$;

  endif

endif

/ update link from vertex to face and edge to face */*

for var $\leftarrow$ *0 to 2 /* repeat 3 times */* **do**

  PUT_UNIQ($\{M_i^2\{M^0\}_{var}\{M^2\}\}$, M_i^2); */* $\lg 35$*+2 steps */*

endfor

RETURN M_i^2; */* 1 step */*

end

/ Time = 1+1+2+2+1(if dir)+2(if FIND)+2+3(*$\lg 35$*+2)+1* $\approx$ *33 */*

Algorithm D.11: Complete MSR: M_createF(M_i^1, M_j^1, M_k^1, *dir*[3])

sData : M_i^0, M_j^0, M_k^0, M_l^0

Result: create M_i^3 bounded by M_i^0, M_j^0, M_k^0, M_l^0

begin
CREATE a region M_i^3; /* *1 step* */
PUT({M{M^3}}, M_i^3); /* *1 step* */
/* *update link from vertex to region and vice versa* */
for var ← *i to l /* repeat 4 times */* **do**
PUT_UNIQ({M_{var}^0{M^3}}, M_{var}^0); /* lg 23+*2* */
{M_i^3{M^0}} ← M_{var}^0; /* *1 step* */
endfor
RETURN M_i^3; /* *1 step* */
end

/* Time = *1+1+4(*lg 23+*2+1)+1* ≈ *33* */

Algorithm D.12: Complete MSR: M_createR(M_i^0, M_j^0, M_k^0, M_l^0)

Data : M_i^2, M_j^2, M_k^2, M_l^2

Result: create M_i^3 bounded by M_i^2, M_j^2, M_k^2, M_l^2

begin
CREATE a region M_i^3; /* *1 step* */
PUT({M{M^3}}, M_i^3); /* *1 step* */
/* *update* {M_i^3{M^0}} */
M_i^3{M^0}$_0$ ← M_i^2{M^0}$_0$; /* *2 steps* */
M_i^3{M^0}$_1$ ← M_i^2{M^0}$_1$;
M_i^3{M^0}$_2$ ← M_i^2{M^0}$_2$;
GET({M_j^2{M^0}}); /* *3 steps* */
for M_i^0 ∈ {M_j^2{M^0}} /* *repeat 3 times* */ **do**
if *FIND(*{M_i^2{M^0}}, M_i^0*) = false /* *lg 3+1 steps* */
M_i^3{M^0}$_3$ ← M_i^0;/* *at most 1 step* */
endif
endfor
/* *update link from vertex to region -* lg 23+*2 steps for each* */
for var ← *0 to 3 /* repeat 4 times */* **do**
PUT_UNIQ({M_i^3{M^0}$_{var}${M^3}}, M_i^3);
endfor
RETURN M_i^3; /* *1 step* */
end

/* Time = *1+1+3·2+3+3(*lg 3+*1)+1+4(*lg 23+*2)+1* ≈ *46* */

Algorithm D.13: Complete MSR: M_createR(M_i^2, M_j^2, M_k^2, M_l^2)