EFFICIENT DISTRIBUTED MESH DATA STRUCTURE FOR PARALLEL AUTOMATED ADAPTIVE ANALYSIS

Mark S. Shephard and E. Seegyoung Seol

A JOHN WILEY & SONS, INC., PUBLICATION

CONTENTS

List of Figures v

List of Tables vii

1 Efficient Distributed Mesh Data Structure for Parallel Automated Adaptive Analysis 1

1.1 Introduction 1
1.1.1 Nomenclature 2
1.2 Requirements for a Parallel Infrastructure for Adaptively Evolving Unstructured Meshes 2
1.3 Structure of the Flexible Mesh Database 3
1.4 Parallel Flexible Mesh DataBase (FMDB) 7
1.4.1 A Partition Model 9
1.5 Mesh Migration for Full Representations 11
1.6 Mesh Migration for Reduced Representations 16
1.6.1 Mesh Representation Adjustment 17
1.6.2 Algorithm of Mesh Migration with Reduced Representations 20
1.6.3 Summary 23
1.7 Parallel Adaptive Applications 24
1.8 Closing Remark 25

References 29

LIST OF FIGURES

1.1. MRM's of 3D mesh representation 4

1.2. Example of 3D MRM union 5

1.3. Example of 3D MRM optimization 7

1.4. Distributed mesh on three partitions P_0, P_1 and P_2 [24] 8

1.5. Example 3D mesh distributed on 3 partitions 9

1.6. Distributed mesh and its association with the partition model via partition classifications 10

1.7. Example of 2D mesh migration 12

1.8. Example 2D mesh with the MSR 17

1.9. MRM adjustments for distributed incomplete meshes 17

1.10. Steps of 2D mesh migration with the MSR 19

1.11. Example of 2D mesh load balancing: (left) partition objects are tagged with their destination pids (right) mesh after load balancing 24

1.12. Parallel mesh adaptation I: (left) initial 36 tet mesh, (right) adapted approx. 1 million tet mesh. 24

1.13. 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). 26

1.14. 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) the front view of adapted mesh from the second adaptive loop (approx. 24 million tet), (d) the back view of (c). 27

LIST OF TABLES

1.1. The contents of vector *entsToUpdt* after Step 1 15

CHAPTER 1

FLEXIBLE DISTRIBUTED MESH DATA STRUCTURE FOR PARALLEL ADAPTIVE ANALYSIS

1.1 INTRODUCTION

An efficient distributed mesh data structure is needed to support parallel adaptive analysis since it strongly influences the overall performance of adaptive mesh-based simulations. In addition to the general mesh-based operations [4], 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.

Issues associated with supporting parallel adaptive analysis on a given unstructured mesh include dynamic mesh load balancing techniques [11, 34, 8, 32], and data structure and algorithms for parallel mesh adaptation [21, 20, 17, 23, 9, 27, 24]. The focus of this chapter is a parallel mesh infrastructure capable of handling general non-manifold [19, 35] models and effectively supporting automated adaptive analysis. The mesh infrastructure, referred to as Flexible distributed Mesh DataBase (FMDB), is a distributed mesh data management system that is capable of shaping its data structure dynamically based on the user's requested mesh representation [29].

1.1.1 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. $d = 0$ for a vertex, $d = 1$ for an edge, $d = 2$ for a face, and $d = 3$ for a region.
$\{\partial(V_i^d)\}$	set of entities on the boundary of V_i^d.
$\{V_i^d\{V^q\}\}$	a set of entities of dimension q in model V that are adjacent to V_i^d.
$V_i^d\{V^q\}_j$	the j^{th} entity in the set of entities of dimension q in model V that are adjacent to V_i^d.
$U_i^{d_i} \sqsubset V_j^{d_j}$	classification indicating the unique association of entity $U_i^{d_i}$ with entity $V_j^{d_j}$, $d_i \leq d_j$, where U, $V \in \{G, P, M\}$ and U is lower than V in terms of a hierarchy of domain decomposition.
$\mathscr{P}[M_i^d]$	set of partition id(s) where entity M_i^d exists.
Examples	
$\{M\{M^2\}\}$	the set of all the faces in the mesh.
$\{M_3^1\{M^3\}\}$	the 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.

1.2 REQUIREMENTS FOR A PARALLEL INFRASTRUCTURE FOR ADAPTIVELY EVOLVING UNSTRUCTURED MESHES

The design of a parallel mesh infrastructure is dictated by the type of meshes to be stored, the mesh level information and functions to be performed by the applications and the parallel computational environment that will be applied. This chapter considers the parallel representation of adaptively evolving conforming unstructured meshes that can include multiple mesh entity topological types.

The mesh information needed and the functions that must be carried out on the mesh are a strong function of the specific application operations to be performed. In the case of adaptive analysis the most demanding of operations performed are the mesh modifications associated with adapting the mesh [1, 5, 9, 16, 15, 20]. In the case of curved geometries the mesh modifications must be performed such that the geometric approximation of the domain is improved as the mesh is modified [15]. This requires the mesh be related back to the original geometry definition. The most common form of geometric representation is a boundary representation defined in terms of topological entities including vertices, edges, faces and regions and the adjacencies between the entities [19, 35]. This leads to consideration of a boundary representation for the mesh in which the mesh entities are easily related to geometric model entities, and the topological entities and their adjacencies are used to support the wide range of mesh information need of mesh modification operations [4, 6, 13, 25]. The three basic functional requirements of a general topology-based mesh data structure are topological entities, geometric classification, and adjacencies between entities.

Topology provides an unambiguous, shape-independent, abstraction of the mesh. With reasonable restrictions on the topology [4], a mesh can be effectively 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.

Geometric classification defines the relation of a mesh to a geometric model. The unique association of a mesh entity of dimension d_i, $M_i^{d_i}$, to the geometric model entity of dimension d_j, $G_j^{d_j}$, where $d_i \leq d_j$, 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.

Adjacencies describe how mesh entities connect to each other. For an entity of dimension d, adjacency, denoted by $\{M_i^d\{M^q\}\}$, returns all 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 ($d < q$).

There are many options in the design of the mesh data structure in terms of the entities and adjacencies stored [4, 6, 13, 25]. If a mesh representation stores all *0* to *d* level entities explicitly, it is a $full$ representation, otherwise, it is a $reduced$ representation. *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 representation is complete, otherwise, it is incomplete. In terms of the requirements to be the basic mesh data structure to be used for parallel adaptive computations it must be able to provide all the mesh adjacencies needed by the operations to be performed and needs to be able to provide them effectively, which does require they be provided in O(1) time since any requirement to provide an adjacency in time at all proportional to the mesh size is not acceptable unless it is only done once. Although this does not strictly require the use of a complete mesh representation, the wide range of adjacencies typically needed by mesh modification will force one to select a complete representation. Note that the ability to meet the requirements of a complete representation does not require it be full [4, 6, 25]. However, the implementation of a complete mesh data structure that is not full is more complex, particularly in parallel [28].

The requirements placed on the mesh are a function of the parallel computing environment. To ensure the greatest flexibility it is desirable to be able to distribute the mesh to the processors in a distributed memory computing environment with the need to store little more than the mesh entities assigned to that processor. Is is also a desirable feature that with the exception of possibly added information, the mesh data structure on the individual processors be identical to that of the serial implementation.

1.3 STRUCTURE OF THE FLEXIBLE MESH DATABASE

One approach to the implementation of a mesh data structure is to select a fixed representation that meets the full set of needs of the applications to be executed. In those cases where the specific adjacencies needed are not know in advance, one will want to be sure the representation selected is complete so that any adjacency can be obtained with acceptable efficiency. Since explicitly storing all the adjacencies used is typically unacceptable, one want to select a representation that can efficiently obtain those that are not stored. Even when a complete representations is used the constant on some of the O(1) time adjacencies recovery operations can be quite large [4]. An alternative approach taken recently is to employ a flexible mesh data representation that at the time an application is initiated can select the to adjacencies stored to be well suited to the application at hand [25, 24, 29, 28].

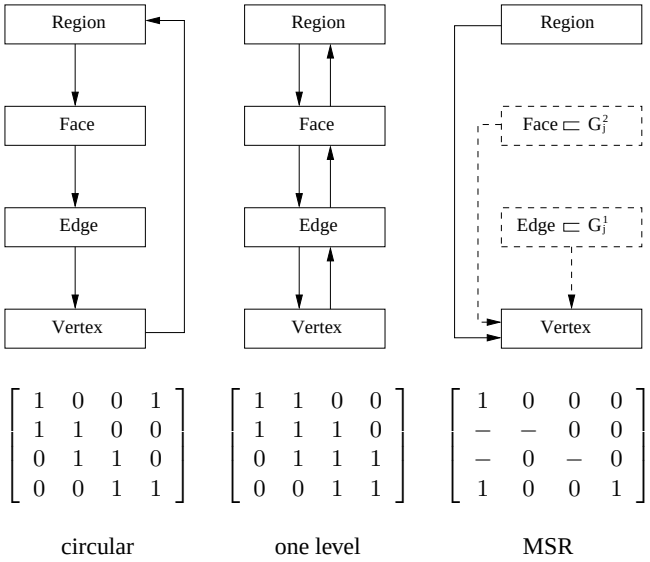


Figure 1.1. MRM's of 3D mesh representation

This section discusses the design of a flexible mesh database, FMDB, which enables the mesh database to shape its structure based on the representational needs. A mesh representation matrix, MRM, is used to define the mesh entities and adjacency information to be maintained. The initial MRM is input by the user based on their knowledge of the entities and adjacencies to be used where the "user" is an application program that interacts with a mesh information in some manner, including changing it. The MRM provided by the user is then modified to optimize its ability to provide the indicated information efficiently without the storage of entities that can be quickly determined based on others that are stored. To ensure the user requested representation remains correct, even with mesh modification, the mesh entity creation/deletion operators used in the application are set to the proper ones dynamically to maintain the needed representation.

For cases when the needed mesh representation is not known in advance, the Dynamic Mesh Usage Monitor (DMUM) was developed [28]. 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.

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 [24] 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.

emphMesh Representation Matrix (MRM): The MRM is 4×4 matrix $\mathcal{R}$ where diagonal element $\mathcal{R}_{i,i}$ is equal to 1 if mesh entities of 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 $0 \leq i, j \leq 3$.

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

Figure 1.2. Example of 3D MRM union

Figure 1.1. gives the MRM's of the circular, one-level and minimum sufficient representation. The circular and one level representation are popular full representations that are complete. The minimum sufficient representation is the minimal representation with no loss of information. In the MSR adjacency graph given in Figure 1.1., $\mathcal{R}_{0,0}$, $\mathcal{R}_{3,3}$, 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 $-$ since 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 the represented edges and faces. The remaining $\mathcal{R}_{i,j}$, $i \neq j$, are 0.

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:

Step 1: Union the user-requested representation with the minimum sufficient representation: Unioning 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 1.2. 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 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.

Step 2: Optimize the representation: The second step alters the MRM to provide the optimal representation that satisfies the user requested representation at the minimum memory cost. Optimization is performed as follows:

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* 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 the second step determines the level of bounding entities in face/region creation to expedite the adjacency update. Note that 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 1.3. 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).

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 when mesh entities are created or deleted. For example, suppose an application requires adjacency $\{M^0\{M^2\}\}$. To keep $\{M^0\{M^2\}\}$, face creation must be followed by adding M^2_i into $\{M^0_i\{M^2\}\}$, and face deletion must be followed by deleting M^2_i from $\{M^0_i\{M^2\}\}$, for each $M^0_i \in \{\partial(M^2_i)\}$.

The mesh data structure is shaped by setting the mesh operators that create or delete the mesh entities to the proper ones in order to preserve the user-requested representation. Shaping the representations using dynamic setting of mesh operators doesn't involve any

representation after union after optimization

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

$$Case\ 2: \quad \mathcal{R}^e = \begin{bmatrix} 1 & 0 & 1 & 1 \\ - & - & 0 & 0 \\ - & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix} \quad \mathcal{R}^f = \begin{bmatrix} 1 & 0 & 1 & 0 \\ - & - & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

$$Case\ 3: \quad \mathcal{R}^h = \begin{bmatrix} 1 & 0 & 0 & 0 \\ - & 1 & 1 & 0 \\ - & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \quad \mathcal{R}^i = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Figure 1.3. Example of 3D MRM optimization

mesh size operation such as search and traversal, and maintains a valid representation under all mesh level operations.

Consider the adjacency request $\{M^d\{M^p\}\}$ and $\{M^d\{M^q\}\}$, $p < d < q$. 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 $M_i^d \in \{\partial(M_i^q)\}$.
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 [3, 10, 26], 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 $M_i^d \in \{\partial(M_i^q)\}$.

1.4 PARALLEL FLEXIBLE MESH DATABASE (FMDB)

A *distributed mesh* is a mesh divided into partitions for distribution over a set of processors for parallel computation.

A *Partition* P_i consists of the set of mesh entities assigned to i^{th} processor. For each partition, the unique partition id can be given. Each partition is treated as a serial mesh

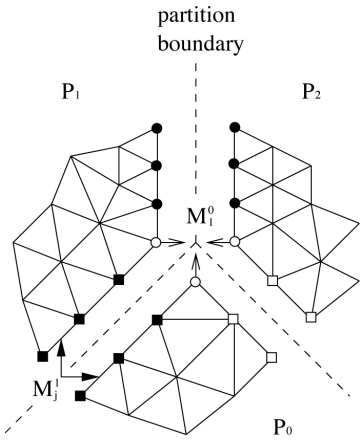


Figure 1.4. Distributed mesh on three partitions P_0, P_1 and P_2 [24]

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 a single partition. Figure 1.4. 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 $\mathscr{P}$ that returns a set of partition id(s) where M_i^d exists. Based on the *Residence partition equation* that operates as follows: If $\{M_i^d\{M^q\}\} = \emptyset$, $d < q$, then $\mathscr{P}[M_i^d]$ = $\{p\}$ where p is the id of a partition on which 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 other 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 any 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 $\{M_i^d\{M_j^{d+1}\}\} = \emptyset$. Otherwise, $\mathscr{P}[M_i^d]$ is $\cup\, \mathscr{P}[M_j^{d+1} \mid M_i^d \in \{\partial(M_j^{d+1})\}]$. For instance, for the 3D mesh depicted in Figure 1.5., where M_1^3 and M_1^2 are on P_0, M_2^3 and M_2^2 are on P_1 (shaded), and M_1^1 is on P_2 (thick line), residence partition ids of M_1^0 (big black dot) 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 determined before moving the mesh entities. The residence partition equation implies that once the destination partition id of 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.

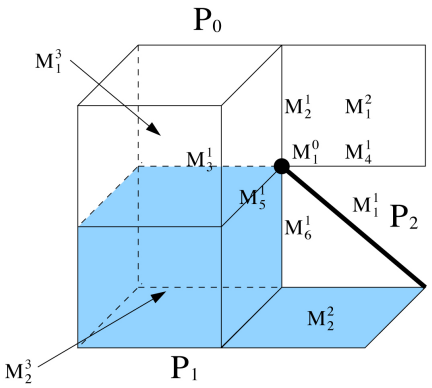


Figure 1.5. Example 3D mesh distributed on 3 partitions

The need for a formal definition of the residence partition is due to the fact that unlike manifold models where only the highest order mesh entities need to assigned to a partition, non-manifold geometries can have lower order mesh mesh entities not bounded by any higher order mesh which thus must be assigned to a partition. Thus, a *Partition object* is 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.

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. This is done by maintaining the *Remote partition* where a mesh entity is duplicated, and the *Remote copy* memory location on the remote partition. In a parallel adaptive analysis the mesh and its partitioning can change thousands of times. Therefore an efficient mechanism to update mesh partitioning that keep the links between partitions updated is 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. For the dynamic entity ownership, there can be several options in determining owning processor of mesh entities. With the FMDB, 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.

1.4.1 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. The partition model can be viewed as a part of hierarchical domain decomposition. Its sole purpose is to represent mesh

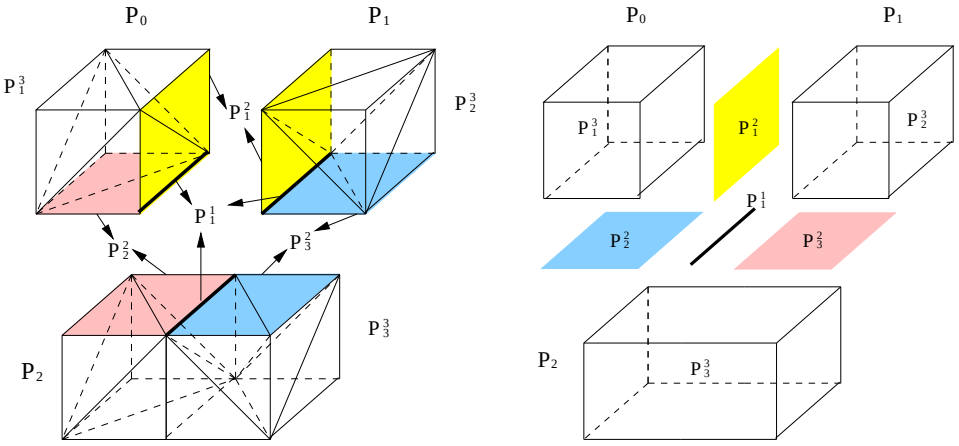


Figure 1.6. Distributed mesh and its association with the partition model via partition classifications

partitioning in topology and support mesh-level parallel operations through inter-partition boundary. 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 operations across multiple processors.

A *Partition (model) entity*, P_i^d, is a topological entity 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. Given this the *Partition classification* is defined as 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 and is denoted $M_i^{d_i} \sqsubset P_j^{d_j}$. Figure 1.6. illustrates a distributed 3D 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 and they are duplicated on three partitions P_0, P_1, and P_2. The mesh vertices, edges and faces on the shaded planes are duplicated on two partitions and they are classified on the partition face pointed with each arrow. The remaining mesh entities are not duplicated, therefore they are classified on the corresponding partition region.

The following rules govern the creation of the 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 (residence 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.

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 1.6.(a), any mesh faces, edges and vertices that are not on shaded planes nor on the thick black line are classified on the partition region by inheriting partition classification from the regions it bounds. 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 a new partition entity of dimension *d* is created for the partition classification of entity M_i^d.

1.5 MESH MIGRATION FOR FULL 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) are the important factors for high performance in parallel adaptive mesh-based simulations. Since the mesh migration process must have be able to deal with any partition mesh entity, it can only be efficient with complete representations. The algorithms presented in this subsection also assume a full representation. The next subsection will indicate the extensions required for migration of reduced representations.

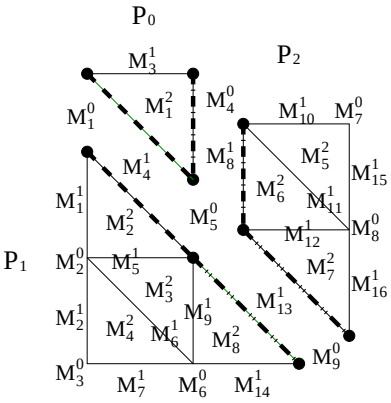
Figure 1.7.(a) and (b) illustrate the 2D partitioned mesh and its associated partition model to be used as an example in this discussion. In Figure 1.7.(a), the partition classification of entities on the partition boundaries is denoted with the lines of the same pattern in Figure 1.7.(b). For instance, M_1^0 and M_4^1 are classified on P_1^1, and depicted with the dashed lines as P_1^1. In Figure 1.7.(b). the owning partition of a partition model edge (resp. vertex) is illustrated with thickness (resp. size) of lines (resp. circle). 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 the three 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, $POsToMove$. Given the initial partitioned mesh in Figure 1.7.(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.

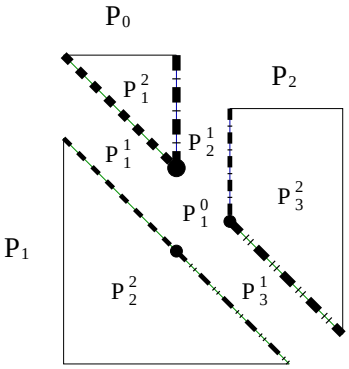
Algorithm 1.1 is the pseudo code of the mesh migration procedure where the pseudo code conventions in Reference [7] are used.

Step 1: Preparation: For a given list of partition objects to migrate, Step 1 collects a set of entities to be updated by migration. The entities collected for the update are maintained in vector *entsToUpdt*, where *entsToUpdt*$[i]$ contains the entities of dimension i, $i = 0, 1, 2, 3$. With a single program multiple data paradigm [22] in parallel, each partition maintains the separate *entsToUpdt*$[i]$ with different contents.

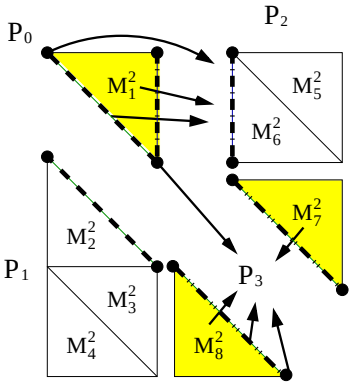
For the example mesh, the contents of *entsToUpdt* by dimension for each partition is given in Table 1.1. Only entities listed in Table 1.1. will be affected by the remaining steps in terms of their location and partitioning-related internal data. *entsToUpdt*$[2]$ contains the



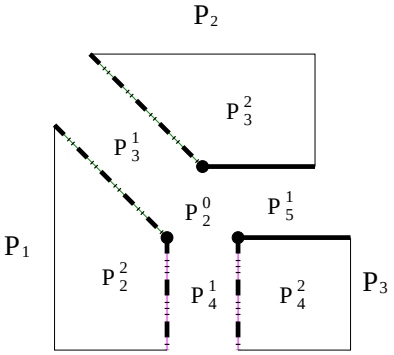
(a) initial mesh



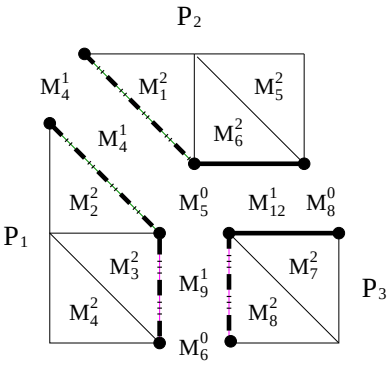
(b) partition model of (a)



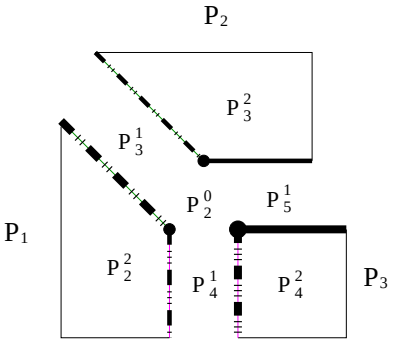
(c) mesh during migration



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



(e) final mesh



(e) final partition model with ownership

Figure 1.7. Example of 2D mesh migration

Data : M, $POsToMove$
Result : migrate partition objects in $POsToMove$
begin
  /* *Step 1: collect entities to process and clear partitioning data.* */
  for *each* $M_i^d \in POsToMove$ **do**
    insert M_i^d into vector *entsToUpdt*$[d]$;
    reset partition classification and $\mathscr{P}$;
    for *each* $M_j^q \in \{\partial(M_i^d)\}$ **do**
      insert M_j^q into *entsToUpdt*$[q]$;
      reset partition classification and $\mathscr{P}$;
    endfor
  endfor
  /* *Step 2: determine residence partition.* */
  for *each* $M_i^d \in$ entsToUpdt$[q]$ **do**
    set $\mathscr{P}$ of M_i^d;
  endfor
  do one-round communication to unify $\mathscr{P}$ of partition boundary entities;
  /* *Step 3: update partition classification and collect entities to remove.* */
  for $d \leftarrow$ *3 to 0* **do**
    for *each* $M_i^d \in$ entsToUpdt$[d]$ **do**
      determine partition classification;
      if $P_{local} \notin \mathscr{P}[M_i^d]$ **do**
        insert M_i^d into $entsToRmv[d]$;
      endif
    endfor
  endfor
  /* *Step 4: exchange entities.* */
  for $d \leftarrow$ *0 to 3* **do**
    M_exchngEnts($entsToUpdt[d]$); /* *Algorithm 1.2* */
  endfor
  /* *Step 5: remove unnecessary entities.* */
  for $d \leftarrow$ *3 to 0* **do**
    for *each* $M_i^d \in$ *entsToRmv*$[d]$ **do**
      if M_i^d *is on partition boundary* **do**
        remove copies of M_i^d on other partitions;
      endif
      remove M_i^d;
    endfor
  endfor
  /* *Step 6: update ownership.* */
  for *each* P_i^d *in* P **do**
    owning partition of $P_i^d \leftarrow$ the poorest partition among $\mathscr{P}[P_i^d]$;
  endfor
end

Algorithm 1.1: M_migrate(M, $POsToMove$)

Data : $entsToUpdt[d]$
Result : create entities on the destination partitions and update remote copies
begin
  /* *Step 4.1: send a message to the destination partitions* */
  for *each* $M_i^d \in entsToUpdt[d]$ **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 *is not a 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 1.2: M_exchngEnts($entsToUpdt[d]$)

Table 1.1. The contents of vector *entsToUpdt* after Step 1

	P_0	P_1	P_2
$entititesToProcess[0]$	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
$entititesToProcess[1]$	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
$entititesToProcess[2]$	M_1^2	M_8^2	M_7^2

mesh faces to be migrated from each partition. *entsToUpdt*$[1]$ contains the mesh edges which bound any mesh face in *entsToUpdt*$[2]$ and their remote copies. *entsToUpdt*$[0]$ contains the mesh vertices that bound any mesh edge in *entsToUpdt*$[1]$ and their remote copies. The partition classification and $\mathscr{P}$ of entities in *entsToUpdt* are cleared for further update.

Step 2: Determine residence partition(s): Step 2 determines $\mathscr{P}$ of the entities according to the residence partition equation. For each entity which bounds the higher order entity, it should be determined if the entity will exist on the current local partition, denoted as P_{local}, after migration to set $\mathscr{P}$. Existence of the entity on P_{local} after migration is determined by checking adjacent partition objects, i.e., checking if there's any adjacent partition object to reside on P_{local}. One round of communication is performed at the end to exchange $\mathscr{P}$ of the partition boundary entities to unify them between remote copies.

Step 3: Update the partition classification and collect entities to remove: For the entities in *entsToUpdt*, based on $\mathscr{P}$, Step 3 refreshes the partition classification to reflect a new updated partition model after migration, and determines the entities to remove from the local partition, P_{local}. An entity is determined to remove from its local partition if $\mathscr{P}$ of the entity doesn't contain P_{local}. Figure 1.7.(d) is the partition model updated based on the new partition topology.

Step 4: Exchange entities: 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, creates entities on the destination partitions, and updates the remote copies of the entities created on the destination partitions. Algorithm 1.2 illustrates the pseudo code that exchanges the entities contained in $entsToUpdt[d]$, $d = 0, 1, 2, 3$.

Step 4.1 sends the messages to destination partitions to create new mesh entities. Consider entity M_i^d duplicated on several partitions needs to be migrated to 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 $broadcaster$, denoted as P_{bc}. The broadcaster is in charge of creating as well as updating M_i^d over all partitions. The arrows in Figure 1.7.(c) indicate the broadcaster of each entity to migrate based on minimum partition id. Before sending a message to P_i, M_i^d is checked if it already exists on P_i using the remote copy information and ignored if exists.

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 unique vertex id (if vertex), entity shape information, required entity adjacencies, geometric classification information, residence partition(s) for setting partition classification, and 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 (Step 4.2). 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 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, they add it to their remote copy.

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.

Step 6: Update ownership: Step 6 updates the owning partition of the partition model entities based on the rule of the poor-to-rich partition ownership. The partition model given in Figure 1.7.(e) is the final partition model with ownership.

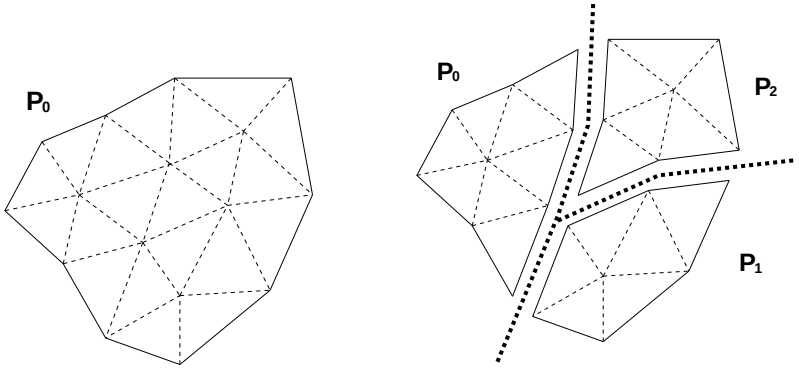
FMDB is implemented in C++, and uses STL (Standard Template Library) [30], functors [10], templates [33], singletons [12], and generic programming [2] for the purpose of achieving reusability of the software. MPI (Message Passing Interface) [22, 18] and Autopack [18] are used for efficient parallel communications between processors. The Zoltan library [36] is used to make partition assignment during dynamic load balancing.

1.6 MESH MIGRATION FOR REDUCED REPRESENTATIONS

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 1.8.(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 1.8.(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 the previous section, the representational requirements for flexible distributed meshes are the following:

- For each partition object M_i^d, downward adjacent entities $\in \{\partial(M_i^d)\}$.



(a) serial mesh (b) partitioned mesh

Figure 1.8. Example 2D mesh with the MSR

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

(a) input MSR (b) after 1^{st} adjustment (c) after 2^{nd} adjustment

Figure 1.9. MRM adjustments for distributed incomplete meshes

- For each downward adjacent entity of M_i^d, M_j^p, the other partition objects adjacent to M_j^p, and the 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. 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.

1.6.1 Mesh Representation Adjustment

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 1.9.(a) which is the MSR, $\mathcal{R}_{0,1}$ and $\mathcal{R}_{0,2}$ are set to $-$ as seen in Figure 1.9.(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.

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.

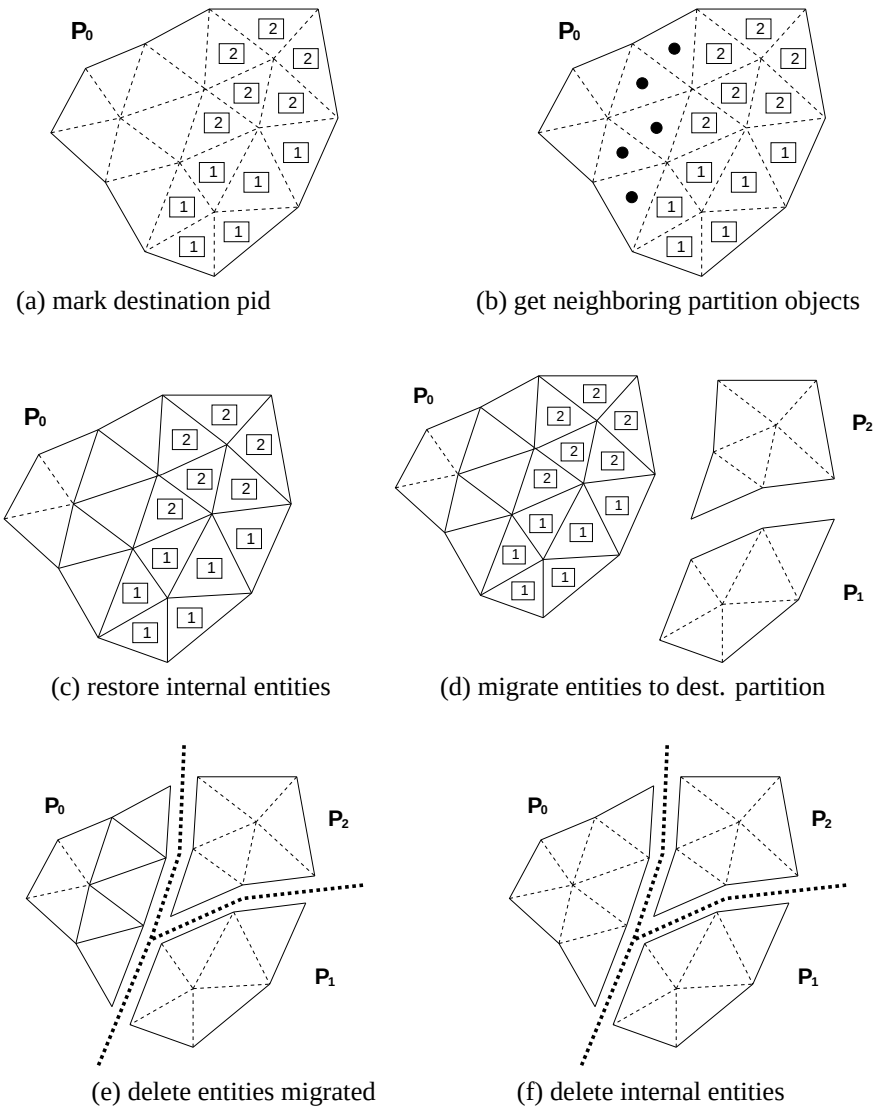
With flexible mesh representations, especially in case where upward adjacency to the level of partition objects is not available, to determine if an entity will exist on the current partition after migration or not while creating partition object M_i^d, we must store adjacency $\{M_i^0\{M_i^d\}\}$ for each $M_i^0 \in \{\partial(M_i^d)\}$ to avoid the need for global searches.

This process 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.

This leads to the second step of MRM adjustment that sets $\{M^0\{M^3\}\}$ to 1 in order to support neighboring partition objects of level 3 as seen in Figure 1.9.(c). The penalty of this option is storing unrequested adjacency information. However, these adjacencies are necessary to avoid mesh-size dependent operations.

Data : M, $POsToMigrate$
Result : migrate partition objects in $POsToMigrate$
begin
  / STEP A: collect neighboring partition objects. */*
  For each partition object in $POsToMigrate$, collect neighboring partition objects and store them in $neighborPOs$;
  / STEP B: restore downward entities. */*
  M_buildAdj_URR(M,$POsToMigrate$,$neighborPOs$);
  / STEP 1: collect entities to process and clear partitioning data. */*
  Run STEP 1 in Algorithm 1.1;
  / STEP 2: determine residence partition. */*
  M_setResidencePartition_URR($POsToMigrate$, $neighborPOs$);
  / STEP 3: update p. classification and collect entities to remove. */*
  Run STEP 3 in Algorithm 1.1;
  / STEP 4: exchange entities.*/*
  for $d \leftarrow$ *0 to 3* **do**
    M_exchangeEnts_URR($entitiesToUpdate[d]$);
  endfor
  / STEP 5: remove unnecessary entities. */*
  Run STEP 5 in Algorithm 1.1;
  / STEP 6: update ownership. */*
  Run STEP 6 in Algorithm 1.1;
  / STEP C: remove unnecessary interior entities and adjacencies. */*
  M_destoryAdj_URR(M, $entitiesToUpdate$, $neighborPOs$);
end

Algorithm 1.3: M_migrate_URR(M, $POsToMigrate$)



(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 1.10. Steps of 2D mesh migration with the MSR

1.6.2 Algorithm of Mesh Migration with Reduced Representations

The mesh migration procedure $M\_migrate$ based on use of complete mesh representations is now extended to work with any mesh representation options. Given the $POsToMove$, the overall procedure for the mesh migration is the following:

1. Collect neighboring partition objects.
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 1.10. depicts the 2D mesh migration procedure with a reduced representation. For the given list of partition objects to migrate, $POsToMove$, (Figure 1.10.(a)), first collect the partition objects which are adjacent to any partition object in $POsToMove$ and store them in a separate container named $neighborPOs$ (Figure 1.10.(b)). Second, for partition objects in $POsToMove$ or $neighborPOs$, restore their interior entities and associated downward adjacencies (Figure 1.10.(c)). 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 (Figure 1.10.(d)). The entities collected to remove are deleted from the local partition (Figure 1.10.(e)). Finally, the interior entities and adjacencies restored in the second step are removed to keep the original requested mesh representation (Figure 1.10.(f)). Algorithm 1.3 is pseudo code that migrates partition objects with flexible mesh representations.

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.

Step B: Restore downward entities: In Step B, iterating over $POsToMigrate$ and $neighborPOs$, $M\_buildAdj\_URR$ restores needed non-existing downward interior entities of each partition object.

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.

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 1.4: M_setResidencePartition_URR($POsToMigrate$, $entitiesToUpdate$, $neighborPOs$)

Step 2: Determine residence partition: Step 2 determines $\mathscr{P}$ of entities collected in $entitiesToUpdate$ (Algorithm 1.4). 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.

Step 3: Determine partition classification and entities to remove: For each entity in $entitiesToUpdate$, determine the partition classification and determine if it will be removed from the local partition.

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
    if $\mathcal{R}_{d,d} \neq 1$
      if M_i^d *will not be on p.boundaries or not equally classified*
        continue;
      endif
    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
  Run STEP 4.2 to 4.4 in Algorithm 1.2;
end

Algorithm 1.5: M_exchangeEnts_URR($entitiesToUpdate[d]$)

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 1.2 has been slightly modified to Algorithm 1.5 in order to work with any mesh representation options. Differences from Algorithm 1.2 are the following:

- The dimension of the entities used to create(define) faces and regions are determined based on the MRM.
- Not all interior mesh entities are migrated to the destination partitions. Interior entities are migrated to destination partitions only when they will be on the partition boundaries in new mesh partitioning topology after migration.

Figure 1.10.(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.

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. Figure 1.10.(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.

Step 6: Update entity ownership: Step 6 updates ownership of partition model entities. See §**??**.

Step C: Restore mesh representation: This step restores the mesh representation modified to have interior entities and associated downward adjacencies in Step B to the original 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.

1.6.3 Summary

The following are the comparisons of the migration procedures, $M\_migrate\_URR$ in Algorithm 1.3 (Steps A, B, 1 to 6, C) and $M\_migrate$ in Algorithm 1.1 (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. [-20pt]

- 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.
- 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.

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 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 [28].

During parallel adaptive analysis, the mesh data needs often re-partitioning to maintain load balance while keeping the size of the inter-partition boundaries minimal [31, 9]. The Zoltan library [36], which is a collection of data management services for parallel, unstructured, adaptive, and dynamic partitioners is used to assign partition entities. FMDB

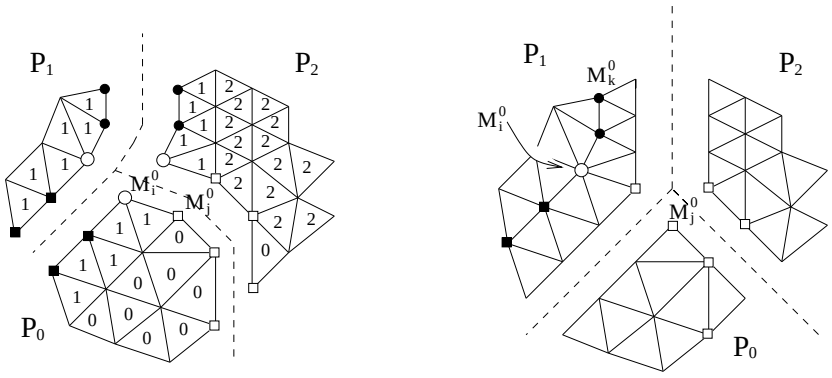
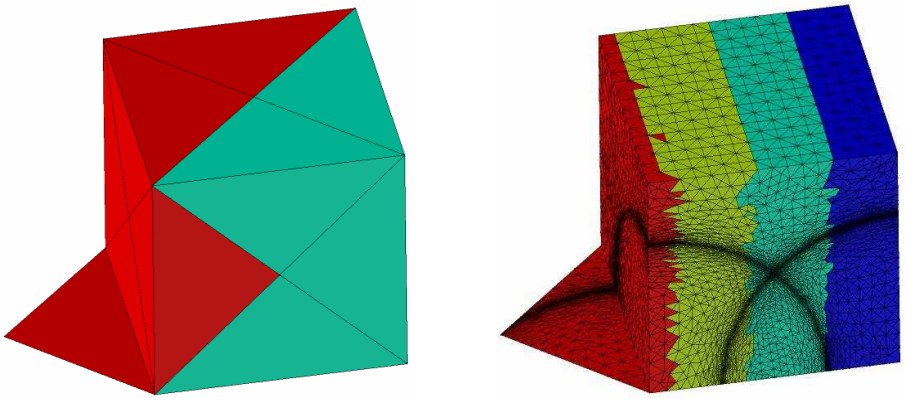


Figure 1.11. Example of 2D mesh load balancing: (left) partition objects are tagged with their destination pids (right) mesh after load balancing



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

computes the input to the Zoltan as a weighted graph or coordinates of partition objects. 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 1.11. 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.

1.7 PARALLEL ADAPTIVE APPLICATIONS

The parallel mesh adaptation procedure has been tested against a wide range of models using either analytical or adaptively defined anisotropic mesh size field definitions [16].

The scalability of a parallel program running on p processors is measured as the $speedup$ or $relativespeedup$.

$$speedup = \frac{\text{run-time on 1 processor}}{\text{run-time on } p \text{ processors}} \tag{1.1}$$

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

$$relative\ speedup = \frac{\text{run time on } \frac{p}{2} \text{ processors}}{\text{run time on } p \text{ processors}} \tag{1.2}$$

Figure 1.12. 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.

Adaptive results have be obtained using the Stanford Linear Accelerator Center (SLAC)'s eigenmode solver Omega3P [14] in conjunction with parallel mesh adaptation. The parallel adaptive procedure has been applied to Trispal model and RFQ model. The speedups given are just for the parallel mesh adaptation portion of the process.

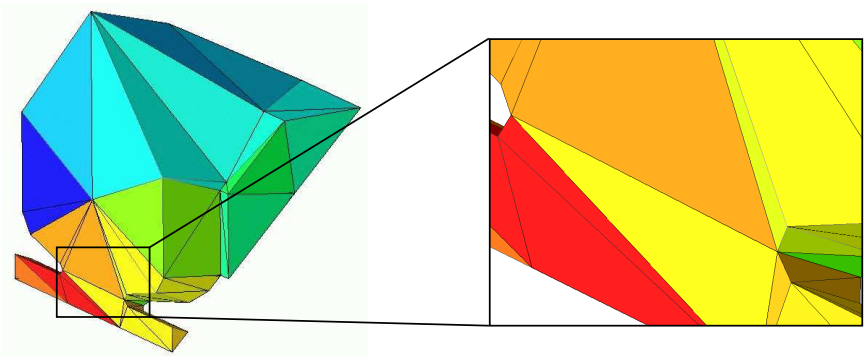
Figure 1.13. shows the Trispal meshes during the parallel adaptive loop, (a) gives the initial mesh composed of 65 tetrahedron, (b) is the adapted, approximately 1 million, mesh after the second adaptive loop on 24 processors, and (c) is the adapted, approximately 12 million, mesh after the eighth adaptive loop.

Figure 1.14. 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, and (c) and (d) are the front and back view of the adapted mesh after the second adaptive loop, which contains about 24 million tetrahedron.

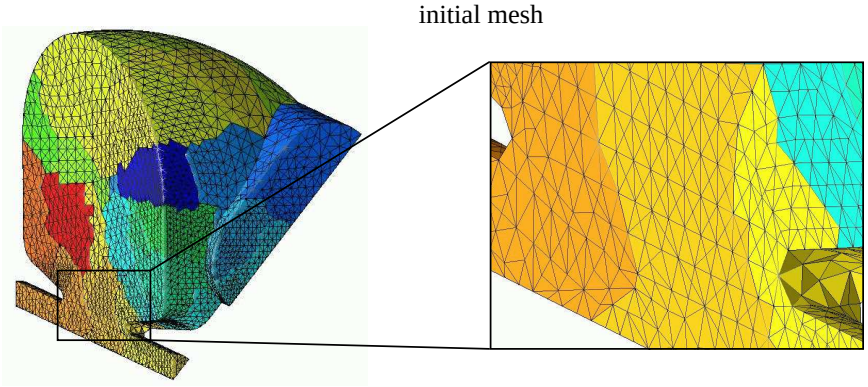
1.8 CLOSING REMARK

A flexible mesh database has been defined and implemented for distributed meshes based on the hierarchical domain decomposition. There procedures are being actively used to support parallel adaptive simulation procedures.

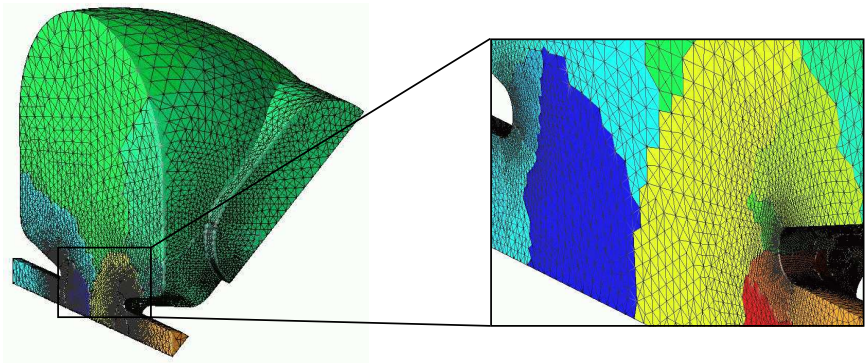
The FMDB is open source available at *http://www.scorec.rpi.edu/FMDB.*



(a) initial mesh



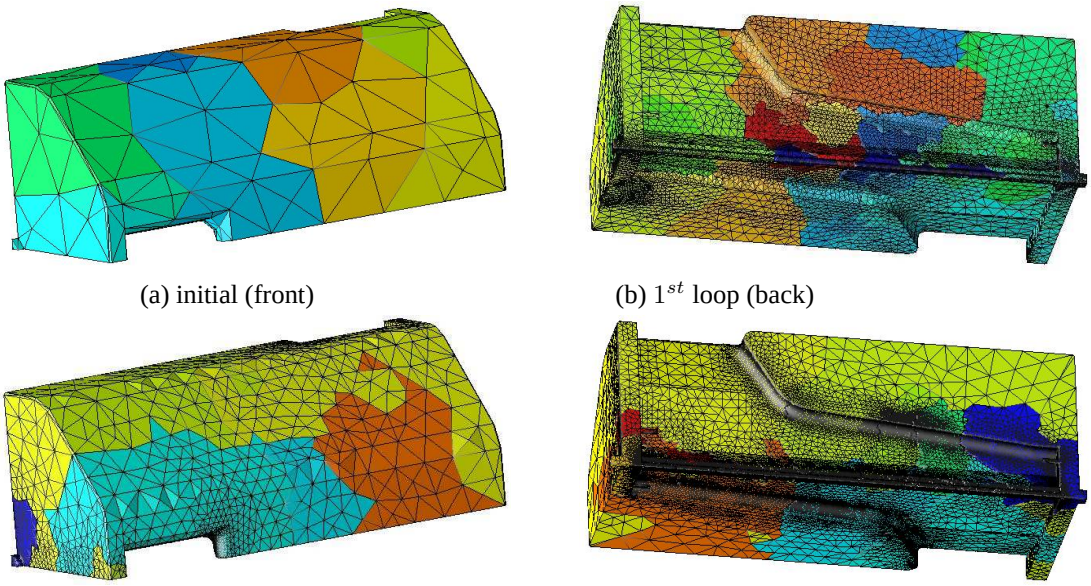
(b) after the 2^{nd} loop



(c) after the 8^{th} loop

# proc	20	40
rel. speedup	-	1.81

Figure 1.13. 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).



(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 1.14. 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) the front view of adapted mesh from the second adaptive loop (approx. 24 million tet), (d) the back view of (c).

REFERENCES

1. F. Alauzet, X. Li, E.S. Seol, and M.S. Shephard. Parallel anisotropic 3d mesh adaptation by mesh modification. *Engineering with Computers*, 21(3):247–258, 2006.
2. A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Aplied*. Addison-Wesley, 2001.
3. M.W. Beall. An object-oriented framework for the reliable automated solution of problems in mathematical physics. Technical report, Rensselaer Polytechnic Institute, Troy NY, 1999. PhD Dissertation Mechanical Engineering Dept.
4. M.W. Beall and M.S. Shephard. A general topology-based mesh data structure. *International Journal for Numerical Methods in Engineering*, 40(9):1573–1596, 1997.
5. R. Biswas and L. Oliker. A new procedure for dynamic adaptation of three-dimensional unstructured grids. *Appl. Numer. Math*, 13:437–452, 1997.
6. W. Celes, G.H. Paulino, and R. Espinha. A compact adjacency-based topological data structure for finite element mesh representation. *Numerical Methods in Engineering*, 64(11):1529–1565, 2005.
7. T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algotithms* 2^{nd} *Ed.* MIT Press, second edition, 2001.
8. H.L. deCougny, K.D. Devine, J.E. Flaherty, and R.M. Loy. Load balancing for the parallel solution of partial differential equations. *Appl. Numer. Math*, 16:157–182, 1995.
9. H.L. deCougny and M.S. Shephard. Parallel refinement and coarsening of tetrahedral meshes. *Int. J. Numer. Meth. Engng*, 46:1101–1125, 1999.
10. Deitel and Deitel. *C++ How To Program*. Prentice Hall, second edition edition, 2001.
11. R. Diekmann, R. Preis, F. Schlimbach, and C. Walshaw. Shape-optimized mesh partitioning and load balancing for parallel adaptive fem. *Parallel Computing*, 26:1555–1581, 2000.

12. E. Gamma, R. Johnson, R. Helm, and J.M Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

13. R.V. Garimella. Mesh data structure selection for mesh generation and fea applications. *International Journal for Numerical Methods in Engineering*, 55:451–478, 2002.

14. L. Ge, L Lee, L. Zenghai, C. Ng, K. Ko, Y Luo, and M.S. Shephard. Adaptive mesh refinement for high accuracy wall loss determination in accelerating cavity design. In *IEEE Conference on Electromagnetic Field Computations*, June 2004.

15. X. Li, M.S. Shephard, and M.W. Beall. Accounting for curved domains in mesh adaptation. *Int. J. Numer. Meth. Engng*, 58:247–276, 2002.

16. X. Li, M.S. Shephard, and M.W. Beall. 3d anisotropic mesh adaptation by mesh modifications. *Comp. Meth. Appl. Mech. Engng*, 2003.

17. libMesh. http://libmesh.sourceforge.net., 2005.

18. R. Loy. Autopack User Manual, 2000.

19. M. Mantyla. *An Introduction to Solid Modeling*. Computer Science Press, Rockville Maryland, 1988.

20. L. Oliker, R. Biswas, and H.N. Gabow. Parallel tetrahedral mesh adaptation with dynamic load balancing. *Parallel Computing*, 26:1583–1608, 2000.

21. C. Ozturan, H.L. de Cougny, M.S. Shephard, and J.E. Flaherty. Parallel adaptive mesh refinement and redistribution on distributed memory. *Comp. Meth. Appl. Mech. Engng*, 119:123–127, 1994.

22. P.S. Pacheco. *Parallel Programming with MPI*. MorganKaufmann Publisher., 1997.

23. Y. Park and O. Kwon. A parallel unstructured dynamic mesh adaptation algorithm for 3-d unsteady flows. *Int. J. Numer. Meth. Fluids*, 48:671–690, 2005.

24. J.F. Remacle, O. Klaas, J.E. Flaherty, and M.S. Shephard. A parallel algorithm oriented mesh database. *Engineering with Computers*, 18:274–284, 2002.

25. J.F. Remacle and M.S. Shephard. An algorithm oriented mesh database. *Int. J. Numer. Meth. Engng*, 58:349–374, 2003.

26. M.L. Scott. *Programming Language Pragmatics*. Kaufmann Publisher, 2000.

27. P.M. Selwood and M. Berzins. Parallel unstructured tetrahedral mesh adaptation: algorithms, implementation and scalability. *Concurrency: Pract.Exper.*, 11(14):863–884, 1999.

28. E.S. Seol. FMDB: Flexible Distributed Mesh Database for Parallel Automated Adaptive Analysis. Technical report, Rensselaer Polytechnic Institute, 2005. Ph.D. Thesis in Computer Science, http://www.scorec.rpi.edu/cgi-bin/reports/GetByYear.pl?Year=2005.

29. E.S. Seol and M.S. Shephard. Efficient distributed mesh data structure for parallel automated adaptive analysis. *Engineering with Computers*, 2006.

30. Sgi Inc. http://www.sgi.com/tech/stl/stl_index.html., 2005.

31. M.S. Shephard, J.E. Flaherty, C.L. Bottasso, and H.L. de Cougny. Parallel automated adaptive analysis. *Parallel Computing*, 23:1327–1347, 1997.

32. J.D. Teresco, M.W. Beall, J.E. Flaherty, and M.S. Shephard. A hierarchical partition model for adaptive finite element computations. *Comput. Methods Appl. Mech. Engrg*, 184:269–285, 2000.

33. D. Vandevoorde and N.M. Josuttis. *C++ Templates*. Addison-Wesley, 2003.

34. C. Walshaw and M. Cross. Parallel optimization algorithms for multilevel mesh partitioning. *Parallel Computing*, 26(12):1635–1660, 2000.

35. K.J. Weiler. The radial-edge structure: a topological representation for non-manifold geometric boundary representations. *Geometric Modeling for CAD Applications*, pages 3–36, 1988.

36. Zoltan. Zoltan: data-management services for parallel applications. http://www.cs.sandia.gov/Zoltan., 2005.