A PARALLEL GHOSTING ALGORITHM FOR THE FLEXIBLE DISTRIBUTED MESH DATABASE (FMDB)

By

Misbah Mubarak

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: COMPUTER SCIENCE

Approved:

Mark S. Shephard, Thesis Adviser

Rensselaer Polytechnic Institute
Troy, New York

April 2011
(For Graduation May 2011)

CONTENTS

LIST OF FIGURES vi

LIST OF TABLES viii

LIST OF ALGORITHMS ix

ABSTRACT x

1. Introduction 1
   - 1.1 Motivation 1
   - 1.2 Organization 3
   - 1.3 Nomenclature 4

2. Overview of FMDB 5
   - 2.1 Geometry-based Analysis Environment 5
     - 2.1.1 Geometric Model 6
     - 2.1.2 Attributes 6
     - 2.1.3 Mesh 7
     - 2.1.4 Fields 9
   - 2.2 ITAPS Mesh Component 9
     - 2.2.1 Entities 10
     - 2.2.2 Entity set 10
     - 2.2.3 Tags 11
     - 2.2.4 Iterators 11
   - 2.3 Topology-based Mesh Data Structure 11
     - 2.3.1 Topological entities 12
     - 2.3.2 Geometric Classification 13
     - 2.3.3 Geometric Information 14
     - 2.3.4 Adjacencies 14
   - 2.4 Distributed Mesh Management 16
     - 2.4.1 Distributed Mesh Representation 18
     - 2.4.2 Functional Requirements of Distributed Meshes 21
   - 2.5 The Partition Model 23
   - 2.6 Dynamic load balancing and mesh modification 25

2.6.1 Entity migration 25
2.6.2 Dynamic load balancing 26
2.6.3 Ghosting 27
2.7 ITAPS parallel mesh component 27
2.7.1 The abstract data model 27
2.7.2 The iMeshP interface 28
2.8 Implementation of FMDB 29
2.8.1 Implementation Structure 30
2.8.2 Design of Testing suites 32
3. N-Layer Ghost Creation & Deletion Algorithm 33
3.1 Historical Review 34
3.2 The Ghosting Process 38
3.3 Algorithm of N-layer ghost creation 41
3.3.1 Step 1: Ghost collection for first layer 43
3.3.2 Step 2: Process next layer 48
3.3.3 Step 3: Eliminate duplicate entities 52
3.3.4 Step 4: Exchange entities and update ghost copies 55
3.3.5 Store ghost rule 58
3.4 N-layer ghost deletion algorithm 59
3.4.1 Step 1: Process first-layer ghosts 59
3.4.2 Step 2: Process next layer ghost entities 62
3.4.3 Step 3: Mark visited bridge entities 62
3.4.4 Step 4: Delete ghosts collected 62
3.5 Ghosting Tests 63
3.6 Summary 64
4. Performance Results 66
4.1 Strong scaling study of Ghost creation/deletion 67
4.2 Weak scaling study of Ghost creation/deletion 73
4.3 N-layer ghost creation with fixed processor count 74
5. Closing Remarks 76
BIBLIOGRAPHY 78
APPENDICES

A. FMDB Unit Tests 85
A.1 Iterator tests 85
A.2 Ownership and remote copy tests 87
B. Mesh Verification Algorithm 91

toc

LIST OF FIGURES

2.1 Example of attributes in a geometry-based problem definition [10] . . . 7

2.2 superquardic: geometric model and the mesh 8

2.3 Example of a field in a geometry-based problem definition [10] 9

2.4 Simple model and the mesh associated through geometric classification [25]. 14

2.5 12 adjacencies possible in a mesh representation [26] 15

2.6 Edge and vertex order on a face 16

2.7 Face ordering in a region [25] 17

2.8 Edge ordering in a region [25] 17

2.9 Distributed Mesh on three parts [29] 19

2.10 Example 3D mesh distributed on two parts 21

2.11 Hierarchy of domain decomposition: geometry model, partition model and distributed mesh on 4 processors [24] 23

2.12 Distributed Mesh and its partition classification [24] 24

2.13 Example of entity migration of a 2-D mesh [33] 26

2.14 FMDB Implementation Structure 30

3.1 A distributed mesh on four parts with ghost entities [19] 39

3.2 A 2-D ghosted mesh 40

3.3 Example of 2-D ghost creation with 1-layer 41

3.4 Example of 2-D ghost creation with 2-layers 49

3.5 Ghost deletion algorithm applied on Part 0 of Figure 3.3 61

4.1 A cubic mesh 67

4.2 Moving air bubbles 68

4.3 Relationship of ghosted entities created with scalability (165M mesh) . 71

4.4 Relationship of ghosted entities deleted with scalability (165M mesh) . . 72

A.1 Steps for testing generic iterator for mesh entities in a part 86

A.2 Steps for testing generic iterator for geometric model entities 88

A.3 Steps for testing remote copy information 89

LIST OF TABLES

3.1 Contents of vector *entitiesToGhost* after Step 1 47

3.2 Contents of vector entitiesToGhost after Step 2 52

3.3 Contents of vector entitiesToGhost after Step 3 54

3.4 Contents of vector *entitiesToRemove* after Step 3 63

4.1 Blue Gene/L vs. Cray XE6 67

4.2 Computation time vs. communication time on Cray XE6 (17M cube) . 69

4.3 1-layer ghost creation time(sec) on Cray XE6 and BG/L 70

4.4 1-layer ghost deletion time(sec) on Cray XE6 and BG/L 72

4.5 Weak scaling of ghost creation algorithm 73

4.6 N-layer ghost creation execution time(s) on 1024 processors 75

List of Algorithms

1 createGhosts(M, g, b, numLayer) 44

2 getGhostEnts(M,g, b, includeCopy, entitiesToGhost, numLayer) 46

3 getDownwardAdjs(M,g, M_i^g, destId, entitiesToGhost) 48

4 processNLayers(M, g, b, includeCopy, lyr, entitiesToGhost) 50

5 removeDuplicateEnts(M, d, entitiesToGhost) 53

6 exchangeGhostEnts(M, g, d, entitiesToGhost(d)) 56

7 deleteGhostEnts(M) 60

8 collectDeleteInfo(M_j^g, g, $entitiesToRemove$) 61

9 Mesh_Verify(M) 93

10 VerifyRemoteCopies(isValid, M_i^d) 94

11 VerifyGhostCopies(isValid, M_i^d) 95

12 VerifyVertices(M, isValid) 95

13 VerifyRegions(M,isValid) 96

14 VerifyEdges(M,isValid) 97

15 VerifyFaces(M,isValid) 98

ABSTRACT

The scalable execution of parallel adaptive analyses requires the application of mesh modification operations to modify the partitioned mesh with balanced work load and minimal communication. The thesis presents a parallel algorithm for ghost creation and deletion that localizes neighborhood data for computation to minimize inter-part communication. The ghosting algorithm provides a third-party application with the complete parallel neighborhood information in a partitioned mesh. This reduces the communication pattern in the application to simple point-to-point transfers of numerical information. The thesis presents a ghost creation and deletion algorithm for the Flexible distributed Mesh Database (FMDB) that can create 1D, 2D or 3D ghost objects in a mesh using bridge entities. The algorithm utilizes neighborhood communication to create any number of ghost layers up-to a point where the whole partitioned mesh is ghosted. Ghosting that becomes invalid due to mesh modification can be synchronized by throwing away old ghosts and creating new ones. For testing purposes, a mesh verification algorithm that verifies the validity of the ghosted mesh is also presented.

Strong and weak scaling analysis results of ghost creation and deletion algorithm is presented up-to a core count of 32,768, using two massively parallel architectures i.e. IBM Blue Gene/L and Cray XE6. Performance results show that the scalability of the ghosting algorithm is dependent on the ratio of inter-part communication to computation and the number of ghost entities that keep on increasing with increasing processor count.

CHAPTER 1
Introduction

1.1 Motivation

The Finite Element Method (FEM) is a powerful tool which is a standard for solving Partial Differential Equations (PDEs) [1]. Adaptive FEMs have gained importance as they provide reliability, robustness, time and space efficiency. As the mesh size continues to grow, and the level of geometric complexity of the domain increases, the application of serial mesh generation become a bottleneck [2]. That is why in such methods, computationally demanding three-dimensional problems make parallel computation essential; though, parallelism introduces complications such as the need to balance processor loading, coordinate inter-process communication, and manage distributed data.

A distributed mesh representation provides the necessary support for finite element analysis in parallel. The approach consists of decomposing the domain into a number of parts, or sub-domains where each part represents a unit of execution. The scalable execution of parallel adaptive analysis requires the application of dynamic load-balancing to re-partition the mesh into sets of parts with balanced work load and minimal communication [3]. As the adaptive meshes being generated reach billions of elements and the analysis are performed on massively parallel computers with 100,000's of computing cores, a number of complexities arise that need to be addressed. First, the adaptive mesh modifications change the number of mesh entities and their adjacencies, thus changing the load balance of the mesh. The application of dynamic load balancing must be applied to regain the load balance. The second issue is related to communication and synchronization among the parts. In an explicit simulation model, parts communicate with each other to exchange data needed for the simulation at each step, and the data must be available for computation in a consistent and up-to-date way. In shared memory systems [4], data communication is straightforward since all processes reside in the same virtual memory space and thus memory can be directly used to send/receive data asynchronously. In distributed memory systems [4], each process has its own private memory space and they are connected by a network. In this case, message passing is the most common way to communicate data among two or more parts, and synchronization can be done explicitly using function calls. Many massively parallel distributed systems have been developed based on large number of computing nodes connected by very fast networks [5].

In a distributed mesh, most of the time communication occurs only among neighboring parts. Even then, network communication overheads can be considerably large. To minimize this communication, applications must consider the size and number of the messages being sent. For each entity, the analysis code usually tends to access adjacent entities to compute corresponding results for the current simulation step. Therefore, it is important to cluster elements or nodes in a way to minimize the number of adjacent entities belonging to different parts, thus minimizing the need for communication [6]. In addition to requiring data associated with shared part boundary mesh entities, specific finite element calculations require data from mesh entities internal to neighboring parts. One means to support providing the data is by the addition of copies of the data referred to as ghost copies as needed on neighboring parts. The basic purpose of ghosts is to localize the data that may be needed by the analysis program for computation on part boundaries. To decrease the inter-process communication for data exchange, ghosts copies bring along the data on part boundaries. If some time during mesh modification, the ghost data becomes invalid, the mesh is synchronized to update the ghost data. In some cases, applications require more than one layer of ghost data from other parts. In such cases, the mesh databases need to support a N-layer ghost creation and deletion algorithm.

The goal of the thesis is to develop an efficient and fully 3-D N-layer ghost creation and deletion algorithm for the Flexible distributed Mesh DataBase (FMDB). The algorithm was designed flexible enough to provide ghost data for all dimensions (1-D, 2-D or 3-D) depending upon the application requirements. The algorithm supports any number of ghost layers as specified by the application. To meet the goal of efficiency, the performance of the N-layer ghost algorithm was tested on massively parallel architectures i.e. Cray XE6 at National Energy Research Scientific Computing Center[7] and IBM BG/L at RPI [8].

1.2 Organization

Chapter 2 introduces general topology based mesh data structures, highlights the requirements and functionality of distributed meshes and presents an overview of dynamic load balancing and mesh modification. Chapter 3 presents a review of the parallel support (including ghosting) provided by various mesh databases, it then presents the design of N-layer ghost creation and deletion algorithm developed for FMDB and explains its working with the help of a 2-D mesh example. Chapter 4 provides the performance results and scalability of the N-layer ghost creation and deletion algorithm. Chapter 5 concludes the thesis by summarizing the contributions and discussing the future work. For readers interested in FMDB unit testing, Appendix A presents the design of some test suites incorporated in FMDB for testing parallel functionality. For readers interested in details of mesh verification and testing, Appendix B presents the mesh verification algorithm that verifies the correctness of a ghosted mesh.

1.3 Nomenclature

V	the model, $V \in \{G, P, M\}$ where V can belong to either of Geometric Model G, partition model P or the mesh model M.
V_i^d	the i^{th} entity of dimension d in the 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.
$\{V_i^d\{V^q\}\}$	set of entities of dimension q that are adjacent to V_i^d. For e.g. $\{M_1^3\{M^0\}\}$ is the set of all vertices that are adjacent to the region M_1^3.
$U_i^{d_i} \sqsubset V_j^{d_j}$	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 hierarchy of domain decomposition.
$\mathscr{P}[M_i^d]$	set of part id(s) where M_i^d exists.
$\{V_i^d\{V^q\}_j\}$	The j^{th} entity in the set of entities of dimension q in model V that are adjacent to V_i^d
$\mathscr{R}[M_i^d]$	Set of remote copies of M_i^d on remote parts.
$\partial(P_{local})$	Set of part boundary entities on current part P_{local}.

CHAPTER 2
Overview of FMDB

The Flexible distributed Mesh Database (FMDB) is a distributed mesh management system that provides a parallel mesh infrastructure capable of handling general non-manifold models while effectively supporting parallel adaptive analysis [9]. This chapter introduces the data sets involved in the geometry-based analysis environment (§2.1) and the ITAPS mesh component (§2.2), followed by a discussion on general topology-based mesh data structure (§2.3) and an overview of distributed mesh data structure developed in FMDB (§2.4, §2.6, §2.5). Finally it presents the ITAPS parallel mesh component (§2.7) and FMDB implementation structures being used (§2.8).

2.1 Geometry-based Analysis Environment

The geometry based analysis framework starts at the level of mathematical problem description, allowing multiple numerical problems to be formulated, solved, and the solution related back to the original problem description. As the analysis framework must take a problem description consisting of a geometrical model with appropriate attributes and construct a solution to the problem specified, it should build on a well-defined set of abstractions for the various types of data that this framework uses. The structures used to support the problem definition, the discretization of the model and their interactions are central to the geometry based analysis framework [10]. The geometry-based analysis environment consists of the four structures of the *geometric model* of the domain, the problem *attributes*, the domain discretization in terms of a *mesh* and the solution *fields* solved for the mesh.

2.1.1 Geometric Model

The geometric model representation is a boundary representation based on the Radial Edge Data Structure[11]. It consists of topology and shape description of the domain of the problem. The model is a hierarchy of topological entities called regions, shells, faces, loops, edges and vertices. Data structures implementing the geometric model support operations to find the various model entities that make up a model and indicate which model entities are adjacent to a given entity. Operations related to performing geometric queries are also supported [12]. It is important to understand that there are associations between the model entities, attributes and mesh entities which are central to support generalized adaptive analysis procedures that operate from a general problem definition [10].

2.1.2 Attributes

To complete the specification of the analysis problem, additional physical information in terms of load, material properties, boundary and initial conditions must be associated with the appropriate geometric model entities. In this form of definition, this information can be considered attributes of the geometric definition of the domain [13, 14]. These attributes are tensorial in nature. This viewpoint provides a means for mathematical consistency in describing the attributes. Thus an attribute can be defined fully given the tensor order, tensor symmetry, distribution of tensor components and the coordinate system in which the tensor is defined [14]. These tensor values attributes may vary in both space and time. A simple example of a problem definition is shown in Figure 2.1, the problem being modeled here is a dam which is subject to load due to gravity and water behind the dam. There are a set of attribute information nodes that are under the attribute case for the problem definition. The attributes here are indicated by triangles with A's inside them. When this case is associated with the model, these attributes are created and

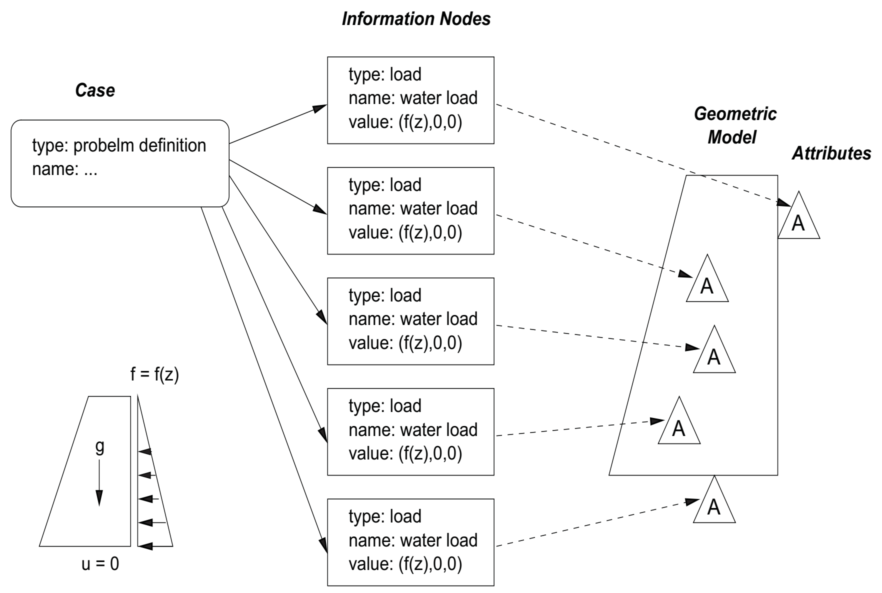


Figure 2.1: Example of attributes in a geometry-based problem definition [10]

associated with the appropriate model entities on which they act.

2.1.3 Mesh

The mesh is a discrete representation of the domain used by the analysis process. Figure 2.2 represents a solid model and its mesh. Understanding how the mesh relates to the geometric model allows an understanding of how the solution relates back to the original problem description [15] and supports the ability to properly adapt the mesh fully accounting for the geometric domain and attributes as the mesh is changed. The representation used for a mesh is also a boundary representation, i.e. there is a topological hierarchy of regions, faces, edges and vertices that make up a mesh [16]. In addition, each mesh entity in the mesh database maintains a relation, called the classification of a mesh entity, to the model entity that it was created to partially represent. The classic element-node approach [17] in the mesh data structures lacks classification information which is critical for mesh generation and enrichment procedures as it allows the mesh to relate back to the geometry.

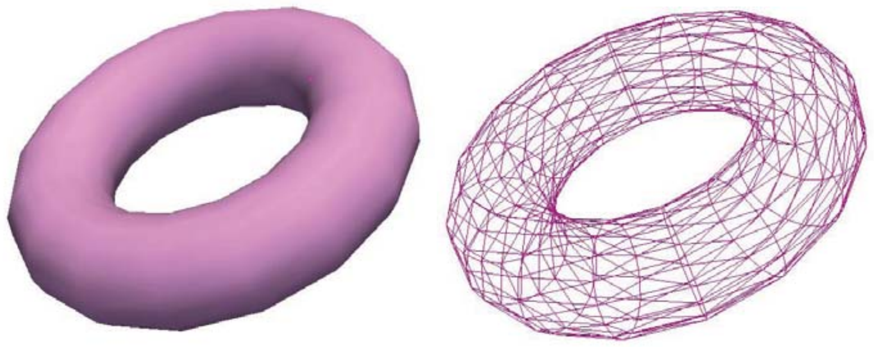


Figure 2.2: superquardic: geometric model and the mesh

Geometric classification also allows the specification of analysis attributes in terms of original geometric model rather than the mesh, which is important in adaptive analysis environments. Many analysis and mesh modification procedures can be easily written when a topological hierarchy of mesh entities is used to represent a mesh. For example, an edge based refinement procedure is much easier to write if it is possible to loop through all edges in a mesh and get connectivity information of the edge.

An important goal in the development of a mesh data structure is ensuring its ability to effectively provide the information required by various procedures that create and/or use that data. The differing needs of such tools requires that such a mesh database must be general and be able to answer all queries about the mesh. This general capability can only be achieved by utilizing more general abstraction of a mesh like the topological hierarchy of mesh entities, not the specific node-element view that were developed for the needs of performing a single analysis on a fixed mesh [18]. The detailed discussion on mesh entities and adjacencies is presented in §2.3.

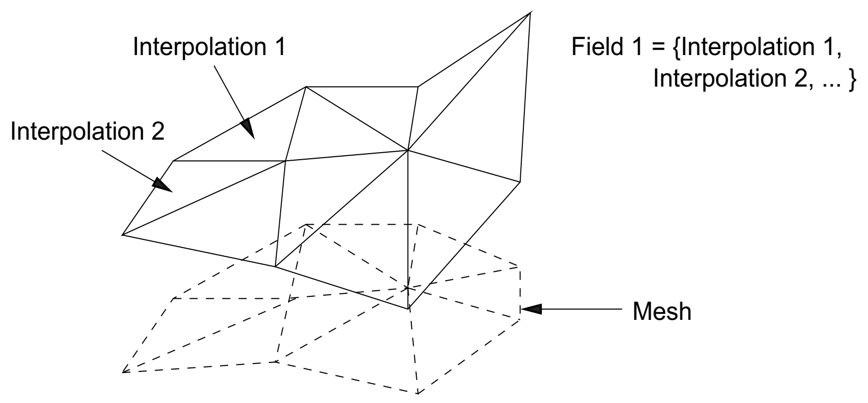


Figure 2.3: Example of a field in a geometry-based problem definition [10]

2.1.4 Fields

Fields describe the variation of tensor quantizers over one or more entities in a mesh model. The spatial variation of the field is defined in terms of interpolations defined over a discrete representation of the geometric model entities, which is currently the finite element mesh. A field is a collection of individual interpolations, all of which are interpolating the same quantity [10]. Each interpolation is associated with one or more mesh entities in the discrete representation of the model (See Figure 2.3).

2.2 ITAPS Mesh Component

This section describes the data model of a mesh component which is developed to provide support for mesh access and manipulation requirements of practical, large-scale scientific computing applications. This component developed as part of the Inter-operable Tools for Advanced Petascale Simulation (ITAPS) center [19] is called *iMesh* [20]. The *iMesh* component is intended to define operations required at a mesh database level so that high level operations including mesh generation, mesh improvement, mesh adaptation, parallel mesh operations can be implemented as services that store and manipulate data by using the *iMesh* component and mesh databases that implement the component's functionality [21]. The key building block of the *iMesh* data model consists of *entities, entity sets and tags*.

2.2.1 Entities

Entities represent the topological entities in the mesh and geometric model. Entity adjacency relationships define how the entities connect to each other and both first-order and second-order adjacencies are supported [22]. The detailed discussions on the entity and adjacencies is presented in §2.3

2.2.2 Entity set

Entity set support the arbitrary grouping of entities. All topological and geometric mesh data as well as other entity sets, are contained in a *root entity set*. Each entity set may be a true set (in the set-theoretic sense) or it may be a possibly non-unique ordered list of entities; in the later case entities are retrieved in the order in which they were added to the entity set [23]. Entity sets are populated by addition or removal of entities from the set (with the exception of a root-set which is present by default in the mesh). In addition, set Boolean operations (subtraction, intersection and union) on entity sets are also supported. An entity set supports two primary operations. First, an entity set may contain one or more entity sets (by definition, all entity sets belong to the root set). An entity set which is contained in another may be a subset or an element (in the set-theoretic sense) of that entity set. Second, parent/child relationship between entity sets are used to represent logical relatioships between sets. Examples of entity set could be the ordered list of vertices bounding a geometric face and the set of all entities in a given level of a multigrid mesh sequence [21, 22].

2.2.3 Tags

A tag is a container of arbitrary data attachable to mesh, entities, geometric model entities and entity set. Different values of a particular tag can be associated with mesh, entity or an entity set: for example tags can be used to mark entities during a specific operation. During mesh smoothing, certain entities are tagged for smoothing. Once the smoothing operation is performed on them, the tag is removed. The ITAPS mesh component supports specialized tag types for improved performance as well as the more general case that allows any type of data to be attached [22].

2.2.4 Iterators

Iterators are a generalization of pointers which are objects that point to other objects. An iterator is an object that allows a programmer to traverse through all elements of a collection, regardless of its specific implementation. If an iterator points to one element in a range, then it is possible to increment it so that it points to the next element.
Various kinds of iterators are desirable for efficient mesh entity traversal with various conditions like entity dimension, entity topology, geometric classification [24].

2.3 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. The functional requirements of a topology-based mesh data structure are: topological entities, classification, geometric information and adjacencies [16].

2.3.1 Topological entities

As the mesh is constructed from a geometric model that is represented using a boundary based scheme, thus using a similar scheme to also represent the mesh is advantageous for several reasons. First, topology provides an unambiguous, shape independent, abstraction of a mesh. Second, maintaining the geometric classification information is simplified as the same type of entities occur both in geometric model and the mesh. Each topological entity of dimension d, M_i^d is defined by a set of lower order topological entities of dimension $d-1$, $\{M_i^d\{M_i^{d-1}\}\}$ which form its boundary. For example, a region is a 3-D entity with a set of faces bounding it. In turn, a face is a 2-D entity bounded with a set of edges whereas an edge is a 1-D entity bounded by a pair of vertices. A vertex is a 0-D entity that is the base of the hierarchy, it has no lower order entities bounding it.

The proper consideration of general geometric domains also requires consideration of loop and shell topological entities, and, in case of non-manifold models, use entities for the vertices, edges, loops and faces [11]. However, there are restrictions on the topology of a mesh that allows a reduced representation which is in terms of only the basic 0 to d dimensional topological entities [18]. For the three dimensional case (d=3), these entities are

$$M = \{M\{M^0\}, M\{M^1\}, M\{M^2\}, M\{M^3\}\} \tag{2.1}$$

where $\{M\{M^d\}\}$, $d = 0, 1, 2, 3$ are respectively the set of vertices, edges, faces and regions defining the primary topological elements of the mesh domain.
The restrictions on the topology of a mesh which allow this reduction include:

- Regions and faces have no interior holes.

- Each entity of order d_i in a mesh M^{d_i}, may use a particular lower order entity,

M^{d_j}, $d_j < d_i$, at most once.

- For any entity $M_i^{d_i}$ there is a unique set of entities of order $d-1$, $\{M_i^{d_i}\{M^{d_i-1}\}\}$

The first item means that regions may be directly represented by the faces that bound them, and faces may be represented by the edges that bound them. The shell and loop entities, required for general models, are therefore not needed in the mesh.

The second item allows an orientation of an entity to be defined in terms of its boundary entities (without the introduction of entity uses). For example, the orientation of an edge, M_i^1 which is bounded by vertices M_j^0 and M_k^0 may be uniquely defined as going from M_j^0 and M_k^0 only if $j \neq k$.

The third item means that an interior entity (defined as $M_i^{d_i} \sqsubset G_i^{d_i}$ where $d_j \leq d_i$ and at-least one of $\partial(M_i^{d_i}) \sqsubset G_i^{d_i}$) can be uniquely specified by the entities that bound it [18].

2.3.2 Geometric Classification

The linkage of the mesh to the geometric model is critical for mesh generation and adaptation procedures. The unique association of a mesh entity of dimension d, M_i^d to a geometric model entity of dimension d_j, ${G_j}^{d_j}$ where $d_i \leq d_j$, ${M_i}^d \sqsubset {G_j}^{d_j}$ where the classification symbol $\sqsubset$, indicates that the left-hand entity, is classified on the right hand entity. Multiple M_i^d can be classified on a ${G_j}^{d_j}$. Mesh entities are always classified with respect to the lowest-order geometric model entity possible [16]. Classification of the mesh against the geometric model entity is central to ensuring that the automatic mesh generator has created a valid mesh [16]. In Figure 2.4, a mesh of a 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 [18].

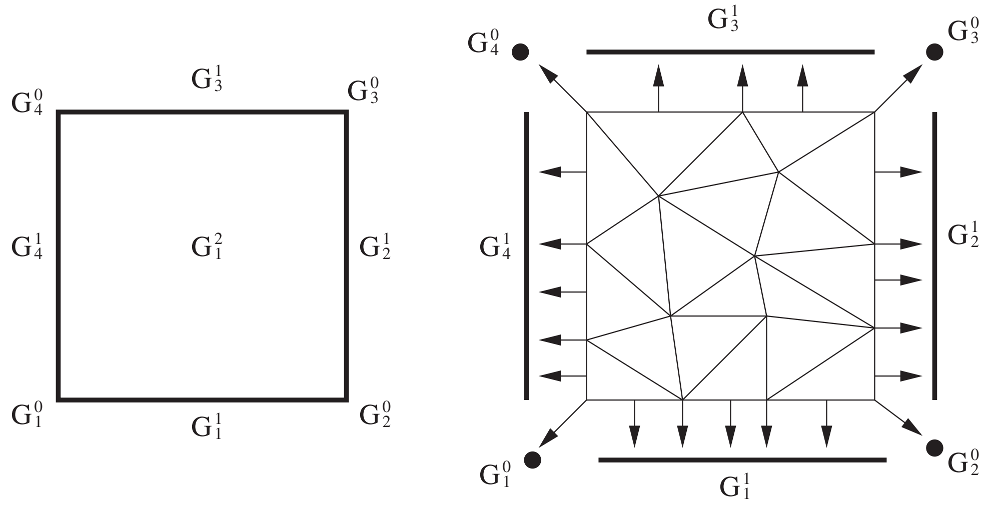


Figure 2.4: Simple model and the mesh associated through geometric classification [25].

2.3.3 Geometric Information

Each topological entity in the geometric model has shape information associated with it that defines the geometry of the model. For the mesh, the geometric information that is required is limited to pointwise information in terms of the parametric coordinates of the model entity that a mesh entity is classified on. Other shape information that is needed to define the mesh can be obtained from the geometric model using the classification information and appropriate queries to the modeler. Zero or more locations may be stored for each mesh entity other than a mesh vertex [18].

2.3.4 Adjacencies

Adjacencies describe how topological entities connect to each other. Certain adjacencies are also used to define higher order entities in terms of the lower order ones (e.g. a region is defined by the faces that bound it). There are orderings for

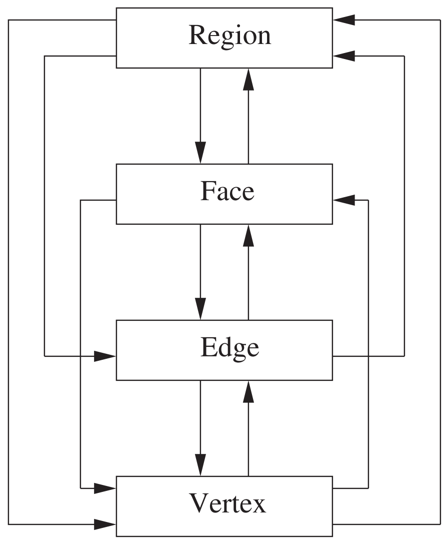


Figure 2.5: 12 adjacencies possible in a mesh representation [26]

adjacencies that are useful when accessing and manipulating the mesh. The form of adjacencies used are: a linear list of entities adjacent to M_i^d, an unordered list of entities adjacent to M_i^d and a cyclic list of entities adjacent to M_i^d. For certain type of entities, there is also a direction component to the adjacency relation that indicates how the entity is used in the specific adjacency. In these cases, the right subscript $\pm$ on the entity $M_{\pm i}^d$ indicates a directional use of the topological entity as defined by its ordered definition in terms of lower order entities. A + indicates use in the same direction whereas - indicates use in the opposite direction for e.g. a face M_i^2 can be defined by a set of edges bounding it as $M_{+i}^1, M_{-k}^1, M_{-l}^1$ meaning that the face is made up of these three edges where M_{+i}^1 is used in the positive direction, from its first to second vertex whereas the edges M_{-k}^1 and M_{-l}^1 are used in the negative direction from their second to first vertex.

First order adjacencies are the set of relations which describe, for a given entity ${M_k}^{d_i}$, all of the entities,M^{d_j} which are either on the closure of the entity ($j < i$) or which it is on the closure of ($j > i$). For example, the adjacency $\{M_i^3, \{M^0\}\}$ represents an ordered list of all the mesh vertices which are on the closure of the mesh regions $M_i{}^3$. Figure 2.5 depicts the 12 first order adjacencies possible in the mesh data structure where a solid box and a solid arrow denote the stored level of entities and stored adjacencies from outgoing to incoming level [16]. Figure 2.6, 2.7 and Figure 2.8 describes a common canonical ordering of bounding entities.

For an entity of dimension d, second order adjacencies describe all mesh entities of dimension q that share any adjacent entities of dimension b where $d \neq b$, $b \neq q$. 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) [18].

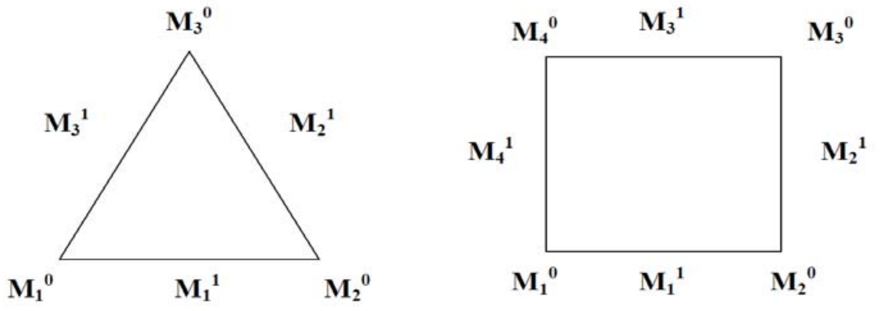


Figure 2.6: Edge and vertex order on a face

2.4 Distributed Mesh Management

Scalable parallel processing techniques are becoming central to the solution of large scale simulations consisting of millions to billions of finite elements. Therefore, consideration must be given to parallel mesh generation and management so that it does not become a computational bottleneck [2]. The development of effective parallel algorithms for adaptive techniques is challenging due to irregular nature of adaptive discretization and the constant modification of the discretization. The

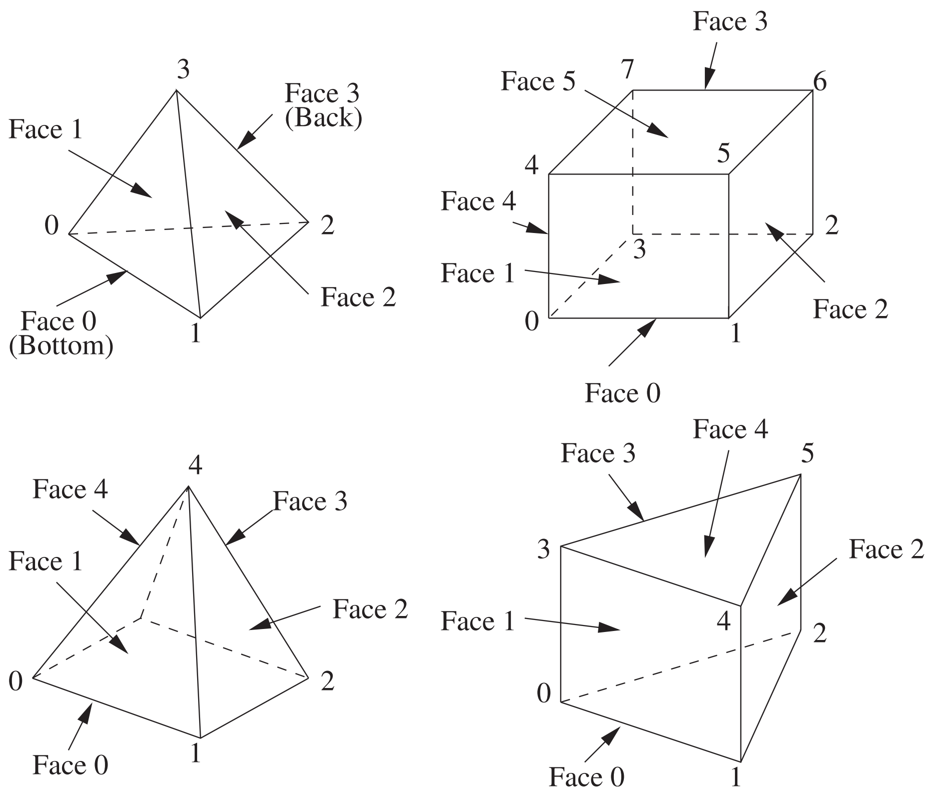


Figure 2.7: Face ordering in a region [25]

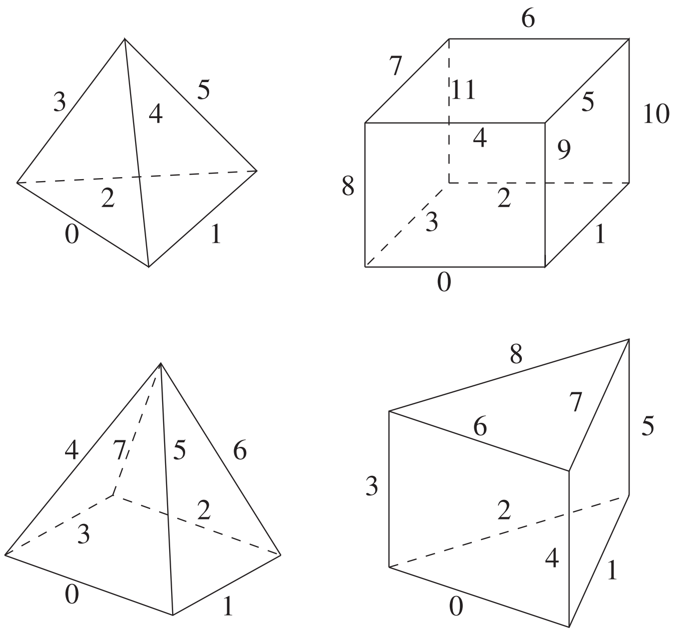


Figure 2.8: Edge ordering in a region [25]

evolving nature of the discretization in an automated adaptive analysis procedure dictates the use of structures and constructs that can effectively account for the processor workload changes. These structures and constructs are dramatically different from those needed for fixed discretization parallel computations in that they must be able to efficiently maintain load balance, while controlling communications, as the distributed discretization evolves [27]. In order to optimize adaptive finite element performance, a static partitioning of the mesh across the cooperating processes is not sufficient. Load imbalance caused by adaptive enrichment necessitates a dynamic partitioning and redistribution of data. A distributed mesh structure builds directly on the topologically based mesh database in the following manner. Each process holds data associated with a subset of the complete mesh. The mesh assigned to each process is housed in a local version of the mesh structure and there are links to neighboring mesh entities on other processes.
The quality of data partitioning is an important efficiency factor. One measure of partition quality is the percentage of elements which require access to off-process data during the computation. A poor partitioning in a distributed mesh results in a higher communication cost during the finite element solution phase [28].

2.4.1 Distributed Mesh Representation

This section describes the abstract data model and terminologies associated with a distributed mesh.

Definition 1 (Distributed Mesh) *A distributed mesh is a mesh divided into parts for distribution over a set of processes for specific reasons, for example parallel computation.*

Definition 2 (Part) *A part consists of the set of mesh entities assigned to a process. For each part, a unique global part id within an entire system and a local*

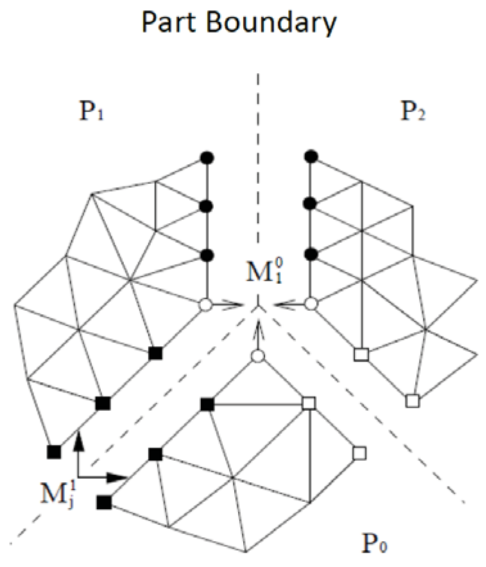


Figure 2.9: Distributed Mesh on three parts [29]

part id within a process can be given. Parts maintain links with other parts through shared mesh entities.

Each part is treated as a serial mesh with the addition of mesh part boundaries to describe entities that are on inter-part boundaries. Mesh entities on part boundaries are shared by more than one part and are maintained by a part boundary data structure. For example in three dimensional distributed mesh, each region is assigned to a unique part, but bounding faces, edges and vertices of regions along part boundaries are duplicated on each part that contains a region using that entity. Figure 2.9 depicts a 2-D mesh distributed on three parts. Vertex M_1^0 is common to three parts and on each part, several edges like M_1^j are common to two parts. The dashed lines are part boundaries that consist of mesh vertices and edges duplicated on multiple parts.

In order to denote the parts on which a mesh entity resides, we define the residence part operator $\mathscr{P}$.

Definition 3 (Residence part equation of M_i^d) *If* $\{M_i^d\{M^q\}\} = 0$, $d < q$, $\mathscr{P}[M_i^d] =$

$\{p\}$ *where* p *is the id of the part where* M_i^d *exists. Otherwise,* $\mathscr{P}[M_i^d] = \cup \mathscr{P}[M_j^q | M_i^d \in \{\partial(M_j^q)\}]$

The residence parts of an entity is determined by the following rules

- For an entity M_i^d on part p which does not exist on the boundary of a higher order mesh entity, $\mathscr{P}[M_i^d]$ returns $\{p\}$. In case, when an entity does not exist on the boundary of a higher order mesh entity, its residence part is only determined by the part where it exists for e.g. a region in a 3-D mesh exists on only one part.

- If M_i^d is not a shared part boundary entity then its bounding part is also $\{p\}$.

- If M_i^d is on the boundary of other higher order mesh entities, M_i^d is duplicated on multiple parts depending on the residence parts of the entities it bounds. Therefore, the residence part(s) of M_i^d is the union of bounding parts 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 {p} if $\{{M_i}^d\{{M_k}^{d+1}\}\} = \phi$. Otherwise,$\mathscr{P}[M_i^d]$ is $\cup \mathscr{P}[M_k^{d+1} | M_i^d \in \{\partial(M_k^{d+1})\}]$. For instance, for the 3D mesh depicted in Figure 2.10 , M_3^1 is shared by P_0 and P_1. For the vertex M_1^0, residence part ids are $\{P_0, P_1\}$ as the union of the bounding parts of its bounding edges $\{M_0^1, M_1^1, M_2^1, M_3^1\}$ are $\{P_0, P_1\}$

Definition 4 (Partition Object) *A partition object is a mesh entity or a set of mesh entities that can be marked for migration.*

In the case of a mesh of a non-manifold geometric domain with no entity grouping for migration, the partition objects are

- All mesh regions.

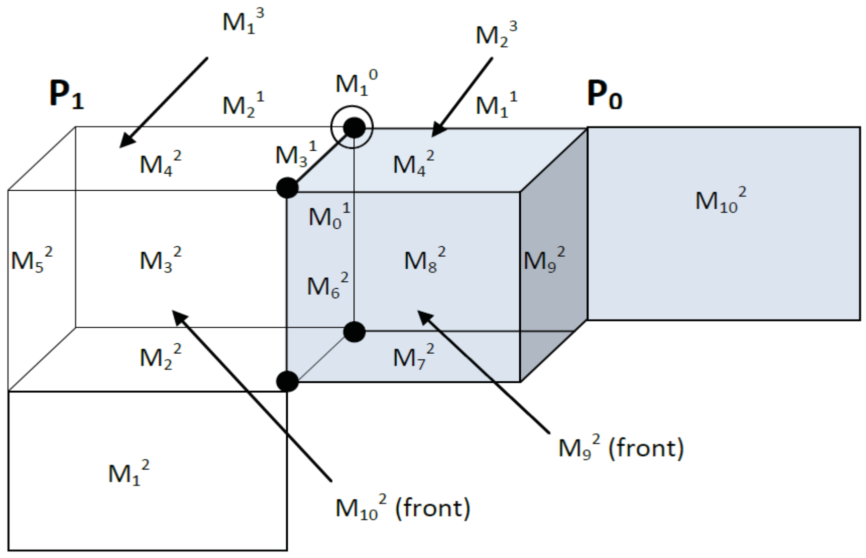


Figure 2.10: Example 3D mesh distributed on two parts

- Mesh faces not bounded by any mesh region.
- Mesh edges not bounded by any mesh face.
- Mesh vertices not bounded by any mesh edge. [24].
- In some applications, sets of mesh entities are always migrated together. For example, the set of mesh regions that form a stack on top of boundary layer nesg face needs to be migrated together. In this case, the set is defined as a single partition object,

2.4.2 Functional Requirements of Distributed Meshes

Functional requirements for supporting mesh operations on distributed meshes are

- Communication Links: Mesh entities on part boundaries must be aware of where they are being duplicated. This is done by maintaining remote parts and remote copies.

Definition 5 (Remote part) *A non-self part where an entity is duplicated.*

Definition 6 (Remote copy) *Refers to memory location of a mesh entity duplicated on remote part p.*

Definition 7 (Neighboring parts) *Part A neighbors Part B if Part A has remote copies of entities owned by part B or if part B has remote copies of entities owned by part A.*

In parallel adaptive analysis, the mesh and its partitioning can change thousands of time during the simulation. Therefore, an efficient mechanism to update mesh partitioning is required which can also update the links between parts [30, 31].

- Entity ownership: For entities on part boundaries, it is beneficial to assign a specific copy as the owner of the others and let the owner manage communication or computation between copies. The two basic strategies for determining the owning part of part boundary entities are
  - *Static entity ownership:* The owning part of a part boundary entity is always fixed to p_i regardless of mesh partitioning [32, 33].
  - *Dynamic entity ownership:* The owning part of a part boundary entity is dynamically specified. There are several options to determine the dynamic entity ownership of part boundary entities.

FMDB determines the part boundary entity ownership based on the rule of *poor-to-rich ownership* which assigns the poorest part as the entity owner, where the poorest part is the one with least number of owner partition objects [9].

2.5 The Partition Model

To meet the goals and functionalities of distributed meshes, a *partition model* concept is introduced between the mesh and the geometric model. Figure 2.11 shows how 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-part boundary links with ease.

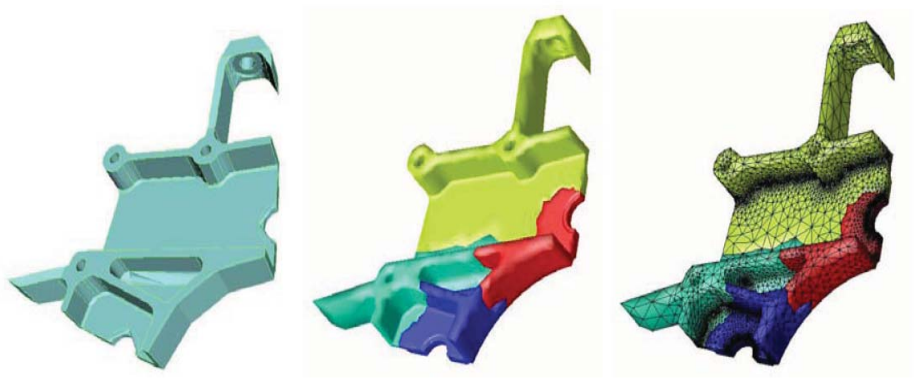


Figure 2.11: Hierarchy of domain decomposition: geometry model, partition model and distributed mesh on 4 processors [24]

Definition 8 (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 can be uniquely determined by $\mathscr{P}$.*

Each partition model entity stores dimension, id, its bounding part(s) and the owning part. By keeping a proper relation to the partition model entity, all needed services for representing mesh partitioning can be provided and the inter-part communication is easily supported.

Definition 9 (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 as $M_i^{d_i} \sqsubset P_j^{d_j}$*

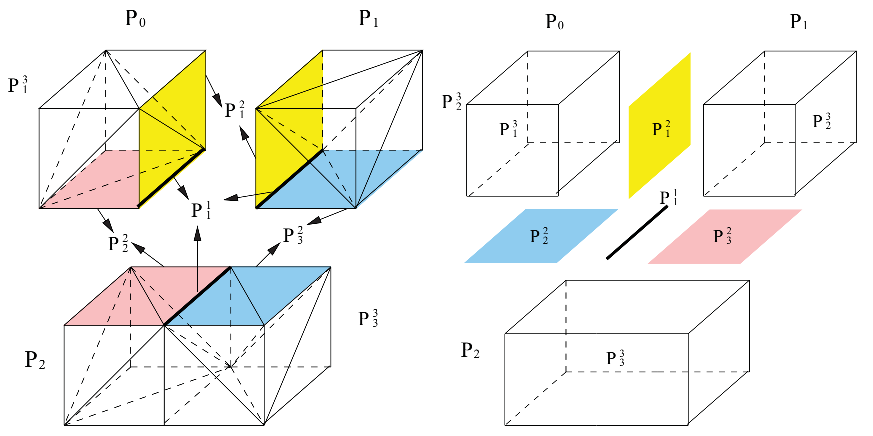


Figure 2.12: Distributed Mesh and its partition classification [24]

Definition 10 (Reverse Partition Classification) *For each partition entity, the set of equal order mesh entities classified on that defines the reverse classification for the partition model entity. The reverse partition classification is denoted as* $RC(P_j^d) = \{M_i^d | M_i^d \sqsubset P_j^{d_j}\}$. *The reverse classification operator only returns the same order mesh entities.*

Figure 2.12 shows a 3D partitioned mesh with mesh entities labeled with arrows indicating the partition classification of the entities onto the partition model entities and the 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-part boundary entities; therefore they are classified on partition regions. The reverse classification of P_1^1 returns mesh edges located on the thick black line, and the reverse partition classification of partition face P_i^2 returns mesh faces on the shaded planes [9].

The implementation of partition model is a parallel extension of FMDB, such that the standard FMDB entities and adjacencies are used on process only with the addition of the partition entity information needed to support operations across multiple processes [9]. The partition model introduces a set of topological entities that represents the collection of mesh entities based on their locations with respect to the partitioning. Grouping mesh entities to define a partition model entity can be done with multiple criteria based on the level of functionalities and needs of distributed meshes. At a minimum, the *residence part* must be a criterion to support inter-part communication. FMDB uses the residence part criterion in its partition model [24].

2.6 Dynamic load balancing and mesh modification

The evolving nature of an adaptive discretization introduces load imbalance into the solution process. Therefore, it is critical that the load be dynamically rebalanced as the calculations proceeds. The tools needed to support dynamic re balancing must determine which mesh entities should be migrated and must be able to move mesh entities from one part to another. The part communication and entity migration routines control the process of moving mesh entities between processors, while either a distributed repartitioner or iterative load balancer can be used to determine which mesh entities should be moved to different parts [27].

2.6.1 Entity migration

The mesh migration procedure uses an owner update rule to collect and update any changes to the part boundaries after moving entities to other parts. The migration of mesh entities from a given part to destination parts proceeds in three stages (See Figure 2.13). In the initial stage, sender parts collect entities to migrate and clear existing partitioning data. The partition classification of entities-to-migrate is updated and entities to be removed are collected. In the second stage, entities are exchanged and the remote copy information is updated. In the last stage, the

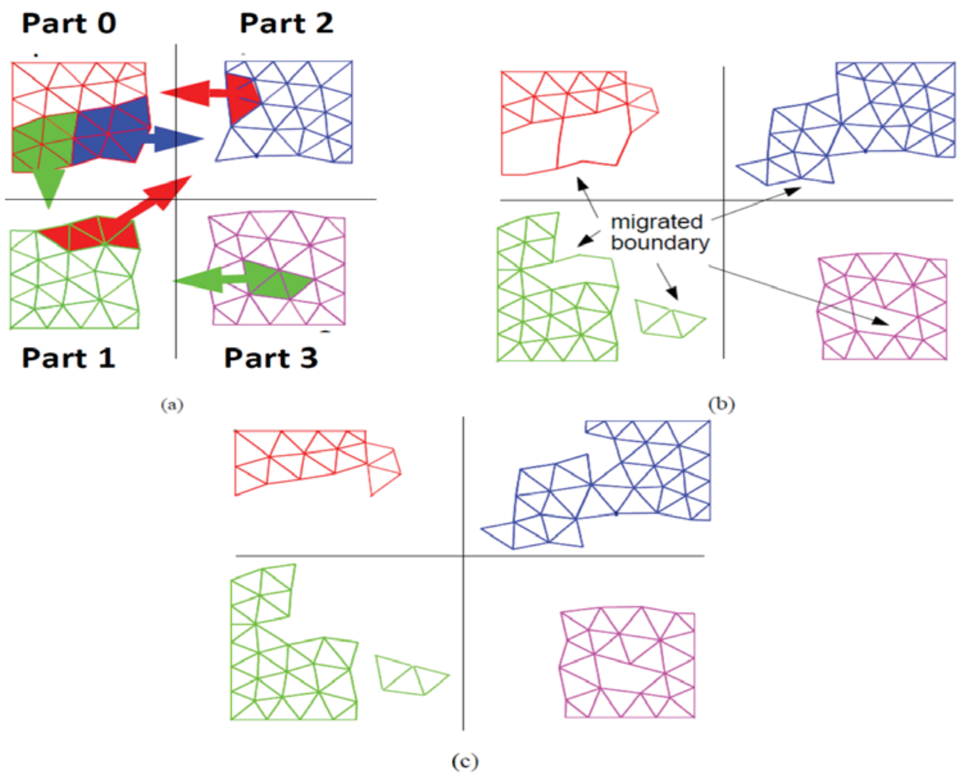


Figure 2.13: Example of entity migration of a 2-D mesh [33]

owning partition of partition model entities is updated and unnecessary entities are removed. A detailed explanation of the procedure and demonstration of it's scalability is given in [9, 3].

2.6.2 Dynamic load balancing

Mesh adaptation introduces load unbalance in parts. This unbalance is not acceptable if one wants to achieve scalable parallel software. The solution to this unbalance is to dynamically re-partition the mesh. For example, the zoltan [34] toolkit is a package which includes dynamic load balancing libraries based on geometric, graph and hyper-graph partitioning. Classically, the balancer takes as input a representation of the parallel mesh and provides as output a partition vector telling on which part a mesh entity has to be in order to restore the load balance. The completion of dynamic re-partitioning consists of dynamically moving the appropriate entities from one part to the other [29].

2.6.3 Ghosting

In order to avoid excessive communication with other parts, it is beneficial to maintain read-only copies of non-part boundary entities from other parts in specific applications (See §3.1 for details). These read-only copies are often referred as *ghost entities*. Through the introduction of ghost entities, data from remote parts can be used locally, as long as the data values have not changed on the original part (the part which owns the entity from which the ghost entity is copied). In the case where the data does change, the ghost copy data must be updated before it is used again on the remote parts having ghost copies. Ghosting relies on the concept of neighborhood of parts to copy data. A detailed account of the ghosting procedure in FMDB is discussed in chapter 3.

2.7 ITAPS parallel mesh component

The ITAPS [19] parallel mesh component, iMeshP, builds on previous work that resulted in the definition of a serial abstract data model and interfaces for serial mesh data (See §2.2). This section describes the ITAPS parallel data model (See §2.7.1).

2.7.1 The abstract data model

The parallel ITAPS data model extends the concepts described in §2.2 to handle the requirements of distributed memory applications. These requirements are addressed through the following additional core concepts.

- A *mesh partition* is a decomposition of the mesh entities (for example vertices, edges, faces, and regions) into *parts*. The *partition* is responsible for mapping the entities to parts and for mapping the parts to processes. Each process may have one or more parts and that each part is wholly contained within a process. Parts are identified globally by unique part IDs and, within a process, by opaque part handles. A partition has a communicator associated with it. Thus *global* operations are performed with respect to data in all parts in the partition's communicator and *local* operations are performed with respect to either a part's or process's data.

- Mesh entities are owned by exactly one part in the partition where ownership imbues the right to modify. It is important to note that ownership is not necessarily static during the course of a computation and can be changed due to a repartitioning of the mesh or due to local micro-migration operations. In addition, some entities will have read-only copies on other parts, for example, along part boundaries and for ghosting operations. No globally unique entity IDs are required or supplied by the data model although they can be constructed by the user as a pair [part ID, entity handle].

- Mesh entities can be further classified as an internal entity (an owned entity not on an inter-part boundary), a part-boundary entity (an entity on an inter-part boundary which are shared between parts), or a ghost entity (a non-owned, non-part-boundary entity). Copies are defined to be all ghost entities plus all non-owned part-boundary entities [20, 35, 22].

2.7.2 The iMeshP interface

Once the abstract data model is defined, the next step to creating interoperable technologies is to define common interfaces that support its functionality. A key aspect of the ITAPS approach is that it does not enforce any particular data structure or implementation with the interfaces, requiring only that certain questions about the geometry, mesh, or field data can be answered through calls to the interface. All data passed through the interface is in the form of opaque handles to objects defined in the data model.

The iMeshP [22, 35] interface is an extension to ITAPS serial mesh interface iMesh [21, 22]. The extension to iMeshP, the parallel mesh interface, requires the definition of a number of additional functions; for example, functions to create and modify partitions, create ghost entities, retrieve ghost and owner entity tag data, and determine an entity's ownership status. Additional iMeshP functions provide information about part boundaries and neighboring parts. Furthermore, the iMeshP interface supports parallel operations needed for efficient computation, load balancing and mesh modification. By necessity, these operations involve parallel communication and both synchronous and asynchronous parallel operations are supported.This design enables such things as updates of tag data in ghost entities during computation, large- or small-scale entity migration for dynamic load balancing or edge swapping, updates of vertex coordinates in non-owned vertices for mesh smoothing, and coordination in the creation of new entities along part boundaries for mesh refinement [22, 35] etc.

2.8 Implementation of FMDB

One important aspect of building a mesh database is its software design and implementation. FMDB is implemented with C++ and provides an API (Application Programming Interface) for C/C++. It uses several advanced C++ elements such as STL(Standard Template Library), templates and generic programming concepts. Moreover, FMDB also employs reusable design patterns like singleton, iterators [36] etc. to achieve re-usability of the software. MPI (Message Passing Interface) [37] and IPComMan [38] are used for efficient parallel communications between pro-

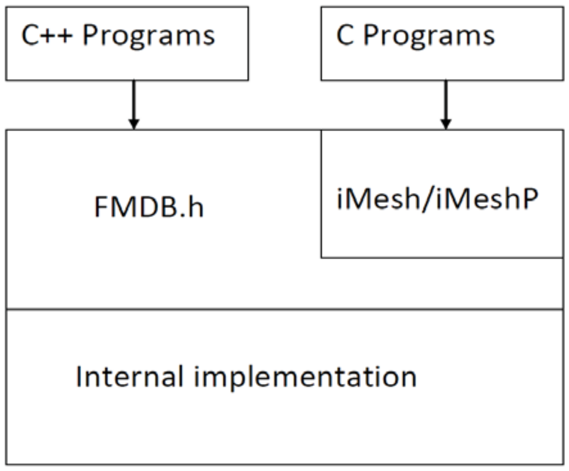


Figure 2.14: FMDB Implementation Structure

cessors. It also uses Zoltan library [34] to make partition assignment during dynamic load balancing. This section briefly describes the design, implementation & testing of the FMDB for serial and parallel operations.

2.8.1 Implementation Structure

Figure 2.14 illustrates the implementation structure of FMDB. Note that the *FMDB.h* interface encapsulates all the internal implementation of FMDB. It is the single point of contact with component services and their implementations running on top of FMDB that support simulations involving complex domains, adaptive techniques and higher order methods for example adaptive mesh refinement, coarsening and swapping through meshAdapt [39]. Moreover, the iMesh and iMeshP component API implementation of FMDB are also built on top of FMDB.h.

- The *'Internal implementation layer'* provides the serial and parallel mesh data structures like entity, entity set, part, generic iterator, mesh and tag. It also supports serial and parallel operations on these data structures like entity/entity set/part/mesh management, mesh migration, ghosting, adjacencies queries, attaching or detaching tag data, dynamic load balancing, etc.

- The *FMDB APIs* is in turn an abstraction describing the FMDB.h interface for interaction with the set of functions provided in the internal implementation layer. The interface functions provide the following functionalities
  - *system*: The operation is system wide for example initializing parallel services, getting system time, getting MPI process ID etc.
  - *tag*: The operation is related to tag data for example create or delete a tag, get tag information etc.
  - *mesh*: The operation is performed on the mesh for example create/delete a mesh, read/write a mesh.
  - *part*: The operation is performed on the part or mesh entities in the entire part for example create/delete a part, read/write a part, get information about entities in a part, get/set tag data on a part, carry out mesh migration, load balancing or ghosting.
  - *entity*: The operation is performed on a specific mesh entity for example create/delete a mesh entity, get entity type/topo/geometric classification/adjacency information, get/set tag on an entity, get/set remote or ghost copy of an entity etc.
  - *entity set*: The operation is performed on a specific entity set for example create/delete entity set, add/remove entity in an entity set, check entity existence, get size and type of entity set, traverse entity set with specific type/topology, get/set tag on entity set, get/set weight of entity set.
  - *iterator*: The operation is performed on a specific iterator (part, entity set or geometric model entity iterator) with conditions like specific type or topology.
- The FMDB.h API has all interface functions which provides services to the

ITAPS mesh interfaces (iMesh & iMeshP) [19].

For a geometry, partition and mesh model, the term instance is used to indicate the model data existing on each process. For example, a mesh instance on process P_i means a pointer to a mesh data structure on process P_i, in which all parts on process P_i are contained and from which they are accessible. For all other data such as entity and entity set, the term handle is used to indicate the pointer to the data. For example, a mesh entity handle means a pointer to the mesh entity data.

2.8.2 Design of Testing suites

Software testing is a fundamental component of software quality assurance and represents a review of the software's specification, design and implementation. The primary purpose of unit testing is to detect software failures such that defects are discovered and corrected. On the other hand, *integration tests* seeks to verify the interfaces between components against a software design. FMDB provides API unit tests to verify the functionality of its data structures and integration tests to verify all of the underlying data structures interact with each other according to the FMDB design. Details on FMDB testing can be found in Appendix A.

CHAPTER 3
N-Layer Ghost Creation & Deletion Algorithm

In a distributed mesh, most of the time communication occurs only among neighboring parts. Even then, network communication overheads can be considerably large. To minimize this communication, applications must consider the size and number of the messages being sent. For each entity, the analysis code usually tends to access adjacent entities to compute corresponding results for the current simulation step. Therefore, it is important to cluster elements or nodes in a way to minimize the number of adjacent entities belonging to different parts, thus minimizing the need for communication. The graph based partitioners used by FMDB account for those needs [3]. In addition to requiring data associated with shared part boundary mesh entities, specific finite element calculations require data from mesh entities internal to neighboring parts. One means to support providing the data is by the addition of copies referred to as ghost entities as needed on neighboring parts. The ghost entities are non-part boundary entities that are asymmetrically shared, one side provides values of the ghost from the owner entities, and the other side accepts read-only copies of these values.

The ghosts are typically adjacent to only part-boundary entities or other ghost entities. Their basic purpose is to provide data that may be needed by the analysis program for computations on the part boundary entities adjacent to ghosts and to ensure local mesh consistency on the boundaries of the mesh. The ghost information may become invalid after mesh modification that is why ghosts are required to be consistent with respect to the local part, not with corresponding mesh entities in other parts. Either the ghost information is synchronized for consistency after mesh modification [40] or it is thrown away and new ghosts are created [35].

This chapter presents an N-layer ghost creation/deletion algorithm in FMDB that can be applied to 1-D, 2-D and 3-D meshes. The ghost algorithm uses distributed system paradigm[4] where each part has its own private memory space and the communication is done using MPI [37]. The chapter first analyzes ghosting support in existing mesh databases and its usage in mesh-adaptive infrastructures (§3.1), it then explains the ghosting process (§3.2), followed by a N-layer ghost creation (§3.3) and deletion (§3.4) algorithm with the help of a 2-D mesh example.

3.1 Historical Review

Papers have been published on the issues of parallel adaptive analysis including parallel mesh generation [41, 42, 43, 44, 45, 46], dynamic load balancing techniques [47, 48, 33, 49, 50, 51], and data structures and algorithms for parallel structured [52, 53] or unstructured mesh adaptation [54, 55, 56].

As communication is expensive in distributed meshes, it is important to design the parallel mesh operations carefully to fully utilize the benefits of parallel programming. Only a small number of libraries handling parallel unstructured meshes exist. Some of them utilize the ghosting concept for specific applications for example during adaptive mesh refinement. Some of them are discussed below.

- ParFUM [40] is a mesh database that deals with unstructured meshes. It is based on Charm++, a framework for development of parallel object-oriented applications [57]. ParFUM implements the classic element-node structure, with domain attributes associated to the mesh entities. It includes some adjacency information like node-to-node, node-to-element and element-to-element for some types of analysis. Mesh partitions are called chunks, and are associated to exactly one MPI process [58]. Communication between chunks is done implicitly through shared and ghost entities. ParFUM allows the creation of ghost layers on the boundary of each chunk. The ghost layer consists of read-only copies of neighboring entities from other parts referred as ghost elements and nodes. Multiple ghost layers can be added by calling the same ghosting routine repeatedly, which adds ghost layers one after the other [59].

- Reference [60] uses ghost layers in *Triangle* which is an implementation of two-dimensional constrained Delaunay traiangulation and refinement algorithm for quality mesh generation [61]. The triangle-based divide and conquer algorithm recursively halves the input vertices until they are partitioned into subsets of two or three vertices each. Each subset is easily triangulated (yielding an edge, two collinear edges, or a triangle), and the triangulations are merged together to form larger ones. Each triangle is surrounded with a layer of *ghost* triangles which is useful in efficiently traversing the edges during the merge step. After the merge step of divide-and-conquer algorithm is completed, the ghost triangles are thrown away [62].
- The SIERRA [63] framework provides a set of general tools for supporting the development of mechanics applications. It includes a distributed unstructured mesh data structure, along with adaptivity and load balancing, an interface to linear solvers, and support for creating multiphysics applications. The SIERRA's FEM data structure represents node, edge, face and element entities (or objects). Mesh entities on the boundary of a partition can be shared with other parts, although only one is chosen the owner of the entity. A mesh entity that resides in a process and is not in the closure of the process' owned subset is termed a *ghost* mesh entity. SIERRA uses ghost entities in computations that gather field data from an owned entity's neighbors. Multiple independent computations may require their own particular subsets of parallel ghosted mesh entities. These computation-specific ghosted subsets are mesh bulk data that

is created, owned, modified and destroyed by an application [64].

- Reference [65, 66] uses ghost layers for parallel adaptive mesh refinement and coarsening in ALPS (Adaptive Large-scale Parallel Simulations) which is a library for parallel octree-based dynamic mesh adaptivity and redistribution.

Many grid-based frameworks use ghosting for certain applications like parallel mesh refinement and boundary conditions calculations [67, 68, 69]. Some selected infrastructures are dicussed below:

- Reference [70] describes Racoon, a framework that offers a grid-based environment for the mesh-adaptive solution of conservative systems and related systems. It exploits both shared and distributed memory architectures as the parallelization strategy is a hybrid of multi-threading and inter-process communication through MPI [37] and POSIX-multithreading [71]. It also supports mesh refinement, re-gridding, load balancing and distribution. During mesh refinement, bands of ghost cells are created and updated around the individual grid blocks. At refinement boundaries, finer blocks receive interpolated ghost cell values from their respective coarser neighbors, while coarser blocks receive averaged values from the finer region. The existence of ghost cells handles every single grid block as a quasi independent piece of work for each integration step during mesh refinement, and assigns the block to processes for execution.

- Reference [72] describes an Adaptive Mesh Refinement (AMR) infrastructure for finite difference calculations on block-structured adaptive meshes and a solver for elliptic Partial Differential Equations. In AMR, grids are often accreted to accommodate a surrounding layer of *ghost cells* for handling various boundary conditions. AMR interpolates ghost values using physical boundary conditions. Basic operations of AMR include copying values from one grid to another such as exchanging ghost values between grids, perform irregular computations for updating boundary values like location dependent quadratic interpolation of ghost values at coarse-fine grid interface. The ghost methods exchange values between neighbors, for examples whose domains are adjacent but disjoint and cache those values in local ghost regions.

Some mesh databases provide support for distributed meshes [44, 55, 73, 74, 75, 33].

- The Parallel Algorithm Oriented Mesh Database (PAOMD) [29] extends the Algorithm Oriented Mesh Database (AOMD) [75] to support distributed meshes. It provides a general parallel mesh management framework in which mesh representation can be adapted to different types of applications. Each partition is assigned to a processor, and the local mesh is represented by a serial AOMD mesh. Mesh entities that are classified on partition boundaries must exist in the parallel data structure and may be shared with other partitions though ghost entities are not supported.

- Reference [76] 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 [33], 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, though the support for ghost entities was not there.

Some of the above mesh databases that provide ghosting support are based on classic element-node representation [55, 40]. Many others provide a partial parallel support [77, 55] or only use shared part boundary entities without ghosting support [29, 76, 74]. Ghosting support is essential in selected applications. Therefore, a general topology-based distributed mesh data structure having all features of a parallel mesh database is important to efficiently support parallel adaptive analysis procedures.

3.2 The Ghosting Process

Ghost entities are read-only copies of remote part entities. They are copies of additional, non-boundary entities which are requested by an application to enable efficient computation. Ghost entities are specified similar to second order adjacencies (§2.3.4) i.e. through a *bridge dimension* [35]. Figure 3.1 depicts a distributed mesh on four parts with ghost entities along the part boundaries.

The ghost creation and deletion process uses the following criteria

- *Ghost dimension*: Dimension of the ghost entities created. Permissible options in a topological mesh representation can be regions, faces or edges. As ghost entities are specified through a bridge dimension [35], the lowest possible dimension of a ghost entity can be an edge since the minimum bridge is a vertex. Vertices are ghosted if they are part of higher dimension ghost entities. For example, in a 1-D mesh, the only possible ghost dimension is an edge and the

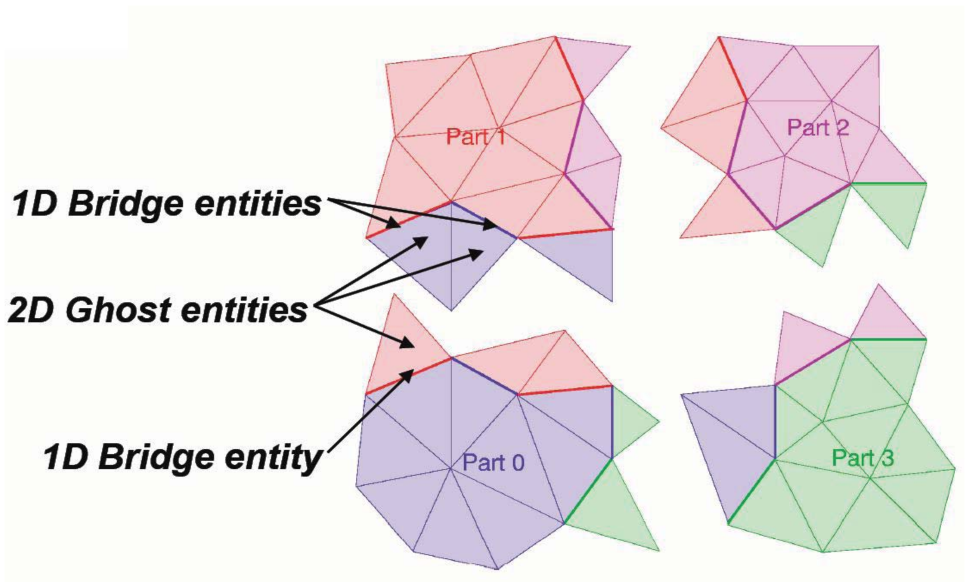


Figure 3.1: A distributed mesh on four parts with ghost entities [19]

vertices that are on the boundary of ghost edges can be ghosted to create the ghost edges.

- *Bridge dimension*: Dimension of the bridge entities used for ghost creation. The bridge entity must be of lower topological order than the ghost and can be a face, edge or a vertex in a topological mesh representation. Bridge entities are on the boundary of higher order ghost entities for example if $\{M_1^1\{M^2\}_1\}$ where M_1^1 is the bridge entity, then M_1^2 becomes a ghost candidate.

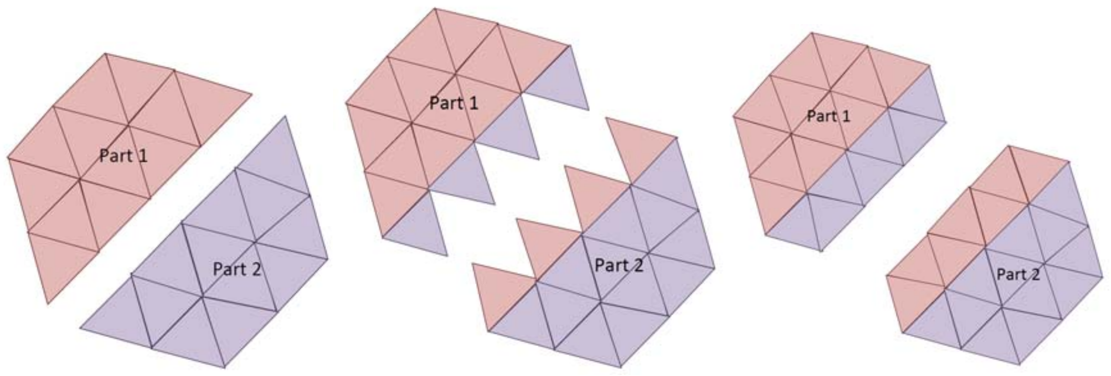
- *Number of layers*: Number of layers of ghost entities. Layers are measured from inter-part boundary of the mesh.

The ghost and the bridge dimensions are represented by symbols g and b through out the thesis. The number of layers of ghost entities are represented by $numLayer$.

Definition 11 (Ghost object) *A ghost object is a mesh entity of dimension g that can be marked for ghosting.*

For example, to get two ghost layers of regions, measured from faces, the ghost dimension is set to region, bridge dimension to faces and number of layers is 2. In Figure 3.1, ghost dimension, bridge dimension and number of layers are 2, 1 and 1 respectively. At a minimum, a ghost entity's owner must store information about its ghost copies that exist. A ghost entity also stores information about its owner entity and the part where the owner entity exists [35]. This provides a mechanism to keep the owners synchronized with their ghost copies.

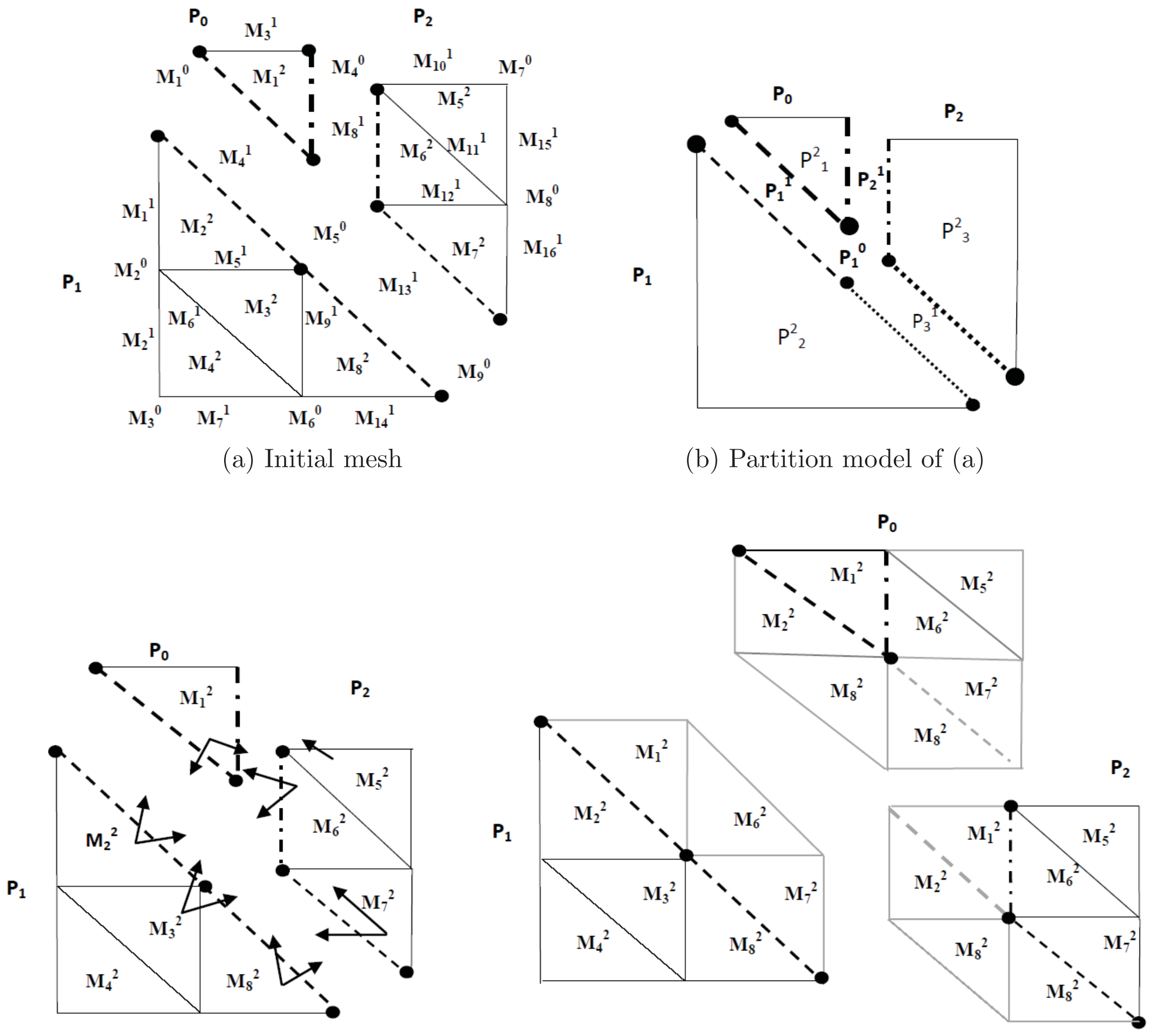
If there are multiple ghost layers, the ghosting process should start with the first (innermost) layer of ghosts adjacent to the part boundary. After collecting entities for the first layer, it collects entities for the second layer by treating the first layer of ghosts as part of the mesh for the second layer. For multiple layers, it also marks the elements that share entities with elements of the first layer. Care is taken that the same element is not added as a ghost again. Figure 3.2 illustrates the ghosting process by creating ghosts of 2-D triangles. Figure 3.2 (b) depicts a ghosted mesh with ghosts of faces using edges as a bridge. Figure 3.2 (c) depicts a ghosted mesh with ghosts of faces using vertices as a bridge.



(a) Mesh without ghosts (b) Ghosted mesh based on *edge bridge* (c) Ghosted mesh based on *vertex* bridge

Figure 3.2: A 2-D ghosted mesh

3.3 Algorithm of N-layer ghost creation



(a) Initial mesh (b) Partition model of (a)

(c) Mesh during 1-layer ghosting (d) Mesh after 1-layer ghosting

Figure 3.3: Example of 2-D ghost creation with 1-layer

This section presents the N-layer ghost creation algorithm with the help of a 2-D mesh example. An efficient ghost creation and deletion algorithm with minimum resources (memory and time) is useful to achieve high performance in parallel adaptive mesh-based simulations. Figure 3.3 (a) and (b) illustrate the 2-D partitioned mesh and its associated partition model to be used for the example of ghost creation throughout this section. In Figure 3.3 (a), the partition classification of entities on the part boundaries is denoted with the lines of the same pattern. For instance, M_1^0 and M_1^4 are classified on P_1^1, and depicted with the dashed line as P_1^1. In Figure 3.3 (b), the owning parts of partition model edge (resp. vertex) is illustrated with thickness (resp. size) of lines (resp. circles).

The input of the ghosting procedure is the dimension of the ghost entities (g), dimension of the bridge entities (b) and number of layers of the ghosts (numLayer). The option whether to include remote copies of bridge entities in the ghosting collection process can also be specified using the *includeCopy* parameter. If includeCopy is true, both remote and owned entities on a part are used as bridge entities. For example, in Figure 3.1, the input of the ghost creation algorithm is $[g = 2, bDim = 1, includeCopy = 1, numLayer = 1]$ as a result of which the non-owned bridge edges (edges shown by color of the other part) on $Part0$ and $Part1$ are also used for ghost collection.

The input to ghost creation algorithm in Figure 3.2 (b) is $[g = 2, bDim = 0, includeCopy = 1, numLayer = 1]$ and for Figure 3.1 is $[g = 2, bDim = 1, includeCopy = 1, numLayer = 1]$. The input to ghost creation algorithm for 2-D mesh in Figure 3.3(a) is $[g = 2, bDim = 0, includeCopy = 1, numLayer = 1]$. Figure 3.3 shows the 2-D mesh during 1-layer ghost creation. A 2-layer ghost creation of the same 2-D mesh will be discussed in §3.3.2. Through out this chapter, we will use the example 2D mesh in Figure 3.3 to explain the ghost creation and deletion algorithm. In Figure 3.3 (c) - (d), first layer bridge entities are depicted with black circles.

The overall procedure for ghost creation is the following:

1. Given the bridge dimension b, ghost dimension g and number of ghost layers $numLayer$, find entities of dimension g adjacent to part boundary bridge entities of dimension b and determine the destination parts of these g dimensional entities. Store these entities-to-ghost in the container $entitiesToGhost[g]$

2. Process next ghost layer using the destination parts of ghost entities collected in the first layer.

3. Ghost collection on every part is independent of ghost collection on other parts. As remote copies exist on multiple parts, more than one part can collect the same entity (original entity or its remote copy) for ghost creation. Carry out one round of communication to eliminate duplicate remote copies from the collected ghost entities.

4. Exchange entities, create ghost entities and update ghost copies.

5. Store ghost rule comprising of ghost entity dimension, bridge entity dimension and number of ghost layers in the part.

Given b, g and $numLayer$, the first procedure finds ghost candidates among the entities of ghost dimension g that are adjacent to part boundary entities of bridge dimension b. If the bridge dimension entity (M_i^b) has a remote copy on a part p but the ghost dimension entity M_j^g does not have a remote copy on p, then M_j^g is collected for ghost creation on part p. Subsequent ghost layers are also termed as $neighbors$ of $neighbors$ and are processed based on the destination part ids of the entities collected for ghost creation in the first layer. As a remote copy can exist on multiple parts, more than one part can send the same entity to the same destination part. Therefore, duplicate remote copies are eliminated from the entities collected for ghost creation. After creating ghost of collected entities on the destination parts, ghost information of the owner entities is updated. Algorithm 1 is the pseudo code of the ghost creation procedure.

3.3.1 Step 1: Ghost collection for first layer

Step 1 collects mesh entities adjacent to part boundary bridge entities for ghost creation. The entities collected for ghosting are maintained in vector $entitiesToGhost$

```
Data: distributed mesh $M$, ghost dimension $g$, bridge dimension $b$,
      include copy $includeCopy$, number of ghost layers $numLayer$
Result: Creates ghosts on the distributed mesh M
begin
    /* Step 1:  collect ghost entities in the first layer     */
    getGhostEnts(M, g, b, includeCopy, entitiesToGhost, numLayer);

    /* Step 2:  Process next ghost layer if $numLayer > 1$      */
    for $lyr \leftarrow 2$ to $numLayer$ do
        processNLayers(M, g, b, includeCopy, lyr, entitiesToGhost);
    end for

    /* Step 3:  Do one round of communication to eliminate
       duplicate remote copies                                  */
    for $d \leftarrow 0$ to $g + numLayer$ do
        removeDuplicateEnts(d, entitiesToGhost[d]);
    end for

    /* Step 4:  Exchange entities and update ghost information
       */
    for $d \leftarrow 0$ to $g + numLayer$ do
        exchangeGhostEnts(M, g, d, entitiesToGhost[d]);
    end for
    /* Step 5:  Store ghost information in the part            */
    $ghostRule \rightarrow [g, b, numLayer, includeCopy]$
end
```

Algorithm 1: createGhosts(M, g, b, numLayer)

where $entitiesToGhost[i]$ contains the entities of dimension $i, i = 0, 1, 2, 3$. Each part maintains the separate $entitiesToGhost[i]$ with different contents. For every entity collected, $ghostParts[M_i^g]$ is a vector that holds its destination parts. An entity, M_i^g is collected for ghost creation using Definition 12. Algorithm 2 collects ghost entities for the first layer. After an entity is marked for ghost creation, its downward adjacent entities are also collected for ghost creation.

Definition 12 (Destination part rule for 1^{st} layer ghosts of M_i^d) *If $d = g$ and $\{M_i^d\{M^b\}_j\}$, $b < g$, and $p \in \mathscr{R}[M_j^b]$, $p \notin \mathscr{R}[M_i^d]$ then $p \in ghostParts[M_i^d]$. Otherwise, if $d < g$ and $M_i^d \in \{M_k^g\{M^d\}\}$ where $p \in ghostParts[M_k^g]$ and $p \notin \mathscr{P}[M_i^d]$*

then $p \in ghostParts[M_k^d]$.

Although the user may only want to add data to use the ghost entities of dimension g, the downward adjacent entities are only carried along for ghost creation as FMDB creates higher order entities using the lower order entity information. For example, to create M_2^2 on P_0, M_5^1 and M_1^1 are carried along to P_0 (M_4^1 is already owned by P_0).

On P_1, the ghost collection algorithm works as follows

1. P_1 has three part boundary vertices M_1^0, M_5^0 and M_9^0. As the bridge dimension specified for ghosting is zero, these three vertices are the first layer bridge entities. Among these vertices, P_1 owns M_1^0 only as show by Figure 3.3 (b).

2. Ghost dimension entities adjacent to M_1^0 are M_2^2, M_5^0 are M_2^2,M_3^2,M_8^2 and M_9^0 is M_8^2 (Algorithm 2 Step 1.1).

3. As includeCopy is true, bridge entities M_5^0 and M_9^0 (remote copies) are considered for ghost creation. If includeCopy is set to false, only M_1^0 would be considered in the ghosting process (Algorithm 2 Step 1.2).

4. M_1^0 has remote copy on P_0 but M_2^2 does not exist on P_0 so according to Definition 12, M_2^2 will be ghosted on P_0 (Algorithm 2 Step 1.3).

5. The next part boundary entity visited is M_5^0. M_5^0 has remote copies on P_0 and P_2 so M_2^2 and M_8^2 will be sent to P_0 and P_2. M_2^2 is was already marked for ghosting (in 4) on P_0 so P_1 will be added to $ghostParts[M_2^2]$

6. The next part boundary entity visited is M_9^0. M_9^0 has remote copies on P_2 but M_8^2 is already marked for ghosting on P_2 (in 5).

7. Downward adjacent entities of M_2^2, M_8^2 and M_3^2 are collected for ghosting using Algorithm 3 (Step 1.4 of Algorithm 2, See Table 3.1).

```
Data: $M$, $g$, $b$, $includeCopy$, $numLayer$
Result: Collect entities to ghost in vector $entitiesToGhost$
begin
    /* Step 1:  Collect entities and their downward adjacencies
       in $entitiesToGhost$                                              */
    /* Step 1.1 For every part boundary bridge entity $M_i^b$, get
       its upward adjacent entity of dimension $g$, $M_j^g$              */
    foreach $M_i^b \in \{\partial(P_{local})\}$ do
        /* Step 1.2 $includeCopy = false$ excludes non-owned bridge
           entities.                                                     */
        if $includeCopy == true$ & $P_{local}$ not owns $M_i^b$ then
            continue;
        end if
        /* Step 1.3 If $M_j^g$ does not exist on a remote part $p$
           where $M_i^b$ exists, collect $M_j^g$ for ghosting            */
        foreach $M_j^g \in \{M_i^b\{M^g\}\}$ do
            foreach $part\ ID\ p \in \mathscr{R}[M_i^b]$ do
                if $p \notin \mathscr{R}[M_j^g]$ then
                    insert $M_j^g$ in entitiesToGhost[g];
                    insert $p$ in ghostParts[$M_j^g$];
                    /* Step 1.4 collect downward adjacent entities
                       of $M_i^g$                                        */
                    getDownwardAdjs(M, g, $M_i^g$, p, entitiesToGhost);
                end if
            end foreach
            /* Step 2:  Mark $M_j^g$ as visited                          */
            if $numLayer > 1$ then
                set $M_j^g \leftarrow visited = true$;
            end if
        end foreach
    end foreach
    /* Step 3:  Mark the bridge entity as visited                        */
    if $numLayer > 1$ then
        set $M_i^b \leftarrow visited = true$;
    end if
end
```

Algorithm 2: getGhostEnts(M,g, b, includeCopy, entitiesToGhost, numLayer)

Table 3.1: Contents of vector *entitiesToGhost* after Step 1

	P_0	P_1	P_2
$entitiesToGhost[0]$	$M_1^0\{2\}$,$M_4^0\{1\}$	$M_1^0\{2\}^*$,$M_2^0\{0,2\}$, $M_9^0\{0\}$,$M_6^0\{0,2\}$	$M_4^0\{0,1\}^*$,$M_7^0\{0\}$, $M_8^0\{0,1\}$,$M_9^0\{2\}^*$
$entitiesToGhost[1]$	$M_3^1\{1,2\}$,$M_8^1\{1\}$, $M_4^1\{2\}$	$M_1^1\{0,2\}$,$M_4^1\{2\}^*$, $M_5^1\{0,2\}^*$,$M_6^1\{0,2\}$, $M_9^1\{0,2\}$,$M_{13}^1\{0\}$, $M_{14}^1\{0,2\}$	$M_8^1\{1\}^*$,$M_{10}^1\{1\}$, $M_{11}^1\{0,1\}$,$M_{12}^1\{0,1\}$, $M_{13}^1\{0\}^*$,$M_{15}^1\{0\}$, $M_{16}^1\{0,1\}$
$entitiesToGhost[2]$	$M_1^2\{1,2\}$	$M_3^2\{0,2\}$,$M_8^2\{0,2\}$, $M_2^2\{0,2\}$	$M_6^2\{0,1\}$, $M_5^2\{0\}$, $M_7^2\{0,1\}$

8. M_2^2, M_8^2 and M_3^2 are marked as visited to process the 2-layer ghost creation (Step 2 of Algorithm 2, See Figure 3.4).

9. Bridge entities M_1^0, M_5^0 and M_9^0 are also marked as visited so that the same bridge entities are not processed repeatedly (Step 3 of Algorithm 2).

For a first-layer entity M_i^g adjacent to part boundary bridge entities, $ghostParts[M_i^g]$ are decided using Definition 12. For a downward adjacent entity M_k^d of ghost entity M_i^g, destination parts are also decided using Definition 12.

The entities after 1st layer ghost collection are

- $M_i^g \in \{M_j^b\{M^g\}\}$ satisfying Definition 12.

- For each M_i^g, downward adjacent entities $M_j^q \in \{\partial(M_i^g)\}$, $q < g$ satisfying Definition 12. Algorithm 3 collects downward adjacent entities of the ghosts.

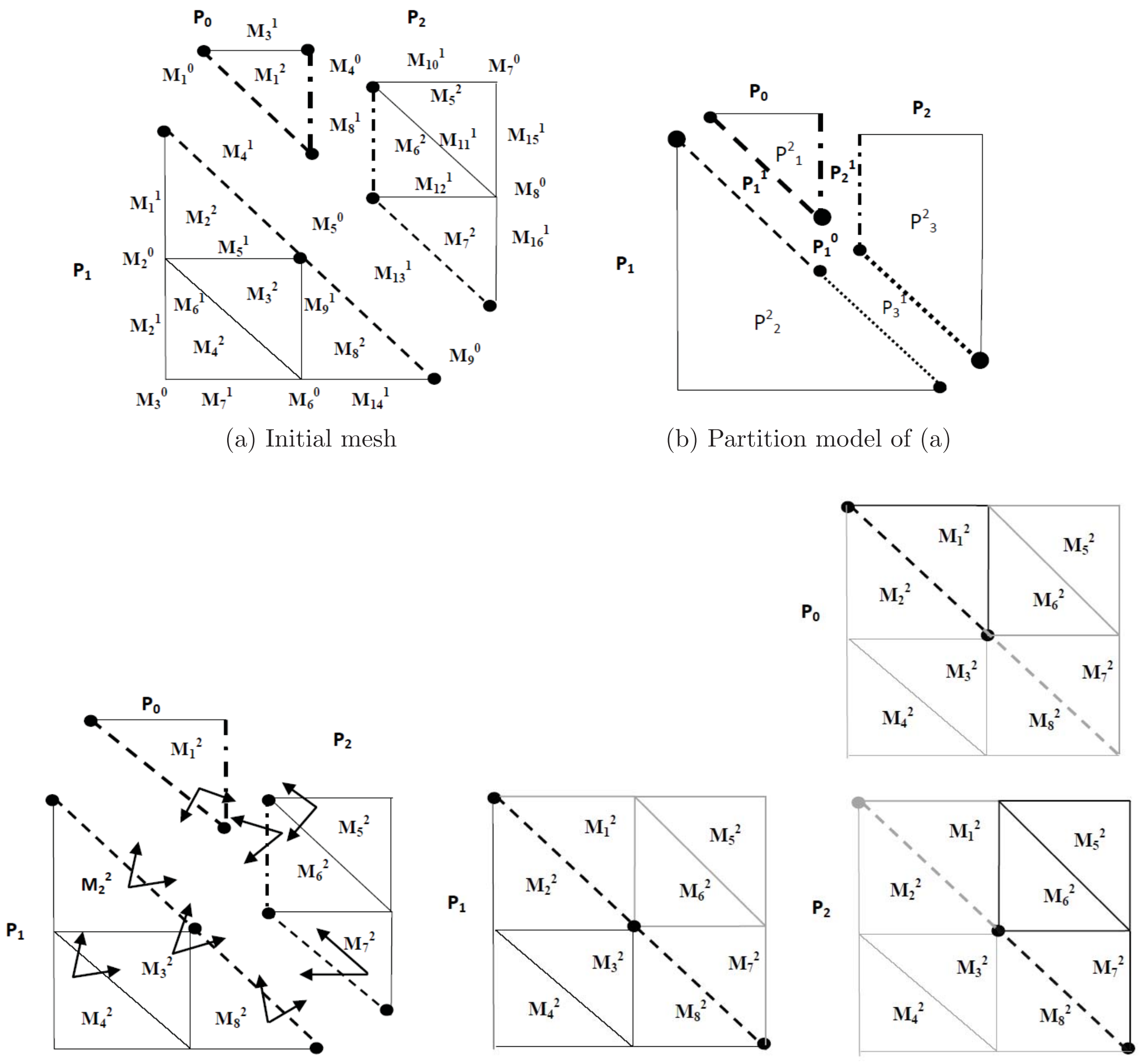
The ghost creation algorithm works only on those entities. The remaining entities are not affected by ghosting, therefore they should not be considered nor touched. For the example given in Figure 3.3(c), the contents of $entitiesToGhost$ are given in Table 3.1. Entities listed in Table 3.1 are collected for ghosting to the destination parts given in curly brackets. $entitiesToGhost[2]$ contains the mesh faces to be ghosted from each part ($g = 2$). $entitiesToGhost[1]$ contains the mesh edges which bound any mesh face in $entitiesToGhost[2]$. $entitiesToGhost[0]$ contains the mesh vertices that bound any mesh edge in *entitiesToGhost*[1] . Note that there are a few duplicate entries for remote copies in entitiesToGhost[0] and entitiesToGhost[1] marked with a *. These entities will be removed after Step 3 (eliminate duplicate entities).

```
Data: $M$, g, $M_i^g$, $destId$, $entitiesToGhost$
Result: Given an entity $M_i^g$, get its downward adjacent entities in
        $entitiesToGhost[i]$ and set destination parts of downward
        adjacent entities using $destId$.
begin
    /* Collect downward adjacent entities of the
       entity-to-ghost                                        */
    for $d \leftarrow 0$ to $g$ do
        foreach $M_k^d \in \{M_i^g\{M^d\}\}$ do
            /* Store $M_k^d$ and update $ghostParts[M_k^d]$ only if it has
               no remote copy on destId                       */
            if $destId \in \mathscr{P}[M_k^d]$ then
                continue;
            end if
            insert $M_k^d$ in entitiesToGhost[d];
            insert $destId$ in ghostParts[$M_k^d$];
        end foreach
    end for
end
```

Algorithm 3: getDownwardAdjs(M,g, M_i^g, destId, entitiesToGhost)

3.3.2 Step 2: Process next layer

If the number of layers specified for ghosting is more than one, the next layer is added after the first one. For the next layer, entities of dimension g that shares a bridge entity with any $M_i^g \in entitiesToGhost[g]$ are collected for ghost creation. A visited tag is set for each M_i^g marked for ghosting so that the same entity is not added as a ghost multiple times (See Algorithm 2 Step 2). The next layer ghost entities are collected in $entitiesToGhost[g + lyr]$, $lyr = 2$ to $numLayer$ i.e. as the number of layers increase, the size of vector *entitiesToGhost* grows proportionately.


P0
P1
P2
M1^0 M1^1 M1^2 M2^0 M2^1 M2^2 M3^0 M3^1 M3^2 M4^0 M4^1 M4^2 M5^0 M5^1 M5^2 M6^0 M6^1 M6^2 M7^0 M7^1 M7^2 M8^0 M8^1 M8^2 M9^0 M9^1 M10^1 M11^1 M12^1 M13^1 M14^1 M15^1 M16^1
P^2_1 P^2_2 P^2_3 P_1^1 P_2^1 P_3^1 P_1^0

(a) Initial mesh (b) Partition model of (a)

(c) Mesh during 2-layer ghosting (d) Mesh after 2-layer ghosting

Figure 3.4: Example of 2-D ghost creation with 2-layers

For an entity M_k^g in the second layer, destination parts are decided using Definition 13. Algorithm 4 gives the pseudo code for N^{th} layer entity collection that decides destination parts using Definition 13. Figure 3.4 shows ghost entity collection for second layer. The second layer bridge vertices are depicted by blue circles around them.

```
Data: $M$, g, b, includeCopy, n, entitiesToGhost
Result: Collects entities for ghosting in $n^{th}$ layer
begin
    /* Step 1 Process $n-1^{th}$ layer to get $n^{th}$ layer */
    foreach $M_i^g \in entitiesToGhost[g+(n-1)]$ do
        /* Step 1.1 For $M_i^g$, get its downward adjacent bridge
           entities */
        foreach $M_k^b \in \{M_i^g\{M^b\}\}$ do
            /* Step 1.2 If $M_k^b$ is visited, continue with next
               bridge entity */
            if $M_k^b \leftarrow visited == true$ then
                continue;
            end if
            /* Step 1.3 For the downward adjacent bridge $M_k^b$, get
               upward adjacent entities of dimension $g$. */
            foreach $M_j^g \in \{M_k^b\{M^g\}\}$ do
                /* Step 1.4 If the upward adjacent entity $M_j^g$ is
                   not visited, update its ghostParts and collect
                   it for ghosting */
                if $M_j^g \leftarrow visited == true$ then
                    continue;
                end if
                insert $M_j^g$ in $entitiesToGhost[g+n]$;
                $ghostParts[M_j^g] \leftarrow ghostParts[M_j^g] \cup ghostParts[M_i^g]$;
                /* Step 1.5 Set visited tag $M_j^g$ so that it is not
                   processed again */
                $M_j^g \leftarrow visited = true$;
            end foreach
        end foreach
        /* Step 1.6 Set visited tag of bridge entity $M_k^b$ so that
           it is not processed again */
        $M_k^b \leftarrow visited = true$;
    end foreach
end
```

Algorithm 4: processNLayers(M, g, b, includeCopy, lyr, entitiesToGhost)

Definition 13 (Destination part rule for next layer ghosts of M_k^d) *: If* $d = g$ *and* $\{M_k^d\{M^b\}_i\}$ *where* $\{M_i^b\{M^g\}_l\}$ .... $\{M_i^b\{M^g\}_m\}$ *s.t.* $\{M^g\}_m....\{M^g\}_l \in entitiesToGhost[g]$*, then*
$ghostParts[M_k^d] = ghostParts[\{M^g\}_m] \cup ... \cup ghostParts[\{M^g\}_l]$*. Otherwise, if* $d < g$*,* $M_i^d \in \{M_k^g\{M^d\}\}$ *where* $p \in ghostParts[M_i^g]$ *and* $p \notin \mathscr{P}[M_k^d]$ *then* $p \in ghostParts[M_k^d]$*.*

For an example, the ghost entity collection example given in Section 3.3.1 is extended for two layers

1. $entitiesToGhost[g]$ has M_2^2, M_3^2 and M_8^2 on part P_1 after Step 1. Its downward adjacent bridge entities that are not yet visited are M_2^0 and M_6^0. The rest of the downward adjacent bridges are part boundary entities that have already been visited during 1-layer ghost collection (Algorithm 4 Step 1.1 and 1.2).

2. The entity of dimension g adjacent to M_2^0 and M_6^0 that is not yet visited during 1-layer ghost collection is M_4^2 (Algorithm 4 Step 1.3).

3. As M_2^2 shares a common bridge entity M_2^0 with M_4^2, the $ghostParts[M_2^2]$ will be added to $ghostParts[M_4^2]$.

4. As M_3^2 and M_8^2 also share a common bridge entity M_6^0 and M_2^0 with M_4^2, the $ghostParts[M_3^2]$ and $ghostParts[M_8^2]$ shall be added to $ghostParts[M_4^2]$ (Algorithm 4 Step 1.4). As M_8^2, M_3^2 and M_2^2 have the same $ghostParts$, the $ghostParts[M_4^2]$ will become $\{0, 2\}$

5. Set visited flag for M_4^2 so that it is not processed again for next layer ghost collection (Algorithm 4 Step 1.5).

6. Set visited flag for M_2^0 and M_6^0 so that the bridge vertices are not processed again for entity collection in current or next layers. Placing visited tag on

bridge entities shrinks the search tree in n-layer ghost collection for example M_2^0 is marked visited when it is processed as a downward adjacent bridge of M_2^2. M_2^0 is also a downward adjacent bridge of M_3^2 which is processed after M_2^2, but as M_2^0 is already marked visited, it's upward adjacent entities will not be processed.

Table 3.2: Contents of vector entitiesToGhost after Step 2

	P_0	P_1	P_2
$entitiesToGhost[0]$	$M_1^0\{2\}$,$M_4^0\{1\}$	$M_1^0\{2\}^*$,$M_2^0\{0,2\}$, $M_3^0\{0,2\}$,$M_6^0\{0,2\}$, $M_9^0\{0\}$	$M_4^0\{1\}^*$,$M_7^0\{0,1\}$, $M_8^0\{0,1\}$,$M_9^0\{0\}^*$
$entitiesToGhost[1]$	$M_3^1\{1,2\}$,$M_4^1\{2\}$, $M_8^1\{1\}$	$M_1^1\{0,2\}$,$M_2^1\{0,2\}$, $M_4^1\{2\}^*$,$M_5^1\{0,2\}$, $M_6^1\{0,2\}$,$M_7^1\{0,2\}$, $M_9^1\{0,2\}$,$M_{13}^1\{0\}$, $M_{14}^1\{0,2\}$	$M_8^1\{1\}^*$,$M_{10}^1\{0,1\}$, $M_{11}^1\{0,1\}$,$M_{12}^1\{0,1\}$, $M_{13}^1\{0\}^*$,$M_{15}^1\{0,1\}$, $M_{16}^1\{0,1\}$
$entitiesToGhost[2]$	$M_1^2\{1,2\}$	$M_3^2\{0,2\}$,$M_8^2\{0,2\}$, $M_2^2\{0,2\}$	$M_5^2\{0\}$,$M_6^2\{0,1\}$, $M_7^2\{0,1\}$
$entitiesToGhost[3]$		$M_4^2\{0,2\}$	$M_5^2\{1\}$

Table 3.2 gives the contents of *entitiesToGhost* vector after processing the second layer.

3.3.3 Step 3: Eliminate duplicate entities

After collecting the ghost entities, the next step is to eliminate any duplicate ghost entities. A remote copy exists on multiple parts so many parts can collect it for ghost creation for the same destination. Suppose M_i^d is owned by part P_i and has a remote copy on P_j. If M_i^d is marked for ghost creation to destination part P_k on both P_i and P_j then P_k will receive the message of creating M_i^d twice.

For this reason, Table 3.1 and Table 3.2 have duplicate entries marked by a *. Algorithm 5 gives the pseudo code for eliminating duplicate ghost creation messages. For notational simplicity, we denote a local part of an entity where an entity is currently located P_{local}. One round of communication is performed in Algorithm

```
Data: $M$, dim, entitiesToGhost[dim]
Result: Eliminate duplicate entities
begin
    /* Step 1:  For all part boundary entities in
       $entitiesToGhost$ send a message to remote parts that have
       their remote copies                                        */
    foreach $M_k^d \in entitiesToGhost[dim]$ do
        /* If $M_k^d$ is not a part boundary entity, proceed with
           the next entity on $entitiesToGhost[dim]$              */
        if $M_k^d \notin \partial(P_{local})$ then
            continue;
        end if
        foreach $destId \in ghostParts[M_k^d]$ do
            foreach $p \in \mathscr{R}[M_k^d]$ do
                send message A(address of $M_k^d$ on $p$, destId) to $p$;
            end foreach
        end foreach
    end foreach
    /* Step 2:  Recieve message from other parts that are
       sending copy of the same part boundary entity              */
    while part $p$ receives message A from $P_{local}$(address of $M_k^d$ on $p$,
    destId) do
        /* Step 3:  Check if the remote part is also sending $M_k^d$
           to the same destination $destId$.                      */
        if $destId \in ghostParts[M_k^d]$ & $ID(p) > ID(P_{local})$ then
            /* Step 4:  If the remote part id $p$ is less than the
               part id of $P_{local}$ then remove $M_k^d$ as $p$ will send it
               to $destId$                                        */
            remove $M_k^d$ from entitiesToGhost[d];
        end if
    end while
end
```

Algorithm 5: removeDuplicateEnts(M, d, entitiesToGhost)

5 to determine duplicate ghost creation messages. If multiple parts intend to send ghost creation message for M_i^d to the same destination part p, after this step only the part having minimum part id will send the ghost creation message of M_i^d to p. For example, in Figure 3.3, both P_0 and P_1 collect M_1^0 for sending it to P_2 but after Algorithm 5 eliminates duplicate entities, only P_0 sends M_1^0 to P_2. Table 3.3

shows the contents of *entitiesToGhost* after eliminating duplicate entries for remote copies.

Table 3.3: Contents of vector entitiesToGhost after Step 3

	P_0	P_1	P_2
$entitiesToGhost[0]$	$M_1^0\{2\}$,$M_4^0\{1\}$	$M_2^0\{0,2\}$,$M_3^0\{0,2\}$, $M_6^0\{0,2\}$,$M_9^0\{0\}$	$M_7^0\{0,1\}$,$M_8^0\{0,1\}$
$entitiesToGhost[1]$	$M_3^1\{1,2\}$,$M_4^1\{2\}$, $M_8^1\{1\}$	$M_1^1\{0,2\}$,$M_2^1\{0,2\}$, $M_5^1\{0,2\}$,$M_6^1\{0,2\}$, $M_7^1\{0,2\}$,$M_9^1\{0,2\}$, $M_{13}^1\{0\}$,$M_{14}^1\{0,2\}$	$M_{10}^1\{0,1\}$,$M_{11}^1\{0,1\}$, $M_{12}^1\{0,1\}$,$M_{15}^1\{0,1\}$, $M_{16}^1\{0,1\}$
$entitiesToGhost[2]$	$M_1^2\{1,2\}$	$M_3^2\{0,2\}$,$M_8^2\{0,2\}$, $M_2^2\{0,2\}$	$M_5^2\{0,1\}$,$M_6^2\{0,1\}$, $M_7^2\{0,1\}$
$entitiesToGhost[3]$		$M_4^2\{0,2\}$	$M_5^2\{1\}$

Extending the example of ghost collection on P_1 from §3.3.2

1. Part boundary vertices on P_1 collected for ghost creation are M_1^0 and M_9^0 where $ghostParts[M_1^0] = \{2\}$ and $ghostParts[M_9^0] = \{0\}$. The remote copy of M_1^0 exists on P_0 and M_9^0 exists on P_2.

2. Part boundary vertex collected on P_0 for ghost creation is M_1^0 with destination part as P_2.

3. At this point, P_0 does not know that P_1 is also sending M_1^0 to P_2 for ghost creation. P_1 also does not know that P_1 is sending M_1^0 to P_2.

4. P_0 (resp. P_1) sends a message to P_1 (resp. P_0) with the address of M_1^0 on P_1 (resp. P_0) and the destination part P_2 of M_1^0 on P_1 (resp. P_0) (Algorithm 5 Step 1).

5. P_1 receives the message from P_0 with the address of M_1^0 on P_1 and destination part idP_2 (Algorithm 5 Step 2).

6. Using the address of M_1^0 on P_1, P_1 checks, if P_2 is in the destination parts of M_1^0 (Algorithm 5 Step 3). As it is there and the part id of P_0 is less than that of P_1, P_1 removes M_1^0 from $entitiesToGhost[0]$ (Algorithm 5 Step 4).

7. P_0 ignores the message from P_1 as the part id of P_0 is less than P_1 (Algorithm 5 Step 4).

3.3.4 Step 4: Exchange entities and update ghost copies

Since an entity of dimension $d > 0$ is bounded by lower dimension entities, mesh entities are exchanged from lower to higher dimension. Step 4 exchanges entities from dimension 0 to 3, and creates entities on the destination parts. Algorithm 6 is the pseudo-code that exchanges the entities contained in $entitiesToGhost[d]$. After creating ghosts on the destination parts, an owner entity and owning part id are assigned to ghosts. The owning entities also update their ghost copies based on the new ghosts created.

- Step 4.1 sends the message to destination parts to create new ghost entities. For each M_i^d collected for ghost creation, P_{local} sends a message composed of the address of M_i^d on P_{local} and the information of M_i^d necessary for entity creation, which consists of the following

  - unique vertex id (if vertex)
  - Owner entity information (to notify the owner about the ghost entity created)
  - Owner part information
  - Geometric classification information (so that the created ghost entity also has the same geometric classification)
  - Shape information

```
Data: $M$, d, entitiesToGhost, g, numLayer
Result: Exchange ghost entities
begin
    /* Step 4.1: Send message with ghost information                    */
    foreach $M_i^d \in entitiesToGhost[d]$ do
        foreach part ID $p \in ghostParts[M_i^d]$ do
            /* For vertices, pack remote copy information               */
            if $d == 0$ then
                pack $\mathscr{R}[M_i^d]$ information;
            end if
            send message A(address of $M_i^d$ on $P_{local}$, owner of $M_i^d$, information of
            $M_i^d$) to $p$;
        end foreach
    end foreach
    /* Step 4.2: Receive message with ghost information                 */
    while $p$ receives message A(address of $M_i^d$, owner of $M_i^d$, information of
    $M_i^d$) from $P_{local}$ do
        create $M_k^d$ with information of $M_i^d$;
        $P_{own} \leftarrow$ owner of $M_i^d$;
        /* Step 4.3:Bounce vertex & ghost information to its owner      */
        if $d == 0$ then
            get remote copy information from A;
            foreach part id $P_i \in \mathscr{R}[M_k^d]$ do
                send message B(address of $M_k^d$ on $p$, address of $M_k^d$ on $P_i$) to $P_i$;
            end foreach
        end if
        if $d == g$ then
            send message B(address of $M_k^d$ on $p$, address of $M_k^d$ on $P_{own}$) to $P_{own}$;
        end if
    end while
    /* Step 4.4: update ghost copy information                          */
    while $P_i$ receives message B(address of $M_k^d$ on p, address of $M_k^d$ on $P_i$) from
    $p$ do
        $M_k^d \leftarrow$ entity located in the address of $M_k^d$ on $P_i$;
        if $d == 0$ then
            /* check if $P_i$ is a neighbor of $p$                      */
            foreach $M_i^g$ where $M_k^d \in \partial(M_i^g)$ do
                if $p \in ghostParts[M_i^g]$ then
                    save the address of $M_k^d$ on $P_i$ as for ghost copy on $P_i$;
                end if
            end foreach
        end if
    end while
end
```

Algorithm 6: exchangeGhostEnts(M, g, d, entitiesToGhost(d))

  - Remote copy information (if vertex since vertex information is bounced back to all its remote copies.)
  - Vertices information (if non-vertex)

  For the example given in §3.3.3, to create the vertex M_1^0 on P_2, P_0 sends a message composed of the address of M_1^0 on P_0 and information of M_1^0 including its remote copy information stored on P_0 (i.e. the address of M_1^0 on P_1 and P_1).

  For non-vertices, the message does not carry any remote copy information. Every non-vertex entity (edge, face or region) carries along the vertex information as it is required to create a new non-vertex entity on the destination part. For example, when M_2^2 is sent to P_2, it takes along address of M_2^2 on P_1 (owner entity information), P_1 (owner part information), geometric classification, shape information (face), address of M_1^0, M_2^0 and M_5^0 (vertices) on the destination part P_2. Sender part P_1 should know the address of vertices on the destination part P_2 that is why whenever a new vertex is created, its information is bounced back to all its remote copies.

- Step 4.2 creates a new entity M_i^d on P_i for the message A received on P_i (sent in Step 4.1). The newly created ghost entity updates information about its owner entity and the part where owner entity exists. This information is extracted from the message received.

  For example, M_2^2 is created on P_2 using the shape information, vertices information and geometric classification from the message A. P_2 then adds the address of M_2^2 on P_0 as its owner entity and sets P_0 as the owning part id of M_2^2.

- Step 4.3 sends back the information of the newly created ghost entity to its owner part. Bouncing back ghost information is only required for vertices and ghost objects. If the new entity created is a vertex, its address should be sent back to all its remote copies that were packed in the sender's message for updating the communication links as this information is required by the sender parts to create higher order ghost entities.

For example, when M_1^0 is created on P_2, P_2 bounces the information of M_1^0 to P_0 and P_1. When P_1 (resp. P_0) send higher order ghost entities to P_2 like M_2^2 (resp. M_1^2), it packs the address of vertex M_1^0 on P_2 in the message. This avoids mesh-level global search on the P_2 whenever a non-vertex ghost entity is created. For example, when M_2^2 is created on P_2, it brings along the vertex address of M_1^0, M_5^0 and M_2^0 on P_2 in message A.

After creating the ghost object M_i^g, its information is also sent back to its owner so that the owner entity can update its ghost information for instance, after M_2^2 is created on P_2, it bounces back the address of M_2^2 on P_2 to the part having its owner entity i.e. P_1.

- In step 4.4, the owner entities of ghost objects and remote copies of ghost vertices receive back the ghost information and update their ghost copies. For example, after 2-layer ghosting process, M_2^2 on P_1 will have the entries [(address of M_2^2 on P_0, P_0), (address of M_2^2 on P_2, P_2)] as its ghost information. The vertex M_1^0 P_1 will have the ghost information of [address of M_1^0 on P_2, P_2]

3.3.5 Store ghost rule

Ghost information is stored in the part so that the ghosting that becomes outdated due to mesh modification can be restored when the mesh is synchronized. Step 5 stores the ghost rule composed of $[g, b, includeCopy, numLayer]$ in the part.

Every part in the example mesh of Figure 3.4 will store the ghost information $[2, 0, 1, 2]$.

3.4 N-layer ghost deletion algorithm

The N-layer ghost deletion algorithm removes ghost entities from a ghosted mesh on a part starting from the part boundary (inner-most) layer. To support a complete N-layer ghost entity deletion algorithm, the following steps are required

- Get part boundary entities of bridge dimension. For every bridge entity M_i^b, get its upward adjacent entity of ghost dimension M_k^g and collect ghost entities for deletion in a vector $entitiesToRemove[d]$ where $d = 0, 1, 2, 3$.
- If the number of layers is more than 1, collect the next layer ghost entities for deletion.
- If the number of layers is more than 1, mark every visited bridge entity.
- Remove the ghost entities collected in the vector $entitiesToRemove$.

Algorithm 7 is the pseudo code of the N-layer ghost deletion algorithm.

3.4.1 Step 1: Process first-layer ghosts

The input of the ghost deletion algorithm is the ghosted mesh. Every part stores its ghost rules i.e. the ghost dimension, bridge dimension and the number of layers. For the example given in Figure 3.3, P_0, P_1 and P_2 store the ghost rule $[g = 2, b = 0, numLayer = 2]$. The ghost deletion algorithm uses this rule to collect ghost entities for deletion. The ghost deletion algorithm clears all ghost information for entities of ghost dimension g adjacent to part boundary bridge entities of dimension b. If an entity of dimension g, M_i^g is a ghost, it is collected for deletion in the vector $entitiesToRemove[g]$. All the downward adjacent entities of M_i^g are also

```
Data: a ghosted mesh $M$
Result: Delete all ghost entities in the mesh
begin
    get numLayer, g, b from M;
    foreach $M_i^b \in \{\partial(P_{local})\}$ do
        /* If the bridge is visited, proceed to the next one.        */
        if $M_i^b \leftarrow visited = true$ then
            continue;
        end if
        set $M_i^b \leftarrow visited = true$;
        /* Step 1: Process entities of dimension $g$ adjacent to part
           boundary bridges. Clear ghosting information of all such
           entities. If it is a ghost, collect it for deletion      */
        foreach $M_j^g \in \{M_i^b\{M\}_g\}$ do
            clear ghost copies of $M_j^g$;
            if $M_j^g \leftarrow isGhost$ then
                $entitiesToRemove[g] \leftarrow M_j^g$;
                collectDeleteInfo($M_j^g$, $g$, $entitiesToRemove$);
            end if
            /* Step 2: If numLayer>1, delete adjacent ghost entities
               */
            for $lyr \leftarrow 2$ to $numLayer$ do
                foreach $M_k^b \in \{M_j^g\{M\}_b\}$ do
                    if $M_k^b \leftarrow visited = true$ then
                        continue;
                    end if
                    foreach $M_l^g \in \{M_k^b\{M\}_g\}$ do
                        clear ghost copies of $M_l^g$;
                        if $M_l^g \leftarrow isGhost$ then
                            collectDeleteInfo($M_l^g$, $g$, $entitiesToRemove$);
                        end if
                        /* Step 3: Mark visited bridge entities         */
                        set $M_k^b \leftarrow visited = true$;
                    end foreach
                end foreach
            end for
        end foreach
    end foreach
    /* Step 4: Delete the collected ghost entities                     */
    for $d \leftarrow g + numLayer$ to $0$ do
        Remove $M_k^d \in entitiesToRemove[d]$;
    end for
end
```

Algorithm 7: deleteGhostEnts(M)

collected for deletion. Algorithm 8 is the pseudo code for collecting downward adjacent entities of M_i^g in the vector *entitiesToRemove*[d]. In the example given

```
Data: ghost entity $M_j^g$, $entitiesToRemove$
Result: Collects downward adjacent entities of $M_j^g$ for deletion
for $d \leftarrow 0$ to $g$ do
    foreach $M_k^d \in \{M_j^g\{M\}_d\}$ do
        clear ghost copies of $M_k^d$;
        $entitiesToRemove[d] \leftarrow M_k^d$;
    end foreach
end for
```

Algorithm 8: collectDeleteInfo(M_j^g, g, *entitiesToRemove*)

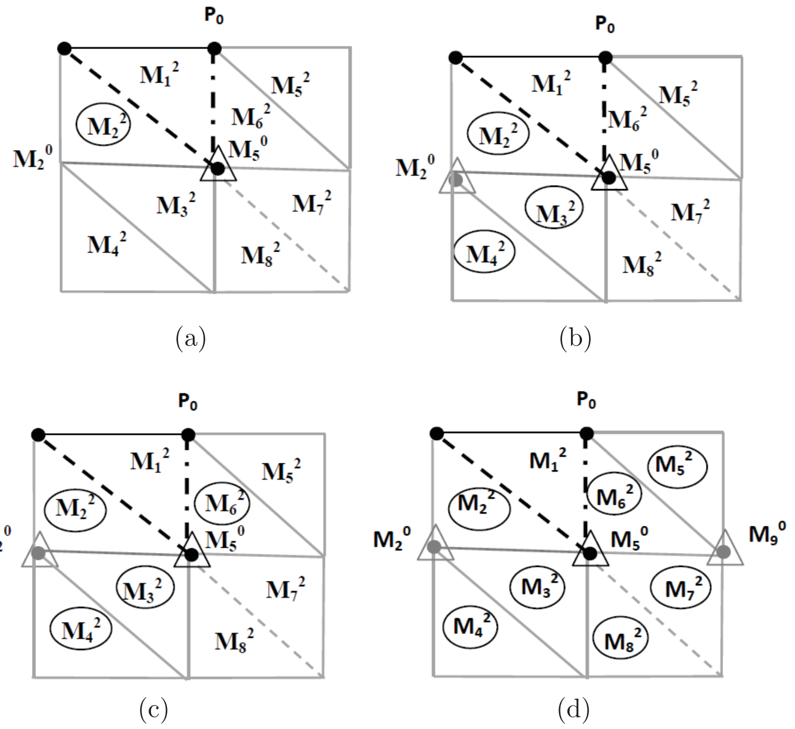


Figure 3.5: Ghost deletion algorithm applied on Part 0 of Figure 3.3

in Figure 3.5, the visited vertices are surrounded by triangles and visited ghost entities are surrounded by ovals. Figure 3.5 shows ghost entity deletion algorithm on part P_0 of the ghosted mesh given in Figure 3.4(d).

3.4.2 Step 2: Process next layer ghost entities

Once a ghost entity, M_i^g is collected for deletion, the algorithm visits entities of dimension g in the second layer that share a bridge entity with M_i^g (Step 2).

3.4.3 Step 3: Mark visited bridge entities

If $numLayer > 1$, we need to keep track of the visited entities to avoid processing the same ghost entity again. For that visited tags are set for entities that have been processed.

3.4.4 Step 4: Delete ghosts collected

Step 4 removes the ghost entities collected in Step 1 and 2. As for the opposite direction of entity creation, entities are removed from higher to lower dimension (Removing lower dimension entities first may create invalid adjacency information for higher order entities).

The ghost deletion algorithm works on part P_0 as follows (See Figure 3.5).

- The part boundary bridge vertices for P_0 are M_1^0, M_4^0, M_5^0. The ghost entity deletion algorithm first visits M_5^0. It then gets the upward adjacent entities of M_5^0 having dimension g i.e. $M_1^2, M_2^2, M_3^2, M_6^2, M_7^2$ and M_8^2. All these entities except M_1^2 are ghosts. The deletion algorithm clears the ghost information of M_1^2, then it proceeds with M_2^2. As M_2^2 is a ghost, it adds it in $entitiesToRemove$ (Algorithm 7 Step 1, Figure 3.5 (a)).

- The bridge entities of M_2^2 are M_1^0, M_2^0, M_5^0. M_5^0 is visited so the algorithm proceeds with M_2^0. M_2^0 has M_3^2 and M_4^2 adjacent to it which are collected for deletion (Algorithm 7 Step 2, Figure 3.5 (b)).

- The processed bridge entity M_2^0 is marked visited (Algorithm 7 Step 3). The next ghost entity adjacent to M_5^0 is M_6^2 which is also collected for deletion

(Figure 3.5 (c)). It shares common vertices M_4^0, M_8^0 with M_5^2, M_7^2, M_8^2 which will also be collected. By now, all ghost entities are collected for deletion (Figure 3.5 (d)).

- *entitiesToRemove* now has the contents given in Table 3.4.

Table 3.4: Contents of vector *entitiesToRemove* after Step 3

	P_1
entitiesToRemove[0]	$M_2^0, M_3^0, M_6^0, M_7^0, M_8^0, M_9^0$
entitiesToRemove[1]	$M_1^1, M_4^1, M_5^1, M_6^1, M_7^1, M_9^1, M_{10}^1, M_{11}^1, M_{12}^1, M_{13}^1, M_{14}^1, M_{15}^1, M_{16}^1$
entitiesToRemove[2]	$M_2^2, M_3^2, M_4^2, M_5^2, M_6^2, M_7^2, M_8^2$

3.5 Ghosting Tests

The tests for ghost creation/deletion algorithm are aimed to verify correctness of the ghosting algorithm. A detailed mesh verification algorithm that verifies the complete ghosted mesh and its adjacencies is presented in Appendix B. Once ghosts are created on parts, the basic steps for verifying the correctness of ghost creation algorithm are

1. Extract ghost dimension g, bridge dimension b and number of layers $numLayer$ from the part.

2. For every non-ghost entity M_i^g of dimension g adjacent to a bridge entity of dimension b, M_k^b, if M_k^b has remote copy on part p but M_i^g does not have a remote copy on p, test the following

    - Verify ghost information of M_i^g if it has the address of ghost copy on part p.

    - Exchange a message with part p to verify if the ghost copy of M_i^g actually exists there.

For the example in Figure 3.4 (d), on part P_1, the bridge entity M_1^0 is adjacent to M_2^2, a non-ghost entity of dimension g. M_1^0 has remote copies on P_0 but M_2^2 has no remote copy on P_0. In this case, the test program verifies the following for M_2^2

- M_2^2 should have its ghost copy information in the form [address of M_2^2 on P_1, P_1].
- A message composed of the address of M_2^2 on P_1 is sent to P_1. If M_2^2 exists as a ghost copy on P_1, a message with its address is sent back to P_1.

3.6 Summary

The steps required for a ghost creation algorithm involves various communication rounds. Algorithm 5 requires neighborhood communication through IPComMan[38] to eliminate duplicate entities. Step 4.1 of Algorithm 6 also employs neighborhood communication to create ghost copies. Step 4.3 and 4.4 of Algorithm 6 employs two rounds of communication. The first round in which the vertices are bounced back requires IPComMan all-to-all communication. It employs communication between neighbors of neighbors. This is because the vertex information is bounced back from the destination ghost part to the vertex's remote copies at the sender part (§3.3.4) and in some cases the destination ghost part may not be neighbors to vertex's remote copy parts at the sender side. Second round where ghost objects are bounced to their owners also requires neighborhood communication. The ghost creation algorithm heavily relies on the neighborhood concept. The more the neighbors of the bridge entities, the more ghost objects will be created. Chapter 4 discusses how the change in number of neighbors affects the performance results of ghost creation.

The ghost deletion algorithm does not employ any form of communication in the ghost collection process. It searches for ghost entities in a way similar to ghost entity collection step during ghost creation (§3.3.1). A detailed discussion on the performance of ghost creation and deletion algorithm is given in Chapter 4.

CHAPTER 4
Performance Results

This chapter presents performance results of the parallel N-layer ghost creation and deletion algorithm on two massively parallel architectures. The ghost creation and deletion algorithm is applied to two example meshes to measure its performance. The first example consider a mesh size field that represents a planar shock on cube geometry (CUBE). The second example consists of mesh size field that represents the motion of air bubbles in a steady uniform flow (BUBBLE). Figure 4.1 shows the CUBE, a uniform mesh of 17 million tetra-hedra. Figure 4.2 represents a BUBBLE mesh, involving movement of five air bubbles by a distance of $1/5^{th}$ of their radius.

Tests were executed on IBM Blue Gene/L[8] and Cray XE6 [78] systems. Table 4.1 presents a high-level comparison of the two architecture's configurations. The differences in processor speed, cache speed, memory access, peak performance and network latency account for the performance speedup of Cray XE6 over BG/L in the results of ghost creation and deletion algorithm presented in this chapter. A detailed discussion on the Blue Gene/L and Cray XE6 architectures is given in [8, 7].

The organization of the chapter is as follows. Section 4.1 presents a strong scaling study of 1-layer ghost creation and deletion algorithm on a 17M cube and 165M bubble mesh (1K-32K processors). Section 4.2 presents a weak scaling study of 1-layer ghost creation and deletion algorithm starting from 32 cores till 16,384 cores. Section 4.3 gives the performance results of the N-layer ghost creation algorithm on 136M cube for a fixed processor count (1K cores).

Table 4.1: Blue Gene/L vs. Cray XE6

	Blue Gene/L	**Cray XE6**
Processor Speed	700 MHz	2.1 GHz
Core count	32,000	153,216
Core per node	2	24
Cache	3 cache levels L1 (32KB per proc.), L2 (2KB per proc. with prefetch buffer) and L3 (4MB embedded DRAM)	3 cache levels L1(64KB per proc.), L2(512 KB per proc.), L3(6MB shared between 6 cores)
Memory per node	512 or 1024 MB double data rate (DDR) dynamic random access memory (DRAM) per node at 350 MHz.	32 GB DDR3 1333 MHz memory per node (for 6008 nodes), 64 GB DDR3 1333 MHz memory per node (for 384 nodes)
Peak Performance per compute node	5.6 GFLOPS	201.6 GFLOPS
Networks	3D torus with 175 MBps per direction, Global collective with 350 MBps, global barrier and interrupt, JTAG and Gigabit Ethernet	3D torus with 168 GB/sec routing capacity and 9.8 GB/sec per custom Gemini chip, scales to 100,000 network endpoints.
Network Latency	1 μs	1.5 μs (Global collective)

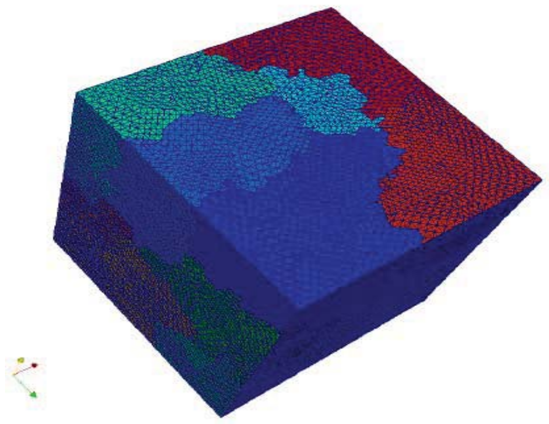


Figure 4.1: A cubic mesh

4.1 Strong scaling study of Ghost creation/deletion

In a strong scaling analysis, the overall problem size remains constant as the processor configuration increases. For the ghost creation and deletion algorithm,

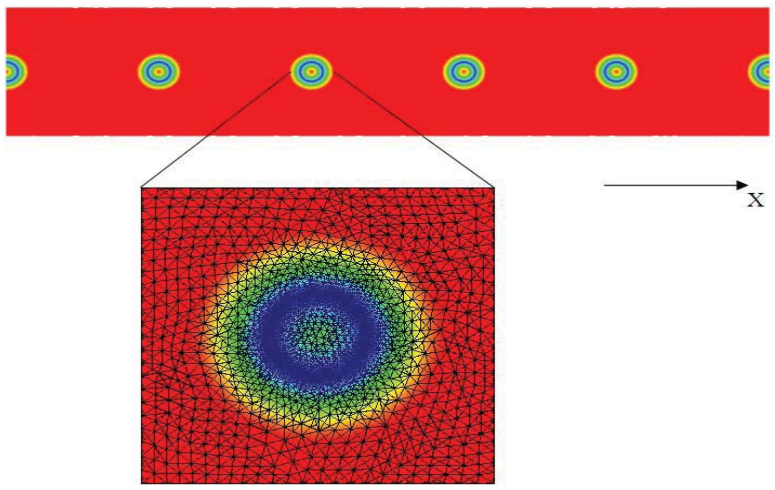


Figure 4.2: Moving air bubbles

strong scaling is influenced by the following factors

- The inter-process communication dominates computational workload per processor.
- The total number of ghost entities increase by a factor dependent on mesh partitioning when the processor count is doubled.

A detailed discussion on the above two factors is given here. The first factor is related to the time spent in computation and communication for ghosting. Completion time for a parallel program has two components: *computation time* and *communication time*. The *completion time* of the parallel application is the maximum of the completion times of all the tasks in the parallel application. The steps in ghost creation algorithm presented in §3.3 fall in two basic categories

- Computation steps: Step 1 (collect ghost entities in the first layer).
- Communication steps: Step 3 (eliminate duplicate entities), Step 4 (Exchange entities and update ghost copies).

Table 4.2: Computation time vs. communication time on Cray XE6 (17M cube)

		1024	2048	4096	8192	16384	32768
CUBE 17M	Comp. time(s)	0.30	0.16	0.10	0.05	0.03	0.021
	% of Total time	11%	8%	10.2%	10.6%	9.6%	10.5%
	Comm. time(s)	2.68	1.74	0.98	0.47	0.31	0.20
	% of Total time	89%	92.0%	89.8%	89.4%	90.4%	89.5%

Table 4.2 gives the ratio of computation time (§3.3 Step 1) vs. communication time (§3.3 Steps 3 and 4) in the ghost creation algorithm on a 17M CUBE test case. On average, the computation time accounts for 10% of the total time in ghost creation as a result of which the inter-process communication dominates the computational workload per process. For the ghost creation algorithm, the load is distributed on each processor such that it spends 10% of its time in computation and 90% in processing ghost creation messages and creating new ghost entities. In case of ghost creation and deletion algorithm, the computational work load per core is insufficient as compared to communication work load as a result of which inter-processor communication dominates which in turn affects strong scalability.

The second factor is related to graph partitioning and neighborhood. Good partitioning schemes (such as graph-based ones) not only balance the work load but also minimize the amount of communication required between parts (software libraries such as ParMETIS [79] and Zoltan [34] are commonly used). With increasing processor count, the total inter-part boundaries typically increase on a fixed size mesh. This increases the number of neighbors for each bridge entity due to which the problem size which is the total number of ghost entities (labeled as E/ghosted) also grows. For example, a bridge entity has 2 neighbors on four processors for a fixed size mesh. When the processor count doubles to 8 for the same fixed size mesh, inter-part boundaries will also increase proportionately but the graph partitioning and load balancing algorithm will try to keep adjacent entities on the same part to reduce inter-part communication. In an effort to keep adjacent entities on the same part, a good partitioning algorithm will try to minimize the increase in neighbors for part boundary entities. The increase in neighbors will typically lie between 1.3 and 1.8 when the processor count is doubled for a fixed size mesh. Thus, when the number of processors is doubled, the load distribution on each processor is not divided into half. As we will see in the upcoming results, these two factors influence strong scalability of the ghosting algorithm.

Scalability is based on the execution time on 1024 processors. For ghost creation, scalability is defined as

$$scalability = (n_{proc-base} * time_{base})/(n_{proc-test} * time_{test}) \tag{4.1}$$

To test the performance of ghost creation algorithm with increasing processor count, the CUBE and BUBBLE test cases were executed on 1024 to 32,768 processors on Cray XE6 and 1024 to 4096 processors on IBM Blue Gene/L. Table 4.3 demonstrates the execution time of ghost creation algorithm with 1-layer on input parameters $[g = 3, b = 0, numLayer = 1, includeCopy = 1]$ (ghost dimension: region, bridge dimension: vertex).

Table 4.3: 1-layer ghost creation time(sec) on Cray XE6 and BG/L

Test case	Machine	N/proc	1024	2048	4096	8192	16384	32768
CUBE 17M	Cray XE6	E/ghosted	11.1M	14.6M	19.1M	18.6M	25.1M	33.9M
		Time(s)	2.98	1.85	1.17	0.51	0.34	0.22
		Scaling	1	0.81	0.64	0.74	0.55	0.42
	BG/L	E/ghosted	11.1 M	14.6 M	19.1 M			
		Time(s)	29.42	17.6	12.6			
		Scaling	1	0.83	0.58			
BUBBLE 165M	Cray XE6	E/ghosted	43.2M	77.1M	94.5M	94.89M	123.7M	160M
		Time(s)	18.64	15.09	9.52	3.2	1.96	1.56
		Scaling	1	0.61	0.50	0.73	0.60	0.37
	BG/L	E/ghosted	43.2M	77.1M	94.5M			
		Time(s)	140.42	122.52	84.72			
		Scaling	1	0.57	0.41			

As it can be seen from Table 4.3, the scaling factor decreases as the number of ghost entities increase with increasing processor count. As the ghost creation and deletion process is dependent on neighborhood communication (See §3.2), the total number of ghost creation messages (resp. total ghost entities to delete) increase when the processor count doubles. Figure 4.3 shows the relationship of increasing processors number of ghost entities with scalability. As the total ghost entities increase, scaling factor keeps on decreasing proportionately.

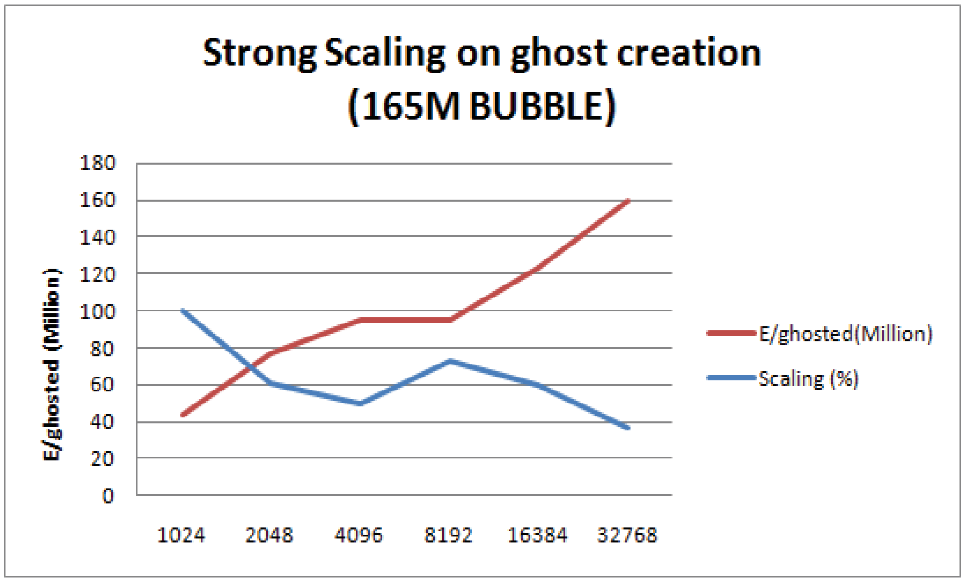


Figure 4.3: Relationship of ghosted entities created with scalability (165M mesh)

For ghost entity deletion algorithm, there is no inter-part communication involved in collecting the ghost entities to be deleted (See §3.4). Factor 2 has no influence on the scalability of ghost deletion. However, the total number of ghost entities to be deleted increase with increasing processor count so factor 1 is present. The scalability for ghost deletion algorithm is based on the execution time on 1024 processors. It can be determined using

$$scalability = (time_{base} * n_{proc-base})/(time_{test} * n_{proc-test}) \tag{4.2}$$

Table 4.4: 1-layer ghost deletion time(sec) on Cray XE6 and BG/L

Test case	Machine	N/proc	1024	2048	4096	8192	16384	32768
CUBE 17M	Cray XE6	E/deleted	11.1M	14.6M	19.1M	18.6M	25.1M	33.9M
		Time(s)	1.05	0.62	0.38	0.19	0.12	0.08
		Scaling	1	0.85	0.69	0.69	0.55	0.41
	BG/L	E/deleted	11.1 M	14.6 M	19.1 M			
		Time(s)	10.54	6.81	4.07			
		Scaling	1	0.77	0.65			
BUBBLE 165M	Cray XE6	E/deleted	43.2M	77.1M	94.5M	94.89M	123.7M	160M
		Time(s)	4.83	4.28	2.84	0.99	0.75	0.5
		Scaling	1	0.57	0.43	0.70	0.55	0.41
	BG/L	E/deleted	43.2M	77.1M	94.5M			
		Time(s)	48.98	43.39	21.58			
		Scaling	1	0.56	0.56			

Figure 4.4 compares scalability with ghost entities deleted. As the number of cores get doubled, ghost entities to be deleted also increase which brings down the scaling factor.

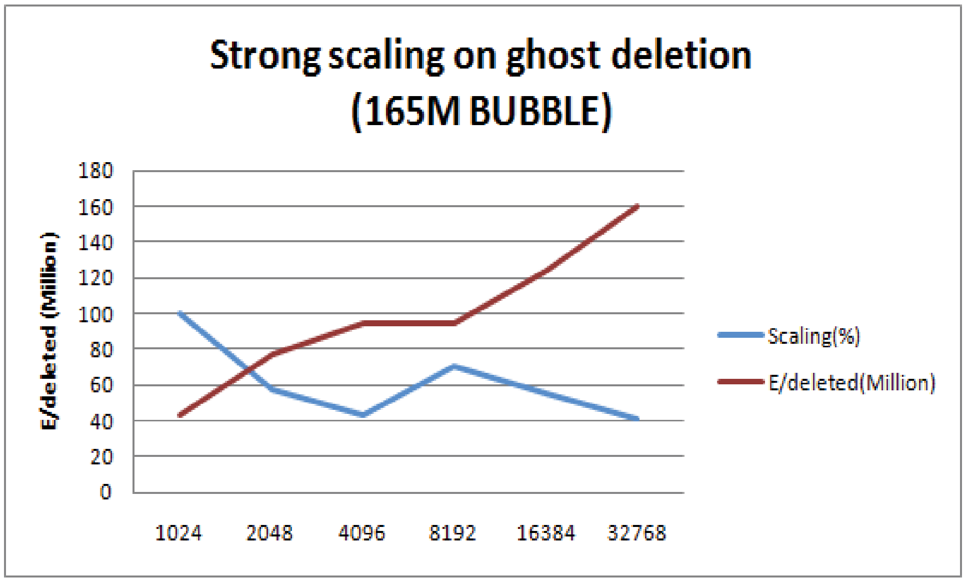


Figure 4.4: Relationship of ghosted entities deleted with scalability (165M mesh)

4.2 Weak scaling study of Ghost creation/deletion

In a weak scalability analysis, the problem size scales with the processor configuration so that the computational load per processor stays constant. In case of ghost creation, the communication load is on average 9 times more than the computational load. Therefore, weak scalability is influenced by this factor. Table 4.5 presents the weak scaling results of the ghost creation and deletion algorithm on input type $[g = 3, b = 0, includeCopy = 0]$ i.e. ghost regions, bridge vertices (remote copies of bridge are not included). The test series increases the number of processors from 32 to 16K in multiples of 8 on the Cray XE6 machine. The mesh size also varies in multiples of 8 (app.) along with processor count. S_G represents scaling based on average number of ghost entities per processor on 1024 processors.

$$S_G = \text{Avg. no. of ghosts on } n_{base} / \text{Avg. no. of ghosts on } n_{test} \tag{4.3}$$

S_{t_create} and S_{t_del} represent scalability of ghost creation and deletion algorithm based on the execution time on 1024 processors. Scalability by execution time can be calculated using Equation 4.4.

$$S_t = time_{base}/time_{test} \tag{4.4}$$

Table 4.5: Weak scaling of ghost creation algorithm

N/Proc	Mesh Size	t_{create}(s)	t_{del}(s)	Avg. ghosts/Proc	S_G	S_{t_create}	S_{t_del}
32	265,872	0.85	0.33	2,547	1	1	1
256	2,126,662	0.95	0.35	2,889	0.88	0.89	0.94
2048	17,010,572	1.24	0.51	3,266	0.78	0.69	0.69
16384	136,084,576	1.23	0.41	3,370	0.76	0.69	0.80

The average number of ghost entities in Table 4.5 varies as the number of ghosts are dependent on entity neighborhood which can be different in every case. Table 4.5 shows a jump of 0.29 seconds (resp. 0.16 seconds) in t_{create} (resp. t_{del}) between

256 and 2048 processors. There are two reasons for this jump. First, the time spent in ghost creation is measured by taking the maximum time spent in ghost creation among all processors. When a part has more ghost creation messages to process, it will likely take more time. Secondly, the ghost creation algorithm currently assign more messages to processor counts with lower ranks (See Algorithm 5 Step 4) as the processor with minimum part id sends more ghost creation messages. This approach will likely be replaced by a round-robin algorithm in the future to avoid any load imbalance of messages.

4.3 N-layer ghost creation with fixed processor count

To test the performance of N-layer ghost creation algorithm with different number of layers, the CUBE test case (136M mesh) was executed on 1024 processors on Cray XE6 and IBM Blue Gene/L with number of layers from 1 to 5. Table 4.6 demonstrates the execution time of N-layer ghost creation algorithm in seconds with input parameters $[g = 3, b = 1, numLyr = 1...5, includeCopy = 1]$ (ghost dimension: region, bridge: edges). Table 4.6 shows how the number of ghosted entities increase with increasing layer count by an amount comparable to the number of entities in the first layer. In Table 4.6, as the number of layer increases to 5, the number of ghost entities reach close to the size of the mesh. The efficiency of N-layer ghost creation algorithm denoted as E_n is based on the execution time on 1024 processors and the increase in total number of ghost entities after adding ghost layers. It is calculated using

$$E_n = \frac{(time_{base} * (1 + \uparrow \text{ in E/ghosted}))}{(time_{test})} \tag{4.5}$$

Visited tag addition, removal and checking for an existing tag are O(1) operations (See §3.2). The first layer ghost creation does not require any tag adding/removal

Table 4.6: N-layer ghost creation execution time(s) on 1024 processors

Test case	Machine		n=1	n=2	n=3	n=4	n=5
CUBE 136M	Cray XE6	E/ghosted	26.5M	55.3M	77.8M	95.8M	110M
		Time(s)	9.77	23.7	35.6	45.3	53.4
		E_n	1	0.86	0.81	0.79	0.78
	BG/L	E/ghosted	26.5 M	55.3 M	77.8 M	95.8 M	110 M
		Time(s)	64.2	167.02	235.76	284.79	322.16
		E_n	1	.81	0.80	0.81	0.81

overhead. That is why processing first layer ghost entities is most efficient (See Table 4.6). As the number of layers increase, tag processing overhead increases. This also affects the scalability of the N-layer ghost creation algorithm.

CHAPTER 5
Closing Remarks

The aim of the thesis was to develop an efficient parallel algorithm of ghost creation and deletion for FMDB that localizes non-part boundary data from remote parts for computation purposes and minimizes inter-part communication. Ghost is a useful algorithm that provides a third-part application with the complete parallel neighborhood information. The performance of the algorithms was evaluated on massively parallel architectures (Blue Gene/L and Cray XE6) up to 32,768 cores. Apart from increasing core count, two other factors that influenced the performance results were inter-part communication and variation in problem size with increasing core count.

The following are the key features of the procedures developed as part of the thesis :

- The ghost creation algorithm was developed that creates 1D, 2D or 3D ghost objects in a mesh using bridge entities.
- The ghost creation algorithm can create ghosts up-till N number of layers. When the number of layers reaches the size of the mesh, the whole mesh can be ghosted.
- The ghost deletion algorithm was developed which deletes all ghosts in a mesh.
- Mesh verification algorithm was extended so that it can verify a ghosted mesh (Appendix B).
- Weak and strong scaling analysis of the ghost creation and deletion algorithm was carried out.

The possible extensions that can be made to the ghost creation and deletion algorithm include:

- *Support for Multiple Parts per Process*: FMDB currently provides the support for single part per process. The upcoming versions shall provide the support for multiple parts per process. The ghost creation and deletion algorithm can be extended to support multiple parts per process.

- *Applications of ghosting*: Ghosting can be applied in certain applications related to mesh modification, refinement and coarsening as it is useful to know which off-part data is neighbor to local part-boundary data. Any third-party application can use ghosting to get neighborhood data information.

- *Account for increasing core count per node*: Ghosting algorithm is currently designed for distributed memory applications using message passing. It can be extended to exploit both shared and distributed memory architectures i.e a hybrid approach of both distributed memory paradigms (MPI) and shared memory paradigms including POSIX threads[4] and OpenMP [80]. In a hybrid approach, message passing is used for inter-node communication whereas shared memory and multi-threading is used for intra-node communication. This approach can be used to account for today's fast growing computing needs i.e. the increasing number of cores per node.

BIBLIOGRAPHY

[1] C. Johnson. *Numerical solution of partial differential equations by the finite element method*, volume 32. Cambridge university press New York, 1987.

[2] M.S. Shephard. Update to: Approaches to the automatic generation and control of finite element meshes. *Applied Mechanics Reviews*, 49:5–16, 1996.

[3] M. Zhou, T. Xie, S. Seol, M.S. Shephard, O. Sahni, and K.E. Jansen. Tools to support Mesh Adaptation on Massively Parallel Computers. *To appear in: Engineering with Computers*, 2011.

[4] I.T. Foster. *Designing and building parallel programs: concepts and tools for parallel software engineering.* Addison-Wesley, 1995.

[5] World's top 500 supercomputers. `http://www.top500.org/`, 2011. Date last accessed: April 20, 2011.

[6] R. Espinha, W. Celes, N. Rodriguez, and G.H. Paulino. ParTopS: compact topological framework for parallel fragmentation simulations. *Engineering with Computers*, 25(4):345–365, 2009.

[7] Hopper 2, Cray XE6 at NERSC. `http://newweb.nersc.gov/users/computational-systems/hopper`, 2011. Date last accessed: April 20, 2011.

[8] IBM Guide to using Blue Gene/L. `http://www.redbooks.ibm.com/abstracts/sg246686.html`, 2007. Date last accessed: April 20, 2011.

[9] M.S. Shephard and S. Seol. Flexible distributed mesh data structure for parallel adaptive analysis. 2007.

[10] M.W. Beall and M.S. Shephard. A geometry-based analysis framework. In *Adv. in Comp. Engg. Science*, pages 557–562, Forsyth, GA, 1996. Atluri, S.N. and Yagawa, G. eds., Tech Science Press.

[11] 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.

[12] iGeom Interface Documentation. `www.itaps.org/software/specifications/iGeom-v0.8.h`, 2011. Date last accessed: April 20, 2011.

[13] M.S. Shephard. The specification of physical attribute information for engineering analysis. *Engineering with Computers*, 4(3):145–155, 1988.

[14] R.M. O'Bara, M.W. Beall, and M.S. Shephard. Analysis model visualization and graphical analysis attribute specification system. *Finite elements in analysis and design*, 19(4):325–348, 1995.

[15] M.S. Shephard, S. Dey, and J.E. Flaherty. A straightforward structure to construct shape functions for variable p-order meshes. *Computer Methods in Applied Mechanics and Engineering*, 147(3-4):209–233, 1997.

[16] 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.

[17] O.C. Zienkiewicz and R.L. Taylor. *The finite element method: basic formulation and linear problems*. McGraw-Hill College, 1989.

[18] M.W. Beall and M.S. Shephard. Mesh data structures for advanced finite element applications. *SCOREC Report*, pages 23–1995.

[19] ITAPS: The Interoperable Technologies for Advanced Petascale Simulations center. `http://www.itaps.org`, 2011. Date last accessed: April 20, 2011.

[20] iMesh Interface Documentation. `http://www.itaps.org/software/iMesh_html/index.html`, 2011. Date last accessed: April 20, 2011.

[21] C. Ollivier-Gooch, L. Diachin, M.S. Shephard, T. Tautges, J. Kraftcheck, V. Leung, X. Luo, and M. Miller. An interoperable, data-structure-neutral component for mesh query and manipulation. *ACM Transactions on Mathematical Software (TOMS)*, 37(3):1–28, 2010.

[22] K.D. Devine, L. Diachin, J. Kraftcheck, K.E. Jansen, V. Leung, X. Luo, M. Miller, C. Ollivier-Gooch, A. Ovcharenko, O. Sahni, et al. Interoperable mesh components for large-scale, distributed-memory simulations. In *Journal of Physics: Conference Series*, volume 180, page 012011. IOP Publishing, 2009.

[23] T.J. Tautges, C. Ernst, C. Stimpson, R.J. Meyers, and K. Merkley. MOAB: a mesh-oriented database. Technical report, Sandia National Laboratories, 2004.

[24] FMDB User's guide. `www.scorec.rpi.edu/FMDB/doc/FMDB.pdf`, 2011. Date last accessed: April 20, 2011.

[25] Symmetrix, Simulation modeling suite. `http://www.simmetrix.com`, 2011. Date last accessed: April 20, 2011.

[26] R.V. Garimella. Mesh data structure selection for mesh generation and FEA applications. *International journal for numerical methods in engineering*, 55(4):451–478, 2002.

[27] M.S. Shephard, J.E. Flaherty, C.L. Bottasso, H.L. de Cougny, C. Ozturan, and M.L. Simone. Parallel automatic adaptive analysis. *Parallel Computing*, 23(9):1327–1347, 1997.

[28] J.E. Flaherty, R.M. Loy, C. Ozturan, M.S. Shephard, B.K. Szymanski, J.D. Teresco, and L.H. Ziantz. Parallel structures and dynamic load balancing for adaptive finite element computation. *Applied Numerical Mathematics*, 26(1-2):241–263, 1998.

[29] J.F. Remacle, O. Klaas, J.E. Flaherty, and M.S. Shephard. Parallel algorithm oriented mesh database. *Engineering with Computers*, 18(3):274–284, 2002.

[30] 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.

[31] L. Oliker, R. Biswas, and H.N. Gabow. Parallel tetrahedral mesh adaptation with dynamic load balancing. *Parallel Computing*, 26(12):1583–1608, 2000.

[32] C. Ozturan, H.L. deCougny, M.S. Shephard, and J.E. Flaherty. Parallel adaptive mesh refinement and redistribution on distributed memory computers. *Computer Methods in Applied Mechanics and Engineering*, 119(1-2):123–137, 1994.

[33] J.D. Teresco, M.W. Beall, J.E. Flaherty, and M.S. Shephard. A hierarchical partition model for adaptive finite element computation. *Computer methods in applied mechanics and engineering*, 184(2-4):269–285, 2000.

[34] Zoltan, Parallel Partitioning, Load Balancing and Data-Management Services. `http://www.cs.sandia.gov/zoltan/`, 2011. Date last accessed: April 20, 2011.

[35] iMeshP Interface Documentation. `http://www.itaps.org/software/iMeshP_html/index.html`, 2011. Date last accessed: April 20, 2011.

[36] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*, volume 206. Addison-wesley Reading, MA, 1995.

[37] Argonne National Laboratory, The Message Passing Interface (MPI) standard library. `http://www-unix.mcs.anl.gov/mpi`, 2011. Date last accessed: April 20, 2011.

[38] A. Ovcharenko, O. Sahni, C.D. Carothers, K.E. Jansen, and M.S. Shephard. Subdomain communication to increase scalability in large-scale scientific applications. In *Proceedings of the 23rd international conference on Supercomputing*, pages 497–498. ACM, 2009.

[39] X. Li, M.S. Shephard, and M.W. Beall. 3D anisotropic mesh adaptation by mesh modification. *Computer methods in applied mechanics and engineering*, 194(48-49):4915–4950, 2005.

[40] X. Zeng, R. Bagrodia, and M. Gerla. GloMoSim: a library for parallel simulation of large-scale wireless networks. *ACM SIGSIM Simulation Digest*, 28(1):154–161, 1998.

[41] J.F. Thompson, B.K. Soni, and N.P. Weatherill. *Handbook of grid generation*. CRC, 1999.

[42] S.J. Owen. A survey of unstructured mesh generation technology. In *7th International Meshing Roundtable*, volume 3, pages 239–267. Citeseer, 1998.

[43] B.G. Larwood, N.P. Weatherill, O. Hassan, and K. Morgan. Domain decomposition approach for parallel unstructured mesh generation. *International journal for numerical methods in engineering*, 58(2):177–188, 2003.

[44] GRUMMP: Generation and Refinement of Unstructured, Mixed-element Meshes in Parallel. `http://tetra.mech.ubc.ca/GRUMMP`, 2011. Date last accessed: April 20, 2011.

[45] R. Said, N.P. Weatherill, K. Morgan, and N.A. Verhoeven. Distributed parallel Delaunay mesh generation. *Computer methods in applied mechanics and engineering*, 177(1-2):109–125, 1999.

[46] B.H.V. Topping and B. Cheng. Parallel and distributed adaptive quadrilateral mesh generation. *Computers & structures*, 73(1-5):519–536, 1999.

[47] J. Chen and V.E. Taylor. ParaPART: parallel mesh partitioning tool for distributed systems. *Concurrency: Practice and Experience*, 12(2-3):111–123, 2000.

[48] R. Diekmann, R. Preis, F. Schlimbach, and C. Walshaw. Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM. *Parallel Computing*, 26(12):1555–1581, 2000.

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

[50] U.V. Catalyurek, E.G. Boman, K.D. Devine, D. Bozdag, R.T. Heaphy, and L.A. Riesen. A repartitioning hypergraph model for dynamic load balancing. *Journal of Parallel and Distributed Computing*, 69(8):711–724, 2009.

[51] U.V. Catalyurek, E.G. Boman, K.D. Devine, D. Bozdag, R. Heaphy, et al. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *2007 IEEE International Parallel and Distributed Processing Symposium*, page 68. IEEE, 2007.

[52] D.S. Balsara and C.D. Norton. Highly parallel structured adaptive mesh refinement using parallel language-based approaches. *Parallel Computing*, 27(1-2):37–70, 2001.

[53] P. MacNeice, K.M. Olson, C. Mobarry, R. de Fainchtein, and C. Packer. PARAMESH: A parallel adaptive mesh refinement community toolkit. *Computer physics communications*, 126(3):330–354, 2000.

[54] H.L. De Cougny and M.S. Shephard. Parallel refinement and coarsening of tetrahedral meshes. *Int. J. Numer. Meth. Engng*, 46(7):1101–1125, 1999.

[55] libMesh: Parallel data structures for finite element computations. `http://www.cfdlab.ae.utexas.edu`, 2011. Date last accessed: April 20, 2011.

[56] Y.M. Park and O.J. Kwon. A parallel unstructured dynamic mesh adaptation algorithm for 3-D unsteady flows. *International journal for numerical methods in fluids*, 48(6):671–690, 2005.

[57] L.V. Kale and S. Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993.

[58] O.S. Lawlor, S. Chakravorty, T.L. Wilmarth, N. Choudhury, I. Dooley, G. Zheng, and L.V. Kalé. Parfum: A parallel framework for unstructured meshes for scalable dynamic physics applications. *Engineering with Computers*, 22(3):215–235, 2006.

[59] L.V. Kaléa, R. Haberb, J. Bootha, S. Thitea, and J. Palaniappanb. An efficient parallel implementation of the spacetime discontinuous galerkin method using CHARM++.

[60] J. Shewchuk. Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. *Applied Computational Geometry Towards Geometric Engineering*, pages 203–222, 1996.

[61] Triangle, A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator. `http://www.cs.cmu.edu/~quake/triangle.html`, 2011. Date last accessed: April 20, 2011.

[62] J.R. Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry*, 22(1-3):21–74, 2002.

[63] J.R. Stewart and H.C. Edwards. A framework approach for developing parallel adaptive multiphysics applications. *Finite elements in analysis and design*, 40(12):1599–1617, 2004.

[64] H.C. Edwards, A. Williams, G.D. Sjaardema, D.G. Baur, and W.K. Cochran. SIERRA Toolkit computational mech conceptual model. *Sandia National Laboratories SAND Series, SAND2010-1192*, 2010.

[65] C. Burstedde, O. Ghattas, M. Gurnis, T. Isaac, G. Stadler, T. Warburton, and L. Wilcox. Extreme-scale AMR. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE Computer Society, 2010.

[66] C. Burstedde, O. Ghattas, G. Stadler, T. Tu, and L.C. Wilcox. Towards adaptive mesh PDE simulations on petascale computers. *Proceedings of Teragrid*, 8, 2008.

[67] A. Vogler, S. Shelyag, M. Schussler, F. Cattaneo, T. Emonet, and T. Linde. Simulations of magneto-convection in the solar photosphere. *Astronomy and astrophysics*, 429(1):335–351, 2005.

[68] T. Gerhold. Overview of the hybrid RANS code TAU. *MEGAFLOW-Numerical Flow Simulation for Aircraft Design*, pages 81–92, 2005.

[69] K.S. Kim and PG Cizmas. Three-dimensional hybrid mesh generation for turbomachinery airfoils. *Journal of propulsion and power*, 18(3):536–543, 2002.

[70] J. Dreher and R. Grauer. Racoon: A parallel mesh-adaptive framework for hyperbolic conservation laws. *Parallel Computing*, 31(8-9):913–932, 2005.

[71] B. Nichols, D. Buttlar, and J.P. Farrell. *Pthreads programming.* O'Reilly Media, 1996.

[72] T. Wen, J. Su, P. Colella, K. Yelick, and N. Keen. An adaptive mesh refinement benchmark for modern parallel programming languages. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 40. ACM, 2007.

[73] MOAB. `http://trac.mcs.anl.gov/projects/ITAPS/wiki/MOAB`, 2011. Date last accessed: April 20, 2011.

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

[75] J.F. Remacle and M.S. Shephard. An algorithm oriented mesh database. *International Journal for Numerical Methods in Engineering*, 58(2):349–374, 2003.

[76] H.L. deCougny, K.D. Devine, J.E. Flaherty, R.M. Loy, C. Ozturan, and M.S. Shephard. Load balancing for the parallel adaptive solution of partial differential equations. *Applied Numerical Mathematics*, 16(1-2):157–182, 1994.

[77] B.S. Kirk, J.W. Peterson, R.H. Stogner, and G.F. Carey. : a C++ library for parallel adaptive mesh refinement/coarsening simulations. *Engineering with Computers*, 22(3):237–254, 2006.

[78] Cray XE6 system. `http://www.cray.com/Products/XE/CrayXE6System.aspx`, 2011. Date last accessed: April 20, 2011.

[79] ParMETIS - Parallel Graph Partitioning and Fill-reducing Matrix Ordering. `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview`, 2011. Date last accessed: April 20, 2011.

[80] The OpenMP API specification for Parallel Programming. `http://openmp.org/wp/`, 2011. Date last accessed: April 20, 2011.

APPENDIX A
FMDB Unit Tests

Unit testing refers to tests that verify the functionality of a specific section of the code, typically at the functional level. FMDB unit tests were carried out to test certain functionalities for example iterators, remote copies, ghost copies and ownership.

A.1 Iterator tests

Iterators are a generalization of pointers which are objects that point to other objects. An iterator is an object that allows a programmer to traverse through all elements of a collection, regardless of its specific implementation. If an iterator points to one element in a range, then it is possible to increment it so that it points to the next element. Various kinds of iterators are desirable for efficient mesh entity traversal with various conditions like entity dimension, entity topology, geometric classification.

Mesh databases can include entity containers and entity set containers. Both mesh entity containers and entity set containers use STL to store mesh entities. A Mesh entity container has an array of four STL containers of the same type for vertices, edges, faces and regions. An entity set container includes a STL container to store arbitrary mesh entities of any dimension. The use of model entity iterator that iterates over the mesh entities mapped to a geometric entity is also required. The solution was to design an iterator that is independent of the STL container type and the underlying data. The FMDB generic iterator is able to iterate over the mesh entities irrespective of the container type (mesh entity container, model entity container or entity set container).

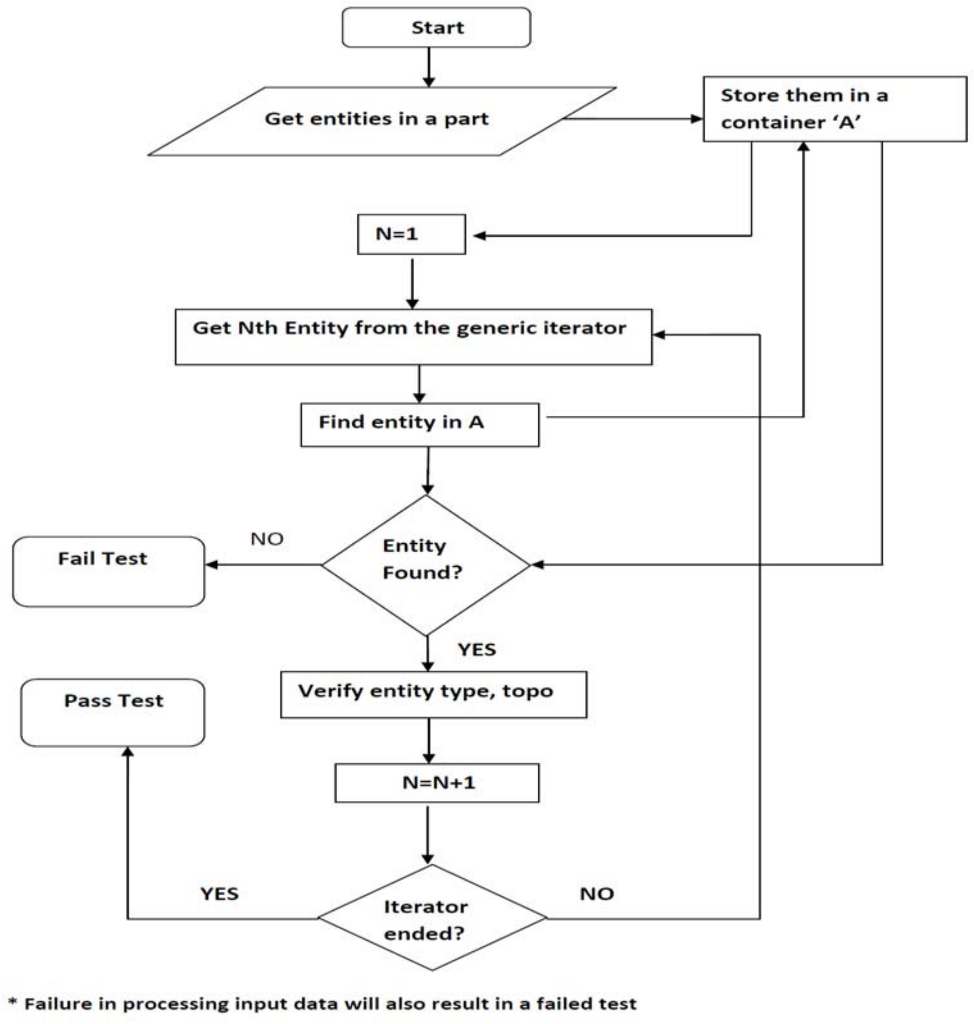

Start
Get entities in a part
Store them in a container 'A'
N=1
Get Nth Entity from the generic iterator
Find entity in A
Entity Found?
NO
Fail Test
YES
Verify entity type, topo
Pass Test
N=N+1
YES
Iterator ended?
NO
* Failure in processing input data will also result in a failed test

Figure A.1: Steps for testing generic iterator for mesh entities in a part

The unit tests for the generic iterator were aimed at testing it for all container types i.e. mesh entities in a part, mesh entities in an entity set and mesh entities mapped to geometric model entity. Figure A.1 presents a flowchart that highlights some of the basic steps for testing the generic iterator for mesh entities in a part. All entities in a mesh part are first stored in a STL container for verification purposes. The generic iterator then initiates over the mesh entities in a part. Each mesh entity returned by the generic iterator is searched in the STL container if it exists there. If the mesh entity exists, its shape information is extracted to verify it is a valid mesh entity. If the iterator does not return an entity which exists in the container, the unit test fails. If an iterator returns an entity which does not exist in the container, the unit test fails. Once the unit test gets all mesh entities returned by iterator, it passes.

Figure A.2 shows the unit testing of generic iterator for mesh entities mapped to geometric model entities (see Sec. 2.1 for details related to mesh entities and geometric entities). The unit test iterates over all geometric model entities. For every geometric model entity G_i^d it does the following

- It gets all mesh entities associated with the geometric model entity G_i^d.
- Generic iterator should return all mesh entities that are classified on G_i^d. For every mesh entity M_k^d that the iterator returns, the unit test performs the reverse classification operator to get back its geometric model entity.
- If the geometric model entity returned by the reverse classification operator is same as G_i^d, the unit test proceeds with next entity otherwise it fails.

A.2 Ownership and remote copy tests

A part-boundary mesh entity can be copied on several parts with one part being its owner (for details, see Sec. 2.4). A remote copy is the memory location of a mesh entity duplicated on another part. Each mesh entity maintains a reference of all other remote copies in a remote copy container. The remote copies of an entity are subject to change after the migration procedure. FMDB associates a remote copy container with every entity which gets updated every time the remote copy information is modified. In order to query and update remote copy information of a mesh entity, FMDB provides several operations like getting all remote copies of an

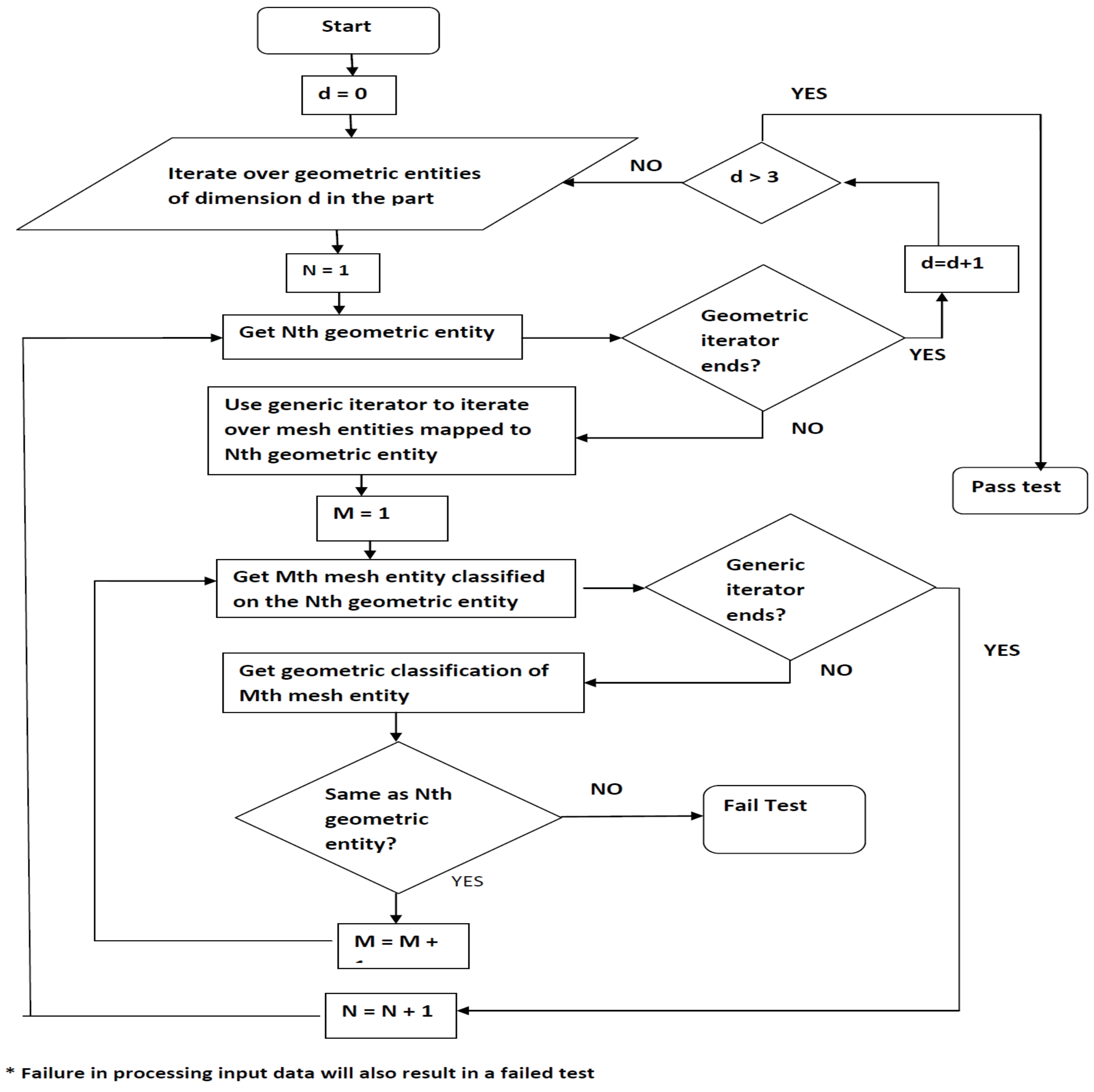

Start
d = 0
Iterate over geometric entities of dimension d in the part
NO
d > 3
YES
N = 1
d=d+1
Get Nth geometric entity
Geometric iterator ends?
YES
Use generic iterator to iterate over mesh entities mapped to Nth geometric entity
NO
Pass test
M = 1
Get Mth mesh entity classified on the Nth geometric entity
Generic iterator ends?
YES
Get geometric classification of Mth mesh entity
NO
Same as Nth geometric entity?
NO
Fail Test
YES
M = M +
N = N + 1
* Failure in processing input data will also result in a failed test

Figure A.2: Steps for testing generic iterator for geometric model entities

entity, checking if remote copy of an entity exists on a specific part, setting remote copy of an entity etc.

Figure A.3 shows the high-level steps for testing the remote copy information of entities. A part-boundary entity has one or more remote copies, the test gets all part-

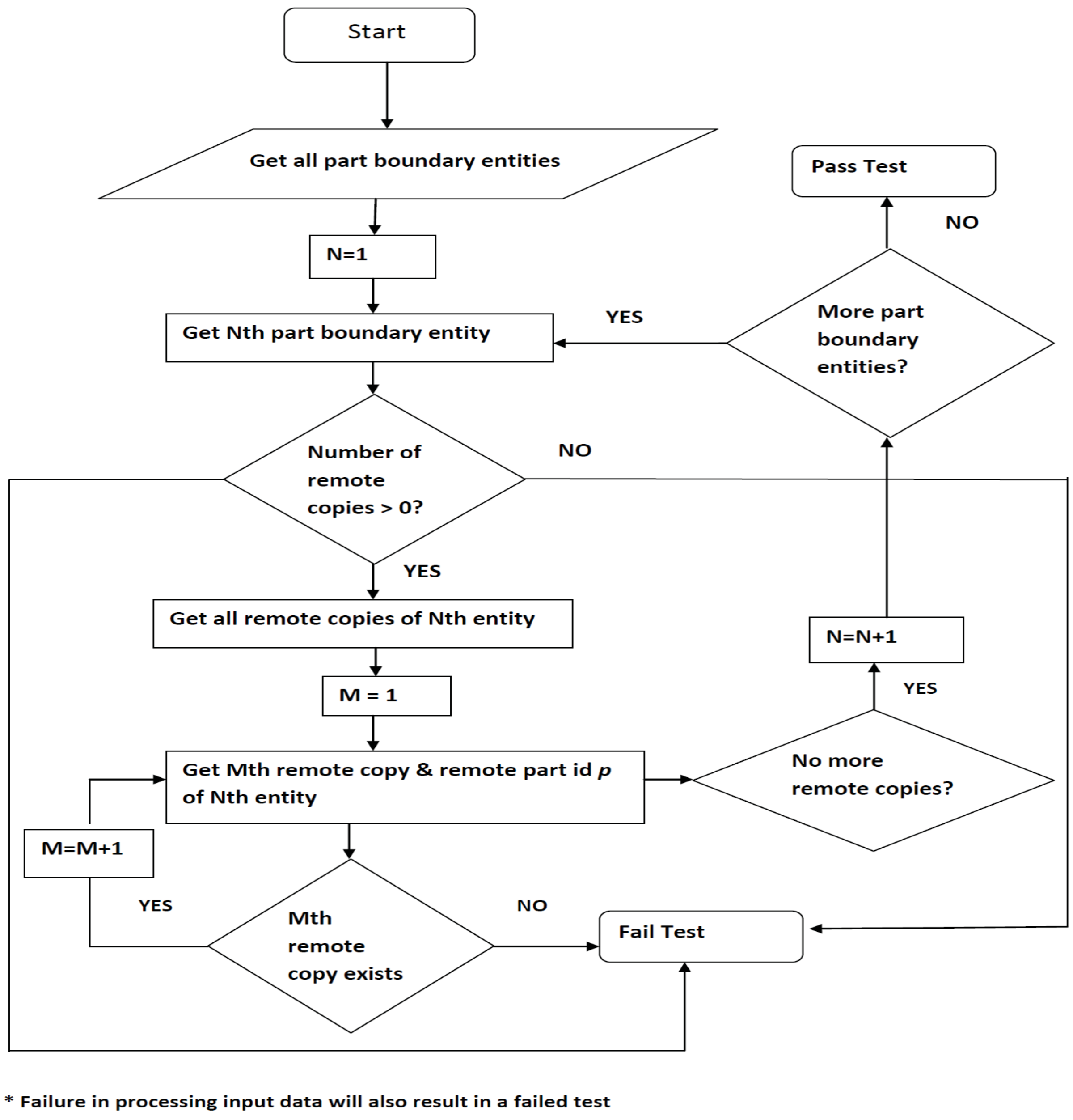

Start
Get all part boundary entities
N=1
Get Nth part boundary entity
Number of remote copies > 0?
NO
YES
Get all remote copies of Nth entity
M = 1
Get Mth remote copy & remote part id p of Nth entity
M=M+1
YES
Mth remote copy exists
NO
Fail Test
No more remote copies?
YES
N=N+1
More part boundary entities?
YES
NO
Pass Test
* Failure in processing input data will also result in a failed test

Figure A.3: Steps for testing remote copy information

boundary entities on the part as the test data. Every part boundary entity maintains remote copy information in the form {*remote part id, remote entity address*}. In order to verify that each part-boundary entity maintains the correct remote copy information, the unit tests checks each remote copy of the part boundary entity to make sure that the remote copy is valid and it exists on the remote part.

APPENDIX B
Mesh Verification Algorithm

The mesh verification algorithm aims to verify the validity of a mesh after ghosting or mesh modification operations are performed on it. The validity of a mesh after ghosting or migration is tested using the following :

- The remote copy information of all part boundary entities is consistent and up-to-date.
- The ghost entities are created on the destinations and all ghost information is up-to-date.
- The adjacencies of ghost entities, part boundary entities and non-part boundary entities are correct. For notational simplicity, a non-part boundary entity is referred as an $INTERNAL$ entity, a part-boundary entity is referred as $BOUNDARY$ entity and a ghost entity is referred as a $GHOST$ entity.
- The geometric classification of all remote and ghost copies is consistent.

Algorithm 9 gives the pseudo code for the mesh verification algorithm. Checks 1-4 verify the adjacencies of the mesh vertices, edges, faces and regions (Algorithms 12, 13, 14, 15). Check 5 verifies that a serial mesh does not hold any partition classification information. Residence parts of an entity consists of its remote copies and the owner. Check 6 verifies if the number of residence parts of an entity is correct (it should be $remoteCopies + 1$). Check 7 and 8 verify the consistency of remote and ghost copies and verifies that all copies of an entity have the same geometric classification information. Note that the algorithm checks adjacencies for a tetra-hedral mesh right now. In the future, it will be extended to support other mesh types. Algorithm 10 verifies the consistency of remote copies of an entity. For an entity M_i^d, the part sends a message to all its remote copies comprising of the address of M_i^d on P_{local}, its remote copy address M_k^d and the geometric classification information G_k^d (Algorithm 10 Step 1). When the part having the remote copy receives this message, it verifies if M_k^d exists and has the address of M_i^d on P_{local} among its remote copies (Algorithm 10 Step 2). It then verifies if the geometric classification information of both M_i^d and G_k^d are the same (Algorithm 10 Step 3).

Algorithm 11 verifies the consistency of ghost copies of an entity. For an entity M_i^d, the part sends a message to all its ghost copies comprising of the owner entity M_i^d, its ghost copy address M_k^d and the geometric classification information G_k^d (Algorithm 11 Step 1). When the part having the ghost copy receives this message, it verifies if M_k^d exists and has the address of M_i^d on P_{local} as its owner entity (Algorithm 11 Step 2). It then verifies if the geometric classification information of both M_i^d and M_k^d are the same (Algorithm 11 Step 3). For notational simplicity, ghost copies of an entity are represented by $\mathscr{G}[ent]$.

Algorithm 12 verifies the vertices in the mesh. For a 2D or 3D mesh, a vertex must have an adjacent edge to it (Algorithm 12 Step 1). Algorithm 13 verifies the regions in the mesh. Check 1 verifies that the mesh has no ghost region if no ghosting exists. Check 2 verifies that regions have no part boundary information (as regions are not part boundary entities, they should have no part boundary information). Mesh regions should be classified on geometric model regions which is verified by Check 3. Note that g denotes existence of ghosts in a mesh. Algorithm 14 verifies the adjacencies of mesh edges. Check 1 verifies that no edge stores any ghosting information if the mesh has no ghosting rule in it. Check 2 verifies that a ghost edge can have zero adjacent faces only if ghost edges are present (Ghost edges can be created independently See §3.3). If ghost faces are present in a mesh (with no

```
Data: Distributed mesh M, isValid
Result: Returns true if the mesh is valid and false otherwise.
/* Test validity of mesh vertices, edges and faces                    */
begin
    $isValid \leftarrow true$;
    /* Check 1:  Verify all vertices in the mesh                      */
    VerifyVertices(M, isValid);
    /* Check 2:  Verify all edges in the mesh                         */
    VerifyEdges(M, isValid);
    /* Check 3:  Verify all faces in the mesh                         */
    VerifyFaces(M, isValid);
    /* Check 4:  Verify all regions in the mesh                       */
    VerifyRegions(M, isValid);
    extract ghost rule $\{b, g, includeCopy\}$ from part;
    for $d \leftarrow 0$ to partDim do
        /* Check 5:  A serial mesh does not have partition
           classification or remote copy information.                 */
        foreach $M_i^d \in part$ do
        if $NUMPROC = 1$ & $\mathscr{R}[M_i^d] > 0$ then
            $isValid = false$;
        else
            /* Check 6:  Residence part count is 1 more than remote
               copy count.                                            */
            if $\mathscr{P}[M_i^d] \neq COUNT(\mathscr{R}[M_i^d]) + 1$ then
                $isValid = false$;
            end if
            /* Check 7:  Verify if the remote copy information is
               consistent                                             */
            VerifyRemoteCopies(isValid, $M_i^d$);
            /* Check 8:Verify if the ghost copy information is
               consistent                                             */
            VerifyGhostCopies(isValid, $M_i^d$);
        end if
    end for
end
```

Algorithm 9: Mesh_Verify(M)

```
Data: isValid, $M_i^d$
Result: Verifies consistency of remote copies
/* Test if remote copies are consistent and all have the same
   geometric information                                          */
begin
   /* Step 1:  Send a message to every remote copy of $M_i^d$       */
   foreach $M_k^d \in \mathscr{R}(M_i^d)$ do
   send message A(adress of $M_i^d$ on $P_{local}$,address of $M_k^d$ on $P_{remote}$,classification
   information of $M_i^d$);
   while $P_{remote}$ receives message A(address of $M_i^d$ on $P_{local}$,address of $M_k^d$ on
   $P_{remote}$,classification information of $M_i^d$) do
      /* Step 2:  Check if $M_k^d$ is a remote copy of $M_i^d$       */
      if $M_i^d \notin \mathscr{R}(M_k^d)$ then
         isValid=false;
      end if
      /* Step 3:  Check if the geometric info.  of remote copies is
         the same                                                 */
      $G_j^d \leftarrow$ classification information of $M_i^d$ on $P_{local}$
      if $G_j^d \neq G_k^d$ where $G_k^d \sqsubset M_k^d$ then
         isValid=false;
      end if
   end while
end
```

Algorithm 10: VerifyRemoteCopies(isValid, M_i^d)

adjacent region), then a ghost edge may only have one adjacent face. Check 3 verifies that a ghost edge can have one adjacent face only if ghost faces are present. Check 4 verifies that every edge in a 3D mesh should have at-least one adjacent face. Check 5 verifies that in a 2D mesh, the number of adjacent faces to a part boundary edge can be 2 if ghost faces exist on part boundary, else number of adjacent faces should be 1.

Algorithm 15 verifies the adjacencies of mesh faces. Note that a face without an adjacent region can only be valid if ghost faces exist in a mesh (ghost faces might not have adjacent regions, Algorithm 15 Check 1). Checks 2-6 verify the geometric classification information of the mesh faces.

```
Data: isValid, $M_i^d$
Result: Verifies consistency of ghost copies.
/* Test if ghost information is up-to-date consistent and if the
   original and ghost entities have the same geometric information
   */
begin
   /* Send a message to every ghost copy of $M_i^d$ and verify it is
      valid                                                        */
   foreach $M_k^d \in \mathscr{G}(M_i^d)$ do
   send message A(adress of $M_i^d$ on $P_{local}$,address of $M_k^d$ on $P_{ghost}$,classification
   information of $M_i^d$);
   while $P_{ghost}$ recieves message A(address of $M_i^d$ on $P_{local}$,address of $M_k^d$ on
   $P_{ghost}$,classification information of $M_i^d$) do
      /* The ghost should have the owner information and ghost flag
         set.                                                      */
      if $M_i^d \notin \mathscr{G}(M_k^d)$ OR $M_i^d \leftarrow isGhost = false$ then
         isValid=false;
      end if
      /* The geometric info.  of ghost copies is the same          */
      $G_j^d \leftarrow$ classification information of $M_i^d$ on $P_{local}$
      if $G_j^d \neq G_k^d$ where $G_k^d \sqsubset M_k^d$ then
         isValid=false;
      end if
   end while
end
```

Algorithm 11: VerifyGhostCopies(isValid, M_i^d)

```
Data: M, isValid
Result: Verifies the vertices in the mesh.
begin
   /* Step 1:  Check if each vertex has 1 or more adjacent edges   */
   . foreach vertex $M_i^0 \in part$ do
   /* For 2D or 3D mesh, number of edges adjacent to the vertices
      should be more than zero                                     */
   if $partDim \geq 2$ & $COUNT(M_i^0\{M^1\}) == 0$ then
      isValid=false;
   end if
end
```

Algorithm 12: VerifyVertices(M, isValid)

```
Data: M, isValid
Result: Verifies the regions in the mesh.
/* Collect entities for ghosting                                    */
begin
    foreach $region M_i^3 \in part$ do /* Check 1: If the mesh is not ghosted
        then $M_i^3$ should not have ghost information                  */
    if $!g$ & $(GHOST[M_i^3])$ then
        $isValid = false$;
    end if
    /* Check 2: Regions are not part boundary entities so $M_i^3$ should
        not be on part boundary.                                        */
    if $BOUNDARY[M_i^3] = true$ then
        isValid=false;
        break;
    end if
    /* Check 3: $M_i^3$ should be classified on a geometric region      */
    if $!M_i^3 \sqsubset G_k^3$ then
        isValid=false;
        break;
    end if
end
```

Algorithm 13: VerifyRegions(M,isValid)

```
Data: M, isValid
Result: Verifies the edges in the mesh.
begin
    foreach edge $M_i^1 \in part$ do
        /* Check 1:  If the mesh is not ghosted, and $M_i^1$ should not have
            ghost information                                                */
        if $!g$ & $(GHOST[M_i^1])$ then
            $isValid = false$;
        end if
        /* Check 2:  If the ghost objects are edges, then there might not
            be adjacent faces                                                */
        if $COUNT(M_i^1\{M^2\}) = 0$ & $g = 1$ & $GHOST(M_i^d)$ then
            continue;
        end if
        /* Check 3:  If the ghost objects are faces, then there may be
            only one face adjacent to $M_i^1$                                */
        if $COUNT(M_i^1\{M^2\}) = 1$ & $g = 2$ & $GHOST(M_i^d)$ then
            continue;
        end if
        if $partDim = 3$ then
            /* Check 4:  For a 3D mesh, every edge should have an adjacent
                face (except for 1D ghosts)                                  */
            if $COUNT(M_i^1\{M^2\}) \leq 1$ then
                isValid=false;
            end if
        else
            if $partDim = 2$ then
                /* Check 5:  For part boundary edge, if ghost faces exist
                    then there can be up-to 2 adjacent faces to the edge.
                    If there are no ghost faces, there can be 1 adjacent
                    face only.  For internal edge, there can be 2 adjacent
                    faces only.                                              */
                if $M_i^1 \in BOUNDARY(M_i^1)$ & $g! = 2$& $COUNT(M_i^1\{M^2\}) \neq 1$ then
                    isValid=false;
                end if
                if $M_i^1 \in BOUNDARY(M_i^1)$ & $g == 2$ &$COUNT(M_i^1\{M^2\}) \neq 2$
                then
                    isValid=false;
                end if
                if $M_i^1 \in INTERNAL(M_i^1)$ & $COUNT(M_i^1\{M^2\}) \neq 2$ then
                    isValid=false;
                end if
            end if
        end if
end
```

Algorithm 14: VerifyEdges(M,isValid)

```
Result: Verifies the faces in the mesh.
begin
    foreach $M_i^2 \in part$ do
    if $partDim = 3$ then
        /* Check1:  If ghost faces exist then there might not be an
            adjacent region to them                                    */
        if $COUNT(M_i^2\{M^3\}) = 0$ & $gDim = 2$ & $GHOST(M_i^d)$ then
            continue;
        end if
        /* Check 2:If the face is classified on a model face then
            there can be 1 adjacent region                             */
        if $M_i^2 \sqsubset G_i^2$ & $COUNT(M_i^2\{M^3\}) \neq 1$ then
            isValid=false;
        else
            /* Check 3:If the mesh face is classified on a model region
                and $M_i^2$ is internal entity then it should have 2 adjacent
                regions                                                */
            if $INTERNAL(M_i^2)$ & $COUNT(M_i^2\{M^3\}) \neq 2$ then
                isValid=false;
            else
                /* Check 4:If $M_i^2$ is a part boundary entity and there are
                    no ghost regions then it can have 1 adjacent regions
                    */
                if $BOUNDARY(M_i^2)$ & $g! = 3$& $COUNT(M_i^2\{M^1\}) \neq 1$ then
                    isValid=false;
                end if
                /* Check 5:If $M_i^2$ is a part boundary entity and there are
                    ghost regions then it can have 2 adjacent regions  */
                if $BOUNDARY(M_i^2)$ & $g = 3$& $COUNT(M_i^2\{M^1\}) \neq 2$ then
                    isValid=false;
                end if
            end if
        end if
    else
        /* Check 6:For 2D mesh, a mesh face should be classified on
            model face and there should be no adjacent regions         */
        if $partDim = 2$ & $(COUNT(M_i^2\{M^3\}) \neq 0$ OR $!(M_i^2 \sqsubset G_i^2)$ then
            isValid=false;
        end if
    end if
end
```

Algorithm 15: VerifyFaces(M,isValid)