Engineering with Computers manuscript No.
(will be inserted by the editor)

Efficient Distributed Mesh Data Structure for Parallel Automated Adaptive Analysis

E. Seegyoung Seol, Mark S. Shephard

Scientific Computation Research Center, Rensselaer Polytechnic Institute, Troy, NY 12180, U.S.A.

Received: date / Revised version: date

Abstract For the purpose of efficiently supporting parallel mesh-based simulations, we developed a partition model and a distributed mesh data management system that is able to shape its mesh data structure dynamically based on the user's representational needs to provide the needed representation at a minimum cost (memory and time), called Flexible distributed Mesh DataBase (FMDB). The purpose of the partition model is to represent mesh partitioning and support mesh-level parallel operations through inter-processor communication links. FMDB has been used to efficiently support parallel automated adaptive analysis processes in conjunction with existing analysis engines.

1 Introduction

An automated adaptive analysis typically starts with a coarse mesh and a low-order numerical solution of a problem and based on an estimate of the local discretization error either refines the mesh (h-refinement), increases the order of numerical solution (p-refinement), moves the mesh (r-refinement), or does combinations of h-, p- and r-refinements to improve the quality of the solution [1,2]. To run adaptive analysis in parallel, solvers and adaptation steps should run on distributed meshes partitioned over multiple processors [3].

A distributed mesh data structure is an infrastructure executing underneath providing all parallel mesh-based operations needed to support parallel adaptive analysis. An efficient and scalable distributed mesh data structure is mandatory to achieve performance since it strongly influences the overall performance of adaptive mesh-based simulations. In addition to the general mesh-based operations [4], such as mesh entity creation/deletion, adjacency and geometric classification, iterators, arbitrary attachable data to mesh entities, etc., the distributed mesh data structure must support (i) efficient communication between entities duplicated over multiple processors, (ii) migration of mesh entities between processors, and (iii) dynamic load balancing.

Papers have been published on the issues of parallel adaptive analysis including parallel mesh generation [5–11], dynamic mesh load balancing techniques [12–16], and data structure and algorithms for parallel structured [17–20] or unstructured mesh adaptation [11,21–27].

Parashar and Browne presented a distributed mesh data structure for parallel non-conforming h-refinement called DAGH (Distributed Adaptive Grid Hierarchy) [20]. DAGH represents a mesh with grid hierarchy. In case of a distributed grid, inter-grid operations are performed locally on each processor without involving any communication or synchronization due to the mesh refinement is non-conforming. The mesh load balancing is performed by varying granularity of the DAGH blocks.

LibMesh [23] is a distributed mesh data structure developed at the university of Texas in order to support parallel finite element simulations with refinement. It opted the classic element-node data structure supporting only h- uniform refinement and serial mesh partitioning for initial distribution.

Reference [28] presented a distributed mesh data strcuture to support parallel adaptive numerical computation based on refinement and coarsening [22]. A mesh data consists of vertices, edges and regions with a linked list data structure and maintains the shared processor lists for entities on partition boundaries through the message passing. Global identifiers are assigned to every entity, thus, all data structure are updated to contain consistent global information during adaptation. It provided the owning processor of shared entities which is randomly selected and the dynamic mesh load balancing with ParMETIS [29].

Correspondence to: seole@scorec.rpi.edu

In reference [26], Selwood and Berzins presented a general distributed mesh data structure that supports parallel mesh refinement and de-refinement. It represents a mesh with all d level mesh entities and adjacencies, and provides dynamic load balancing with the Zoltan [30] library. In order to be aware of the part of the mesh distributed on other processors, the pointers to the adjacent tetrahedron that are on other processors are kept for each processor.

Reference [14,21] 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 interprocessor 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. The owning processor of an entity on the partition boundary was determined to the processor with minimum processor id. In reference [15], PMDB was enhanced with addition of RPM (Rensselaer Partition Model) that represents heterogeneous processor and network of workstations, or some combination of these for the purpose of improving performance by accounting for resources of parallel computers.

Most of distributed mesh data structures published to date are shaped to specific mesh applications [11,20, 23,28] or support only part of adaptive analysis such as refinement step [18–20,23,24] or are able to handle only manifold meshes [11,15,19,20,23,24,26–28]. The development of the *general* distributed mesh data structure to efficiently support parallel adaptive analysis procedures including the solvers and the adaptation procedures is not trivial due to data structure comlexity, the nature of the mesh with non-manifold models, the consistently evolving nature in the mesh as it is adapted, and the needs for dynamic load balancing.

The current paper presents a new parallel mesh infrastructure capable of handling general non-manifold [31,32] models and effectively supporting automated adaptive analysis. The resulting parallel mesh infrastructure, referred to as Flexible distributed Mesh DataBase (FMDB), is a distributed mesh data management system that is capable of shaping its data structure dynamically based on the user's requested mesh representation [33]. This paper focuses on the design and implementation of the distributed mesh data structure in the FMDB. Two related papers cover issues of flexibility in the FMDB [34] and the software engineering aspects of the FMDB [33].

1.1 Nomenclature

V	the model, $V \in \{G, P, M\}$ where G signifies the geometric model, P signifies the partition model, and M signifies the mesh model.
$\{V\{V^d\}\}$	a set of topological entities of dimension d in model V.
V_i^d	the i^{th} entity of dimension d in model V. $d = 0$ for a vertex, $d = 1$ for an edge, $d = 2$ for a face, and $d = 3$ for a region.
$\{\partial(V_i^d)\}$	set of entities on the boundary of V_i^d.
$\{V_i^d\{V^q\}\}$	a set of entities of dimension q in model V that are adjacent to V_i^d.
$V_i^d\{V^q\}_j$	the j^{th} entity in the set of entities of dimension q in model V that are adjacent to V_i^d.
$U_i^{d_i} \sqsubset V_j^{d_j}$	classification indicating the unique association of entity $U_i^{d_i}$ with entity $V_j^{d_j}$, $d_i \leq d_j$, where U, $V \in \{G, P, M\}$ and U is lower than V in terms of a hierarchy of domain decomposition.
$\mathscr{P}[M_i^d]$	set of partition id(s) where entity M_i^d exists.
Examples	
$\{M\{M^2\}\}$	the set of all the faces in the mesh.
$\{M_3^1\{M^3\}\}$	the mesh regions adjacent to mesh edge M_3^1.
$M_1^3\{M^1\}_2$	the 2nd edge adjacent to mesh region M_1^3.

2 General Topology-based Mesh Data Structure

The structures used to support the problem definition, the discretization of the model and their interactions are central to mesh-based analysis methods like finite element and finite volumes. The geometric model houses the topological and shape description of the domain of the problem of interest. The mesh describes the discretized representation of the domain used by the analysis method. The linkage of the mesh to the geometric model, referred to as geometric classification, is critical for mesh generation and adaptation procedures since it allows the specification of analysis attributes in terms of the original geometric model, the proper approximation of the geometry during mesh adaptation and supports direct links to the geometric shape information of the original domain needed to improve geometric approximation and useful in p-version element integration [4, 35].

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 three functional requirements of a general topology-based mesh data structure are topological entities, geometric classification, and adjacencies between entities [4].

Topology provides an unambiguous, shape-independent, abstraction of the mesh. With reasonable restrictions on the topology [4], a mesh is represented with only the basic *0* to *d* dimensional topological entities, where d is the dimension of the domain of the interest. The full set of mesh entities in 3D is $\{\{\text{M}\{M^0\}\}, \{\text{M}\{M^1\}\}, \{\text{M}\{M^2\}\}, \{\text{M}\{M^3\}\}\}$, where $\{\text{M}\{M^d\}\}$, $d = 0, 1, 2, 3$, are, respectively, the set of vertices, edges, faces and regions. Mesh edges, faces, and regions are bounded by the lower order mesh entities.

Geometric classification defines the relation of a mesh to a geometric model. The unique association of a mesh entity of dimension d_i, $M_i^{d_i}$, to the geometric model entity of dimension d_j, $G_j^{d_j}$, where $d_i \leq d_j$, on which it lies is termed geometric classification and is denoted $M_i^{d_i} \sqsubset G_j^{d_j}$, where the classification symbol, $\sqsubset$, indicates that the left hand entity, or a set of entities, is classified on the right hand entity.

Adjacencies describe how mesh entities connect to each other. For an entity of dimension d, adjacency, denoted by $\{M_i^d\{M^q\}\}$, returns all the mesh entities of dimension q, which are on the closure of the entity for a downward adjacency ($d > q$), or for which the entity is part of the closure for an upward adjacency ($d < q$).

There are many options in the design of the mesh data structure in terms of the entities and adjacencies stored. If a mesh representation stores all *0* to *d* level entities explicitly, it is a *full* representation, otherwise, it is a *reduced* representation. *Completeness of adjacency* indicates the ability of a mesh representation to provide any type of adjacencies requested without involving an operation dependent on the mesh size such as the global mesh search or mesh traversal. Regardless of full or reduced, if all adjacency information is obtainable in O(1) time, the representation is complete, otherwise, it is incomplete.

We assume full and complete mesh representations throughout this paper.

3 Distributed Mesh Representation

3.1 Definitions and properties

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

Definition 1 (Partition) *A partition P_i consists of the set of mesh entities assigned to i^{th} processor. For each partition, the unique partition id can be given.*

Each partition is treated as a serial mesh with the addition of mesh partition boundaries to describe groups of mesh entities that are on inter-partition boundaries.

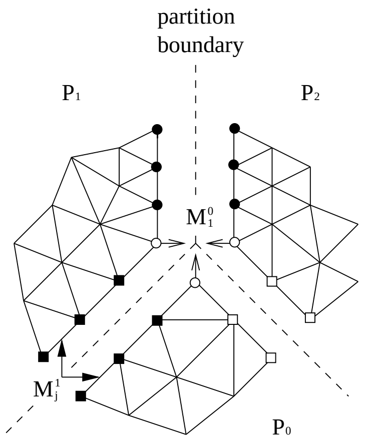


Fig. 1 Distributed mesh on three partitions P_0, P_1 and P_2 [27]

Mesh entities on partition boundaries are duplicated on all partitions on which they are used in adjacency relations. Mesh entities not on the partition boundary exist on a single partition. Figure 1 depicts a mesh that is distributed on 3 partitions. Vertex M_1^0 is common to three partitions and on each partition, several mesh edges like M_j^1 are common to two partitions. The dashed lines are *partition boundaries* that consist of mesh vertices and edges duplicated on multiple partitions.

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

Definition 2 (Residence partition operator $\mathscr{P}[M_i^d]$) *An operator that returns a set of partition id(s) where M_i^d exists.*

Definition 3 (Residence partition equation) *If $\{M_i^d\{M^q\}\} = \emptyset$, $d < q$, $\mathscr{P}[M_i^d] = \{p\}$ where p is the id of a partition on which M_i^d exists. Otherwise, $\mathscr{P}[M_i^d] = \cup\ \mathscr{P}[M_j^q \mid M_i^d \in \{\partial(M_j^q)\}]$.*

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

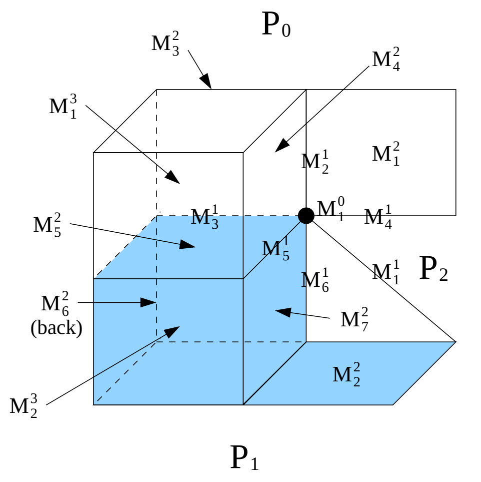


Fig. 2 Example 3D mesh distributed on 3 partitions

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

Definition 4 (Partition object) *The basic unit to which a destination partition id is assigned. The full set of partition objects is the set of mesh entities that are not part of the boundary of any higher order mesh entities. In a 3D mesh, this includes all mesh regions, the mesh faces not bounded by any mesh regions, the mesh edges not bounded by any mesh faces, and mesh vertices not bounded by any mesh edges.*

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

3.2 Functional requirements of distributed meshes

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

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

  Definition 5 (Remote partition) *Non-self partition*[1] *where a mesh entity is duplicated.*

  Definition 6 (Remote copy) *The memory location of a mesh entity duplicated on remote partition.*

  In parallel adaptive analysis, the mesh and its partitioning can change thousands of time during the simulation. Therefore, at the mesh functionality level, efficient mechanism to update mesh partitioning and keep the links between partitions updated are mandatory to achieve scalability.
- *Entity ownership*: For entities on partition boundaries, it is beneficial to assign a specific copy as the owner of the others and let the owner be in charge of communication or computation between the copies. There are 2 common strategies in determining the owning partition of partition boundary entities.
  - *Static entity ownership*: The owning partition of a partition boundary entity is always fixed to P_i regardless of mesh partitioning [15, 21]. It has been observed that the static entity ownership produces mesh load imbalance in adaptive analysis.
  - *Dynamic entity ownership*: The owning partition of the partition boundary entity is dynamically specified [28].

For the dynamic entity ownership, there can be several options in determining owning processor of mesh entities. With the FMDB, entity ownership is determined based on the rule of *the poor-to-rich ownership*, which assigns the poorest partition to the owner of entity, where the poorest partition is the partition that has the least number of partition objects among residence partitions of the entity. With this scheme, mesh load balance is kept during adaptive mesh control simulations since the local mesh migration procedure performed during mesh adaptation to gain the necessary entities for a specific mesh modification operator [25, 36] always migrates entities to poor partitions improving the overall performance of the parallel simulation.

4 A Partition Model

To meet the goals and functionalities of distributed meshes, a partition model has been developed between the mesh and the geometric model. As illustrated in Figure 3, the partition model can be viewed as a part of hierarchical domain decomposition. Its sole purpose is to represent mesh partitioning in topology and support mesh-level parallel operations through inter-partition boundary links with ease.

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

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

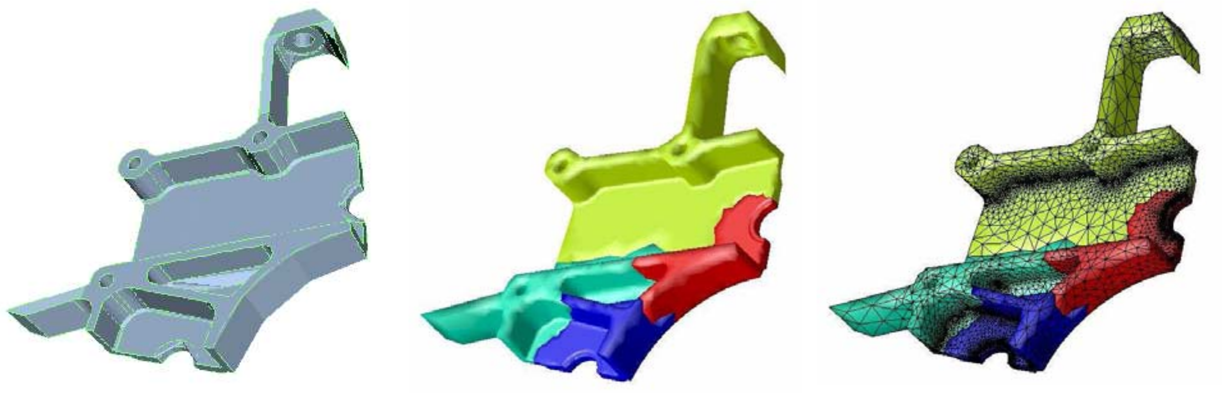


Fig. 3 Hierarchy of domain decomposition: geometry model, partition model, and the distributed mesh on 4 processors

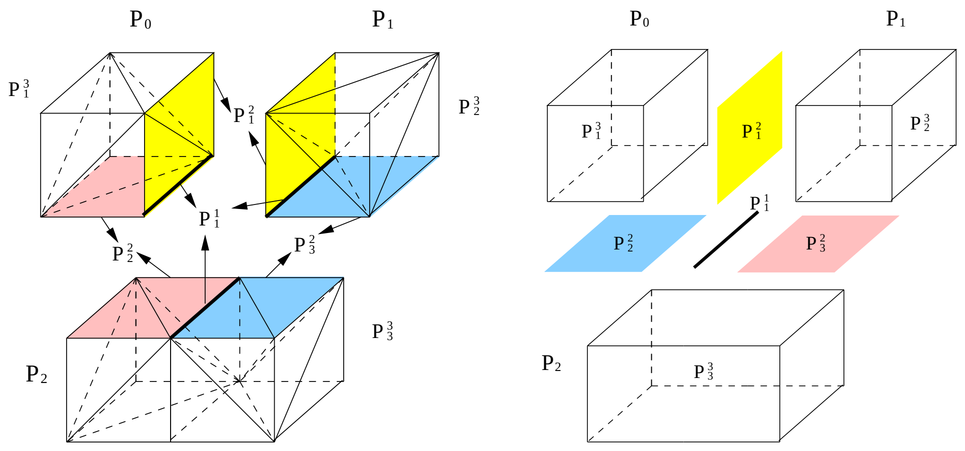


Fig. 4 Distributed mesh and its association with the partition model via partition classifications

4.1 Definitions

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

At a minimum, *residence partition* must be the criterion to be able to support the inter-partition communications. *Connectivity* between entities is also desirable for the criterion to support some operations quickly and can be used optionally. Two mesh entities are *connected* if they are on the same partition and reachable via adjacency operations. The connectivity is expensive but useful in representing separate chunks in a partition. It enables to diagnose the quality of mesh partitioning immediately at the partition model level. In our implementation, for the efficiency purpose, only residence partition is used for the criterion. Definition 7 defines the partition model entity based on the residence partition criterion.

Definition 7 (Partition (model) entity) *A topological entity in the partition model, P_i^d, which represents a group of mesh entities of dimension d that have the same $\mathscr{P}$. Each partition model entity is uniquely determined by $\mathscr{P}$.*

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

Definition 8 (Partition classification) *The unique association of mesh topological entities of dimension d_i, $M_i^{d_i}$, to the topological entity of the partition model of dimension d_j, $P_j^{d_j}$ where $d_i \le d_j$, on which it lies is termed partition classification and is denoted $M_i^{d_i} \sqsubset P_j^{d_j}$.*

Definition 9 (Reverse partition classification) *For each partition entity, the set of equal order mesh entities classified on that entity defines the reverse partition classification for the partition model entity. The reverse partition classification is denoted as $RC(P_j^d) = \{M_i^d \mid M_i^d \sqsubset P_j^d\}$.*

Figure 4 illustrates a distributed 3D mesh with mesh entities labeled with arrows indicating the partition classification of the entities onto the partition model entities and its associated partition model. The mesh vertices and edges on the thick black lines are classified on partition edge P_1^1 and they are duplicated on three partitions P_0, P_1, and P_2. The mesh vertices, edges and faces on the shaded planes are duplicated on two partitions and they are classified on the partition face pointed with each arrow. The remaining mesh entities are not duplicated, therefore they are classified on the corresponding partition region. Note the reverse classification returns only the same order mesh entities. The reverse partition classification of P_1^1 returns mesh edges on the thick black lines. The reverse partition classification of partition face P_i^2 returns mesh faces on each corresponding shaded plane, $i = 1, 2, 3$.

4.2 Building a partition model

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

1. *High-to-low mesh entity traversal*: The partition classification is set from high order to low order entity (residence partition equation).
2. *Inheritance-first*: If $M_i^d \in \{\partial(M_j^q)\}$ and $\mathscr{P}[M_i^d] = \mathscr{P}[M_j^q]$, M_i^d inherits the partition classification from M_j^q as a subset of the partitions it is on.
3. *Connectivity-second*: If M_i^d and M_j^d are connected and $\mathscr{P}[M_i^d] = \mathscr{P}[M_j^d]$, M_i^d and M_j^d are classified on the same partition entity.
4. *Partition entity creation-last*: If neither of rule 2 nor 3 applies for M_i^d, a new partition entity P_j^d is created.

Rule 2 means if the residence partitions of M_i^d is identical to those of its bounding entity of higher order, M_j^q, it is classified on the partition entity that M_j^q is classified on. For example, in Figure 4(a), any mesh faces, edges and vertices that are not on shaded planes nor on the thick black line are classified on the partition region by inheriting partition classification from the regions it bounds. Note multiple inheritance produces unique partition classification. For instance, internal mesh faces on partition P_0 which are not on shaded planes can inherit partition classification from any of its bounding regions. However, the derived partition classification will always be P_1^3 regardless of the region it was derived from. Rule 3 is applied when Rule 2 is not satisfied. Rule 3 means if residence partitions of M_i^d and M_j^d are the same and they are connected, M_i^d is classified on the same partition entity where M_j^d classified on. When neither Rule 2 nor Rule 3 is satisfied, Rule 4 applies, thus a new partition entity of dimension d is created for the partition classification of entity M_i^d.

5 Efficient Mesh Migration

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

Figure 5(a) and (b) illustrate the 2D partitioned mesh and its associated partition model to be used as for the example of mesh migration throughout this section. In Figure 5(a), the partition classification of entities on the partition boundaries is denoted with the lines of the same pattern in Figure 5(b). For instance, M_1^0 and M_4^1 are classified on P_1^1, and depicted with the dashed lines as P_1^1. In Figure 5(b). the owning partition of a partition model edge (resp. vertex) is illustrated with thickness (resp. size) of lines (resp. circle). For example, the owning partition of partition vertex P_1^0 is P_0 since P_0 has the least number of partition objects among 3 residence partitions of P_1^0. Therefore P_1^0 on P_0 is depicted with a bigger-sized circle than P_1^0 on P_1 or P_2 implying that P_0 is the owning partition of P_1^0.

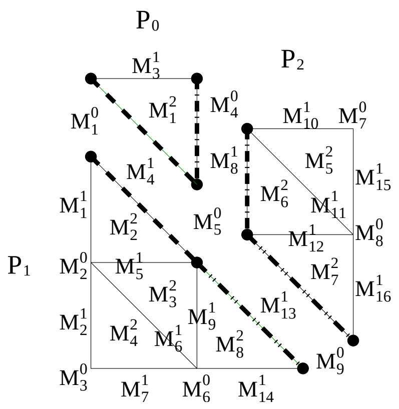
The input of the mesh migration procedure is a list of partition objects to migrate and their destination partition ids, called, for simplicity, $POsToMove$. Given the initial partitioned mesh in Figure 5(a), we assume that the input of the mesh migration procedure is <(M_1^2,2), (M_7^2,3), (M_8^2,3)>; M_1^2 will migrate to P_2 and M_7^2 and M_8^2 will migrate to P_3. Partition P_3 is currently empty.

Algorithm 1 is the pseudo code of the mesh migration procedure.

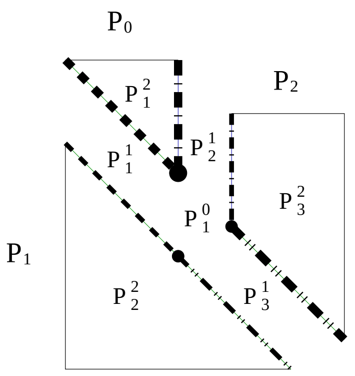
5.1 Step 1: Preparation

For a given list of partition objects to migrate, Step 1 collects a set of entities to be updated by migration. The entities collected for the update are maintained in vector *entsToUpdt*, where *entsToUpdt*$[i]$ contains the entities of dimension i, $i = 0, 1, 2, 3$. With a single program multiple data paradigm [37] in parallel, each partition maintains the separate *entsToUpdt*$[i]$ with different contents.

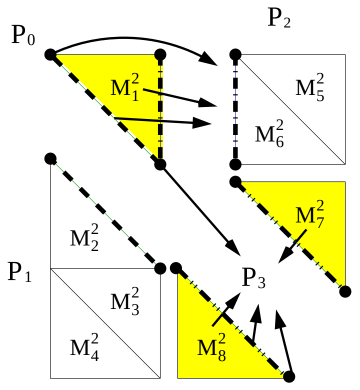
For the example mesh, the contents of *entsToUpdt* by dimension for each partition is given in Table 1. Only entities listed in Table 1 will be affected by the remaining steps in terms of their location and partitioning-related internal data. *entsToUpdt*$[2]$ contains the mesh faces to be migrated from each partition. *entsToUpdt*$[1]$ contains the mesh edges which bound any mesh face in *entsToUpdt*$[2]$ and their remote copies. *entsToUpdt*$[0]$



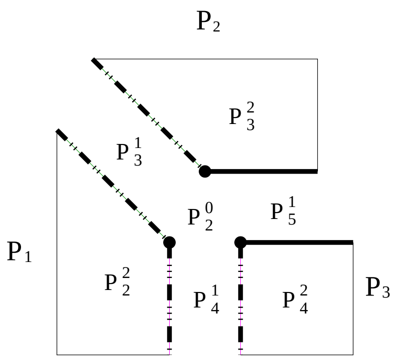
(a) initial mesh



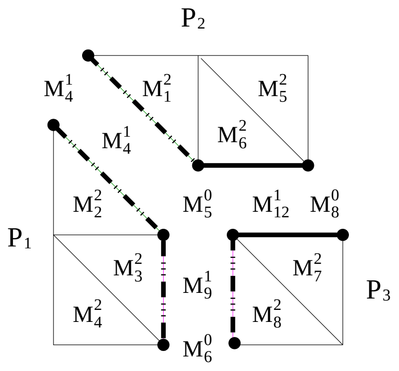
(b) partition model of (a)



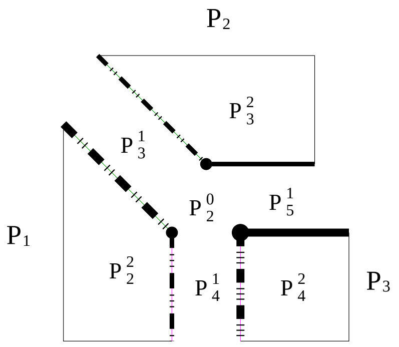
(c) mesh during migration



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



(e) final mesh



(e) final partition model with ownership

Fig. 5 Example of 2D mesh migration

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

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

```
Data   : $M$, $POsToMove$
Result : migrate partition objects in $POsToMove$
begin
    /* Step 1: collect entities to process and clear
    partitioning data. See §5.1 */
    for each $M_i^d \in POsToMove$ do
        insert $M_i^d$ into vector $entsToUpdt[d]$;
        reset partition classification and $\mathscr{P}$;
        for each $M_j^q \in \{\partial(M_i^d)\}$ do
            insert $M_j^q$ into $entsToUpdt[q]$;
            reset partition classification and $\mathscr{P}$;
        endfor
    endfor
    /* Step 2: determine residence partition. See
    §5.2 */
    for each $M_i^d \in$ entsToUpdt$[q]$ do
        set $\mathscr{P}$ of $M_i^d$;
    endfor
    do one-round communication to unify $\mathscr{P}$ of
    partition boundary entities;
    /* Step 3: update partition classification and
    collect entities to remove. See §5.3 */
    for $d \leftarrow 3$ to $0$ do
        for each $M_i^d \in$ entsToUpdt$[d]$ do
            determine partition classification;
            if $P_{local} \notin \mathscr{P}[M_i^d]$ do
                insert $M_i^d$ into $entsToRmv[d]$;
            endif
        endfor
    endfor
    /* Step 4: exchange entities. See §5.4 */
    for $d \leftarrow 0$ to $3$ do
        M_exchngEnts($entsToUpdt[d]$); /* Algorithm
        2 */
    endfor
    /* Step 5: remove unnecessary entities. See §5.5
    */
    for $d \leftarrow 3$ to $0$ do
        for each $M_i^d \in entsToRmv[d]$ do
            if $M_i^d$ is on partition boundary do
                remove copies of $M_i^d$ on other
                partitions;
            endif
            remove $M_i^d$;
        endfor
    endfor
    /* Step 6: update ownership. See §5.6 */
    for each $P_i^d$ in $P$ do
        owning partition of $P_i^d \leftarrow$ the poorest
        partition among $\mathscr{P}[P_i^d]$;
    endfor
end
```

Algorithm 1: M_migrate(M, $POsToMove$)

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

Algorithm 2: M_exchngEnts($entsToUpdt[d]$)

contains the mesh vertices that bound any mesh edge in *entsToUpdt*[1] and their remote copies. The partition classification and $\mathscr{P}$ of entities in *entsToUpdt* are cleared for further update.

5.2 Step 2: Determine residence partition(s)

Step 2 determines $\mathscr{P}$ of the entities according to the residence partition equation. For each entity which bounds the higher order entity, it should be determined if the entity will exist on the current local partition, denoted as P_{local}, after migration to set $\mathscr{P}$. Existence of the entity on P_{local} after migration is determined by checking adjacent partition objects, i.e., checking if there's any adjacent partition object to reside on P_{local}. One round of communication is performed at the end to exchange $\mathscr{P}$ of the partition boundary entities to unify them between remote copies. See §6.2.2 and §6.2.3 for the typical pseudo codes for a round of communication.

5.3 Step 3: Update the partition classification and collect entities to remove

For the entities in *entsToUpdt*, based on $\mathscr{P}$, Step 3 refreshes the partition classification to reflect a new updated partition model after migration, and determines the entities to remove from the local partition, P_{local}. An entity is determined to remove from its local partition if $\mathscr{P}$ of the entity doesn't contain P_{local}. Figure 5(d) is the partition model updated based on the new partition topology.

5.4 Step 4: Exchange entities

Since an entity of dimension > 0 is bounded by lower dimension entities, mesh entities are exchanged from low to high dimension. Step 4 exchanges entities from dimension 0 to 3, creates entities on the destination partitions, and updates the remote copies of the entities created on the destination partitions. Algorithm 2 illustrates the pseudo code that exchanges the entities contained in $entsToUpdt[d]$, $d = 0, 1, 2, 3$.

Step 4.1 sends the messages to destination partitions to create new mesh entities. Consider entity M_i^d duplicated on several partitions needs to be migrated to P_i. In order to reduce the communications between partitions, only one partition sends the message to P_i to create M_i^d. The partition to send the message to create M_i^d is the partition of the minimum partition id among residence partitions of M_i^d. The partition that sends messages to create a new entity is called *broadcaster*, denoted as P_{bc}. The broadcaster is in charge of creating as well as updating M_i^d over all partitions. For instance, among 3 copies of vertex M_5^0 in Figure 5(a), P_0 will be the *broadcaster* of M_5^0 since its partition id is the minimum among $\mathscr{P}[M_5^0]$. The arrows in Figure 5(c) indicate the broadcaster of each entity to migrate. Before sending a message to P_i, M_i^d is checked if it already exists on P_i using the remote copy information and ignored if exists.

For each M_i^d to migrate, P_{bc} of M_i^d sends a message composed of the address of M_i^d on P_{bc} and the information of M_i^d necessary for entity creation, which consists of unique vertex id (if vertex), entity shape information, required entity adjacencies, geometric classification information, residence partition(s) for setting partition classification, and remote copy information.

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

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

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

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

5.5 Step 5: Remove unnecessary entities

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

5.6 Step 6: Update ownership

Step 6 updates the owning partition of the partition model entities based on the rule of the poor-to-rich partition ownership. The partition model given in Figure 5(e) is the final partition model with ownership.

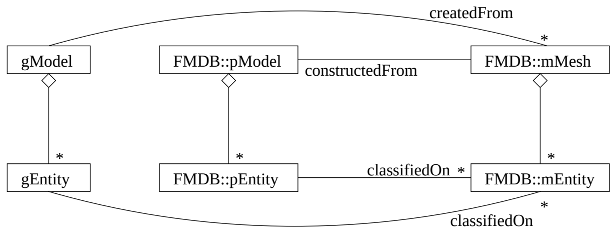


Fig. 6 Class diagram of G, M and P

6 S/W Design and Implementations

FMDB is implemented in C++ and includes advanced C++ programming elements, such as the STL (Standard Template Library) [38], functors [39], templates [40], singletons [41], and generic programming [42] for the purpose of achieving reusability of the software. MPI (Message Passing Interface) [37,43] and Autopack [44] are used for efficient parallel communications between processors. The Zoltan library [30] is used to make partition assignment during dynamic load balancing.

6.1 Class definition

Figure 6 illustrates the relationship between the geometric model, the partition model, the mesh and their entities using the Unified Modeling Language notation [45]. A geometric model, *gModel*, is a collection of geometric model entities, *gEntity*. A partition model, *pModel*, is a collection of partition model entities, *pEntity*, and it is constructed from a set of mesh entities assigned to partitions. A mesh, *mMesh*, is a collection of mesh entities, *mEntity*, and it is created from *gModel* using a mesh generation procedure. The mesh maintains its current classification against a geometric model, *gModel*, and a partition model, *pModel*.

When a mesh entity is on the partition boundaries, its remote copies and remote partitions are maintained. The pairs of remote copy and remote partition of each mesh entity is stored in the STL *map* since remote copy and remote partition are one-to-one mapped. Iteration over remote copies is provided through STL-like iterators.

The partition model acts as a container for partition model entities. All partition model entities are stored in an STL *set* since they are unique and iterated through STL-like iterators. It provides member functions such as updating the owning partition of partition model entities and computing partition classification of a mesh entity. Since a partition model must be instantly updated as the mesh partitioning is changed, each partition model instance maintains the relation back to the corresponding distributed mesh.

Each partition model entity stores id, dimension, the owning partition id, and the set of residence partitions as its member data. Residence partitions of a partition entity are stored in an STL *set* and also iterated through STL-like iterators.

6.2 Parallel fuctionalities

6.2.1 Parallel services. The parallel utility class supports various services for parallel programming. Its main purpose is to hide the details of parallelization and let the user program do parallel operations without knowing details of parallel components. The parallel utility class is a singleton (i.e., it is based on the Singleton pattern [41,42]) so only one single instance can exist overall and be accessible globally. The main goal in the design of distributed meshes is to have a serial mesh be a distributed mesh on a single processor. All parallel utility functions are also available in serial.

6.2.2 Efficient communications: Autopack. Since communication is costly for distributed memory systems, it is important to group small pieces of messages together and send all out in one inter-processor communication. The message packing library Autopack [44] is used for the purpose of reducing the number of message fragments exchanged between partitions. The general non-blocking codes embedded in parallel mesh-based algorithms to minimize communications begin by allocating a local buffer on each processor. Then for mesh entities on partition boundaries, the messages for remote copies to be sent to remote partitions are collected in the buffer. When all desired messages have been processed, the messages collected in local buffer are sent to remote partitions. Then the remote partitions process what they received.

The following is the template of a program used for communications between remote copies of partition boundary entities using Autopack, where *AP_size* and *AP_size* return, respectively, the total number of processors and the processor id of the local processor.

```
#include "autopack.h"

// send phase
int* sendcounts = new int[AP_size()];
for (int i = 0; i < AP_size(); ++i)
   sendcounts[i] = 0;
 for each entity on the partition boundary  {
    for each remote copy of the entity  {
      void* buf = AP_alloc(...,
          remote partition id, );
      fill the buffer ;
      AP_send(buf);
      ++sendcounts[remote partition id];
   }
}
// receive phase
AP_check_sends(AP_NOFLAGS);
AP_reduce_nsends(sendcounts);
int count, message = 0;
```

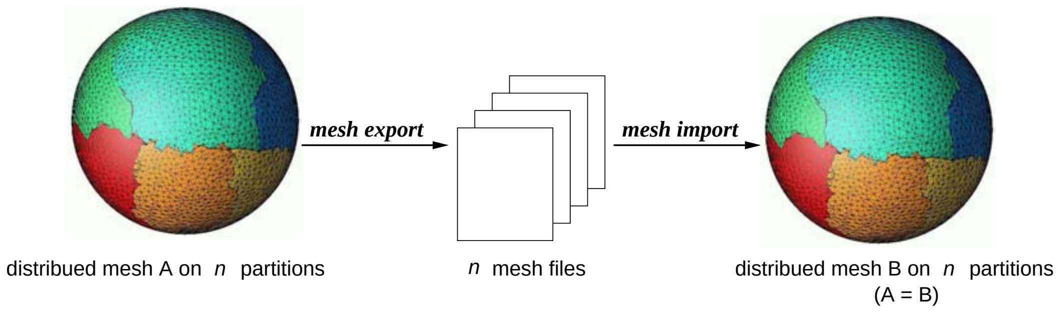

mesh export
mesh import
distribued mesh A on n partitions
n mesh files
distribued mesh B on n partitions
(A = B)

Fig. 7 Parallel mesh I/O

```
while (!AP_recv_count(&count) || message<count)
{
   void* msg;
   int rc = AP_recv(..., &msg, ...);
   if (rc)  {
      ++message;
      process msg received ;
      AP_free(msg);
   }
}
AP_check_sends(AP_WAITALL);
delete[] sendcounts;
```

The C integer array *sendcounts* is a counter for the number of messages sent to each partition. After initializing *sendcounts*, for each remote copy of each entity, *AP_alloc* allocates memory for a message. After filling the message buffer to send, *AP_send* sends the message to the remote partition where the remote copy exists. *AP_recv* receives the message and the appropriate operation is performed on the remote copy. *AP_recv_count* keeps track of the number of messages received ensuring no change in the number of messages sent and received. Packing many small messages into larger messages is hidden from the user.

6.2.3 Generic data communicator. The typical pattern of communications between remote copies of a mesh entity on the partition boundary is the following: for each remote copy of the mesh entity on the partition boundaries, (*i*) fill the message buffer to send to the remote copy on the remote partition. One message per remote copy is filled allowing sending different messages depending on the destination partition. (*ii*) Send the messages to the remote copy of the mesh entity. (*iii*) The remote copy of the entity receives the message from the *sender* partition and processes the received data.

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

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

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

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

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

6.3 Parallel mesh I/O

Figure 7 illustrates the parallel mesh I/O. The parallel mesh *import/export* procedures let the user export the distributed mesh into mesh files and recover the mesh later from the files. The parallel mesh export operator writes a distributed mesh on n partitions into n mesh files and the parallel mesh import operator reads the n mesh files and constructs the identical distributed mesh and the partition model on n partitions as before the export. The extra information kept in the mesh file to recover the distributed mesh from the file includes:

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

6.4 Dynamic mesh load balancing

During parallel adaptive analysis, the computation and mesh data need to be partitioned in such a way that the load balance is achieved in each partition and the size of the inter-partition boundaries is kept minimal to maximize utilization of all processors by minimizing idling of processors and inter-processor communication (i.e. the size of partition boundaries) [3,25].

The Zoltan library is a collection of data management services for parallel, unstructured, adaptive, and dynamic applications [30]. It includes a suite of parallel algorithms for dynamically partitioning problems over sets of processors. The FMDB interfaces with the Zoltan to obtain distribution information of the mesh. It computes the input to the Zoltan that is a model for computational load, i.e., a representation of the distributed mesh, usually a weighted graph or coordinates of partition objects. The load of a given processor P_i is defined as the number of elements in its partition multiplied by a weight that can be determined based on, for example, the elements computational demands. If no weight information is provided, all weights are set to *1.0* by default so that the load is simply proportional to the number of elements. The user also can provide weights for partition boundaries in order to take into account differing communication costs.

With the distribution information from Zoltan, the re-partitioning or initial partitioning step is completed by calling the mesh migration procedure that moves the appropriate entities from one partition to another. Figure 8 illustrates an example of 2D mesh load balancing. In the left, the partition objects (all mesh faces in this case) are tagged with their destination partition ids. The final balanced mesh is given on the right.

6.4.1 User interface: Zoltan callback.

The interface for load balancing with a weighted graph is the following:

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

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

Class *pmLBCallbacks* is purely virtual. It serves as a base class of the Zoltan callback class, *pmZoltanCallbacks*, to provide functions to retrieve partition information from the Zoltan in a form of an integer array of which the i^{th} item denotes the destination partition of i^{th} partition object in the mesh and pack/unpack arbitrary attached data to mesh entities for communication.

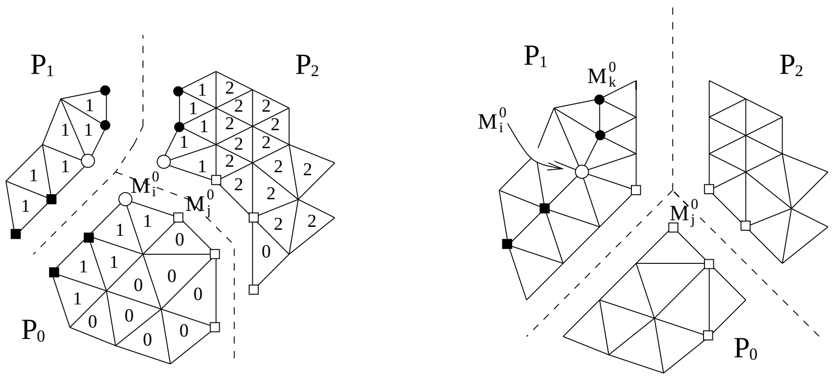


Fig. 8 Example of 2D mesh load balancing: (left) partition objects are tagged with their destination pids (right) mesh after load balancing

Class *pmZoltanCallbacks* is a derived class from *pmLBCallbacks* with addition of a function that enables one to choose one specific load balancing algorithm among various partitioning services of the Zoltan.

A typical set of instructions for doing a dynamic load balancing with the FMDB is to declare a derived class from *pmZoltanCallbacks*, choose a Zoltan algorithm, fill in the behaviors for packing/unpacking attached data to mesh entities, set weights of entities. Reference [30] provides more detailed discussions on the partitioning algorithms provided by Zoltan. The following is a simple user-defined Zoltan callback class for serial mesh partitioning:

```
class myCB: public pmZoltanCallbacks  {
public :
  myCB() : pmZoltanCallbacks(
             pmZoltanCallbacks::Serial){}
  virtual void * getUserData (mEntity*, int, int&)
  { return; }
  virtual void receiveUserData (mEntity*, int, int,
    void*)  { return; }
  virtual void deleteUserData (void *buf)
  { free(buf); }
};

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

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

7 Applications

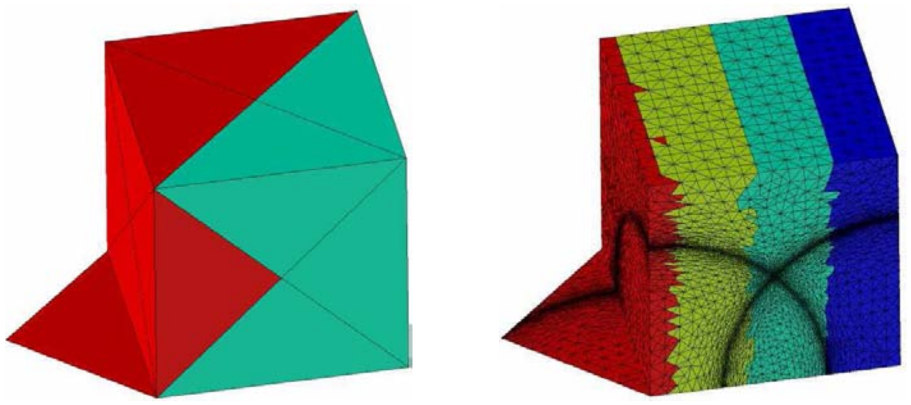
7.1 Parallel Anisotropic 3D Mesh Adaptation

Anisotropic mesh modification [46,47] provides a general mesh adaptation procedure that applies local mesh modification operations to yield a mesh of elements matching the required sizes and shapes. The mesh adaptation procedure is governed by a discrete anisotropic metric field specified at each mesh vertex of the current mesh [46]. The procedure consists of the four interacted high-level components of refinement, coarsening, swapping [46], and projecting new vertices created onto curved model boundaries onto the boundaries [47].

The serial mesh modification procedure has been extended to work with the distributed meshes in parallel [36,46].

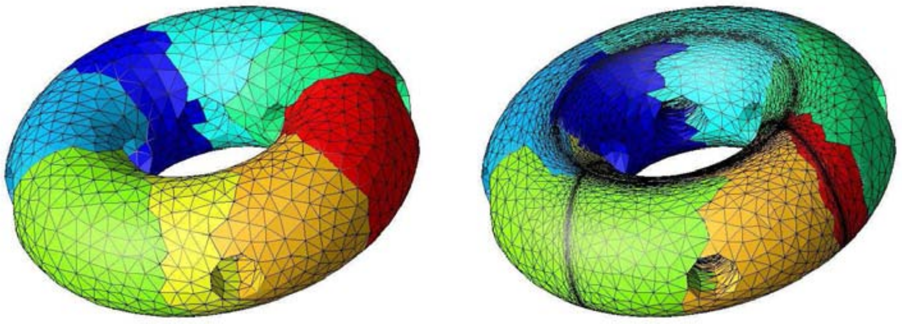
The distributed mesh data structure of the FMDB provides all needed functionalities for supporting parallel mesh adaptation such as dynamic mesh load balancing, individual local mesh migration for coarsening, projecting and swapping, easy-to-customize templated communication, etc.. The mesh migration procedures combined with *the poor-to-rich ownership* maintains the mesh load balance during processes, reducing the frequency of calls to the mesh load balancing procedure. The metric field of each mesh vertex is implemented with attachable data to the mesh vertices. While migrating mesh vertices, the metric field attached to the vertex must be exchanged properly to proceed the adaptation. As discussed in §6.4, the Zoltan callback of the FMDB supports the functionality to customize packing/unpacking any arbitrary attached data to mesh entities in the migration procedure. Reference [36] presents detailed discussions on parallelizing each mesh modification component.

The parallel mesh adaptation procedure has been tested against a wide range of models under either analytical or adaptively defined mesh size field definitions [46]. Some results of the parallel mesh adaptation are presented to demonstrate the scalability of the distributed mesh data structure developed. The scalability of a par-



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

Fig. 9 Parallel mesh adaptation I: (left) initial 36 tet mesh, (right) adapted approx. 1 million tet mesh.



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

Fig. 10 Parallel mesh adaptation: (left) initial 20,067 tet mesh, (right) adapted approx. 2 million tet mesh.

allel program running on p processors is defined as the *speedup*.

$$speedup = \frac{\text{run-time on 1 processor}}{\text{run-time on } p \text{ processors}} \quad (1)$$

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

$$relative\ speedup = \frac{\text{run time on } \frac{p}{2} \text{ processors}}{\text{run time on } p \text{ processors}} \quad (2)$$

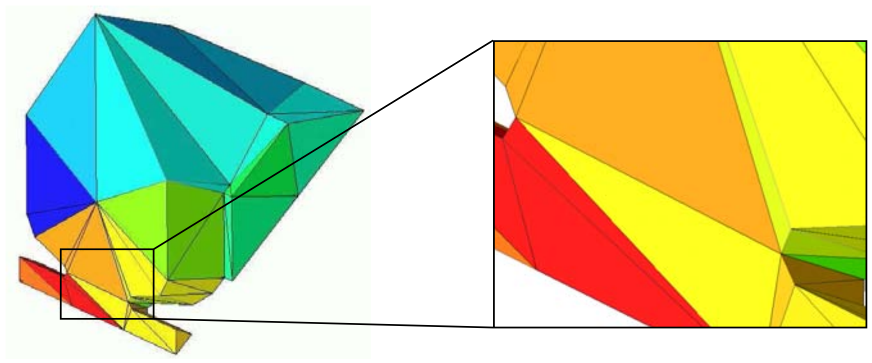
Figure 9 shows a uniform initial non-manifold mesh of a $1 \times 1 \times 1$ cubic and triangular surface domain and the adapted mesh with two spherical mesh size fields on 4 processors. Different color represents different partitions.

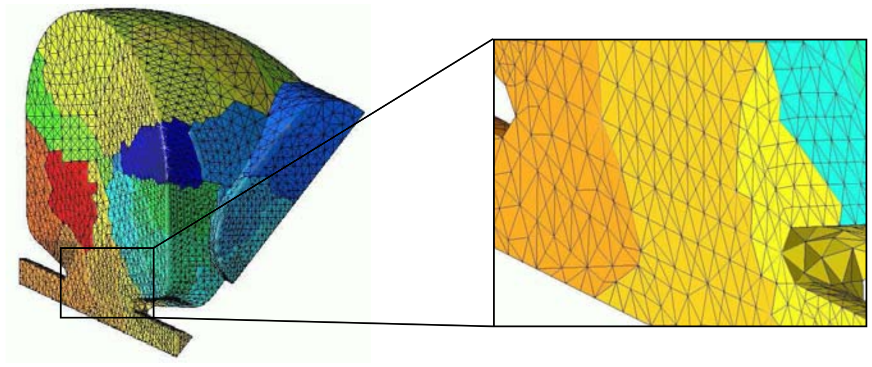
The geometry of the mesh in Figure 10 is a torus with four circular holes. The initial mesh is 20,067 tetrahedron. The adapted, approximately 2 million tetrahedron mesh with two spherical shocks is given with the speedup.

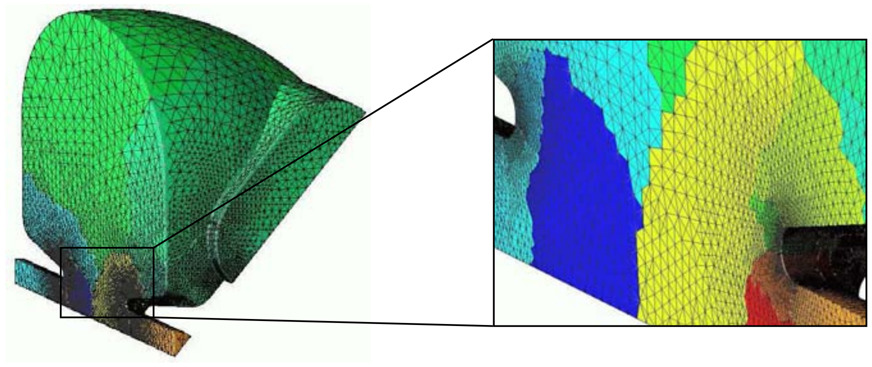
7.2 Parallel adaptive loop for accelerator design

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

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


(a) initial mesh


(b) after the 2^{nd} loop


(c) after the 8^{th} loop

# proc	20	40
rel. speedup	-	1.81

Fig. 11 Parallel adaptive loop for SLAC I: (a) initial coarse Trispal mesh (65 tets), (b) adapted mesh after the second adaptive loop (approx. 1 million tet), (c) the final mesh converged to the solutions after the eighth adaptive loop (approx. 12 million tets).

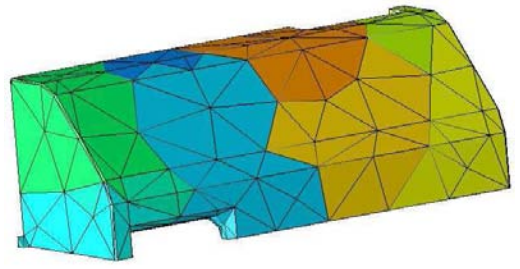
Figure 11 shows the Trispal meshes during the parallel adaptive loop, (a) gives the initial mesh composed of 65 tetrahedron, (b) is the adapted, approximately 1 million, mesh after the second adaptive loop on 24 processors, and (c) is the adapted, approximately 12 million, mesh after the eighth adaptive loop, which converged to the solution fields.

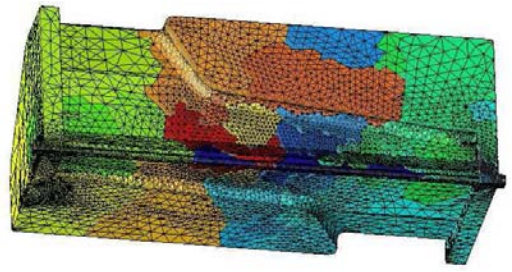
Figure 12 gives the RFQ meshes during the parallel adaptive loop, (a) gives the initial coarse mesh of 1,595 tetrahedron, (b) is the adapted mesh after the first adaptive loop, which is approximately 1 million tetrahedron, and (c) and (d) are the front and back view of the adapted mesh after the second adaptive loop, which contains about 24 million tetrahedron.

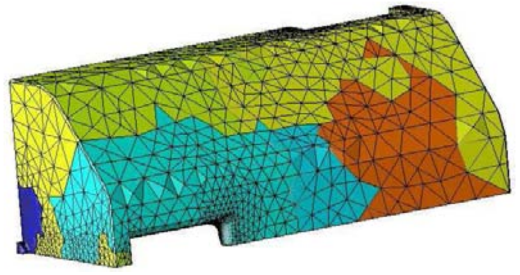
In both of Trispal and RFQ cavities, the parallel adaptive procedure reliably produced the results with the desired accuracy and quality factors.

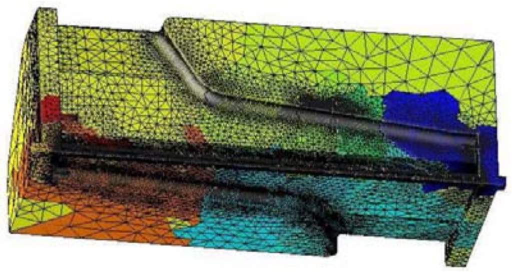
8 Closing Remarks

Data structures for distributed meshes were designed based on the hierarchical domain decomposition, providing a partition model as intermediate domain decomposition between the geometric model and the mesh. For


(a) initial (front)


(b) 1st loop (back)


(c) 2nd loop (front)


(d) 2nd loop (back)

# proc	28	56
rel. speedup	-	1.97

Fig. 12 Parallel adaptive loop for SLAC II: (a) initial coarse RFQ mesh (1,595 tet), (b) adapted mesh from the first adaptive loop (approx. 1 million tet), (c) the front view of adapted mesh from the second adaptive loop (approx. 24 million tet), (d) the back view of (c).

that purpose, the definitions and properties of the partition model and relations between the distributed mesh and the partition model were identified. The support of partitioning at the partition model level, optimal order algorithms to construct and use it, local mesh entity migration and dynamic load balancing are supported effectively. The distributed mesh environment was designed to handle meshes on general non-manifold geometries.

The DOE SciDAC [50] Terascale Simulation Tools and Technologies (TSTT) center [51] defined a mesh interface that enables the application to use several different mesh databases via language-interoperability provided by SIDL-Babel [52]. The FMDB was developed to be TSTT compliant and represent core functionalities of the TSTT meshing tools. The effort to run Mesquite [51] the mesh quality improvement library and the mesh adaptation procedure through the TSTT Mesh API of the FMDB is undergoing.

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

Acknowledgements This work is supported by the U.S. Department of Energy's Scientific Discovery through Advanced Computing (SciDAC) program as part of the Terascale Simulation Tools and Technology (TSTT) center.

References

1. Hughes TJR (2000) The Finite Element Method: Linear Static and Dynamic Finite Element Analysis. Dover Publication Inc..
2. Zienkiewicz OC (1997) The Finite Element Method 3rd ed. McGraw-Hill.
3. Shephard MS, Flaherty JE, Bottasso CL, de Cougny HL, Özturan C, Simone ML (1997) Parallel automated adaptive analysis *Parallel Computing* 23:1327-1347.
4. Beall MW, Shephard MS (1997) A general topology-based mesh data structure *Int. J. Numer. Meth. Engng* 40:1573-1596.
5. de Cougny HL, Shephard MS (1999) Parallel unstructured grid generation *CRC Handbook of Grid Generation* Thompson JF, Soni Bk, Wetherill NP, Eds. CRC Press Inc. Boca Raton p24.1-24.18.
6. de Cougny HL, Shephard MS, Özturan C (1996) Parallel three-dimensional mesh generation on distributed memory (MIMD) computers *Engineering with Computers* 12(2):94-106.
7. Lämmer L, Burghardt M (2000) Parallel generation of triangular and quadrilateral meshes *Advances in Engineering Software* 31(12):929-936.
8. Topping BHV, Cheng B (1999) Parallel and distributed adaptive quadrilateral mesh generation *Computers & Structures* 73:519-536.
9. Said R, Weatherill NP, Morgan K, Verhoeven NA (1999) Distributed parallel Delaunay mesh generation *Comp. Meth. Appl. Mech. Engng* 177:109-125.
10. Larwood BG, Weatherill NP, Hassan O, Morgan K (2003) Domain decomposition approach for parallel unstructured mesh generation *Int. J. Numer. Meth. Engng* 58(2)2:177-188.
11. Ollivier-Gooch C (2005) Generation and Refinement of Unstructured, Mixed-element meshes in parallel. http://tetra.mech.ubc.ca/GRUMMP/index.html.

12. Diekmann R, Preis R, Schlimbach F, Walshaw C (2000) Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM *Parallel Computing* 26:1555-1581.
13. Walshaw C, Cross M (2000) Parallel optimisation algorithms for multilevel mesh partitioning *Parallel Computing* 26(12):1635-1660.
14. de Cougny HL, Devine KD, Flaherty JE, Loy RM, Özturan C, Shephard MS (1995) Load balancing for the parallel solution of partial differential equations *Appl. Numer. Math* 16:157-182.
15. Teresco JD, Beall MW, Flaherty JE, Shephard MS (2000) A hierarchical partition model for adaptive finite element computations *Comput. Methods Appl. Mech. Engrg* 184:269-285.
16. Chen J, Taylor VE (2000) ParaPART: parallel mesh partitioning tool for distributed systems *Concurrency: Pract. Exper.* 12:111-123.
17. Kohn SR, Baden SB (2001) Parallel software abstractions for structured adaptive mesh methods *J. of Parallel and Distributed Computing* 61(6):713-736.
18. Balsara DS, Norton CD (2001) Highly parallel structured adaptive mesh refinement using parallel language-based approaches *Parallel Computing* 27:37-70.
19. MacNeice P, Olson KM, Mobarry C, Fainchtein RD, Packer C (2000) PARAMESH: A parallel adaptive mesh refinement community toolkit *Computer Physics Communications* 126(3):330-354.
20. Parashar M, Browne JC (2005) DAGH: Data Management for Parallel Adaptive Mesh Refinement Techniques. http://www.caip.rutgers.edu/∼parashar/DAGH.
21. Özturan C, de Cougny HL, Shephard MS, Flaherty JE (1994) Parallel adaptive mesh refinement and redistribution on distributed memory computers *Comp. Meth. Appl. Mech. Engng* 119:123-127.
22. Oliker L, Biswas R, Gabow HN (2000) Parallel tetrahedral mesh adaptation with dynamic load balancing *Parallel Computing* 26:1583-1608.
23. libMesh: Parallel Data Structures for Finite Element Computations (2005) University of Texas at Austin. http://www.cfdlab.ae.utexas.edu.
24. Park Y, Kwon O (2005) A parallel unstructured dynamic mesh adaptation algorithm for 3-D unsteady flows *Int. J. Numer. Meth. Fluids* 48:671-690.
25. de Cougny HL, Shephard MS (1999) Parallel refinement and coarsening of tetrahedral meshes *Int. J. Numer. Meth. Engng* 46:1101-1125.
26. Selwood PM, Berzins M (1999) Parallel unstructured tetrahedral mesh adaptation: algorithms, implementation and scalability *Concurrency: Pract. Exper.* V11(14) 863-884.
27. Remacle JF, Klaas O, Flaherty JE, Shephard MS (2002) A parallel algorithm oriented mesh database *Engineering with Computers* 18:274-284.
28. Biswas R, Oliker L (1994) A new procedure for dynamic adaptation of three-dimensional unstructured grids *Appl. Numer. Math* 13:437-452.
29. Karypis G, Schloegel K, Kumar V (1998) ParMETIS: Parallel Graph Partitioning & Sparse Matrix Ordering Library V. 2.0. University of Minnesota, Computer Science Dept. Army HPC Research Center Minneapolis MN.
30. Sandia National Laboratories (2005) Zoltan: data-management services for parallel applications. http://www.cs.sandia.gov/Zoltan.
31. Mäntylä M (1988) An Introduction to Solid Modeling. Computer Science Press Rockville Maryland.
32. Weiler KJ (1988) The radial-edge structure: a topological representation for non-manifold geometric boundary representations *Geometric Modeling for CAD Applications* p3-36.
33. Seol ES, Shephard MS, Musser DR (2005) FMDB: Flexible distributed Mesh DataBase. http://www.scorec.rpi.edu/FMDB.
34. Seol ES, Shephard MS (2005) General flexible mesh database for parallel adaptive analysis *Submitted to Int. J. Numer. Meth. Engng.*
35. Shephard MS (2000) Meshing environment for geometry-based analysis *Int. J. Numer. Meth. Engng* 47:169-190.
36. Alauzet F, Li X, Seol ES, Shephard MS (2005) Parallel anisotropic 3D mesh adaptation by mesh modification *In preparation to submit to J. Computing and Information Science.*
37. Pacheco PS (1997) Parallel Programming with MPI. Morgan Kaufmann Publisher.
38. Sgi Inc. (2005) Standard Template Library. http://www.sgi.com/tech/stl/stl_index.html.
39. Deitel & Deitel (2001) C++ How To Program 2^{nd} Ed.. Prentice Hall.
40. Vandevoorde D, Josuttis NM (2003) C++ Templates. Addison-Wesley.
41. Gamma E, Johnson R, Helm R, Vlissides JM (1994) Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
42. Alexandrescu A (2001) Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley.
43. The Message Passing Interface (MPI) standard library (2005) Argonne National Laboratory. http://www-unix.mcs.anl.gov/mpi.
44. Loy R (2000) Autopack User Manual. Science Division Argonne National Laboratory.
45. Booch G, Jacobson I, Rumbaugh J (1999) Unified Modeling Language for Object-Oriented Development Documentation Set Version 0.91 Addendum, Rational Software Corporation, Santa Clara, CA.
46. Li X, Shephard MS, Beall MW (2003) 3D anisotropic mesh adaptation by mesh modifications *Submitted to Comp. Meth. Appl. Mech. Engng.*
47. Li X, Shephard MS, Beall MW (2002) Accounting for curved domains in mesh adaptation *Int. J. Numer. Meth. Engng* 58:247-276.
48. Lee LQ, et al. (2004) Solving large sparse linear systems in end-to-end accelerator structure simulations. SLAC-PUB-10320 January.
49. Shephard MS, Seol ES, Wan J, Bauer AC (2004) Component-based adaptive mesh control procedures *Conference on Analysis, Modeling and Computation on PDE and Multiphase Flow* Stony Brook NY.
50. SciDAC: Scientific Discovery through Advanced Computing (2005) http://www.scidac.org.
51. The SciDAC Terascale Simulation Tools and Technology (TSTT) center (2005) http://www.tstt-scidac.org.
52. Center for component technologies for terascale simulation science (2005) SIDL/Babel user's guide. http://www.llnl.gov/CASC/components/babel.html.