The FMDB User's Guide

– WORKING DRAFT –

The FMDB Team
Scientific Computation Research Center
Rensselaer Polytechnic Institute

April 3, 2013

Contents

1 Introduction

1.1 Background and Motivation

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

Issues associated with supporting parallel adaptive analysis on a given unstructured mesh include dynamic mesh load balancing techniques [6, 8, 27, 28], and data structure and algorithms for parallel mesh adaptation [7, 12, 15, 17, 18, 19, 21].

The Flexible distributed Mesh DataBase (FMDB) is a distributed mesh data management system that is capable of a parallel mesh infrastructure capable of handling general non-manifold [13, 29] models and effectively supporting automated adaptive analysis.

This document describes an overview of the FMDB design, and presents its interface functions and how to use them to develop the applications.

1.2 Organization

Chapter 2 introduces the data sets involved with geometry-based analysis and the role of a topological mesh representation. Chapter 3 introduces distributed mesh data structure in accordance with a partition model. Chapter 4 describes the structure of FMDB interface. Chapter 5 presents the FMDB interface functions categorized based on the target of function and the information provided by them. Chapter 6 presents present example codes per category. Chapter 7 provides how to compile the FMDB and run the application on top of the FMDB either through its C++ language interface and ITAPS C language interface [10].

1.3 Nomanclature

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. (e.g., $\{M\{M^2\}\}$ is the set of all the faces in the mesh.)
V_i^d	the i^{th} entity of dimension[1] 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. (e.g., $\{M_3^1\{M^3\}\}$ are the mesh regions adjacent to mesh edge M_3^1.)
$V_i^d\{V^q\}_j$	the j^{th} entity in the set of entities of dimension q in model V that are adjacent to V_i^d. (e.g., $M_1^3\{M^1\}_2$ is the 2^{nd} edge adjacent to mesh 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 a hierarchy of domain decomposition.
$\mathscr{P}[M_i^d]$	set of part id(s) where entity M_i^d exists.

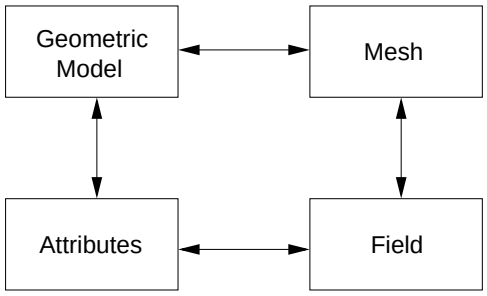


Figure 1: The relationship between components of the geometry-based analysis environment [3]

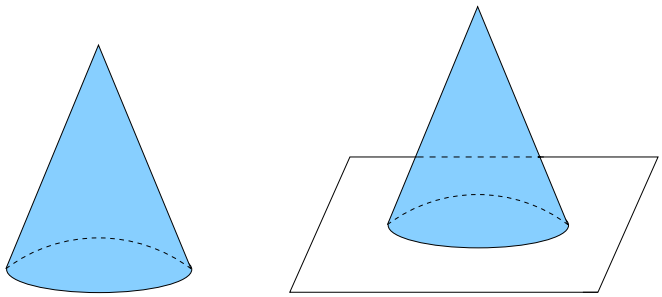


Figure 2: Example of (left) manifold and (right) non-manifold models

2 Geometry-based Analysis

2.1 Key Data

The structures used to support the problem definition, the discretization of the model and their interactions are central to mesh-based analysis methods like finite element and finite volumes. The geometry-based analysis environment consists of four parts: *the geometric model* which houses the topological and shape description of the domain of the problem, *attributes* describing the rest of information needed to define and solve the problem, *the mesh* which describes the discretized representation of the domain used by the analysis method, and *fields* which describe the distribution of solution tensors over the mesh entities [3, 14]. Figure 1 represents the general interactions between the four components.

2.1.1 Geometric model

The most common geometric representation is a boundary representation. A general representation of general non-manifold domains is the Radial Edge Data Structure [29]. Non-manifold models are common in engineering analyses. Simply speaking, non-manifold models consist of general combinations of solids, surfaces, and wires. Figure 2 illustrates examples of manifold and non-manifold model.

In the boundary representation, the model is a hierarchy of topological entities called regions, shells, faces, loops, edges, vertices, and in case of non-manifold models, use entities for vertices,

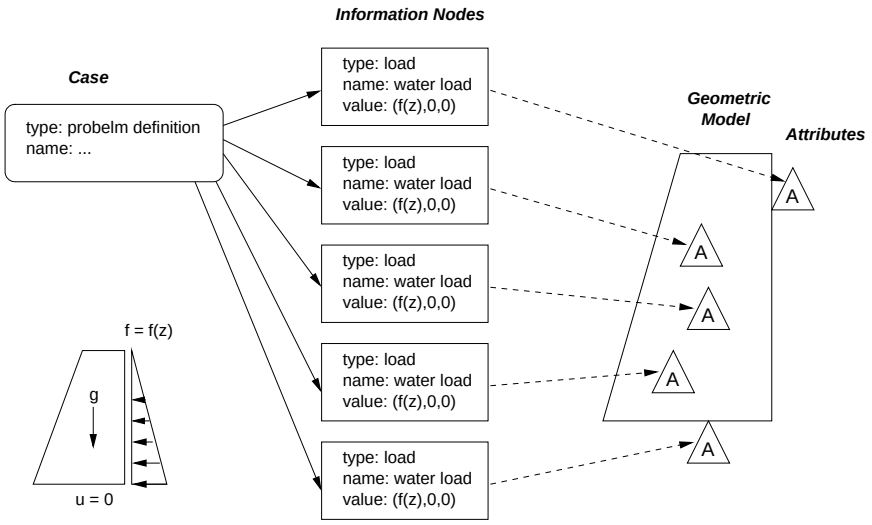


Figure 3: Example geometry-based problem definition [3]

edges, loops, and faces. The data structure implementing the geometric model supports operations to find the various model entities that make up a model, information about which model entities are adjacent to a given entity, operations relating to perform geometric shape queries, and queries about what attributes are associated with model entities.

2.1.2 Attribute

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

Figure 3 illustrates an example of a problem definition. The problem being modeled is a dam subjected to loads due to gravity and due to the water behind the dam. There is a set of attribute information nodes that are all under the attribute case for the problem definition. When this case is associated with the geometric model, attributes are created and attached to the individual model entities on which they act [3, 14]. The attributes are indicated by triangles with A's inside of them.

2.1.3 Mesh

A mesh is a geometric discretization of a domain. With restrictions on the mesh entity topology [4], a mesh is represented with a hierarchy of regions, faces, edges and vertices. Each mesh entity maintains a relation, called geometric classification [4, 24], to the model entity that it was created to partially represent. Geometric classification allows an understanding of which attributes (e.g. boundary conditions or material properties) are related to the mesh entities and the how the solution relates back to the original problem description, and is critical in mesh generation and adaptation [3, 4, 24]. More discussion on the mesh representation is presented in §2.2.

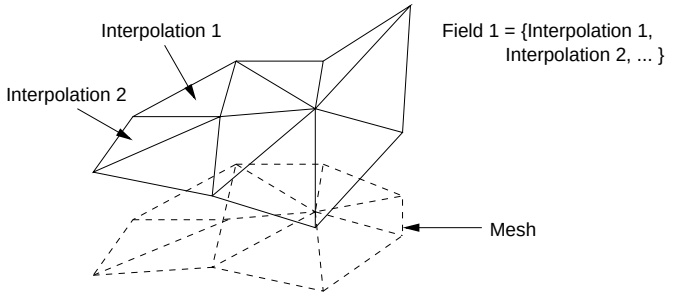


Figure 4: Representation of a field defined over a mesh [3]

In a geometry-based analysis environment, mesh data structures house the discretization of the domain, a mesh, and provide the mesh-level services to applications.

2.1.4 Field

A field describes the variation of solution tensors over the mesh entities discretizing one or more entities in a geometric model. The spatial variation of the field is defined in terms of mesh level distribution functions [3]. Figure 4 demonstrates the concept of a field written in terms of C^0 interpolating distribution functions.

2.2 General Topology-Based Mesh Data Structure

The mesh consists of a collection of mesh entities of controlled size, shape, and distribution. The relationships of the entities defining the mesh are well described by topological adjacencies, which form a graph of the mesh [4, 5, 9, 20]. A critical capability needed by automated, adaptive geometry-based analysis procedures is to manipulate the mesh of the analysis domain. A mesh data structure is a toolbox that provides the mesh-level services to the applications that create/use the mesh data. The differing needs of the applications dictate that the database be able to answer to the needed queries about the mesh. The five essential components of a general topology-based mesh data structure are: topological entities, geometric classification, adjacencies between entities [4], entity set and arbitrary user data attachable to the topological entities or entity sets, referred as *tag data* [10, 16].

2.2.1 Topological entities

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

Restrictions on the topology of a mesh are:

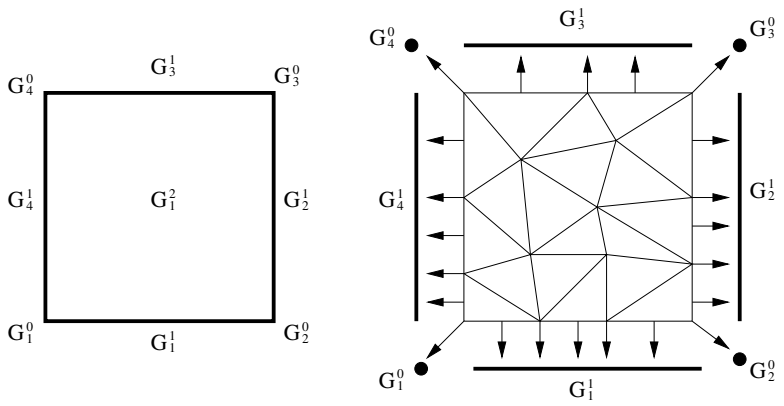


Figure 5: Example of simple model(left) and mesh(right) showing their association via geometric classification [14]

- Regions and faces have no interior holes.
- Each entity of order d in a mesh, M_i^d, may use a particular entity of lower order, p, M_j^p, $p < d$, at most once.
- For any entity M_i^d, there is the unique set of entities of order $d-1$, $\{M_i^d\{M^{d-1}\}\}$ that are on the boundary of M_i^d. (Note, based on mesh entity classification, it is possible to relax this restriction in the case of equal order classification [4])

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

2.2.2 Geometric classification

The linkage of the mesh to the geometric model is critical for mesh generation and adaptation procedures since it allows the specification of analysis attributes in terms of the original geometric model, the proper approximation of the geometry during mesh adaptation and supports direct links to the geometric shape information of the original domain need to improve geometric approximation and useful in p-version element integration [3, 4, 24].

The unique association of a mesh entity of dimension d_i, $M_i^{d_i}$, to the geometric model entity of dimension d_j, $G_j^{d_j}$, $d_i \leq d_j$, on which it lies is termed geometric classification, and is denoted $M_i^{d_i} \sqsubset G_j^{d_j}$, where the classification symbol, $\sqsubset$, indicates that the left hand entity, or a set of entities, is classified on the right hand entity. In Figure 5, a mesh of simple square model with entities labeled is shown with arrows indicating the classification of the mesh entities onto the model entities. All of the interior mesh faces, mesh edges, and mesh vertices are classified on the model face G_1^2.

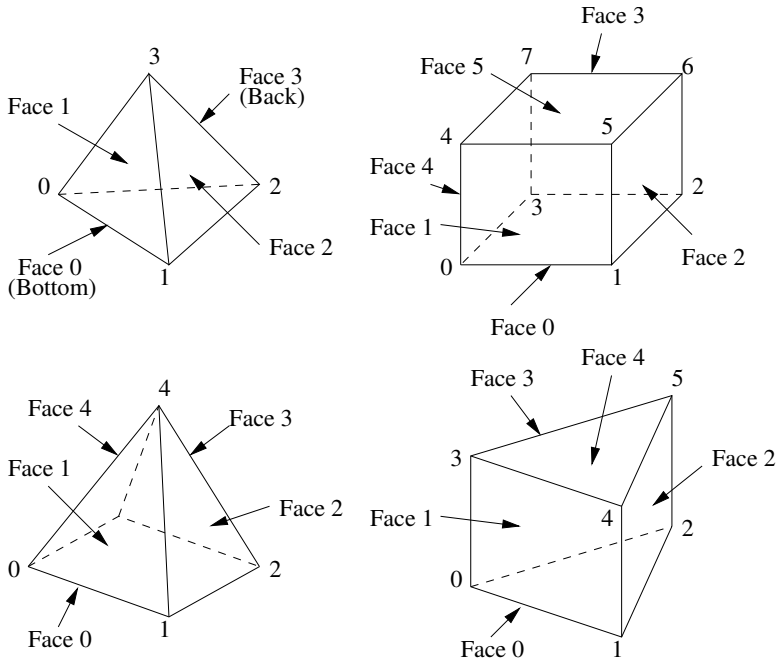


Figure 6: Vertex and face order on a region [14]

2.2.3 Adjacencies

Adjacencies describe how mesh entities connect to each other. For an entity of dimension d, first-order adjacency returns all of the mesh entities of dimension q, which are on the closure of the entity for a downward adjacency ($d > q$), or for which the entity is part of the closure for an upward adjacency ($d < q$). For denoting specific downward first-order adjacent entity, $M_i^d\{M^q\}_j$, the ordering conventions can be used to enforce the order. Figure 6, 7, and 8 illustrate a common canonical order of bounding entities. Figure 9 is an adjacency graph that depicts 12 first-order adjacencies possible in the mesh data structure where a solid box and a solid arrow denote, respectively, explicitly stored level of entities and explicitly stored adjacencies from outgoing level to incoming level.

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

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

2.2.4 Entity set

An entity set provides a mechanism for creating arbitrary groupings of entities for various purposes such as representing boundary layer, boundary condition and materials. Each entity set can be either of a set with unique entity or a list with insertion order preserved. The following are the

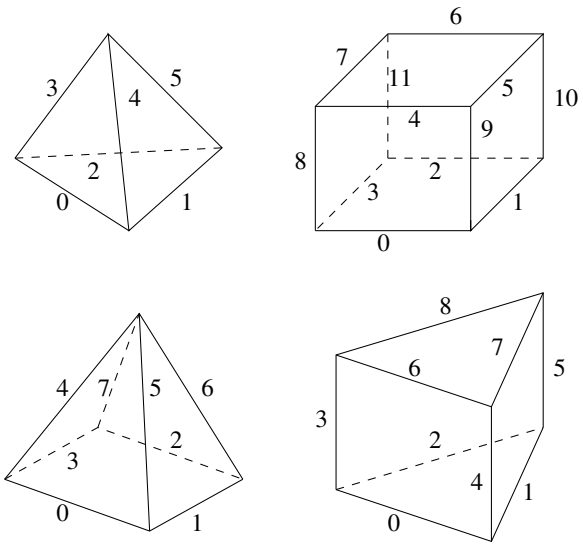


Figure 7: Edge order on a region [14]

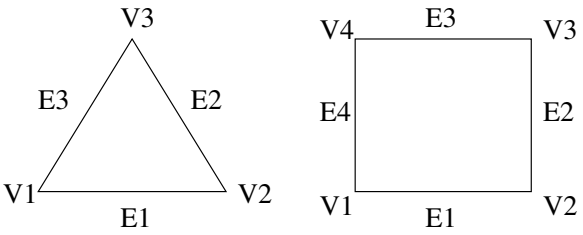


Figure 8: Edge order on a face [14]

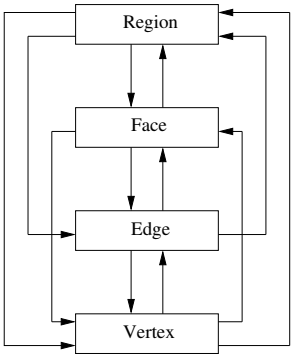


Figure 9: 12 adjacencies possible in the mesh representation [9]

functionalities of entity set to effectively support the application needs [10, 16].

- populating by addition or removal of entities from the set
- traversal through an iterator with various conditions such as topology, and type of the entity
- set boolean operations of subtraction, intersection, and union
- relationships among entity sets: subset, parent/child

In parallel computing environment, the mesh is distributed over multiple parts across the processes. Therefore, there are two kinds of set available in distributed mesh.

- mesh set: entity set created in a mesh. entities in the set can be in different part.
  - $L-SET$: a list type entity set created in mesh. Insertion order is preserved and an entity can be inserted multiple times.
  - $S-SET$: a set type entity set created in mesh. Insertion order is not preserved and an entity can be inserted at most once.
- part set or $P-SET$: a set type entity set created in part. Insertion order is preserved and only entities without higher order adjacency can be inserted.

For more information on distributed mesh representation, see Chapter 4.

2.2.5 Iterator

Iterators are a generalization of pointers which are objects that point to other objects. Iterators are often used to iterate over a range of objects: 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 [25].

Various kinds of iterators are desirable for efficient mesh entity traversal with various conditions such as entity dimension, entity topology, geometric classification. Furthermore, the iterator validity shall be guaranteed with mesh modification through entity creation/deletion.

2.2.6 Tag

A tag is a container of arbitrary data attachable to meshes, entities, and entity sets. Different values of a particular tag can be associated with mesh, entity, or entity set [10, 16].

For efficient manipulation of tags and their association with meshes, entities and entity sets, tags consist of the following data.

- tag name: character string for identifying tag

- tag data: data stored in the tag
- tag type: data type for tag data
  For better performance and management, five specialized tag types, integer, double, mesh entity, entity set and byte type data are supported through interface. If the tag data consists of multiple units (e.g. array of integer data), the size of tag data in byte and the number of units are needed for efficient tag manipulation.
- tag size: the number of units of *tag type* in tag data
- tag byte: the size of tag data in bytes

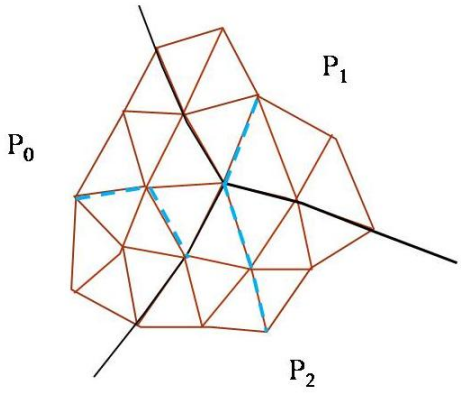


Figure 10: Distributed mesh on three processes P_0, P_1 and P_2 with two parts per each process

3 Distributed Mesh Management

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, the distributed mesh data structure must support (*i*) efficient communication between entities duplicated over multiple processes, (*ii*) migration of entities between processes, and (*iii*) dynamic load balancing.

This chapter presents the concept and functionalties of distributed mesh management. §3.3 describes a partition model that is developed in FMDB for the purpose of effectively meeting the specific functionalities of distributed meshes [22, 23]. The readers not interested in the internal design and implementation of the distributed meshes in FMDB might skip §3.3

3.1 Distributed Mesh Representation

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

Definition 3.1 *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 part id within a process can be given.

Each part will be treated as a serial mesh with the addition of mesh part boundaries to describe groups of mesh entities that are on inter-part boundaries. Mesh entities on part boundaries are duplicated on all parts on which they are used in adjacency relations. Mesh entities not on the part boundary exist on only one part and referred as *internal entities*. In implementation, for effective manipulation of multiple parts on each process, a single mesh data is defined on each process so

multiple parts are contained in the mesh data where the mesh data is assigned to a process. The mesh data defined on each process is referred as *mesh instance*. Figure 10 depicts a mesh that is distributed on 6 parts where the mesh instance on each process has two parts respectively. The dashed lines are *part boundaries* within a process and the solid black lines are *part boundaries* between the processes. The vertices and edges on part boundaries are duplicated on multiple parts.

In order to simply denote a set of parts where a mesh entity physically exist, termed *residence part set*, we define an operator $\mathscr{P}$.

Definition 3.2 *Residence part set operator* $\mathscr{P}[M_i^d]$
An operator that returns a set of global part id(s) where M_i^d exists.

Definition 3.3 *Residence part equation of* M_i^d
If $\{M_i^d\{M^q\}\} = \emptyset$, $d < q$, $\mathscr{P}[M_i^d] = \{p\}$ where p is the id of a part where M_i^d exists. Otherwise, $\mathscr{P}[M_i^d] = \cup\, \mathscr{P}[M_j^q \mid M_i^d \in \{\partial(M_j^q)\}]$.

For any entity M_i^d not on the part boundary of any higher order mesh entities and on part 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 part set is determined simply to be the part where it resides. If entity M_i^d is on the boundary of other higher order mesh entities, M_i^d is duplicated on multiple parts depending on the residence part set of its bounding entities since M_i^d exists wherever a mesh entity it bounds exists.

Therefore, the residence part set of M_i^d is the union of residence part set of all entities that it bounds. For a mesh topology where the entities of order $d > 0$ are bounded by entities of order $d-1$, $\mathscr{P}[M_i^d]$ is determined to be {p} if $\{M_i^d\{M_k^{d+1}\}\} = \emptyset$. Otherwise, $\mathscr{P}[M_i^d]$ is $\cup\, \mathscr{P}[M_k^{d+1} \mid M_i^d \in \{\partial(M_k^{d+1})\}]$. For instance, for the 3D non-manifold mesh depicted in Figure 11, 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 part set of M_1^0 are $\{P_0, P_1, P_2\}$ since the union of residence part set of its bounding edges, $\{M_1^1, M_2^1, M_3^1, M_4^1, M_5^1, M_6^1\}$, are $\{P_0, P_1, P_2\}$.

To migrate mesh entities to other parts, the destination part id's of mesh entities must be specified before moving the mesh entities. The residence part set equation implies that once the destination part id of a M_i^d that is not on its 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 part id in the mesh migration procedure.

Definition 3.4 *Partition object*
The basic unit to which a destination part 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 or regions, and mesh vertices not bounded by any mesh edges, faces or regions. A set of unique mesh entities refered as entity set can also be a partition object if designated to be a migration unit.

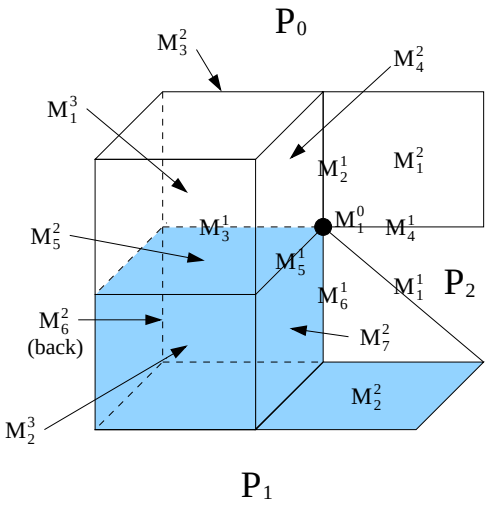


Figure 11: Example 3D mesh distributed on three parts

In case of a manifold model, partition objects are all mesh regions in 3D and all mesh faces in 2D. In case of a non-manifold model, the careful lookup for entities not being bounded is required over the entities of one specific dimension. For example, partition objects of the mesh in Figure 11 are M_1^1, M_1^2, M_2^2, M_1^3, and M_2^3.

3.2 Functional Requirements

3.2.1 Communication links

Mesh entities on the part boundaries (shortly, part boundary entities) must be aware of where they are duplicated.

Definition 3.5 *Remote part*
Non-self part[2] where a mesh entity is duplicated.

Definition 3.6 *Remote copy*
Non-owned part boundary entities, in other words, the memory location of a mesh entity duplicated on remote part.

3.2.2 Ownership

In parallel adaptive analysis, the mesh and its partitioning can change thousands of time during the simulation [1, 7, 15, 26]. Therefore, at the mesh functionality level, an efficient mechanism to update the mesh partitioning and keep the links between parts updated are mandatory to achieve scalability.

[2] A part that is not in the current local part

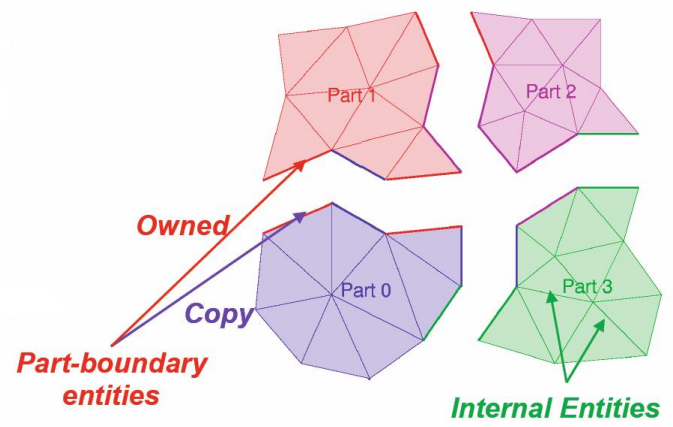


Figure 12: A distributed mesh on four processes with one part per process [10]

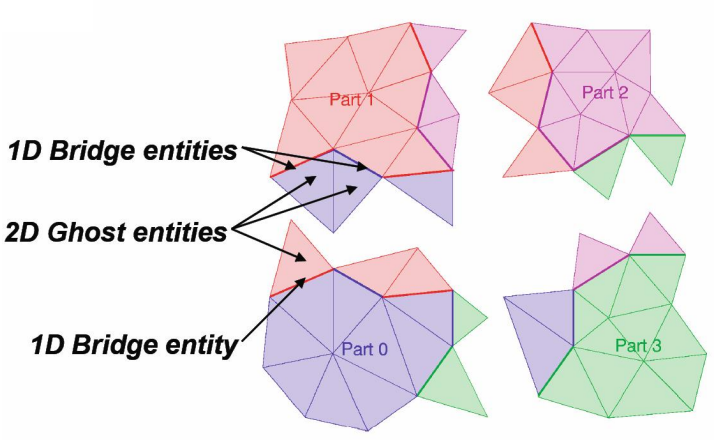


Figure 13: A distributed mesh on four parts with ghost entities [10]

For entities on part boundaries duplicated on multiple parts, it is beneficial to assign a specific part as the owner with charge of modification, communication or computation of the copies. For the purpose of simple denotation, a part bounday entity owned by the self part is referred as *owner* of other entities copied on other parts.

Figure 12 depicts a mesh that is distributed on four processes with a single part per process. Entities on part boundaries are either of owner or copies. Internal entities are owners.

3.2.3 Ghosting

To avoid communications between the parts, it is beneficial to support the ability to have a copy of non-part boundary entities on other part, referred as *ghosting* [10].

Definition 3.7 *Ghost copy or ghost entity*
Non-owned, non-part-boundary entity in a part

Figure 13 depicts a distributed mesh on four parts with ghost entities along the part boundaries. Similar to ownership of part boundary entities, the original ghosted entity is designated as *owner* of all ghosted copies on other parts.

To create ghost entities, three pieces of information are needed.

- ghost (entity) dimension
- bridge (entity) dimension
- number of layers

Ghost entities are specified through a *bridge* dimension. The number of layers is measured from the inter-part interfaces. For example, to get two layers of region entities in the ghost layer, measured from faces on the interface, ghost dimention, bridge dimension, and the number of layers shall be, respectively, 3, 2, 2. The number of layers specified is with respect to the global mesh, that is, ghosting may extend beyond a single neighboring process if the number of layers is high.

In Figure 13, ghost dimension, bridge dimension and the number of layers are 2, 1, and 1, respectively.

3.2.4 Migration

For effective management of distributed mesh with multiple parts per process, the following migration procedures are needed.

- Migrating entities and entity sets between parts with tag
- Migrating whole parts between processes
- Redistributing mesh. For example, splitting n part mesh to m parts, $n \neq m$, to load the mesh on an m process machine.

3.3 A Partition Model

To meet the goals and functionalities of distributed meshes, a partition model has been developed between the mesh and the geometric model. As illustrated in Figure 14, 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.

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

The partition model introduces a set of topological entities that represents the collections of mesh entities based on their location with respect to the partitioning. Grouping mesh entities to define

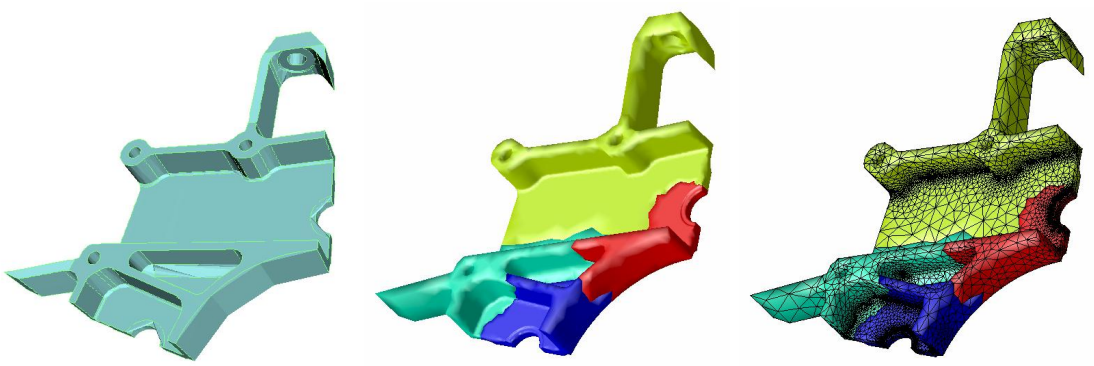


Figure 14: Hierarchy of domain decomposition: geometry model, partition model, and distributed mesh on 4 processes

a partition entity can be done with multiple criteria based on the level of functionalities and needs of distributed meshes. These constructs are consistent with the ITAPS iMeshP specification [10].

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

Definition 3.8 *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 is uniquely determined by $\mathscr{P}$.

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

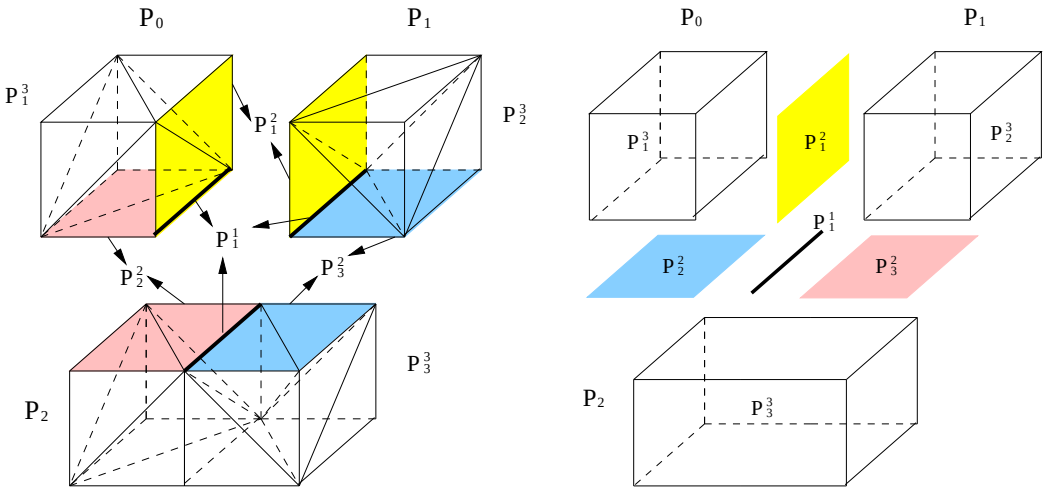
Definition 3.9 *Partition classification*

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 $M_i^{d_i} \sqsubset P_j^{d_j}$.

Definition 3.10 *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 $\mathrm{RC}(P_j^d) = \{M_i^d \mid M_i^d \sqsubset P_j^d\}$.

Figure 15 illustrates a 3D distributed mesh with mesh entities labeled with arrows indicating the partition classification of the entities onto the partition model entities and its associated partition



(a) distributed mesh (b) partition model

Figure 15: Distributed mesh and its association with the partition model via partition classifications

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 the partition regions. Note the reverse classification returns only the same order mesh entities. The reverse partition classification of P_1^1 returns mesh edges located on the thick black lines, and the reverse partition classification of partition face P_i^2 returns mesh faces on the shaded planes.

4 Interface Structure

The application can take advantage of FMDB through either of its C++ language interface or ITAPS C language interface [10]. This chapter presents the general information on FMDB C++ interface and the next chapter presents the interface specification. See reference [10] for ITAPS interface specifications.

4.1 Interface Header File

All interface functions are declared in *FMDB.h*.

4.2 Function Categories

The function name begins with *FMDB* followed by the operation target and operation description. For example, the function which returns the number of mesh entities in a part with specific type is named *FMDB_Part_GetNumEntType*. The operation target is the following:

- system: the function is not performed on any mesh related data and is system wide operation.
- tag: the function name contains Tag. The function is performed on tag.
- mesh: the function name contains $Mesh$. The function is performed on mesh or all parts contained in the mesh.
- part: the function name contains $Part$. The function is performed on a specific part.
- entity: the function name contains Ent. The function is performed on a specific mesh entity.
- entity set: the function name contains $EntSet$. The function is performed on a (any kind) specific entity set.
- iterator: the function name contains $Iter$. The function is performed on a specific iterator (part iterator or entity set iterator) with designated condition such as entity type, topology, etc.

We group the interface functions based on the information provided by them.

- service: start/stop the SCOREC S/W
- tag management: create/delete a tag, get tag information (size, type, name, etc.)
- mesh/part
  - mesh/part management: create/delete a mesh, read/write a mesh, create/delete a part
  - mesh/part information: parts, entities, dimension, associated geometry model

  - part traversal: entity iterator of specific type/topology, reverse classification iterator, and part boundary entity iterator
  - part tag: set/get tag to/from part, delete tag, check tag data existence
  - distributed mesh functionality: load balancing, migration, create/delete ghost entities
- entity
  - entity management: create/delete an entity, entity search
  - entity information: general information (e.g. type, topology), geometric classification, entity adjacency (first and second order), direction of one-level downward adjacent entity
  - entity tag: set/get tag to/from entity, delete tag, check tag data existence
  - parallel entity: owning part, entity status (internal, owner, remote copy, ghost copy, etc.), set/get remote copy, get remote/ghost copy information, set/get entity weight
- entity set
  - entity set management: create/delete an entity set, add/remove an entity to/from the entity set
  - entity set information: get size and type, check entity existence
  - entity set traversal: entity set iterator with specific type/topology
  - entity set tag: set/get tag to/from entity set, delete tag, check tag data existence
  - parallel entity set: set/get entity set weight

4.3 Parameter Type

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 i means a pointer to a mesh data structure on process i, in which all parts on process 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. The predefined data type has a prefix p to indicate the pointer data type.

The following are predefined data types used in the interface function parameters.

pGeomMdl	geometric model instance
pGeomEnt	geometric entity handle
pPart	part handle
pMeshMdl	mesh instance
pMeshEnt	mesh entity handle
pMeshVtx	mesh vertex handle
pMeshEdge	mesh edge handle
pMeshFace	mesh face handle
pMeshRgn	mesh region handle
pNode	higher order node
pEntSet	entity set handle (L-set, S-set or P-set)
pPartSet	part set handle
pTag	tag handle
pMeshSetIter	iterator traversing over entity sets in mesh
pPartSetIter	iterator traversing over entity sets in part
pPartEntIter	iterator traversing over entities in part
pSetEntIter	iterator traversing over entities in entity set
MigrCB	(supported for backward compatibility only). A deprecated callback routine consisting of *pMigrPackFn* and *pMigrUnpackFn* for secifying how to process tag attached to the mesh entities and entity sets along with the migration or ghosting.

To enhance the code readibility, the enumeration types, *TagType*, *EntType*, *EntTopo*, *EntStatus* are definded for tag data type, entity type (dimension), entity topology, and entity status, respectively. Since *TagType* is common to all s/w libraries, it is defined in SCUtil.h which is an api header of SCUtil (Scientific Computation Utility) library.

```
enum SCUTIL_TagType {
    SCUtil_BYTE       = 0,
    SCUtil_INT,      /* 1 */
    SCUtil_DBL,      /* 2 */
    SCUtil_ENT,      /* 3 */
    SCUtil_SET       /* 4 */
}

enum FMDB_EntType {
    FMDB_VERTEX     = 0,
    FMDB_EDGE,     /* 1 */
    FMDB_FACE,     /* 2 */
    FMDB_REGION,   /* 3 */
    ALLTYPE        /* 4 */
}
```

```
enum FMDB_EntTopo {
    FMDB_POINT = 0,    /* 0 - a general 0D entity  */
    FMDB_LINE,         /* 1 - a general 1D entity  */
    FMDB_POLYGON,      /* 2 - a general 2D element (NOT SUPPORTED) */
    FMDB_TRI,          /* 3 - a three-sided 2D element */
    FMDB_QUAD,         /* 4 - a four-sided 2D element  */
    FMDB_POLYHEDRON,   /* 5 - a general 3D element (NOT SUPPORTED)
    FMDB_TET,          /* 6 - a four-sided, 3D element whose faces are triangles */
    FMDB_HEX,          /* 7 - a six-sided 3D element whose faces are quadrilaterals */
    FMDB_PRISM,        /* 8 - a five-sided 3D element which has three quadrilateral
                           faces and two triangular faces  */
    FMDB_PYRAMID,      /* 9 - a five-sided 3D element which has one quadrilateral
                           face and four triangular faces */
    FMDB_SEPTA,        /* 10 - a hexahedral entity with one collapsed edge
                            (NOT SUPPORTED) */
    FMDB_ALLTOPO       /* 11 - all topologies */
}

enum FMDB_EntStatus {
    FMDB_INTER  = 0,
    FMDB_BDRY, /* 1 */
    FMDB_GHOST /* 2 */
}

enum FMDB_SetType {
    FMDB_SSET     = 0,
    FMDB_LSET,  /* 1 */
    FMDB_PSET   /* 2 */
}
```

Currently, polygon (a general 2D element), polyhedron (a general 3D element), and septahedron (a hexahedral entity with one collapsed edge) topologies are not supported in FMDB. They can be easily added based on a given specification of the desired representation.

4.4 Return Value

Except for the functions returning ID of entity set or entity, all FMDB functions return an integer value indicating if the operation succeeded or failed. If the function succeeded, it returns 0, otherwise, it returns a positive integer error code defined in *SCUtil.h*.

```
enum SCUTIL_Err {
    SCUtil_SUCCESS = 0,       // no error
    SCUtil_MESH_ALREADY_LOADED,
```

```
    SCUtil_FILE_NOT_FOUND,
    SCUtil_FILE_WRITE_ERROR,
    SCUtil_NULL_ARRAY,
    SCUtil_BAD_ARRAY_SIZE,
    SCUtil_BAD_ARRAY_DIMENSION,
    SCUtil_INVALID_ENT_HANDLE,
    SCUtil_INVALID_ENT_COUNT,
    SCUtil_INVALID_ENT_TYPE,
    SCUtil_INVALID_ENT_TOPO
    SCUtil_BAD_TYPE_AND_TOPO,
    SCUtil_ENTITY_CREATION_ERROR,
    SCUtil_INVALID_TAG_HANDLE,
    SCUtil_TAG_NOT_FOUND,
    SCUtil_TAG_ALREADY_EXISTS,
    SCUtil_TAG_IN_USE,
    SCUtil_INVALID_SET_HANDLE,
    SCUtil_INVALID_ITERATOR,
    SCUtil_INVALID_ARGUMENT,
    SCUtil_MEMORY_ALLOCATION_FAILED,
    SCUtil_NOT_SUPPORTED,
    SCUtil_FAILURE,
    SCUtil_INVALID_MESH_INSTANCE,
    SCUtil_INVALID_GEOM_MODEL,
    SCUtil_INVALID_PART_HANDLE,
    SCUtil_INVALID_PART_ID,
    SCUtil_INVALID_SET_TYPE,
    SCUtil_ENTITY_NOT_FOUND,
    SCUtil_ENTITY_ALREADY_EXISTS,
    SCUtil_REMOTE_NOT_FOUND,
    SCUtil_GHOST_NOT_FOUND,
    SCUtil_CB_ERROR
}
```

5 Interface Functions

5.1 System-Level Functions

```
int SCUTIL_Init (MPI_Comm /* in */ comm)
```

Given MPI communicator, initialize parallel services pertinent to FMDB including MPI. In serial, set *NULL* for the input.

```
int SCUTIL_Finalize ()
```

Finalize parallel services and clean the memory.

```
int SCUTIL_SplitComm (int /* in */ numProc)
```

Given the number of processes, *numProc*, where the total number of processes in MPI communicator modular *numProc* equals zero, group *numProc* processes so each group can load a mesh and perform migration, load balancing, modification, and file i/o independently. If the number of processes modular *numProc* is not equal to zero, the error code *SCUtil_INVALID_ARGUMENT* is returned.

```
int SCUTIL_SetComm (MPI_Comm /* in */ comm)
```

(*Temporarily Unavailable*) Given a communicator, set the communicator.

```
int SCUTIL_GetComm (MPI_Comm /* out */ *comm)
```

Get the communicator.

```
int SCUTIL_CommSize ()
```

Return the number of processes in communicator.

```
int SCUTIL_CommRank ()
```

Return the MPI rank in communicator. Rank starts from 0.

```
int SCUTIL_CommGrp ()
```

Return the group ID in communicator. Group ID starts from 0. In case of 12 processes which are splitted in three groups by *SCUTIL_SplitComm*(3), *SCUTIL_CommGrp* returns 0 on processes from MPI rank 0 to 3, 1 on processes from MPI rank 4 to 7, and 2 from on processes from MPI rank 8 to 11.

```
int SCUTIL_GrpSize ()
```

Return the number of processes in process group. In case of 12 processes which are splitted in three groups by *SCUTIL_SplitComm*(3), *SCUTIL_GrpSize* returns 4 on any process.

```
int SCUTIL_GrpRank ()
```

Return the rank in process group. Rank starts from 0. In case of 12 processes which are splitted in three groups by *SCUTIL_SplitComm*(3), *SCUTIL_GrpRank* returns 0 on processes with MPI rank 0, 4, 8, 1 on processes with MPI rank 1, 5, 9, 2 on processes with MPI rank 2, 6, 10, and 3 on processes with MPI rank 3, 7, 11.

```
int SCUTIL_Sync ()
```

Synchronize all processes in communicator

```
int SCUTIL_GetCurMem (double*  /* out */  mem)
```

Get the memory increase (MB) since *SCUTIL_ResetRsrc()* call. In parallel, the sum of memory increase of all processes in communicator.

```
int SCUTIL_DspCurMem (const char*  /* in */  msg)
```

Display input string *msg* and the memory increase (MB) since *SCUTIL_ResetRsrc()* call. In parallel, the sum of memory increase of all processes in communicator is displayed.

```
int SCUTIL_ResetRsrc ()
```

Reset time and memory counter.

```
int SCUTIL_GetRsrcDiff (
        double*  /* out */  time,
        double*  /* out */  mem)
```

Get time (sec) and memory (MB) increase since *SCUTIL_ResetRsrc()* call.

```
int SCUTIL_DspRsrcDiff (const char*  /* in */  msg)
```

Display input string *msg*, the time (sec) and memory (MB) increase since *SCUTIL_ResetRsrc()* call.

```
int SCUTIL_DspRsrc (
        const char*  /* in */  msg,
        double  /* in */  time,
        double  /* in */  mem)
```

Display input string *msg*, time (sec) and memory (MB).

```
int SCUTIL_DspSysInfo ()
```

Display system information such as hostname, os, etc. *(Example)* Linux node10.borg.scorec.rpi.edu 2.6.9-89.ELsmp SMP Mon Jun 22 12:31:33 EDT 2009 x86_64.

5.2 Tag Management

```
int FMDB_Mesh_CreateTag (
        pMeshMdl  /* in */  mesh,
        const char*  /* in */  tagName,
        int  /* in */  tagType,
        int  /* in */  tagSize,
        pTag&  /* out */  tag);
```

Given mesh instance, tag name, tag size, and tag type (SCUtil_BYTE: byte, SCUtil_INT: integer, SCUtil_DBL: double, SCUtil_ENT: mesh entity, SCUtil_SET: entity set), create a unique tag in mesh instance.

```
int FMDB_Mesh_DelTag (
        pMeshMdl  /* in */  mesh,
        pTag  /* in */  tag,
        int  /* in */  forceDel);
```

Given a mesh instance and tag handle, destroy the tag from the mesh instance. If *forceDel* is non-zero, both tag and tag data associated with the tag is deleted. If *forceDel* is zero, the tag is deleted only if there's no tag data associated with the tag.

Note: Since FMDB doesn't keep track of tag data attachment, forced tag deletion is a mesh size function.

```
int FMDB_Mesh_FindTag (
        pMeshMdl  /* in */  mesh,
        const char*  /* in */  tagName,
        pTag&  /* out */  tag)
```

Given mesh instance and character string, get the handle of an existing tag in the mesh. If no tag exists with the specified name, *tag* is *null* and error code *SCUtil_TAG_NOT_FOUND* is returned.

```
int FMDB_Mesh_HasTag (
        pMeshMdl  /* in */  mesh,
        pTag  /* in */  tag,
        int*  /* out */  has)
```

Given mesh instance and tag handle, if the tag exists in the mesh, *has* is 1, otherwise, 0.

```
int FMDB_Mesh_GetTag (
        pMeshMdl  /* in */  mesh,
        std::vector<pTag>&  /* out */  vecTag)
```

Given mesh instance, get the vector *vecTag* filled with all tag handles created in the mesh.

```
int FMDB_Tag_InUse (
        pMeshMdl  /* in */  mesh,
        pTag  /* in */  tag,
        int*   /* out */  use)
```

Given mesh instance and tag handle, if the tag is in used for any part, entity or set, *use* is 1, otherwise 0.

```
int FMDB_Tag_GetType (
        pTag  /* in */  tag,
        int*  /* out */  tagType)
```

Given a tag handle, get the data type of the tag (SCUtil_BYTE: byte, SCUtil_INT: integer, SCUtil_DBL: double, SCUtil_ENT: mesh entity, SCUtil_SET: entity set).

```
int FMDB_Tag_GetName (
        pTag  /* in */  tag,
        char*  /* out */  tagName,
        int   /* in */  name_length)
```

Given a tag handle, get the name for it.

```
int FMDB_Tag_GetSize (
        pTag  /* in */  tag,
        int*  /* out */  tagSize)
```

Given a tag handle, get the number of units in tag of tag data type

```
int FMDB_Tag_GetByte (
        pTag  /* in */  tag,
        int*  /* out */  tagByte)
```

Given a tag handle, get the size of tag data in byte.

5.3 Mesh/Part Functions

5.3.1 Mesh/Part management

```
int FMDB_Mesh_Create (
        pGeomMdl  /* in */  geom,
        pMeshMdl&  /*out */  mesh)
```

Given a geomeric model, create a mesh instance. If no geometry model is provided (*geom* is *null*), a mesh model is generated automatically by the FMDB. By default, one empty part is created in the mesh instance.

```
int FMDB_Mesh_Del (
        pMeshMdl  /*inout*/  mesh)
```

Delete the mesh instance and deallocate the memory.

```
int FMDB_Mesh_LoadSerialFromFile (
        pMeshMdl  /*in*/  mesh,
        const char*  /* in */  fileName)
```

Load a serial mesh onto the first part of the master process from a mesh file. The mesh format supported are *sms*, *vtk*, *ncdf*(ascii), or *nc* (binary). To use the binary ncdf mesh file I/O (.nc), NCDF library should be linked. If the file extension doesn't match, the mesh is read in *sms* file format.

```
int FMDB_Mesh_LoadFromFile (
        pMeshMdl  /*in*/  mesh,
        const char*  /* in */  fileName,
        int  /* in */  distributed)
```

Load a mesh from a mesh file. The mesh format supported are *sms*, *vtk*, *ncdf*(ascii), or *nc* (binary). To use the binary ncdf mesh file I/O (.nc), NCDF library should be linked. If the file extension doesn't match, the mesh is read in *sms* file format. If *distributed* is 0, mesh data from *meshFile* is loaded onto the first part of the master process and partitioned. If *distributed* is non-zero, mesh data loaded from $meshFile_i$ is stored in the $part_i$, where i is the global part id.

```
int FMDB_Mesh_WriteToFile (
        pMeshMdl  /* in */  mesh,
        const char*  /* in */  fileName,
        int  /* in */  distributed)
```

Write a mesh to mesh file(s). The file name extension sould be one of *sms*, *vtk*, *ncdf* (ascii), or *nc* (binary). To use the binary ncdf mesh file I/O (.nc), NCDF library should be linked. If file extension doesn't match, the mesh is written in *sms* mesh format. If there are multiple parts in all, one mesh file is created per non-empty part with global part id appended to the file name. For example, if a mesh on three processes has three parts (single part per process) and *mesh.sms* is given as *fileName*, the mesh data are written into three files; *mesh0.sms*, *mesh1.sms* and *mesh2.sms*. If *distributed* is non-zero, mesh data in $part_i$ is written to $meshFile_i$ where i is the global part id. If *distributed* is zero, only mesh data on $part_0$ is written to *meshFile*.

FMDB v1.3.8 or higher version support tag, entity set file i/o in sms format.

```
int FMDB_Mesh_DelPart (
        pMeshMdl  /* in */  mesh,
        pPart  /* in */ part)
```

Given a mesh instance and an empty part handle, delete the part from the mesh instance.

```
int FMDB_Mesh_IsPtnSynced(
        pMeshMdl /* in */ mesh,
        int* /* out */ sync);
```

Given a mesh instance, check if the global part ids and partition classification and owner part of all m esh entities are synchronized.

```
int FMDB_Mesh_SyncPtn (pMeshMdl mesh)
```

Given a mesh instance, synchronize the global part ids and partition classification and owner part of all mesh entities. It's strongly recommended to call this function whenever the mesh partitioning topology is changed by *FMDB_Mesh_SetNumPart*, or manual part boundary update via *FMDB_Ent_SetRmt*, *FMDB*_Ent_*RmvRmt*, and *FMDB_Ent_ClrRmt*.

5.3.2 Mesh/Part information

```
int FMDB_Mesh_SetLastError (
        pMeshMdl  /* in */  mesh,
        int  /* in */  error_code)
```

Set the last error code occurred in the mesh.

```
int FMDB_Mesh_GetLastError (
        pMeshMdl  /* in */  mesh,
        int*  /* out */  error_code)
```

Get the last error code occurred in the mesh.

```
int FMDB_Mesh_IsEmpty (
        pMeshMdl  /* in */  mesh,
        int*  /* out */ empty)
```

Given a mesh, get whether the mesh is empty of not. If empty mesh, *empty* is non-zero, otherwise, 0.

```
int FMDB_Mesh_GetGeomMdl (
        pMeshMdl  /* in */  mesh,
        pGeomMdl&  /* out */  geom)
```

Given a mesh instance, get the geometry model instance associated with.

```
int FMDB_Mesh_GetDim (
        pMeshMdl  /* in */  mesh,
        int*  /* out */  dim)
```

Given a mesh instance, get the dimension.

```
int FMDB_Mesh_SetNumPart (
        pMeshMdl  /* in */  mesh,
        int  /* in */  numPartPerMeshOnProc)
```

Given a mesh instance and the desired number of parts per mesh on each process *numPartPerMeshOnProc*, set the number of parts-per-process across all processes. New parts are created on mesh if the number of existing parts is less than *numPartPerMeshOnProc*.

```
int FMDB_Mesh_GetNumPart (
        pMeshMdl  /* in */  mesh,
        int*  /* out */  numPart)
```

Given a mesh instance, get the number of parts created in mesh.

```
int FMDB_Mesh_GetPart (
        pMeshMdl  /* in */  mesh,
        int  /* in */  ith,
        pPart  /* out */  part)
```

Given a mesh instance and the order *ith*, get $i'th$ local part in mesh.

```
int FMDB_Mesh_SetEntID (
        pMeshMdl  /* in */ mesh,
        int  /* in */  type)
```

Given a mesh instance and entity type (FMDB_VERTEX, FMDB_EDGE, FMDB_FACE, FMDB_REGION, FMDB_ALLTYPE), set unique integer id to all mesh entities of type. The unique entity id is not maintained during mesh modification such as mesh adaptation or migration.

```
int FMDB_Mesh_DelEntID (
        pMeshMdl  /* in */ mesh,
        int  /* in */  type)
```

Given a mesh instance and entity type (FMDB_VERTEX, FMDB_EDGE, FMDB_FACE, FMDB_REGION, FMDB_ALLTYPE), delete unique integer id.

```
int FMDB_Mesh_GetAllPart (
        pMeshMdl  /* in */  mesh,
        int  /* in */  processRank,
        std::vector<pPart>&  /* inout*/  vecPart)
```

Given a mesh instance and a process rank, get the container *vecPart* filled with part handles on the process.

```
int FMDB_Mesh_GetAllPartID (
        pMeshMdl  /* in */  mesh,
        std::vector<int>&  /* inout*/ vecPartID)
```

Given a mesh instance, get the container *vecPartID* filled with id's of part handles on the process.

```
int FMDB_Mesh_GetNumSet (
        pMeshMdl  /* in */  mesh,
        int*  /* out */  numEntSet)
```

Given a mesh, get the number of mesh set (LSET or SSET) created. Note PSET is not counted.

```
int FMDB_Mesh_GetSet (
        pMeshMdl  /* in */  mesh,
        vector<pEntSet>&  /* inout */  vecEntSet)
```

Given a mesh, get the vector container *vecEntSet* filled with mesh set (LSET or SSET) handles created in a mesh.

```
int FMDB_Mesh_GetNborProcID (
        pMeshMdl  /* in */  mesh,
        std::vector<int>&  /* out */  procRank)
```

Given a mesh, get the integer container *procRank* filled with process ranks neighboring the local process. A process i neighbors process j if they share mesh entities on part boundary between parts located in process i and process j.

```
int FMDB_Part_ID(pPart  /* in */  part)
```

Given a part handle, return the global part id. For i'th part p on process rank j, where the number of parts per process is M and the number of processes is X (i=0,...,M-1, j=0,...,X-1), global id is M*j+i.

```
int FMDB_Part_LocalID (pPart  /* in */  part)
```

Given a part handle, return the local part id. For i'th part p on process rank j, where the number of parts per process is M and the number of processes is X (i=0,...,M-1, j=0,...,X-1), local id is i.

```
int FMDB_PartID_CommRank (int  /* in */  partID);
```

Given global part ID, get process rank.

```
int FMDB_PartID_LocalID (int /* in */ partID);
```

Given global part ID, get local part ID.

```
int FMDB_Part_GetDim (
        pPart  /* in */  part,
        int*  /* out */  dim)
```

Given a part handle, get the dimension of the part.

```
int FMDB_Part_GetNumSet (
        pPart  /* in */  part,
        int*  /* out */  numEntSet)
```

Given a part, get the number of P-sets created in a part.

```
int FMDB_Part_GetSet (
        pPart  /* in */  part,
        std::vector<pEntSet>&  /* inout */  vecEntSet)
```

Given a part, get the vector *vecEntSet* filled with P-set handles created in a part.

```
int FMDB_Part_FindEnt (
        pPart  /* in */  part,
        pMeshEnt  /* in */  ent,
        int*  /* out */  found)
```

Given a part and mesh entity handle, check if the part contains the mesh entity. If the part contains the entity set, *found* is non-zero integer, otherwise, *found* is 0.

```
int FMDB_Part_GetNumEnt (
        pPart  /* in */  part,
        int  /* in */  type,
        int  /* in */  topo,
        int*  /* out */  numEnt)
```

Given a part handle, entity type and topology, get the total number of entities of type/topo in the part.

```
int FMDB_Part_GetNumPartBdry (
        pPart  /* in */  part,
        int  /* in */  dest_partID,
        int  /* in */  type,
        int  /* in */  topo,
        int*  /* out */  numEnt)
```

Given a part handle, destination global part id, entity type and topology, get the number of part boundary entities of type and topology existing on the destination part. Specify -1 to *dest_partid* for all part boundary entities of given type and topo.

```
int FMDB_Part_GetNumGhost (
        pPart  /* in */  part,
        int  /* in */  type,
        int*  /* out */  numEnt)
```

Given a part handle and integer specifying the entity type (FMDB_VERTEX, FMDB_EDGE, FMDB_FACE, FMDB_REGION, FMDB_ALLTYPE), get the number of ghost entities of the type. If *type* is FMDB_VERTEX, get the number of ghost vertices in the part, if *type* is FMDB_EDGE, get the number of ghost edges in the part, if *type* is FMDB_FACE, get the number of ghost faces in the part, if *type* is FMDB_REGION, get the number of ghost regions in the part, if *type* is FMDB_ALLTYPE, get the number of ghost entities in all types (FMDB_VERTEX to FMDB_REGION) in the part.

```
int FMDB_Part_GetNumNborPart (
        pPart  /* in */  part,
        int  /* in */  type,
        int*  /* out */  numNbor)
```

Given a part handle and entity type (FMDB_VERTEX, FMDB_EDGE, FMDB_FACE, FMDB_REGION, FMDB_ALLTYPE), get the number of neighboring parts. A part i neighbors part j over entity type d if they share d dimensional mesh entities on part boundary.

```
int FMDB_Part_GetNborPartID (
        pPart  /* in */  part,
        int  /* in */  type,
        std::vector<int>&  /* inout */  vecPartID)
```

Given a part handle and entity type (FMDB_VERTEX, FMDB_EDGE, FMDB_FACE, FMDB_REGION, FMDB_ALLTYPE), get the global part ids of parts with mesh entities of type on part boundaries shared with the part.

5.3.3 Part traversal

```
int FMDB_PartEntIter_Init (
        pPart  /* in */  part,
        int  /* in */  type,
        int /* in */  topo,
        pPartEntIter&  /* out */  iter)
```

Given a part and entity type (FMDB_VERTEX, FMDB_EDGE, FMDB_FACE, FMDB_REGION, FMDB_ALLTYPE) and entity topology (0 to 11), get a pointer to the first mesh entity of the type and the topo. If no entity exists or *type* or *topo* is invalid, *iter* is set to $NULL$ and error code is returned.

```
int FMDB_PartEntIter_InitRevClas (
        pPart  /* in */  part,
        pGeomEnt  /* in */  geomEnt,
        int  /* in */ type,
        pPartEntIter&  /* out */  iter)
```

Given a part, geometric entity, and desired mesh entity type (FMDB_VERTEX, FMDB_EDGE, FMDB_FACE, FMDB_REGION, FMDB_ALLTYPE), get a pointer to the first mesh entity of *type*

classified on the geometric model entity. If no mesh entity exists, *iter* is set to $NULL$ and error code is returned.

```
int FMDB_PartEntIter_InitPartBdry (
        pPart  /* in */  part,
        int  /* in */  dest_glob_partid,
        int  /* in */  type,
        int /* in */  topo,
        pPartEntIter&  /* out */  iter)
```

Given a part, destination global part id, entity type (0 to 4) and entity topology (0 to 11), get a pointer to the first part boundary entity of the type and the topo existing on the destination part. If no mesh entity exists or *type* or *topo* is invalid, *iter* is *null*. If *dest_glob_partid* is set to FMDB_ALL, all neighboring parts are checked for part boundary entities of given type and topo. If no mesh entity exists, *iter* is set to $NULL$ and error code is returned.

```
int FMDB_PartEntIter_GetNext(
        pPartEntIter  /* inout */  iter,
        pMeshEnt /* out */ ent)
```

Increment the iterator to the next position and get the mesh entity being pointed.

```
int FMDB_PartEntIter_Del(pPartEntIter  /* inout */  iter)
```

Delete the iterator.

```
int FMDB_PartEntIter_IsEnd(
        pPartEntIter  /* in */  iter,
        int*  /* out */  isEnd)
```

If the iterator advanced beyond the last entity, *isEnd* is 1, otherwise, 0.

```
int FMDB_PartEntIter_Reset(pPartEntIter  /* inout */  iter)
```

Reset the iterator (move to the first entity position).

5.3.4 ID Management

```
int FMDB_Mesh_SetSetID (pMeshMdl  /* in */  mesh)
```

Given a mesh, assign unique ID to entity sets (NP or P) in the mesh. The unique ID's must be deleted via *FMDB_Mesh_DelSetID* to avoid memory leak. The ID begins from 0. Note the unique IDs become invalid when entity sets are modified through modifications/migrations or new set is created or deleted.

```
int FMDB_Mesh_DelSetID (pMeshMdl  /* in */  mesh)
```

Given a mesh, delete unique ID assigned to entity sets (NP or P) in the mesh.

```
int FMDB_Part_SetEntID (
        pPart  /* in */  part,
        int  /* in */  type)
```

Given a part handle and entity type (0 to 4), assign unique ID to mesh entities of the type in the part. The unique ID's must be deleted via *FMDB_Part_DelEntID* to avoid memory leak. The ID begins from 1. Note the unique IDs become invalid when a mesh is modified through mesh modifications or migrations.

```
int FMDB_Part_DelEntID (
        pPart  /* in */  part,
        int  /* in */  type)
```

Given a part handle and entity type (0 to 4), delete unique ID assigned to mesh entities of the type in the part.

```
int FMDB_Part_SetSetID (pPart /* in */ part);
```

Given a part, assigns a unique id starting from 0 for all p-sets created in part.

```
int FMDB_Part_DelSetID (pPart /* in */ part);
```

Given a part, for all p-sets created in part, delete unique id generated by FMDB_Part_SetSetID.

5.3.5 Part tag

```
int FMDB_Part_DelTag (
        pPart  /* in */  part,
        pTag  /* in */  tag)
```

Given a part handle and tag handle, delete the tag data from the part. If the tag doesn't exist with the part, the error code *SCUtil_TAG_NOT_FOUND* is returned.

```
int FMDB_Part_HasTag (
        pPart  /* in */  part,
        pTag  /* in */  tag,
        int*  /* out */  hasTag)
```

Given a part handle and tag handle, if the tag is attached to the part, *hasTag* is 1, otherwise, 0. If the tag doesn't exist with the part, the error code *SCUtil_TAG_NOT_FOUND* is returned.

```
int FMDB_Part_GetTag (
        pPart  /* in */  part,
        std::vector<pTag>& /* out */  vecTag)
```

Given a part handle, get the vector *vecTag* filled with all tag handles attached to the part.

```
int FMDB_Part_SetByteTag (
        pPart  /* in */  part,
        pTag  /* in */  tag,
        const void*  /* in */  tagVal,
        int  /* in */  tagByte)
```

Given a part handle, tag handle of any tag type, data in bytes, and size of data in byte, set or update the tag value of the part. *tagByte* indicates the size of tag data in byte. If the tag doesn't exist with the part, the error code *SCUtil_TAG_NOT_FOUND* is returned.

```
int FMDB_Part_GetByteTag (
        pPart  /* in */  part,
        pTag  /* in */  tag,
        void**  /* out */  tagVal,
        int*  /* out */  tagByte)
```

Given a part handle and tag handle, get the byte type data tagged to the part. *tagByte* indicates the size of tag data in byte. If the tag doesn't exist with the part, the error code *SCUtil_TAG_NOT_FOUND* is returned.

```
int FMDB_Part_SetIntTag (
        pPart  /* in */  part,
        pTag  /* in */  tag,
        const int  /* in */  tagVal)
```

Given a part handle, tag handle, and integer, set or update the tag value of the part. If tag type is not integer type or tag size is not 1, *INVALID_TAG* is returned.

```
int FMDB_Part_GetIntTag (
        pPart  /* in */  part,
        pTag  /* in */  tag,
        int*  /* out */  tagVal)
```

Given a part handle and tag handle, get integer type data tagged to the part. If the tag doesn't exist with the part, the error code *SCUtil_TAG_NOT_FOUND* is returned. If tag type is not integer type or tag size is not 1, the error code *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Part_SetDblTag (
        pPart  /* in */  part,
        pTag  /* in */  tag,
        const double  /* in */  tagVal)
```

Given a part handle, tag handle, and double, set or update the tag value of the part. If tag type is not double type or tag size is not 1, the error code *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Part_GetDblTag (
        pPart  /* in */  part,
        pTag  /* in */  tag,
        double*  /* out */  tagVal)
```

Given a part handle and tag handle, get double type data tagged to the part. If the tag doesn't exist with the part, the error code *SCUtil_TAG_NOT_FOUND* is returned. If tag type is not double type or tag size is not 1, the error code *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Part_SetEntTag (
        pPart  /* in */  part,
        pTag  /* in */  tag,
        const pMeshEnt /* in */  tagVal)
```

Given a part handle, tag handle, and another mesh entity, set or update the tag value of the part. If tag type is not entity type or tag size is not 1, *INVALID_TAG* is returned.

```
int FMDB_Part_GetEntTag (
        pPart  /* in */  part,
        pTag  /* in */  tag,
        pMeshEnt*  /* out */  tagVal)
```

Given a part handle and tag handle, get the entity tagged to the part. If the tag doesn't exist with the part, the error code *SCUtil_TAG_NOT_FOUND* is returned. If tag type is not entity type or tag size is not 1, the error code *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Part_SetSetTag (
        pPart  /* in */  part,
        pTag  /* in */  tag,
        const pEntSet  /* in */  tagVal)
```

Given a part handle, tag handle, and entity set, set or update the tag value of the part. If tag type is not entity set type or tag size is not 1, *INVALID_TAG* is returned.

```
int FMDB_Part_GetSetTag (
        pPart  /* in */  part,
        pTag  /* in */  tag,
        pEntSet*  /* out */  tagVal)
```

Given a part handle and tag handle, get the entity set tagged to the part. If the tag doesn't exist with the part, the error code *SCUtil_TAG_NOT_FOUND* is returned. If tag type is not entity set type or tag size is not 1, the error code *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Part_SetIntArrTag (
        pPart  /* in */  part,
        pTag  /* in */  tag,
        const int*  /* in */  data_arr,
        int  /* in */  data_size)
```

Given a part handle, tag handle, and integer array data, set or update the tag value of the part. If tag type is not integer type or tag size is 1, *INVALID_TAG* is returned.

```
int FMDB_Part_GetIntArrTag (
        pPart  /* in */   part,
        pTag  /* in */  tag,
        int**  /* out */  data_arr,
        int*  /* out */  data_size)
```

Given a part handle and tag handle, get integer array data tagged to the part. *data_size* is the size of *data_arr*. If the tag doesn't exist with the entity, the error code *SCUtil_TAG_NOT_FOUND* is returned. If tag type is not integer type or tag size is 1, *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Part_SetDblArrTag (
        pPart  /* in */   part,
        pTag  /* in */  tag,
        const double*  /* in */  data_arr,
        int  /* in */  data_size)
```

Given a part handle, tag handle, and double array data, set or update the tag value of the part. If tag type is not double type or tag size is 1, *INVALID_TAG* is returned.

```
int FMDB_Part_GetDblArrTag (
        pPart  /* in */  part,
        pTag  /* in */  tag,
        double**  /* out */  data_arr,
        int*  /* out */  data_size)
```

Given a part handle and tag handle, get double array data tagged to the part. *data_size* is the size of *data_arr*. If the tag doesn't exist with the entity, the error code *SCUtil_TAG_NOT_FOUND* is returned. If tag type is not double type or tag size is 1, *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Part_SetEntArrTag (
        pPart  /* in */  part,
        pTag  /* in */  tag,
        const pMeshEnt*  /* in */  data_arr,
        int  /* in */  data_size)
```

Given a part handle, tag handle, and entity array data, set or update the tag value of the part. If tag type is not entity type or tag size is 1, *INVALID_TAG* is returned.

```
int FMDB_Part_GetEntArrTag (
        pPart  /* in */  part,
        pTag  /* in */  tag,
        pMeshEnt**  /* out */  data_arr,
        int*  /* out */  data_size)
```

Given a part handle and tag handle, get entity array data tagged to the part. *data_size* is the size of *data_arr*. If the tag doesn't exist with the entity, the error code *SCUtil_TAG_NOT_FOUND* is returned. If tag type is not entity type or tag size is 1, *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Part_SetSetArrTag (
        pPart  /* in */  part,
        pTag  /* in */  tag,
        const pEntSet*  /* in */  data_arr,
        int  /* in */  data_size)
```

Given a part handle, tag handle, and entity set array data, set or update the tag value of the part. If tag type is not entity set type or tag size is 1, *INVALID_TAG* is returned.

```
int FMDB_Part_GetSetArrTag (
        pPart  /* in */  part,
        pTag  /* in */  tag,
        pEntSet**  /* out */  data_arr,
        int*  /* out */  data_size)
```

Given a part handle and tag handle, get entity set array data tagged to the part. *data_size* is the size of *data_arr*. If the tag doesn't exist with the entity, the error code *SCUtil_TAG_NOT_FOUND* is returned. If tag type is not entity set type or tag size is 1, *SCUtil_INVALID_TAG* is returned.

5.3.6 Distributed mesh functionality

```
int FMDB_Mesh_SetPtnParam (
        pMeshMdl  /* in */  mesh,
        int      /* in */   ptnMethod,
        int      /* in */   ptnApproach,
        double   /* in */  imbalance_tol,
        int      /* in */  debug_level)
```

Given a mesh instance, set the zoltan parameters: Zoltan LB_Method (*FMDB_RCB*, *FMDB_RIB*, *FMDB_GRAPH*, *FMDB_HYPERGRAPH*, *FMDB_PARMETIS*), Zoltan LB_Approach (*FMDB_PARTITION*, *FMDB_REPARTITION*, *FMDB_REFINE*, *PartKway, PartGeom, PartGeomKWay, AdaptiveRep art, RefineKway*), imbalance tolerance and debug level

```
int FMDB_Tag_SetAutoMigrOn (
        pMeshMdl  /* in */  mesh,
        pTag  /* in */  tag,
        int  /* in */  ent_type)
```

Given a mesh instsance, tag handle and entity type, register a tag handle for automatic tag data migration along the mesh migration. If *ent_type* is *FMDB_ALLTYPE*, migrate tag data for all entity types. The supported tag types are SCUtil_INT (integer) and SCUtil_DBL (double) of any length (single or array). If tag type is SCUtil_BYTE, SCUtil_ENT or SCUtil_SET, the error code *SCUtil_INVALID_TAG_TYPE* is returned. If the tag is already registered for automatic migration, the error code *SCUtil_TAG_ALREADY_EXISTS* is returned.

```
int FMDB_Tag_SetAutoMigrOff (
        pMeshMdl  /* in */  mesh,
        pTag  /* in */  tag,
        int  /* in */  ent_type)
```

Given a mesh instance, tag handle and entity type, cancel automatic migration for a tag handle along the migration. If *ent_type* is *FMDB_ALLTYPE*, cancel automatic tag data migration for all entity types.

```
int FMDB_Mesh_GlobPtn (
        pMeshMdl  /* in */  mesh,
        MigrCB&  /* in */  migrCB,
        int      /* in */  LB_Method,
        int      /* in */  LB_Approach,
        int      /* in */  Parmetis_Method,
        double   /* in */  imbalance_tol,
        int      /* in */  debug_level)
```

Given a mesh instance, migration callback, Zoltan LB_Method (*FMDB_RCB*, *FMDB_RIB*, *FMDB_GRAPH*, *FMDB_HYPERGRAPH*, *FMDB_PARMETIS*), Zoltan LB_Approach (*FMDB_PARTITION*, *FMDB_REPARTITION*, *FMDB_REFINE*, *PartKway, PartGeom, PartGeomKWay, AdaptiveRepart, RefineKway*), imbalance tolerance and debug level, globally partition the mesh to make equally loaded. Speficy *FMDB_NONE* in *Parmetis_Method* for non-parmetis LB_Method. The user can set the weight of partition object which is either of entity or entity set (See §5.4.8 and §5.5.6). In old version of FMDB, the migration callback was used to specify how to handle the tag attached to the entity or entity set along with migration. Migration callback is still supported for backward compatibility but not recommeded to use any more for tag data migration. Instead, use *FMDB_Tag_SetAutoMigrOn* and *FMDB_Tag_SetAutoMigrOff* to turn on and off the automatic tag migration. For tag handle not registered for automatic data migration, tag data attached to entities are discarded when entities and entitiy sets are migrated and removed from the part. If input mesh's part boundary is not synchronized, *FMDB_Mesh_SyncPtn* is performed internally.

```
int FMDB_Mesh_SerialGlobPtn(
        pMeshMdl  /* in */  mesh,
        int      /* in */   ptnMethod,
        int      /* in */   ptnApproach,
        double   /* in */  imbalance_tol,
        int      /* in */  debug_level)
```

Global partitioning function developed for one process run - faster version of FMDB_Mesh_GlobPtn on one proccess.

```
int FMDB_Mesh_LocalPtn (
        pMeshMdl  /* in */  mesh,
        MigrCB&  /* in */  migrCB,
        int      /* in */  LBMethod,
        int      /* in */  LBApproach,
```

```
        int      /* in */  parmetisMethod,
        double   /* in */  imbalance_tol,
        int      /* in */  debug_level)
```

Reserved

```
int FMDB_Mesh_Migr (
        pMeshMdl  /* in */  mesh,
        MigrCB&  /* in */  migrCB,
std::map<pMeshEnt, std::pair<int, int> >&  /* in */  EntToMigr,
        std::map<pEntSet, std::pair<int, int> >&  /* in */  SetToMigr);
```

Given a mesh instance, migration callback, and two map containers with entity and P-sets where each entity and P-set has a pair of global source part id and destination part id, migrate entity and P-sets to designated part. The entities and P-sets designated to migrate are termed *partition object* and any entities contained in *EntToMigr* or a set in *SetToMigr* should be regions in 3D mesh and faces in 2D mesh. In old version of FMDB, the migration callback was used to specify how to handle the tag attached to the entity or entity set along with migration. Migration callback is still supported for backward compatibility but not recommeded to use any more for tag data migration. Instead, use *FMDB_Tag_SetAutoMigrOn* and *FMDB_Tag_SetAutoMigrOff* to turn on and off the automatic tag migration. For tag handle not registered for automatic data migration, tag data attached to entities are discarded when entities and entitiy sets are migrated and removed from the part. If input mesh's part boundary is not synchronized, *FMDB_Mesh_SyncPtn* is performed internally.

```
int FMDB_Ent_Migr (
        pMeshMdl  /* in */  mesh,
        pmMigrationCallbacks&  /* in */  migrCB,
        pMeshEnt  /* in */  ent,
        int  /* in */  source_partid,
        int  /* in */  dest_partid,
        std::vector<pMeshEnt>&  /* out */  *newEnts,
        std::vector<pMeshEnt>&  /* out */  *oldEnts,
        std::vector<MPI_Request*>&  /* out */  &requests);
```

Given a mesh instance, migration callback, target entity, and source_partid migrate a single entity. In old version of FMDB, the migration callback was used to specify how to handle the tag attached to the entity or entity set along with migration. Migration callback is still supported for backward compatibility but not recommeded to use any more for tag data migration. Instead, use *FMDB_Tag_SetAutoMigrOn* and *FMDB_Tag_SetAutoMigrOff* to turn on and off the automatic tag migration. For tag handle not registered for automatic data migration, tag data attached to entities are discarded when entities and entitiy sets are migrated and removed from the part.

```
int FMDB_Mesh_MigrAdv (
        pPart  /* in */  mesh,
        MigrCB&  /* in */  migrCB,
        std::map<pMeshEnt, std::pair<int, int> >&  /* in */  EntToMigr,
```

```
        std::map<pEntSet, std::pair<int, int> >&  /* in */  SetToMigr,
        std::vector<pMeshEnt>*  /* out */  newMeshEnt,
        std::vector<pMeshEnt>*  /* out */  rmvMeshEnt,
        std::vector<pEntSet>*  /* out */  newEntSet,
        std::vector<pEntSet>*  /* out */  rmvEntSet);
```

Given a mesh instance, migration callback, and two map containers with entity and P-sets where each entity and P-set has a pair of global source part id and destination part id, migrate entity and P-sets to designated part. The entities and P-sets designated to migrate are termed *partition object* and any entities contained in *EntToMigr* or a set in *SetToMigr* should be regions in 3D mesh and faces in 2D mesh. In old version of FMDB, the migration callback was used to specify how to handle the tag attached to the entity or entity set along with migration. Migration callback is still supported for backward compatibility but not recommeded to use any more for tag data migration. Instead, use *FMDB_Tag_SetAutoMigrOn* and *FMDB_Tag_SetAutoMigrOff* to turn on and off the automatic tag migration. For tag handle not registered for automatic data migration, tag data attached to entities are discarded when entities and entitiy sets are migrated and removed from the part. After migration is completed, the vector containers $newMeshEnt$, $rmvMeshEnt$, $newEntSet$, $rmvEntSet$ are filled as the following: *newMeshEnt[i]* is filled with newly created mesh entities of dimension i on local parts. *rmvMeshEnt[i][j]* is filled with removed mesh entities of dimension i from local parts. *newEntSet[i]* is filled with newly created P-set created on i'th local part. *rmvEntSet[i]* is filled with removed P-set from i'th local part. How to handle the tag attached to the entity or P-set along with migration is specified in *migrCB*. Tags not specified in *migrCB* are discarded when entities and entitiy sets are migrated and removed from the part. If input mesh's part boundary is not synchronized, *FMDB_Mesh_SyncPtn* is performed internally.

```
int FMDB_Mesh_Merge (
pMeshMdl /* in */ mesh,
pmMigrationCallbacks& /* in */ migrCB);
```

Given a mesh instance, and migation callback, merge all parts (migrate all parts to part 0 on master process). In old version of FMDB, the migration callback was used to specify how to handle the tag attached to the entity or entity set along with migration. Migration callback is still supported for backward compatibility but not recommeded to use any more for tag data migration. Instead, use *FMDB_Tag_SetAutoMigrOn* and *FMDB_Tag_SetAutoMigrOff* to turn on and off the automatic tag migration. For tag handle not registered for automatic data migration, tag data attached to entities are discarded when entities and entitiy sets are migrated and removed from the part. For N part mesh merged to part 0, FMDB Mesh WriteToFile call still creates N mesh files where N - 1 mesh files are empty.

```
int FMDB_Mesh_CreateGhost (
        pMeshMdl  /* in */  mesh,
        MigrCB  /* in */  migrCB,
        int  /* in */ ghostType,
        int  /* in */ brgType,
        int  /* in */ numLayer,
        int  /* in */ includeCopy)
```

Given a mesh instance, migration callback to handle tag data attached to entity, desired ghost entity type, desired bridge entity type, number of ghost layers, and the flag indicating

whether to create ghosts (1: yes, 0: no), create ghost entities. In old version of FMDB, the migration callback was used to specify how to handle the tag attached to the entity or entity set along with migration. Migration callback is still supported for backward compatibility but not recommeded to use any more for tag data migration. Instead, use *FMDB_Tag_SetAutoMigrOn* and *FMDB_Tag_SetAutoMigrOff* to turn on and off the automatic tag migration. For tag handle not registered for automatic data migration, tag data attached to entities are not copied to ghost entities. If *includeCopies* equals to 0 and part boundary entity of type *brgType* is not owned by a self part (shortly, non-owned bridge type entity), *ghostType* dimensional entities adjacent to the non-owned bridge type entity is not ghost copied. If *includeCopies* is non-zero integer, all *ghostType* dimensional entities adjacent to the bridge type entities on part boundaries are ghost copied.

```
int FMDB_Mesh_DelGhost (pMeshMdl  /* in */  mesh)
```

Given a mesh instance, delete ghost entities.

```
int FMDB_Mesh_GetGhostInfo (pMeshMdl mesh, std::vector<int>& ghostInfo);
```

Given mesh instance, return the historical ghosting information in order which consists of four integers, ghost type, bridge type, the number of layers, and an option for include copy (0 or 1) for each FMDB_Mesh_CreateGhost call. For instance, if FMDB_Mesh_CreateGhost was called twice in the following order (mesh, migrCB, FMDB_ALLTYPE, FMDB_VERTEX, 1, 1) and (mesh, migrCB, FMDB_REGION, FMDB_EDGE, 3, 0), *FMDB_Mesh_GetGhostInfo* returns an array of integers [4, 1, 1, 1, 3, 2, 3 0].

5.3.7 Miscellaneous

```
int FMDB_Mesh_Verify (
        pMeshMdl  /* in */  mesh,
        int*  /* out */  isValid)
```

Given a mesh instance, check if the mesh is valid or not. If the mesh is invalid, *isValid* is 0, otherwise *isValid* is non-zero. See Appendix A for more information on the mesh verification algorithm.

```
int FMDB_Mesh_DspNumEnt (pMeshMdl  /* in */  mesh)
```

Given a mesh instance, display number of owned entities per part.

```
int FMDB_Mesh_DspStat (pMeshMdl  /* in */  mesh)
```

Given a mesh, display mesh statistics such as the number of p-sets, entities per part, and load imbalance ratio, etc. See Appendix A for more information.

5.4 Entity Functions

5.4.1 Entity management

```
int FMDB_Vtx_Create (
        pPart  /* in */  part,
        pGeomEnt  /* in */  geomEnt,
        double*  /* in */  coord,
        double*  /* in */  param,
        pMeshEnt&  /* out */  meshVtx)
```

Given a part handle, geometric model entity, coodinates and parametric values, create a mesh vertex on the part. The new vertex is returned through *meshVtx*.

```
int FMDB_Edge_Create (
        pPart  /* in */  part,
        pGeomEnt  /* in */  geomEnt,
        pMeshEnt  /* in */  meshVtx_1,
        pMeshEnt  /* in */  meshVtx_2,
        pMeshEnt&  /* out */  meshEdge)
```

Given a part handle, geometric model entity, and two mesh vertices, create a mesh edge on the part. If an edge with two mesh vertices already exists, error code *SCUtil_ENTITY_ALREADY_EXIST* is returned. The new or existing edge is returned through *meshEdge*.

```
int FMDB_Face_Create (
        pPart  /* in */  part,
        pGeomEnt  /* in */  geomEnt,
        int  /* in */  downEnt_topo,
        pMeshEnt*  /* in */ downEnts,
        int*   /* in */  dirs,
        pMeshEnt&  /* out */  meshFace)
```

Given a mesh instance, geometric model entity, the number of downward adjacent entities, downward adjacent entity array *downEnt*, and array of downward adjacent entity direction *dir*, create a mesh face on the part. *numDownEnt* should be either 3 or 4, otherwise, the error code *SCUtil_INVALID_ARGUMENT* is returned. If *numDownEnt* is 3, a triangle is created. If *numDownEnt* is 4, a quadrilateral is created. Mesh entities in *downEnt* should be the same type entities, either vertices or edges. If a face with downward entities already exists, error code *SCUtil_ENTITY_ALREADY_EXIST* is returned. The new or existing face is returned through *meshFace*.

```
int FMDB_Rgn_Create (
        pPart  /* in */  part,
        pGeomEnt  /* in */  geomEnt,
        int  /* in */  topo,
        pMeshEnt*  /* in */ downEnt,
```

```
        int  /* in */  numDownEnt,
        pMeshEnt&  /* out */  meshRgn)
```

Given a mesh instance, geometric model entity, region topology (FMDB_TET, FMDB_HEX, FMDB_PRISM, FMDB_PYRAMID), downward adjacent entity array *downEnt* and the number of downward adjacent entities, create a mesh region on the part. Mesh entities in *downEnt* should be the same type entities; vertices or edges or faces. The newly created region is returned through *meshRgn*.

```
int FMDB_Ent_Find (
        int  /* in */ topo,
        pMeshEnt*  /* in */  downEnts,
        int /* in */ num_down_ents,
        pMeshEnt&  /* out */  meshEnt)
```

Given entity topology and an array of downward adjacent entities *downEnt*, search if an entity bounded by downward adjacent entities exists on the part. If the entity exists, the found entity is returned through *meshEnt* and the function returns *SCUtil_SUCCESS*. Otherwise, *NULL* is returned through *meshEnt* and the function returns error code *SCUtil_ENTITY_NOT_FOUND*. Mesh entities in *downEnt* should be the same type entities. If *entType* is less than or equal to downward adjacent entity type or *entType* is 0, the error code *SCUtil_INVALID_ARGUMENT* is returned.

```
int FMDB_Ent_Del (
        pPart  /* in */  part,
        pMeshEnt  /* in */  meshEnt)
```

Given a part handle and entity handle, delete the entity and all associated tag data from the part. If the entity is contained in entity sets, it is removed from entity sets. The entity copies on off-part (remote copies or ghost copies) are not removed.

5.4.2 Entity information

```
int FMDB_Ent_SetID(pMeshEnt meshEnt, int ID)
```

Given a mesh entity and integer number, set the ID of the entity to the number.

```
int FMDB_Ent_DelID(pMeshEnt meshEnt)
```

Given a mesh entity, delete the integer ID generated by *FMDB_Mesh_SetEntID* or *FMDB_Ent_SetID*.

```
int FMDB_Ent_ID (pMeshEnt  /* in */  meshEnt)
```

Given a mesh entity, return the ID generated by *FMDB_Mesh_SetEntID* or *FMDB_Ent_SetID*. If ID is not available, return -1. Note by default, vertex ID is generated during mesh loading.

```
std::string FMDB_Ent_GetStrID (pMeshEnt  /* in */  meshEnt)
```

Given a mesh entity, return the string type ID consisting of a letter and vertex ID. For a mesh vertex, the string ID is the letter 'V' and integer ID. E.g. V10 For a mesh edge, the string ID is the letter 'E' and ID's of consisting vertices. E.g. E10_7 For a mesh face, the string ID is the letter 'F' and ID's of consisting vertices. E.g. F3_7_10 For a mesh region, the string ID is the letter 'R' and ID's of consisting vertices. E.g. R_3_7_10 To use this function for edges, faces and regions, the part-wide unique entity ID should be created via *FMDB_Part_*SetEntID beforehand. Note the part-wide unique entity ID becomes invalid when the mesh is changed either by mesh modifications or migrations.

```
int FMDB_Ent_GetType (
        pMeshEnt  /* in */  meshEnt,
        int*  /* out */  type)
```

Given a mesh entity, get the type of the entity: FMDB_VERTEX, FMDB_EDGE, FMDB_FACE, and FMDB_REGION.

```
int FMDB_Ent_GetTopo (
        pMeshEnt  /* in */  meshEnt,
        int*  /* out */  topo)
```

Given a mesh entity, get the topology of the entity. Supported topologies are: FMDB_POINT, FMDB_LINE, FMDB_TET, FMDB_HEX, FMDB_PRISM, and FMDB_PYRAMID.

```
int FMDB_Ent_SetGeomClas (
        pMeshEnt  /* in */  meshEnt,
        pGeomEnt  /* in */  geomEnt)
```

Given a mesh entity and geometric model entity, set the geometric classification of mesh entity.

```
int FMDB_Ent_GetGeomClas (
        pMeshEnt  /* in */  meshEnt,
        pGeomEnt&  /* out */  geomEnt)
```

Given a mesh entity, get the geometric entity classified on.

```
int FMDB_Ent_GetGeomClasType (
        pMeshEnt  /* in */  meshEnt,
        int*  /* out */  geomType)
```

Given a mesh entity, get the type of geometric entity classified on. FMDB_VERTEX, FMDB_EDGE, FMDB_FACE, and FMDB_REGION.

```
int FMDB_GeomEnt_GetNumRevClas (
        pGeomEnt  /* in */  geomEnt,
        pPart  /* in */  part,
        int   /* in */ type,
        int*  /* out */  numEnt)
```

Given a geometric entity, part handle and desired entity type (FMDB_VERTEX, FMDB_EDGE, FMDB_FACE, FMDB_REGION, FMDB_ALLTYPE), get the number of mesh entities classified on the geometric model entity.

```
int FMDB_Ent_GetNumAdj (
        pMeshEnt  /* in */  meshEnt,
        int  /* in */  tgtType,
        int*  /* out */  numAdj)
```

Given a mesh entity and desired adjacency type *tgtType*, get the number of adjacent entities of type *tgtType*. If *tgtType* equals to the type of *meshEnt*, *numAdj* is 0.

```
int FMDB_Ent_GetAdj (
        pMeshEnt  /* in */  meshEnt,
        int  /* in */  tgtType,
        int  /* in */  dir,
        std::vector<pMeshEnt>&  /* inout */  vecAdj)
```

Given a mesh entity and desired adjacency type *tgtType*, get the vector *vecAdj* filled with the adjacent entities of type *tgtType*. If *tgtType* equals to the type of *meshEnt*, vector is returned as empty.

```
int FMDB_Ent_Get2ndAdj (
        pMeshEnt  /* in */  meshEnt,
        int  /* in */  brgType,
        int  /* in */  tgtType,
        std::vector<pMeshEnt>&  /* inout */  vecAdjEnt)
```

Given a mesh entity, bridge type *brgType*, and desired adjacency type *tgtType*, get the vector *vecAdj* filled with 2^{nd} order adjacent entities of type *tgtType* obtained through the bridge type *brgType*. If *tgtType* equals to *brgType* and *brgType* is not *4* (all types), the error code *SCUtil_INVALID_ARGUMENT* is returned.

5.4.3 Vertex functions

```
int FMDB_Vtx_GetLoadPartOrderPair (
        pPart  /* in*/  part,
        pMeshEnt  /* in */  vtx,
        int*  /* in */  load_partid,
        int*  /* out */  load_order)
```

Given a part handle and vertex handle *vtx*, get loading part id and read-in order from mesh file. If vtx doesn't exist in part or vtx is not a vertex handle, the error code *SCUtil_INVALID_ENTITY_HANDLE* is returned. Restriction: only one or zero partitioning in a given process - load_partid is global part id where the global id of i'th part on process j is j*M+i, the number of parts per process is M and the number of processes is X (i=0,...,M-1, j=0,...,X-1). - load_order starts from 1 - Note: global

part id is changed dynamically every time the number of parts per process changes. Suppose N is the number of parts per process at time of loading and M is the number of parts per process at time of partitioning. If N is not qual to M, for proper part id mapping, load_partid returned by FMDB_Vtx_GetLoadPartOrderPair should be converted to the part id at time of loading .

[Equation] For a vertex, global part id at time of loading = FMDB_PartID_CommRank(pid)*N + FMDB_PartID_LocalID(pid), where N: the number of parts per process at time of loading pid: loading part id returned by FMDB_Vtx_GetLoadPartOrderPair

```
int FMDB_Vtx_SetCoord (
        pMeshVtx  /* in */  meshVtx,
        double*   /* in */  coord)
```

Given a vertex and coordinate values in double array of minimum size 3, set *xyz* coordinates of the vertex.

```
int FMDB_Vtx_GetCoord (
        pMeshVtx  /* in */  meshVtx,
        double**  /* out */  coord)
```

Given a vertex and a double array of minimum size 3, get *xyz* coordinates of the vertex.

```
int FMDB_Vtx_SetParamCoord (
        pMeshVtx /* in */  meshVtx,
        double*  /*  in */  xyz)
```

Given a vertex and coordinate values in double array of minimum size 3, set the parametric coordinates of the vertex.

```
int FMDB_Vtx_GetParamCoord (
        pMeshVtx  /* in */  meshVtx,
        double*  /* in */  param)
```

Given a vertex and a double array of minimum size 3, get *xyz* parametric coordinates of the vertex.

5.4.4 Edge and higher-order node functions

```
int FMDB_Node_SetCoord (
        pNode  /* in */  node,
        double*  /* in */  xyz)
```

Given a node and coordinate values in double array of minimum size 3, set *xyz* coordinates of the node.

```
int FMDB_Node_GetCoord (
        pNode  /* in */  node,
        double*  /* inout */  xyz)
```

Given a node and a double array of minimum size 3, get *xyz* coordinates of the node.

```
int FMDB_Node_SetParamCoord (
        pNode  /* in */  node,
        double*  /* in */  param)
```

Given a node and coordinate values in double array of minimum size 3, set the parametric coordinates of the node.

```
int FMDB_Node_GetParamCoord (
        pNode  /* in */  node,
        double*  /* inout */  param)
```

Given a node and a double array of minimum size 3, get *xyz* parametric coordinates of the node.

```
int FMDB_Edge_CreateNode (
        pMeshEnt  /* in */  meshEdge,
        double*  /* in */  xyz,
        pNode&  /* out */  node)
```

Given an edge and coordinate values in double array of minimum size 3, create a node on the edge.

```
int FMDB_Edge_GetNumNode (
        pMeshEnt  /* in */  meshEdge,
        int*  /* out */  numNode)
```

Given an edge, get the number of nodes on the edge.

```
int FMDB_Edge_GetNode (
        pMeshEnt  /* in */  meshEdge,
        int  /* in */  n,
        pNode&  /* out */  node)
```

Given an edge, get n'th higher node of the edge. Note that n is 0 for the first node on the edge. If n is greater than or equal to the number of nodes on the edge, the error code *SCUtil_INVALID_ARGUMENT* is returned.

```
int FMDB_Edge_DelNode (
        pMeshEnt  /* in */  meshEdge,
        int  /* in */  n)
```

Given an edge, delete n'th higher node of the edge. Note that n is 0 for the first node on the edge. If n is *FMDB_ALL*, delete all nodes on the edge. If n is greater than or equal to the number of nodes on the edge, the error code *SCUtil_INVALID_ARGUMENT* is returned.

5.4.5 Face-specific functions

5.4.6 Region-specific functions

5.4.7 Entity tag

```
int FMDB_Ent_DelTag (
        pMeshEnt  /* in */  meshEnt,
        pTag  /* in */  tag)
```

Given an entity handle and tag handle, delete the tag from the entity. If the tag doesn't exist with the entity, the error code *SCUtil_TAG_NOT_FOUND* is returned.

```
int FMDB_Ent_HasTag (
        pMeshEnt  /* in */  meshEnt,
        pTag  /* in */  tag,
        int*  /* out */  hasTag)
```

Given an entity handle and tag handle, if the tag is attached to the entity, *hasTag* is 1, otherwise, 0. If the tag doesn't exist with the entity, the error code *SCUtil_TAG_NOT_FOUND* is returned.

```
int FMDB_Ent_GetTag (
        pMeshEnt  /* in */  meshEnt,
        std::vector<pTag>& /* out */  vecTag)
```

Given an entity handle, get the vector *vecTag* filled with all tag handles attached to the entity.

```
int FMDB_Ent_SetByteTag (
        pMeshMdl  /* in */  mesh,
        pEntSet  /* in */  entSet,
        pTag  /* in */  tag,
        const void*  /* in */  tagVal,
        int  /* in */  tagByte)
```

Given mesh, entity handle, tag handle of any type (byte, integer, double, entity, entity set), byte type data and the size of data in byte (*tagByte*), set or update the tag value of the entity. If the tag doesn't exist with the entity, the error code *SCUtil_TAG_NOT_FOUND* is returned.

```
int FMDB_Ent_GetByteTag (
        pMeshMdl  /* in */  mesh,
        pMeshEnt  /* in */  meshEnt,
        pTag  /* in */  tag,
        void**  /* out */  tagVal,
        int*  /* out */  tagByte)
```

Given mesh, entity handle and tag handle, get the byte type data tagged to the entity. *tagByte* indicates the size of tag data in byte. If the tag doesn't exist with the entity, the error code *SCUtil_TAG_NOT_FOUND* is returned.

```
int FMDB_Ent_SetIntTag (
        pMeshMdl  /* in */  mesh,
        pMeshEnt  /* in */  meshEnt,
        pTag  /* in */  tag,
        const int  /* in */  tagVal)
```

Given mesh, entity handle, tag handle, and integer data, set or update the tag value of the entity.

```
int FMDB_Ent_GetIntTag (
        pMeshMdl  /* in */  mesh,
        pMeshEnt  /* in */  meshEnt,
        pTag  /* in */  tag,
        int*  /* out */  tagVal)
```

Given mesh, entity handle and tag handle, get integer type data tagged to the entity. If the tag doesn't exist with the entity, the error code *SCUtil_TAG_NOT_FOUND* is returned.

```
int FMDB_Ent_SetDblTag (
        pMeshMdl  /* in */  mesh,
        pMeshEnt  /* in */  meshEnt,
        pTag  /* in */  tag,
        const double  /* in */  tagVal)
```

Given mesh, entity handle, tag handle, and double data, set or update the tag value of the entity. If tag type is not double type or tag size is not 1, *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Ent_GetDblTag (
        pMeshMdl  /* in */  mesh,
        pMeshEnt  /* in */  meshEnt,
        pTag  /* in */  tag,
        double*  /* out */  tagVal)
```

Given mesh, entity handle and tag handle, get double type data tagged to the entity. If the tag doesn't exist with the entity, the error code *SCUtil_TAG_NOT_FOUND* is returned. If tag type is not double type or tag size is not 1, *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Ent_SetEntTag (
        pMeshMdl  /* in */  mesh,
        pMeshEnt  /* in */  meshEnt,
        pTag  /* in */  tag,
        const pMeshEnt /* in */  tagVal)
```

Given mesh, entity handle, tag handle, and another mesh entity, set or update the tag value of the entity. If tag type is not entity type or tag size is not 1, *INVALID_TAG* is returned.

```
int FMDB_Ent_GetEntTag (
        pMeshMdl  /* in */  mesh,
        pMeshEnt  /* in */  meshEnt,
        pTag  /* in */  tag,
        pMeshEnt*  /* out */  tagVal)
```

Given mesh, entity handle, tag handle, get the entity tagged to the mesh entity. If the tag doesn't exist with the mesh entity, the error code *SCUtil_TAG_NOT_FOUND* is returned. If tag type is not entity type or tag size is not 1, *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Ent_SetSetTag (
        pMeshMdl  /* in */  mesh,
        pMeshEnt  /* in */  meshEnt,
        pTag  /* in */  tag,
        const pEntSet  /* in */  tagVal)
```

Given mesh, entity, tag handle, and entity set, set or update the tag value of the entity. If tag type is not entity set type or tag size is not 1, the error code *SCUtil_ INVALID_TAG* is returned.

```
int FMDB_Ent_GetSetTag (
        pMeshMdl  /* in */  mesh,
        pMeshEnt  /* in */  meshEnt,
        pTag  /* in */  tag,
        pEntSet*  /* out */  tagVal)
```

Given mesh, entity and tag handle, get the entity set tagged to the entity. If the tag doesn't exist with the mesh entity, the error code *SCUtil_TAG_NOT_FOUND* is returned. If tag type is not entity set type or tag size is not 1, *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Ent_SetIntArrTag (
        pMeshMdl  /* in */  mesh,
        pMeshEnt  /* in */  meshEnt,
        pTag  /* in */  tag,
        const int*  /* in */  data_arr,
        int  /* in */  data_size)
```

Given mesh, entity handle, tag handle, and integer array data, set or update the tag value of the entity. If tag type is not integer type or tag size is 1, *INVALID_TAG* is returned.

```
int FMDB_Ent_GetIntArrTag (
        pMeshMdl  /* in */  mesh,
        pMeshEnt  /* in */  meshEnt,
        pTag  /* in */  tag,
        int**  /* out */  data_arr,
        int*  /* out */  data_size)
```

Given mesh, entity handle and tag handle, get integer array data tagged to the entity. *data_size* is the size of *data_arr*. If the tag doesn't exist with the entity, the error code *SCUtil_TAG_NOT_FOUND* is returned. If tag type is not integer type or tag size is 1, *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Ent_SetDblArrTag (
        pMeshMdl  /* in */  mesh,
        pMeshEnt  /* in */  meshEnt,
        pTag  /* in */  tag,
        const double*  /* in */  data_arr,
        int  /* in */  data_size)
```

Given mesh, entity handle, tag handle, and double array data, set or update the tag value of the entity. If tag type is not double type or tag size is 1, *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Ent_GetDblArrTag (
        pMeshMdl  /* in */  mesh,
        pMeshEnt  /* in */  meshEnt,
        pTag  /* in */  tag,
        double**  /* out */  data_arr,
        int*  /* out */  data_size)
```

Given mesh, entity handle and tag handle, get double array data tagged to the entity. *data_size* is the size of *data_arr*. If the tag doesn't exist with the entity, the error code *SCUtil_TAG_NOT_FOUND* is returned. If tag type is not double type or tag size is 1, *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Ent_SetEntArrTag (
        pMeshMdl  /* in */  mesh,
        pMeshEnt  /* in */  meshEnt,
        pTag  /* in */  tag,
        const pMeshEnt*  /* in */  data_arr,
        int  /* in */  data_size)
```

Given mesh, entity handle, tag handle, and entity array data, set or update the tag value of the entity. If tag type is not entity type or tag size is 1, *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Ent_GetEntArrTag (
        pMeshMdl  /* in */  mesh,
        pMeshEnt  /* in */  meshEnt,
        pTag  /* in */  tag,
        pMeshEnt**  /* out */  data_arr,
        int*  /* out */  data_size)
```

Given mesh, entity handle and tag handle, get entity array data tagged to the entity. *data_size* is the size of *data_arr*. If the tag doesn't exist with the entity, the error code *SCUtil_TAG_NOT_FOUND* is returned. If tag type is not entity type or tag size is 1, *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Ent_SetSetArrTag (
        pMeshMdl  /* in */  mesh,
        pMeshEnt  /* in */  meshEnt,
        pTag  /* in */  tag,
        const pEntSet*  /* in */  data_arr,
        int  /* in */  data_size)
```

Given mesh, entity handle, tag handle, and entity set array data, set or update the tag value of the entity. If tag type is not entity set type or tag size is 1, *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Ent_GetSetArrTag (
        pMeshMdl  /* in */  mesh,
        pMeshEnt  /* in */  meshEnt,
        pTag  /* in */  tag,
        pEntSet**  /* out */  data_arr,
        int*  /* out */  data_size)
```

Given mesh, entity handle and tag handle, get entity set array data tagged to the entity. *data_size* is the size of *data_arr*. If the tag doesn't exist with the entity, the error code *SCUtil_TAG_NOT_FOUND* is returned. If tag type is not entity set type or tag size is 1, *SCUtil_INVALID_TAG* is returned.

5.4.8 Entity in parallel

```
int FMDB_Ent_IsOnPartBdry (
        pMeshEnt  /* in */  meshEnt,
        int*  /* out */  isOnPrtBdry)
```

Given a mesh entity, check if the entity is on the part boundary or not. If the entity is on part boundary, *isOnPrtBdry* is non-zero integer, otherwise, *isOnPrtBdry* is 0.

```
int FMDB_Ent_IsGhost (
        pMeshEnt  /* in */  meshEnt,
        int*  /* out */  isGhost)
```

Given a mesh entity, check if the entity is a ghost copy. If the entity is a ghost copy, *isGhost* is non-zero, otherwise, *isGhost* is 0.

```
FMDB_Ent_IsGhosted (
        pMeshEnt  /* in */  meshEnt,
        int*  /* out */  ghosted)
```

Given a mesh entity, check if the entity is ghost copied. If the entity has ghost copies on non-self part, *ghosted* is non-zero, otherwise, 0.

```
int FMDB_Ent_GetOwnPartID (
        pMeshEnt  /* in */  meshEnt,
        int*  /* out */  partID)
```

Given a mesh entity, owning part ID is returned.

```
int FMDB_Ent_GetOwnEnt (
        pMeshEnt  /* in */  meshEnt,
        pMeshEnt&  /* out */  ownerEnt)
```

Given a mesh entity, owning entity is returned.

```
int FMDB_Ent_GetNumRmt (
        pMeshEnt  /* in */  meshEnt,
        int*  /* out */  numRmt)
```

Given a mesh entity, get the number of remote copies.

```
int FMDB_Ent_GetAllRmt (
        pMeshEnt  /* in */  meshEnt,
        std::vector<std::pair<int, pMeshEnt> >&  /* inout */  vecRmt)
```

Given a mesh entity, get the vector *vecRmt* filled with pair(s) of global remote part id and the memory address of the entity on the remote part.

```
int FMDB_Ent_GetRmt (
        pMeshEnt  /* in */  meshEnt,
        int  /* in */  glob_partID,
        pMeshEnt&  /* out */  rmtEnt)
```

Given a mesh entity and global remote part ID, get the memory address of the entity on the remote part.

```
int FMDB_Ent_SetRmt (
        pMeshEnt  /* in */  meshEnt,
        int  /* in */  glob_partID,
        pMeshEnt  /* in */  rmtMeshEnt)
```

Given a mesh entity, global remote part ID, and address of remote mesh entity, add the remote copy to the entity. If $NULL$ is specified as $rmtMeshEnt$, reset the remote copy of entity on the remote part.

```
int FMDB_Ent_RmvRmt (
        pMeshEnt  /* in */  meshEnt,
        int  /* in */  glob_partID)
```

Given a mesh entity and global remote part id, delete the remote copy of entity on the remote part.

```
int FMDB_Ent_ClrRmt (pMeshEnt  /* in */  meshEnt)
```

Given a mesh entity, clear the remote copy. As a result, the status of entity changed from *FMDB_BDRY* to *FMDB_INTER*.

```
int FMDB_Ent_SetPtnTopo (
        pMeshEnt /* in */ meshEnt,
        pPart /*  in */part)
```

Given a mesh entity, update partition classification (a link to partition model entity) based on the remote part information.

```
int FMDB_Ent_GetResPart(
        pMeshEnt /* in */  meshEnt,
        pPart  /* in */  part,
        std::vector<int>& /* out */ resPartId)
```

Given a mesh entity and part handle on which the entity exists (termed source part handle), get a list of residence part ids.

```
int FMDB_Ent_GetNumGhost (
        pMeshEnt  /* in */  meshEnt,
        pPart  /* in */  part,
        int*  /* out */  numGhost)
```

Given a mesh entity and its source part handle, get the number of ghost copies.

```
int FMDB_Ent_GetAllGhost (
pMeshEnt  /* in */  meshEnt,
pPart  /* in */  part,
std::vector<std::pair<int, pMeshEnt> >& /* inout */ vecGhost)
```

Given a mesh entity and its source part handle, get the vector *vecGhost* filled with pair(s) of global part id and the memory address of ghost copies.

```
int FMDB_Ent_GetGhost (
        pMeshEnt  /* in */  meshEnt,
        int  /* in */  part_id,
        pMeshEnt&  /* out */  ghostEnt)
```

Given a mesh entity and part id, get ghost copy on the part.

```
int FMDB_Topo_SetWeight (
        pMeshMdl /* in */ mesh,
        int /* in */ topo,
        double /* in */ weight)
```

Given a mesh instance, entity topology topo, and double weight value weight, set the weight of the entities of topology to weight for dynamic mesh partitioning control. The specified entity weight is maintained even after the entity is migrated to other part. In FMDB_Mesh_GlobPtn, if no specific weight is set, the default weight for each entity is 1.0. Weight control is not available for static mesh partitioning (initial mesh partitioning performed in FMDB_Mesh_LoadFromFile).

```
int FMDB_Topo_DelWeight (
        pMeshMdl /* in */ mesh,
        int    /* in */ topo)
```

Given a mesh instance and entity topology topo, delete the weight of the entities of topology.

```
int FMDB_Ent_SetWeight (
        pMeshEnt  /* in */  meshEnt,
        double  /* in */  weight)
```

Given a mesh entity handle and double weight value, set the weight of the entity.

```
int FMDB_Ent_GetWeight (
        pMeshEnt /* in */ meshEnt,
        double* /* out */ weight)
```

Given a mesh entity handle, get the weight of the entity.

```
int FMDB_Ent_DelWeight (pMeshEnt  /* in */  meshEnt)
```

Given a mesh entity handle, delete the weight of the entity.

5.4.9 Miscellaneous

```
int FMDB_Ent_DspInfo (
        pPart /* in */ part,
        pMeshEnt /* in */ ent)
```

Displays entity's information (such as id (if vertex), geometric classification, partition classification, the number of remote copies, etc.)

```
int FMDB_Ent_DspAllAdj (pMeshEnt /* in */  ent)
```

Reserved.

```
int FMDB_Topo_Type(int /* in */ topo)
```

Given entity topology, returns entity type.

```
int FMDB_Topo_NumDownAdj (
        int /* in */ topo,
        int /* in */ downType)
```

Given FMDB topology (FMDB_TET, FMDB_HEX, etc.) and downward entity type (FMDB_VERTEX, FMDB_EDGE, etc.) return the number of downward entities of the type e.g. FMDB_Topo_NumDownAdj (FMDB_TET, FMDB_FACE) returns 4. e.g. FMDB_Topo_NumDownAdj (FMDB_PRISM, FMDB_EDGE) returns 9.

```
int FMDB_Topo_GetTopoFromFaces(
        int  /* in */  numFace,
        pMeshEnt*  /* out */  pMeshEnt)
```

Given a list of mesh faces, return region topology.

5.5 Entity Set Functions

5.5.1 Entity set management

Except it is explicitly stated in an API, all entity set level APIs can be used for any type of entity sets, including list or set type, partition object type or non partition object type.

```
int FMDB_Set_Create (
        pMeshMdl  /* in */ mesh,
        pPart  /* in */  part,
        int  /* in */  set_type,
        pEntSet&  /* out */  entSet)
```

Given a mesh and/or a part handle, and desired set type, create an entity set. The supported set type is FMDB_LSET, FMDB_SSET, or FMDB_PSET. FMDB_LSET: a list type entity set without single part constraint. Stored in a mesh instance so part handle is not required for FMDB_LSET creation. FMDB_SSET: a set type entity set without single part constraint. Stored in the mesh instance so part handle is not required for FMDB_SSET creation. FMDB_PSET: a set type entity set with single part constraint. Stored in the part and can be designated as partition object for migration. The part handle must be specified for FMDB_PSET creation.

```
int FMDB_Set_Del (
        pMeshMdl  /* in */ mesh,
        pPart  /* in */  part,
        pEntSet  /* inout */  entSet)
```

Given a mesh and/or a part handle, delete the entity set. FMDB_LSET: delete the set from the mesh instance so part handle is not required for FMDB_LSET deletion. FMDB_SSET: delete the set from the mesh instance so part handle is not required for FMDB_SSET deletion. FMDB_PSET: delete the set from the part handle.

```
int FMDB_Set_AddEnts (
        pEntSet  /* inout */  entSet,
        vector<pMeshEnt>&  /* in */ vecMeshEnt)
```

Given an entity set handle and a vector container of entities, add all the entities in the vector into the entity set.

```
int FMDB_Set_AddEnt (
        pEntSet  /* inout */  entSet,
        pMeshEnt  /* in */  meshEnt)
```

Given an entity set handle and a mesh entity handle, add the entity into the entity set. If *entSet* is a P-Set, *meshEnt* should not be in any other P-Set, and it should not be in the give *entSet* already.

```
int FMDB_Set_RmvEnt (
        pEntSet  /* inout */  entSet,
        pMeshEnt  /* in */  meshEnt)
```

Given an entity set handle and a mesh entity handle, remove the entity from the entity set.

```
int FMDB_Set_Clr (pEntSet  /* inout */  entSet)
```

Given an entity set handle, clear its content.

5.5.2 Entity set information

```
int FMDB_Set_IsEmpty (
        pEntSet  /* in */  entSet
        int* /* out */  isEmpty)
```

Given an entity set handle, check if the entity set is empty. If yes, *isEmpty* is non-zero integer, if no, *isEmpty* is zero.

```
int FMDB_Set_ID (pEntSet  /* in */  entSet)
```

Given an entity set, return the unique ID. The part-wide unique entity set ID should be created via *FMDB_Part_*SetEntSetID prior to calling this function. Note the part-wide unique entity set ID becomes invalid when the entity set is created or deleted either by mesh modifications or migrations.

```
int FMDB_Set_IsEmpty (
        pEntSet  /* in */  entSet
        int* /* out */  isEmpty)
```

Given an entity set handle, check if the entity set is empty. If yes, *isEmpty* is non-zero integer, if no, *isEmpty* is zero.

```
int FMDB_Set_FindEnt (
        pEntSet  /* in */  entSet,
        pMeshEnt  /* in */  meshEnt,
        int*  /* out */  found)
```

Given an entity set and mesh entity, check if the entity set contains the mesh entity. If the entity set contains the entity set, *found* is non-zero integer, otherwise, *found* is 0.

```
int FMDB_Set_GetNumEnt {
        pEntSet  /* in */  entSet,
        int*  /* out */  numEnt)
```

Given an entity set, get the number entities stored in the entity set.

```
int FMDB_Set_GetType {
        pEntSet  /* in */  entSet,
        int*  /* out */  set_type)
```

Given an entity set, decide its type. set_type is FMDB_SSET, FMDB_LSET, or FMDB_PSET.

5.5.3 Traversal over entity sets

```
int FMDB_MeshSetIter_Init (
        pMeshMdl  /* in */  mesh,
        pMeshSetIter&  /* out */  iter)
```

Given a mesh, initiate an iterator *iter* and let it point to the first entity set in the mesh. This will not iterate any set inside each part of the mesh. This does not require deleting the iterator after the iteration is done.

```
int FMDB_MeshSetIter_GetNext (
        pMeshMdl  /* in */  mesh,
        pMeshSetIter  /* in */  iter,
        pEntSet& /* out */ entSet)
```

Given a mesh and a set iterator, get the current entity set that *iter* is pointing to. Then forward *iter* to the next entity set sequentially.

```
int FMDB_MeshSetIter_IsEnd (
        pMeshMdl  /* in */  mesh,
        pMeshSetIter  /* in */  iter,
        int* /* out */ isEnd)
```

Given a mesh and a set iterator, check if *iter* is the end of entity set sequence in the mesh.

```
int FMDB_MeshSetIter_Reset (
        pMeshMdl  /* in */  mesh,
        pMeshSetIter&  /* inout */  iter)
```

Given a mesh and a mesh set iterator, reset *iter* to point to the first entity set in the mesh.

```
int FMDB_MeshSetIter_Del(pMeshSetIter  /* in */  iter);
```

Delete mesh set iterator – see http://redmine.scorec.rpi.edu/projects/fmdb/wiki/Iterator

```
int FMDB_PartSetIter_Init (
        pPart  /* in */  part,
        pPartSetIter&  /* out */  iter)
```

Given a part, initiate an part set iterator and let it point to the first set in the part. This does not require deleting the iterator after the iteration is done.

```
int FMDB_PartSetIter_GetNext (
        pPart  /* in */  part,
        pPartSetIter  /* in */  iter,
        pEntSet& /* out */ entSet)
```

Given a part and a part set iterator, get the current set pointed by the iterator. Then forward the iterator to the next entity set sequentially.

```
int FMDB_PartSetIter_IsEnd (
        pPart  /* in */  part,
        pPartSetIter  /* in */  iter,
        int* /* out */ isEnd)
```

Given a part and a part set iterator, check if *iter* is the end of entity set sequence in the part.

```
int FMDB_PartSetIter_Reset (
        pPart  /* in */  part,
        pPartSetIter&  /* inout */  iter)
```

Given a part and a part set iterator, reset *iter* to point to the first entity set in the part.

```
int FMDB_PartSetIter_Del (pPartSetIter /* in */ iter);
```

Delete the part set iterator – see http://redmine.scorec.rpi.edu/projects/fmdb/wiki/Iterator

5.5.4 Set traversal over entities

```
int FMDB_SetEntIter_Init (
        pEntSet  /* in */  entSet,
        int  /* in */  type,
        int  /* in */  topo,
        pSetEntIter&  /* out */  iter)
```

Given an entity set, an entity type (0 to 3) and an entity topology (0 to 7), get a pointer to the first mesh entity of the type and topo. If no entity exists or *type* is invalid, a null pointer is returned. Whenever a set entity iterator is created, it must be deleted after the iteration is done.

```
int FMDB_SetEntIter_GetNext (
        pSetEntIter  /* in */  iter,
        pMeshEnt& /* out */  meshEnt)
```

Given a set entity iterator, get the current mesh entity that *iter* is pointing to. Then forward the iterator to the next entity sequentially.

```
int FMDB_SetEntIter_IsEnd (
        pSetEntIter  /* in */  iter,
        int* /* out */ isEnd)
```

Given a set entity iterator, check if *iter* is the end of entity sequence.

```
int FMDB_SetEntIter_Reset (pSetEntIter  /* inout */  iter)
```

Given a set entity iterator, reset *iter* to point to the first satisfying entity.

```
int FMDB_SetEntIter_Del (pSetEntIter  /* inout */  iter)
```

Given a set entity iterator, delete it in order to avoid memory leak.

5.5.5 Entity set tag

```
int FMDB_Set_DelTag (
        pEntSet  /* in */  entSet,
        pTag  /* in */  tag)
```

Given an entity set handle and tag handle, delete the tag from the entity set. If the tag doesn't exist with the entity set, the error code *SCUtil_TAG_NOT_FOUND* is returned.

```
int FMDB_Set_HasTag (
        pEntSet  /* in */  entSet,
        pTag  /* in */  tag,
        int*  /* in */  hasTag)
```

Given an entity set handle and tag handle, if the tag is attached to the entity set, *hasTag* is 1, otherwise, 0. If the tag doesn't exist with the entity set, the error code *SCUtil_TAG_NOT_FOUND* is returned.

```
int FMDB_Set_GetTag (
        pEntSet  /* in */  entSet,
        std::vector<pTag>& /* out */  vecTag)
```

Given an entity set handle, get the vector *vecTag* filled with all tag handles attached to the entity set.

```
int FMDB_Set_SetByteTag (
        pEntSet  /* in */  entSet,
        pTag  /* in */  tag,
        const void*  /* in */  tagVal,
        int  /* in */  tagByte)
```

Given an entity set, tag handle of any type, byte type data and the size of data in byte (*tagByte*), set or update the tag value of the entity set. If the tag doesn't exist with the entity set, the error code *SCUtil_TAG_NOT_FOUND* is returned.

```
int FMDB_Set_GetByteTag (
        pMeshEnt  /* in */  meshEnt,
        pTag  /* in */  tag,
        void*  /* out */  tagVal,
        int*  /* out */  tagByte)
```

Given an entity set handle and tag handle, get the byte type data tagged to the entity set. *tagByte* indicates the size of tag data in byte. If the tag doesn't exist with the entity set, the error code *SCUtil_TAG_NOT_FOUND* is returned.

```
int FMDB_Set_SetIntTag (
        pEntSet  /* in */  entSet,
        pTag  /* in */  tag,
        const int  /* in */  tagVal)
```

Given an entity set handle, tag handle, and integer data, set or update the tag value of the entity set. If tag type is not integer type or tag size is not 1, *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Set_GetIntTag (
        pEntSet  /* in */  entSet,
        pTag  /* in */  tag,
        int*  /* out */  tagVal)
```

Given an entity set handle and tag handle, get the integer data tagged to the entity set. If the tag doesn't exist with the entity set, the error code *SCUtil_TAG_NOT_FOUND* is returned. If tag type is not integer type or tag size is not 1, *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Set_SetDblTag (
        pEntSet  /* in */  entSet,
        pTag  /* in */  tag,
        const double  /* in */  tagVal)
```

Given an entity set handle, tag handle, and double data, set or update the tag value of the entity set. If tag type is not double type or tag size is not 1, *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Set_GetDblTag (
        pEntSet  /* in */  entSet,
        pTag  /* in */  tag,
        double*  /* out */  tagVal)
```

Given an entity set handle and tag handle, get the double data tagged to the entity set. If the tag doesn't exist with the entity set, the error code *SCUtil_TAG_NOT_FOUND* is returned. If tag type is not double type or tag size is not 1, *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Set_SetEntTag (
        pEntSet  /* in */  entSet,
        pTag  /* in */  tag,
        const pMeshEnt /* in */  tagVal)
```

Given an entity set, tag handle, and mesh entity, set or update the tag value of the entity set. If tag type is not entity type or tag size is not 1, *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Set_GetEntTag (
        pEntSet  /* in */  entSet,
        pTag  /* in */  tag,
        pMeshEnt*  /* out */  tagVal)
```

Given an entity set and tag handle, get the entity tagged to the entity set. If the tag doesn't exist with the entity set, the error code *SCUtil_TAG_NOT_FOUND* is returned. If tag type is not entity type or tag size is not 1, *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Set_SetSetTag (
        pEntSet  /* in */  entSet,
        pTag  /* in */  tag,
        const pEntSet  /* in */  tagVal)
```

Given an entity set, tag handle, and another entity set, set or update the tag value of the entity set. If tag type is not entity set type or tag size is not 1, the error code *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Set_GetSetTag (
        pEntSet  /* in */  entSet,
        pTag  /* in */  tag,
        pEntSet*  /* out */  tagVal)
```

Given an entity set and tag handle, get the entity set tagged to the entity set. If the tag doesn't exist with the entity set, the error code *SCUtil_TAG_NOT_FOUND* is returned. If tag type is not entity set type or tag size is not 1,the error code *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Set_SetIntArrTag (
        pEntSet  /* in */  entSet,
        pTag  /* in */  tag,
        const int*  /* in */  data_arr,
        int  /* in */  data_size)
```

Given an entity set, tag handle, and integer array of size *data_size*, set or update the tag value of the entity set. If tag type is not integer type or tag size is 1, the error code *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Set_GetIntArrTag (
        pEntSet  /* in */  entSet,
        pTag  /* in */  tag,
        int**  /* out */  data_arr,
        int*  /* out */  data_size)
```

Given an entity set handle and tag handle, get integer array data tagged to the entity set. *data_size* is the size of *data_arr*. If the tag doesn't exist with the entity set, the error code *SCUtil_TAG_NOT_FOUND* is returned. If tag type is not integer type or tag size is 1, the error code *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Set_SetDblArrTag (
        pEntSet  /* in */  entSet,
        pTag  /* in */  tag,
        const double*  /* in */  data_arr,
        int  /* in */ data_size)
```

Given an entity set, tag handle, and double array of size *data_size*, set or update the tag value of the entity set. If tag type is not double type or tag size is 1, the error code *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Set_GetDblArrTag (
        pEntSet  /* in */  entSet,
        pTag  /* in */  tag,
        double**  /* out */  data_arr,
        int*  /* out */  data_size)
```

Given an entity set handle and tag handle, get double array data tagged to the entity set. *data_size* is the size of *data_arr*. If the tag doesn't exist with the entity set, the error code *SCUtil_TAG_NOT_FOUND* is returned. If tag type is not double type or tag size is 1, the error code *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Set_SetEntArrTag (
        pEntSet  /* in */  entSet,
        pTag  /* in */  tag,
        const pMeshEnt*  /* in */  data_arr,
        int  /* in */  data_size)
```

Given an entity set handle, tag handle, and entity array data of size *data_size*, set or update the tag value of the entity set. If tag type is not entity type or tag size is 1, *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Set_GetEntArrTag (
        pEntSet  /* in */  entSet,
        pTag  /* in */  tag,
        pMeshEnt**  /* out */  data_arr,
        int*  /* out */  data_size)
```

Given an entity set handle and tag handle, get entity array data tagged to the entity set. *data_size* is the size of *data_arr*. If the tag doesn't exist with the entity set, the error code *SCUtil_TAG_NOT_FOUND* is returned. If tag type is not entity type or tag size is 1, *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Set_SetSetArrTag (
        pEntSet  /* in */  entSet,
        pTag  /* in */  tag,
        const pEntSet*  /* in */  data_arr,
        int  /* in */  data_size)
```

Given an entity set handle, tag handle, and entity set array data of size *data_size*, set or update the tag value of the entity set. If tag type is not entity set type or tag size is 1, the error code *SCUtil_INVALID_TAG* is returned.

```
int FMDB_Set_GetSetArrTag (
        pEntSet  /* in */  entSet,
        pTag  /* in */  tag,
        pEntSet**  /* out */  data_arr,
        int*  /* out */  data_size)
```

Given an entity set handle and tag handle, get entity set array data tagged to the entity set. *data_size* is the size of *data_arr*. If the tag doesn't exist with the entity, the error code *SCUtil_TAG_NOT_FOUND* is returned. If tag type is not entity set type or tag size is 1, *SCUtil_INVALID_TAG* is returned.

5.5.6 Entity set in parallel

```
int FMDB_Set_GetWeight (
        pEntSet  /* in */  entSet,
        double*  /* out */  weight)
```

Given an entity set handle, get the weight of the entity set. The weight of entity set is the summation of consisting entities's weight.

5.5.7 Miscellaneous

```
int FMDB_Set_DspInfo (pEntSet  /* in */  entSet)
```

Given an entity set handle, display set information such as consisting entities, subsets, parents, and children.

5.6 Partition Object Functions

```
int FMDB_Ent_GetAdjPtnObj (
        pMeshEnt  /* in */  meshEnt,
        vector<pPtnObj>&  /* out */  adjPtnObj)
```

Given a mesh entity handle, get the adjacent partition objects.

```
int FMDB_PtnObj_GetType (
        pPtnObj  /* in */  ptn_obj,
        int*  /* out */  type)
```

Given a partition object handle, get the type, FMDB_PSET, FMDB_REGION, or FMDB_FACE.

6 Sample Program

6.1 Loading/Exporting Mesh

```
// ****************************************************************************
// The program gets two command line arguments
// argv[1] is input mesh file and argv[2] is output mesh file
// ****************************************************************************

#include <iostream>
#include "FMDB.h"

// this function print out "msg" and if oK is positive,
// displays "OK" otherwise, "FAILED"
void CHECK(char *msg, int oK, ...);

int main (int argc, char *argv[])
{
  char mesh_file[1024], out_file [1024];

  SCUTIL_Init(NULL);     // initialize services

  pMeshMdl mesh;
  FMDB_Mesh_Create (NULL, mesh);               // create mesh instance

  pGeomMdl geom;
  FMDB_Mesh_GetGeomMdl (mesh, geom);
  CHECK ("\n* Checking Geom Model...", geom==NULL);

  FMDB_Mesh_LoadFromFile(mesh, argv[1], 0);  // load a serial mesh
  FMDB_Mesh_WriteToFile(mesh, argv[2], 0);   // export mesh into file
  FMDB_Mesh_DspNumEnt (mesh);                  // display mesh size information

  FMDB_Mesh_Del (mesh);                        // delete mesh instance

  SCUTIL_Finalize();                             // finalize services
  return 0;
}

void CHECK(char *msg, int oK, int myrank=0, ...)
{
  assert(oK);
  if (myrank) return;
  char buff[1024];
  va_list  args;
  va_start (args, msg);
  vsprintf(buff, msg, args);
  va_end (args);
  if (oK)
    fprintf(stdout,"%s OK \n",buff);
  else
    fprintf(stdout,"%s FAILED\n", buff);
```

```
  fflush(stdout);
}
```

6.2 Tag Management

```
template <class T>
void TEST_TAG (pMeshMdl mesh, pTag tag, char* in_name, int name_len,
               int in_type, int in_size)
{
  char tag_name[name_len];
  int tag_type, tag_size, tag_byte;
  // verifying byte tag info
  FMDB_Tag_GetName(mesh, tag, tag_name);
  FMDB_Tag_GetType (mesh, tag, &tag_type);
  FMDB_Tag_GetSize (mesh, tag, &tag_size);
  FMDB_Tag_GetByte (mesh, tag, &tag_byte);
  assert(!strncmp(tag_name, in_name, name_len));
  assert(tag_type == in_type);
  assert(tag_size == in_size);
  assert(tag_byte==sizeof(T)*tag_size);
  CHECK ("* Checking Tag Information...", !strncmp(tag_name, in_name, name_len)
         && tag_type == in_type && tag_size == in_size && tag_byte==sizeof(T)*tag_size,
         SCUTIL_CommRank());
}

int main()
{
   ...
  pTag int_tag, dbl_tag, ent_tag, byte_tag, clone_tag;
  int traceable=1, tag_type, tag_size, exist;
  char tag_name[256];

  CHECK("\n* Creating Single Data Tags...", 1, SCUTIL_CommRank());
  FMDB_Mesh_CreateTag (mesh, "byte", SCUtil_BYTE, 7, 1,  byte_tag);
  FMDB_Mesh_CreateTag (mesh, "integer", SCUtil_INT, 1,  1, int_tag);
  FMDB_Mesh_CreateTag (mesh, "double", SCUtil_DBL, 1, 1, dbl_tag);
  FMDB_Mesh_CreateTag (mesh, "entity", SCUtil_ENT, 1, 1, ent_tag);

  CHECK("* Checking Aborting Duplicate Tags...",
        FMDB_Mesh_CreateTag (mesh, "byte", SCUtil_BYTE, 7, 0, byte_tag)
        && FMDB_Mesh_CreateTag (mesh, "integer", SCUtil_INT, 1, 1, int_tag)
        && FMDB_Mesh_CreateTag (mesh, "double", SCUtil_DBL, 1, 1, dbl_tag)
        && FMDB_Mesh_CreateTag (mesh, "entity", SCUtil_ENT, 1, 1, ent_tag)
        , SCUTIL_CommRank());

  // verifying tag info
  TEST_TAG<char>(mesh, byte_tag, "byte", 100, SCUtil_BYTE, 7);
  TEST_TAG<int>(mesh, int_tag, "integer", 100, SCUtil_INT, 1);
  TEST_TAG<double>(mesh, dbl_tag, "double", 100, SCUtil_DBL, 1);
  TEST_TAG<pMeshEnt>(mesh, ent_tag, "entity", 100, SCUtil_ENT, 1);

  FMDB_Mesh_HasTag(mesh, int_tag, &exist);
```

```
CHECK ("* Checking Tag Existence...", exist, SCUTIL_CommRank());

FMDB_Mesh_FindTag (mesh, "generic byte", clone_tag);
CHECK ("* Checking Tag Handle Retrieval..", clone_tag == byte_tag, SCUTIL_CommRank());

FMDB_Mesh_GetTag (mesh, tag_ids);
CHECK ("* Checking Tag Retrieval..", tag_ids.size()==4, SCUTIL_CommRank());
```

6.3 Part Information

```
int partDim, numEnt, numVt, numEg, numFc, numRg;
vector<pPart> parts;

FMDB_Mesh_GetAllPart (mesh, SCUTIL_CommRank(), parts);
CHECK ("* Checking Initial Part Retrieval...", parts.size()==1, SCUTIL_CommRank());
pPart part = parts.at(0);

FMDB_Part_GetNumEnt(part, FMDB_VERTEX, FMDB_ALLTOPO, &numVt);
FMDB_Part_GetNumEnt(part, FMDB_EDGE, FMDB_ALLTOPO, &numEg);
FMDB_Part_GetNumEnt(part, FMDB_FACE, FMDB_ALLTOPO, &numFc);
FMDB_Part_GetNumEnt(part, FMDB_REGION, FMDB_ALLTOPO, &numRg);

FMDB_Part_GetNumEnt(part, FMDB_ALLTYPE,  FMDB_ALLTOPO, &numEnt);
CHECK ("* Checking Part Ent Type/Topo Functions...", numEnt==numVt+numEg+numFc+numRg,
        SCUTIL_CommRank());
```

6.4 Entity Adjacency

```
pPartEntIter entIter;
vector<pMeshEnt> adjEnts;
for (int type=FMDB_VERTEX; i<FMDB_REGION; ++i)
{
  int iterEnd = FMDB_PartEntIter_Init (part, type,  FMDB_ALLTOPO, entIter);
  while (!iterEnd)
  {
    iterEnd = FMDB_PartEntIter_GetNext(entIter, ent);
    if (iterEnd) break;
    for (int j=FMDB_VERTEX; j<FMDB_REGION; ++j)
    {
      if (i==j) continue;
      adjEnts.clear();
      FMDB_Ent_GetAdj(ent, j, 1, adjEnts);
      FMDB_Ent_GetNumAdj (ent, j, &numAdj);
      assert (adjEnts.size()==((size_t)numAdj));
    }
  }
  FMDB_PartEntIter_Del (entIter);
}
```

6.5 Entity Creation

```
pMeshEnt* new_vtx=new pMeshEnt[3];

double coords_0[] = {0.0, 0.0, 0.0};
double params[] = {0.0, 0.0, 0.0};
FMDB_Vtx_Create (part, (pGeomEnt)NULL, coords_1, params, new_vtx[0]);

double coords_1[] = {0.0, 0.5. 0.5};
FMDB_Vtx_Create (part, (pGeomEnt)NULL, coords_2, params, new_vtx[1]);

double coords_2[] = {1.0, 0.5. 0.5};
FMDB_Vtx_Create (part, (pGeomEnt)NULL, coords_3, params, new_vtx[2]);

pMeshEnt new_edge;
FMDB_Edge_Create(part, (pGeomEnt)NULL, new_vtx[0], new_vtx[1], new_edge);

pMeshEnt* new_face=new pMeshEnt[4];
FMDB_Face_Create(part, (pGeomEnt)NULL, FMDB_TRI, new_vtx, 0, new_face[0]);
...

pMeshEnt new_tet;
if (FMDB_Ent_Find(FMDB_TET, new_face, 4, new_tet) == SCUtil_ENTITY_NOT_FOUND)
  FMDB_Rgn_Create(part, (pGeomEnt)NULL, FMDB_TET, new_face, 4, new_tet);
```

6.6 Part Entity Iterator

```
pMeshEnt ent;
pPartEntIter entIter;

int iterEnd = FMDB_PartEntIter_Init (part, i, FMDB_ALLTOPO, entIter);
while (!iterEnd)
{
  iterEnd = FMDB_PartEntIter_GetNext(entIter, ent);
  if(iterEnd) break;
  ...
} // while
FMDB_PartEntIter_Del (entIter);
```

6.7 Reverse Clssification Iterator

```
pGeomModel geomModel;
FMDB_Mesh_GetGeomMdl (mesh, geomModel);

// Get geometric model vertices and iterate over the mesh entities
// classified on them
GVIter vtxIter = GM_vertexIter(geomModel);
pGVertex gVtx;
pMeshEnt ent;
pGeomEnt geomEnt;
```

```
while(gVtx = GVIter_next(vtxIter))
{
  pPartEntIter revIter;

  int iterEnd = FMDB_PartEntIter_InitRevClas (part, (pGEntity)gVtx, revIter);
  while (!iterEnd)
  {
    iterEnd = FMDB_PartEntIter_GetNext(revIter, ent);
    if(iterEnd) break;
    ...
  }
  FMDB_PartEntIter_Del (revIter);
}
GVIter_delete(vtxIter);
```

6.8 Part Boundary Iterator

```
pMeshEnt ent;
int partDim;
FMDB_Part_GetDim (part, &partDim);

// traverse on all part boundary entities (vertices, edges and faces)
for(int type = 0; type < partDim; type++)
{
  pPartEntIter pbiter;
  int iterEnd =  FMDB_PartEntIter_InitPartBdry(part, -1, type, pbIter);

  while(!iterEnd)
  {
    iterEnd = FMDB_PartEntIter_GetNext(pbIter, ent);
    if(iterEnd) break;
    ...
  }
  FMDB_PartEntIter_Del (pbIter);
}
```

6.9 Part Tag

```
pMeshEnt ent;
FMDB_PartEntIter_Init (part, FMDB_VERTEX, FMDB_POINT, iter); // retrieve the first vertex
FMDB_PartEntIter_GetNext(iter, ent);
FMDB_PartEntIter_Del(iter);

char data[]= "abcdefg";
void* byte_data = (void*)calloc(256, sizeof(char));

FMDB_Part_SetIntTag (part, int_tag, 1000);
FMDB_Part_SetDblTag (part, dbl_tag, 1000.37);
FMDB_Part_SetEntTag (part, ent_tag, ent);
```

```
FMDB_Part_SetByteTag (part, byte_tag, data, 7);

FMDB_Part_GetIntTag (part, int_tag, &int_data);
FMDB_Part_GetDblTag (part, dbl_tag, &dbl_data);
FMDB_Part_GetEntTag (part, ent_tag, &ent_data);
FMDB_Part_GetByteTag (part, byte_tag, &byte_data, &byte_size);

// using Set/GetByteTag with integer tag
char byte_int_data[sizeof(int)];
int* intPntData = new int(1000);
char *charPntData = (char*)intPntData;

for ( int i=0; i<sizeof(int); i++ )
  byte_int_data[i] =  charPntData[i];
delete intPntData;

FMDB_Part_SetByteTag(part, int_tag, byte_int_data, sizeof(int));
FMDB_Part_GetByteTag (part, int_tag, &byte_data, &byte_size);
CHECK ("Cheking Set/GetByteTag with integer tag...", byte_size == (int)sizeof(int)
       && *(int*)byte_data == 1000, SCUTIL_CommRank());

FMDB_Part_DelTag (part, int_tag);
FMDB_Part_DelTag (part, dbl_tag);
FMDB_Part_DelTag (part, ent_tag);
FMDB_Part_DelTag (part, byte_tag);
```

6.10 Entity Set Management

```
// This program creates a P-entity set
// and traverses entities contained in the set
pEntSet mySet, set;
FMDB_Set_Create(mesh, part, FMDB_PSET, mySet);
int iterEnd = FMDB_PartSetIter_Init (part, psetIter);
while(!iterEnd)
{
  iterEnd = FMDB_PartSetIter_GetNext(part, psetIter, set);
  if(iterEnd) break;
  ...
}
iterEnd = FMDB_PartSetIter_Reset (part, psetIter);
FMDB_Set_Del (mesh, part, mySet);
```

6.11 Entity Set Manipualation

```
iterEnd = FMDB_PartEntIter_Reset(iter);
while (!iterEnd)
{
  iterEnd = FMDB_PartEntIter_GetNext(iter, ent);
  if(iterEnd) break;
  FMDB_Set_AddEnt (listEntSet, ent);
```

```
  FMDB_Set_AddEnt (listEntSet, ent);
  FMDB_Set_AddEnt (setEntSet, ent);
  FMDB_Set_AddEnt (setEntSet, ent);
}
FMDB_Set_GetNumEnt (listEntSet, &numEnt);
CHECK ("* Checking List-type Set Data Insertion...", numEnt==num_type[0]*2,
        SCUTIL_CommRank());
FMDB_Set_GetNumEnt (setEntSet, &numEnt);
CHECK ("* Checking Set-type Set Data Insertion...", numEnt==num_type[0], SCUTIL_CommRank());

FMDB_Set_Clr (listEntSet);
FMDB_Set_Clr (setEntSet);
int isListSetEmpty=0, isSetSetEmpty=0;
FMDB_Set_IsEmpty (listEntSet, &isListSetEmpty);
FMDB_Set_IsEmpty (setEntSet, &isSetSetEmpty);
CHECK ("* Checking Empty Set...", isListSetEmpty && isSetSetEmpty, SCUTIL_CommRank());
```

6.12 Entity Set Iterator

```
pSetEntIter esIter;
int iterEnd = FMDB_SetEntIter_Init (listEntSet, FMDB_ALLTYPE, FMDB_POINT, esIter);
while (!iterEnd)
{
  iterEnd = FMDB_SetEntIter_GetNext(esIter, ent);
  if(iterEnd) break;
  ...
}
FMDB_SetEntIter_Reset (esIter);
```

6.13 Ownership

```
pMeshEnt ent, ownerEnt;
int isEnd, ownerPartID;

pPartEntIter iter;

isEnd =  FMDB_PartEntIter_InitPartBdry(part, FMDB_NONE, FMDB_ALLTYPE, FMDB_ALLTOPO, iter);
while(!isEnd)  // traverse on all part boundary edges
{
  iterEnd = FMDB_PartEntIter_GetNext(iter, ent);
  if (iterEnd) break;
  // Get entity's owner part, part boundary entities are shared
  // so owner can be the local or remote part
  FMDB_Ent_GetOwnPartID(ent, part, &ownerPartID);

  // Get owner entity, if ent is a copy, this function returns the owning entity of ent
  FMDB_Ent_GetOwnEnt(ent, part, ownerEnt);
}
// delete part boundary iterator
FMDB_PartEntIter_Del (iter);
```

6.14 Remote Copies

```
pMeshEnt ent, rmt, rmtCopy;
pPartEntIter iter;
int isEnd, numCopies, pid, numCopiesNew;
vector<pair<int, pMeshEnt> > vecRmtCpy;
vector<pair<int, pMeshEnt> >::iterator itVec;

isEnd =  FMDB_PartEntIter_InitPartBdry(part,-1, 1, iter);
while(!isEnd) // traverse on all part boundary edges
{
  isEnd =  FMDB_PartEntIter_GetNext(iter, ent);
  if (iterEnd) break;

  // Get all remote copies of the part boundary entity
  vecRmtCpy.clear();
  FMDB_Ent_GetAllRmt(ent, vecRmtCpy);

  // Iterate through all remote copies
  for(itVec = vecRmtCpy.begin(); itVec != vecRmtCpy.end(); itVec++ )
  {
    pid = itVec->first;
    rmt = itVec->second;
    // There should exist a remote copy on pid then, Cross-check with function
    // GetRmt(A O(1) operation for getting remote copy on a target part id)
    FMDB_Ent_GetRmt(ent, pid, rmtCopy);
    CHECK ("* Checking remote copy existence on part... ", rmtCopy==rmt, SCUTIL_CommRank());
  }

  // Get the number of remote copies for each entity
  FMDB_Ent_GetNumRmt(ent,  &numCopies);

  // Clear all remote copies
  FMDB_Ent_ClrRmt(ent);

  // Now add the remote copies again
  for(itVec = vecRmtCpy.begin(); itVec != vecRmtCpy.end(); itVec++ )
  {
    pid = itVec->first;
    rmt = itVec->second;
    FMDB_Ent_SetRmt(ent, pid, rmt);
  }

  // Get the number of remote copies again
  FMDB_Ent_GetNumRmt(ent, &numCopiesNew);
  CHECK("* Checking remote copies after adding copies..." , numCopiesNew == numCopies,
         SCUTIL_CommRank());
}
FMDB_PartEntIter_Del (iter);   // delete part boundary iterator
```

6.15 Mesh Distribution on Multiple Parts

```
for (int numPart=2; numPart<20; ++numPart)
{
  zoltanCB zlb;
  FMDB_Mesh_SetNumPart(mesh, numPart);
  FMDB_Mesh_GlobPtn (mesh, cb, FMDB_HYPERGRAPH, FMDB_REPARTITION, FMDB_NONE, 1.1, 1);

  FMDB_Mesh_Verify(mesh, &isValid);
  CHECK("* Checking Multi-Part Distributed Mesh Validity...", isValid, SCUTIL_CommRank());

  char out_file[256];
  sprintf(out_file,"%dpart_p.sms",numPart);
  CHECK("* Exporting Multi-Part Distributed Mesh ...",
 !FMDB_Mesh_WriteToFile (mesh, out_file, 1), SCUTIL_CommRank());
}
FMDB_Mesh_Del(mesh);
SCUTIL_Finalize();
```

6.16 Migration and Load Balancing

```
// This program randomly picks the entities to migrate and destination part,
// then do migration and load balancing.
MigrCB cb;
pMeshEnt ent;
pPartEntIter iter;
int numProc, procID, meshDim, isValid, numPart;

std::vector<pPart> parts;
FMDB_Mesh_GetAllPart (mesh, SCUTIL_CommRank(), parts);

FMDB_Mesh_GetDim (mesh, &meshDim);
FMDB_Mesh_GetNumPart (mesh, &numPart);

 for (std::vector<pPart>::iterator part_it=parts.begin();
       part_it!=parts.end(); ++part_it)
{
  std::map<pMeshEnt, std::pair<int, int> > EntToMigr;
  std::map<pEntSet, std::pair<int, int> > SetToMigr;

  int iterEnd = FMDB_PartEntIter_Init (*part_it, meshDim, FMDB_ALLTOPO, iter);
  while(!iterEnd)
  {
    iterEnd = FMDB_PartEntIter_GetNext(iter, ent);
    if (iterEnd) break;
    if (rand()%3==1)   // Randomly select partition objects for migration
      EntToMigr.insert(std::map<mEntity*, std::pair<int, int> >::value_type
       (ent, std::make_pair(FMDB_Part_ID(*part_it), rand()%(SCUTIL_CommSize()*numPart))));
  }
  FMDB_PartEntIter_Del (iter);

  FMDB_Mesh_Migr(mesh, cb, EntToMigr, SetToMigr); // migration
```

```
  FMDB_Mesh_GlobPtn (mesh, cb, FMDB_PARMETIS, FMDB_PartKway, 1.03, 0);        // load balance

  FMDB_Mesh_Verify(mesh, &isValid);
  CHECK("\n* Checking Re-partitioned Mesh Validity...", isValid, SCUTIL_CommRank());
}
```

6.17 Ghosting

```
for (int iGhostType = FMDB_REGION; iGhostType <= FMDB_REGION; iGhostType++)
  for (int iBrgType = FMDB_VERTEX; iBrgType < iGhostType; iBrgType++)
    for (int include_copy = 0; include_copy <= 1; include_copy++)
      FMDB_Mesh_CreateGhost(mesh, cb, iGhostType, iBrgType, iNumLayer, include_copy);

FMDB_Mesh_DelGhost(mesh);

int isValid;
FMDB_Mesh_Verify(mesh, &isValid);
CHECK("\n* Checking Ghosted Mesh Validity...", isValid, SCUTIL_CommRank());
```

6.18 Time and Memory Cost

```
// This program displays system information, # processes running and initial memory cost.
// Then, it displays the time and memory cost for mesh loading.

SCUTIL_DspSysInfo();
cout<<"Running Test on "<<SCUTIL_CommSize()<<" processes\n";

FMDB_Mesh_Create (model, mesh);
SCUTIL_DspCurMem("Initial cost: ");

SCUTIL_ResetRsrc();
if (FMDB_Mesh_LoadFromFile (mesh, mesh_file, distributedFlag))
{
  FMDB_Mesh_Del(mesh);
  SCUTIL_Finalize();
  return SCUtil_FAILURE;
}
else
  SCUTIL_DspRsrcDiff("Cost for mesh loading: ");
```

7 Compilation and Execution

7.1 H/W Requirements

FMDB functionality has been developed and tested on the architectures listed in Figure 16.

7.2 S/W Requirements

Figure 16 lists the operating system - processor - compiler combinations that have been tested and are supported; other similar combinations are likely to work. There is no official support for MacOSX. There is no plan to support Microsoft operating systems.

OS	Processor / Word Size (bits)	Compiler
Red Hat 4	Intel Xeon / 32	gcc 3.4
Red Hat 5	AMD Opteron / 64	gcc 4.4
Red Hat 4	AMD Athlon 64 X2 / 64	gcc 3.4
Debian GNU/Linux 5.0	AMD Athlon 64 X2 / 64	gcc 4.3
Debian GNU/Linux 5.0	Intel Core i7 920 / 64	intel 11.1
Cray XE6	12 Core AMD 'MagnyCours' / 64	pgi 10.9

Figure 16: Supported OS-Processor-Compiler Combinations

AutoConf 2.59, M4 6.10, Libtool 1.5.26, and AutoMake 1.10.1 compatible versions are required for compilation.

For compilation with support for distributed meshes Parmetis 3.1.1 and Zoltan 3.1 are required.

OpenMPI is the recommended MPI implementation and was used in testing.

7.3 Installation

The GNU Autotools [2] are used for compilation of FMDB.

Note, in order to install FMDB, all other PUMI components, Zoltan, and ParMETIS should be installed. The following illustrates the instructions to download and install PUMI components including FMDB, See http://www.scorec.rpi.edu/FMDB/buildS.html for detailed compilation instructions.

7.3.1 Checkout the PUMI components

```
svn co http://redmine.scorec.rpi.edu/anonsvn/buildutil/trunk/GNUautoTools/m4Macros m4
svn co http://redmine.scorec.rpi.edu/anonsvn/ipcomman/trunk pcu
svn co http://redmine.scorec.rpi.edu/anonsvn/fmdb/software/trunk/SCORECUtil/SCORECUtil scu
svn co http://redmine.scorec.rpi.edu/anonsvn/gmi/trunk gmi
```

```
svn co http://redmine.scorec.rpi.edu/anonsvn/fmdb/software/trunk/FMDB/FMDB fmdb
```

7.3.2 Set environment variables

```
export export SCOREC_SOFTWARE=/fasttmp/seol/ALBANY/scorec-sw
export DEBUG=-DDEBUG  // debug options
export ZOLTAN_HOME= path where zoltan include and lib folders are found
export PARMETIS_HOME= path where parmetis libraries are found
export MPIHOME= path where MPI include and lib folders are found
export CXX=$MPIHOME/bin/mpicxx;
export CC=$MPIHOME/bin/mpicc;
export FC=$MPIHOME/bin/mpif90;
```

7.3.3 Compile

```
cd $SCOREC_SOFTWARE/pcu
ln -s ../m4 .
autoreconf -fi
export MPI_INSTALL=$MPIHOME
./configure CFLAGS="$DEBUG" CXXFLAGS="$DEBUG" FCFLAGS="$DEBUG" --with-debug=3 \
  MPI_MISSING=false --prefix=$SCOREC_SOFTWARE
make
make install


cd $SCOREC_SOFTWARE/gmi
ln -s ../m4 .
autoreconf -fi
./configure CFLAGS="$DEBUG" CXXFLAGS="$DEBUG" FCFLAGS="$DEBUG" --with-debug=3 \
  --with-scorecutil=$SCOREC_SOFTWARE/scu --with-fmdb=$SCOREC_SOFTWARE/fmdb \
  --prefix=$SCOREC_SOFTWARE
make -j 8
make install

cd $SCOREC_SOFTWARE/scu
ln -s ../m4 .
autoreconf -fi
./configure CFLAGS="$DEBUG" CXXFLAGS="$DEBUG" FCFLAGS="$DEBUG" --with-debug=3 \
  --with-gmi=$SCOREC_SOFTWARE --with-fmdb=$SCOREC_SOFTWARE/fmdb \
  --prefix=$SCOREC_SOFTWARE
make -j 8
make install

cd $SCOREC_SOFTWARE/fmdb
ln -s ../m4 .
autoreconf -fi
./configure CFLAGS="$DEBUG" CXXFLAGS="$DEBUG" FCFLAGS="$DEBUG" --with-debug=3 \
  --enable-paralell MPI_INSTALL=$MPIHOME ZOLTAN_DIR=$ZOLTAN_HOME \
  PARMETIS_DIR=$PARMETIS_HOME PKG_CONFIG_PATH=$SCOREC_SOFTWARE/lib/pkgconfig \
  --prefix=$SCOREC_SOFTWARE
```

```
make -j 8
make install
```

7.3.4 Configuration option for GMI

Pass –enable-meshmodel flag to the configure script to enable mesh model support.

7.3.5 Configuration option for FMDB

- Pass the –enable-parallel flag to the configure script to enable parallel support.
- Pass the –enable-imesh flag to the configure script to enable iMesh support.
- Pass the –enable-imeshp flag to the configure script to enable iMeshP support.
- Pass the –enable-albany flag to the configure script to enable Sandia Albany support.

7.4 Execution

After compilation enter the command 'make check' to execute a series of on part tests.

8 Closing Remark

FMDB is supported by the US Department of Energy's Scientific Discovery through Advanced Computing (SciDAC) program as part of the Interoperable Technologies for Advanced Petascale Simulations (ITAPS) center (http://www.itaps.org).

The latest source and the user's guide is downloadable from http://www.scorec.rpi.edu/FMDB. For all inquiries on FMDB, email to fmdb@scorec.rpi.edu.

References

[1] F. Alauzet, X. Li, E.S. Seol, and M.S. Shephard, "Parallel anisotropic 3d mesh adaptation by mesh modification," *Engineering with Computers,* **21**(3):247–258 (2006).

[2] John Calcote, "Autotools: A Practioner's Guide to GNU Autoconf, Automake, and Libtool," (2010).

[3] M.W. Beall, An object-oriented framework for the reliable automated solution of problems in mathematical physics [dissertation]. Troy (NY): Rensselaer Polytechnic Institute (1999).

[4] M.W. Beall and M.S. Shephard, "A general topology-based mesh data structure," *Int. J. Numer. Meth. Engng.,* **40**(9):1573–1596 (1997).

[5] W. Celes, G.H. Paulino, and R. Espinha, "A compact adjacency-based topological data structure for finite element mesh representation," *Numerical Methods in Engineering,* **64**(11):1529–1565 (2005).

[6] H.L. de Cougny, K.D. Devine, J.E. Flaherty, and R.M. Loy, "Load balancing for the parallel solution of partial differential equations" *Appl. Numer. Math.,* **16**:157–182 (1995).

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

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

[9] R.V. Garimella, "Mesh data structure selection for mesh generation and fea applications" *Int. J. Numer. Meth. Engng.,* **55**:451–478 (2002).

[10] ITAPS: The Interoperable Technologies for Advanced Petascale Simulations center (*http://www.itaps.org*) (2010).

[11] X. Li, M.S. Shephard, and M.W. Beall, "3D anisotropic mesh adaptation by mesh modifications" *Comp. Meth. Appl. Mech. Engng.,* **194:**4915–4950 (2005).

[12] libMesh, *http://libmesh.sourceforge.net* (2005).

[13] M. Mantyla, An Introduction to Solid Modeling. Rockville Maryland: Computer Science Press (1988).

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

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

[16] C. Ollivier-Gooch, K. Chand, T. Dahlgren, L.F. Diachin, B. Fix, J. Kraftcheck, X. Li, E.S. Seol, M.S. Shephard, T. Tautges, H. Trease "The TSTT Mesh Interface" *44th AIAA Aerospace Sciences Meeting and Exhibit,* **AIAA 2006**–529 (2006).

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

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

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

[20] J.F. Remacle and M.S. Shephard, "An algorithm oriented mesh database" *Int. J. Numer. Meth. Engng.,* **58**:349–374 (2003).

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

[22] E.S. Seol, FMDB: Flexible Distributed Mesh Database for Parallel Automated Adaptive Analysis [dissertation]. Troy (NY): Rensselaer Polytechnic Institute (2005). Available from: *http://www.scorec.rpi.edu/cgi-bin/reports/GetByYear.pl?Year=2005.*

[23] E.S. Seol and M.S. Shephard, "Efficient distributed mesh data structure for parallel automated adaptive analysis" *Engineering with Computers*, **22**:197–213 (2006).

[24] M.S. Shephard, "Meshing environment for geometry-based analysis" *Int. J. Numer. Meth. Engng*, 47:169-190 (2000).

[25] Sgi Inc. *http://www.sgi.com/tech/stl/stl_index.html* (2005).

[26] M.S. Shephard, J.E. Flaherty, C.L. Bottasso, and H.L. de Cougny, "Parallel automated adaptive analysis" *Parallel Computing,* **23**:1327–1347 (1997).

[27] J.D. Teresco, M.W. Beall, J.E. Flaherty, and M.S. Shephard, "A hierarchical partition model for adaptive finite element computations" *Comp. Meth. Appl. Mech. Engng.,* **184**:269–285 (2000).

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

[29] K.J. Weiler, "The radial-edge structure: a topological representation for non-manifold geometric boundary representations" *Geometric Modeling for CAD Applications*, p 3–36 (1988).

[30] Zoltan: data management services for parallel applications (2005). Available from: *http://www.cs.sandia.gov/Zoltan.*

A Mesh Verification

/* describe mesh verification api */

A Mesh Statistics

[Example]

```
[MESH STATISTICS]
#process: 2, #part per process: 5, #total part: 10
LB_Method: HYPERGRAPH, LB_Approach: Refine, imbalance_tol: 3%

-- Per-Process Count Statistics --
Global:
#p-set: avg 6558.5, max 7337, min 5780
avg p-set size (#ent): 10

#po-ent & weight: avg 1.29e+05-2.606e+05, max 144427-2.609e+05, min 113621-2.604e+05
Imbalance ratio [max/avg weight]: 0.1105 %
Imbalance ratio [avg/min weight]: 0.1106 %

Imbalance ratio [max/avg #po-ent]: 11.94 %
Imbalance ratio [avg/min #po-ent]: 13.56 %

#p-set & size: 13117-131170
(p0: 5780-57800) (p1: 7337-73370)

#owned+non-owned vertex: 113342
(p0: 55820) (p1: 57522)
#owned+non-owned edge: 479488
(p0: 249232) (p1: 230256)
#owned+non-owned face: 624150
(p0: 337843) (p1: 286307)

#owned vertex: 91289
(p0: 44603) (p1: 46686)
#owned edge: 424057
(p0: 221402) (p1: 202655)
#owned face: 590817
(p0: 321164) (p1: 269653)
#region & weight: 258048-5.213e+05
(p0: 144427-2.609e+05) (p1: 113621-2.604e+05)

-- Per-Part Count Statistics --
Global:
#p-set: avg 1312, max 1566, min 459
avg p-set size (#ent): 10

#po-ent & weight: avg 2.58e+04-5.213e+04, max 39057-5.369e+04, min 19517-4.824e+04
Imbalance ratio [max/avg weight]: 2.999 %
Imbalance ratio [avg/min weight]: 8.07 %

Imbalance ratio [max/avg #po-ent]: 51.36 %
Imbalance ratio [avg/min #po-ent]: 32.22 %

Local (per process) avg/max/min #p-set and avg size:
(p0: avg 1156, max 1419, min 459, size 10) (p1: avg 1467, max 1566, min 1308, size 10)
```

```
Local (per process) #po-ent & weight:
(p0: avg 2.889e+04-5.219e+04, max 39057-5.369e+04, min 24892-4.824e+04)
    (p1: avg 2.272e+04-5.207e+04, max 27234-5.339e+04, min 19517-5.024e+04)


Local (per process) imbalance ratio [max/avg weight]:
(p0: 2.885%) (p1: 2.538%)

Local (per process) imbalance ratio [avg/min weight]:
(p0: 8.19%) (p1: 3.657%)

Local (per process) imbalance ratio [max/avg #po-ent]:
(p0: 35.21%) (p1: 19.85%)

Local (per process) imbalance ratio [avg/min #po-ent]:
(p0: 16.04%) (p1: 16.43%)

#p-set & size: 13117-131170
(p0: 1341-13410 1419-14190 1384-13840 1177-11770 459-4590)
    (p1: 1308-13080 1566-15660 1519-15190 1563-15630 1381-13810)


#owned+non-owned vertex: 113342
(p0: 10963 11734 11981 11096 10046) (p1: 15408 10949 10516 10506 10143)

#owned+non-owned edge: 479488
(p0: 46987 49366 48454 49639 54786) (p1: 57990 44279 43485 41466 43036)

#owned+non-owned face: 624150
(p0: 61976 62963 61325 67763 83816) (p1: 69670 55413 55201 50449 55574)


#owned vertex: 91289
(p0: 10732 8595 9732 8143 7401) (p1: 6493 10609 9809 10506 9269)

#owned edge: 424057
(p0: 46533 41969 42323 42780 47797) (p1: 33963 43764 42071 41466 41391)

#owned face: 590817
(p0: 61750 58735 57417 63848 79414) (p1: 54688 55211 54497 50449 54808)

#region & weight: 258048-5.213e+05
(p0: 25951-5.368e+04 25313-5.369e+04 24892-5.257e+04 29214-5.276e+04 39057-4.824e+04)
    (p1: 27234-5.339e+04 22053-5.337e+04 22202-5.258e+04 19517-5.078e+04 22615-5.024e+04)
```