The PUMI User's Guide

– VERSION 2.1 –

Scientific Computation Research Center
Rensselaer Polytechnic Institute

March 16, 2021

********** SCOREC 3-CLAUSE BSD LICENSE **********

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Questions to shephard@rpi.edu
Derived from the OSI 3-clause BSD

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 unstructured meshes include dynamic mesh load balancing techniques [7, 10, 29, 30], and data structure and algorithms for parallel mesh adaptation [2, 8, 14, 16, 18, 19, 22, 23, 24].

The PUMI is a unstructured, distributed mesh data management system that is capable of a parallel mesh infrastructure capable of handling general non-manifold [15, 31] models and effectively supporting automated adaptive analysis [12].

This document describes an overview of the PUMI design, its interface functions and example codes.

1.2 Organization

Chapter 2 introduces the data sets involved with geometry-based analysis and the role of a topological mesh representation. Chapters 3-5 introduce the PUMI libraries which include parallel control, geometric model, distributed mesh data structure in accordance with a partition model, and mesh partitioning control. Chapters 6-11 present the PUMI API specifications. Chapter 12 presents example programs. Chapter 13 provides compilation and installation instructions.

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.

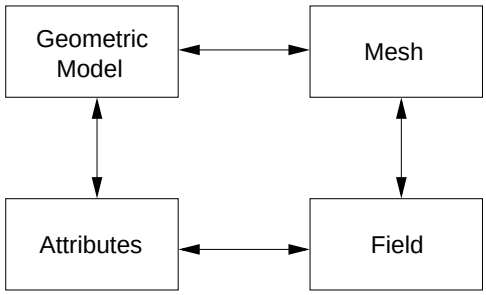


Figure 1: The relationship between components of the geometry-based analysis environment [3]

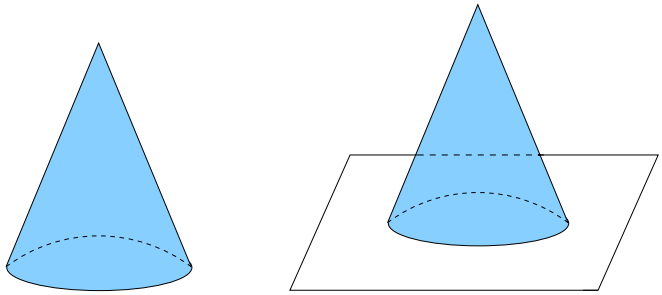


Figure 2: Example of (left) manifold and (right) non-manifold models

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, 28]. 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 [31]. 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,

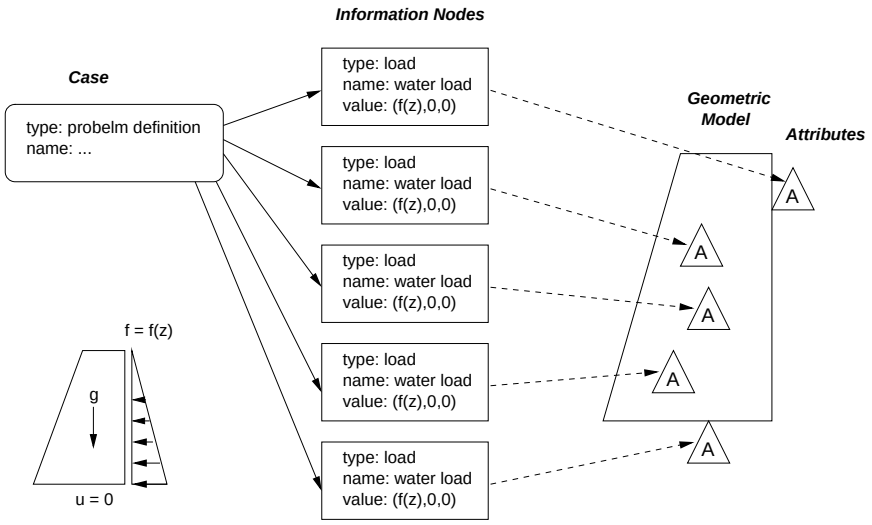


Figure 3: Example geometry-based problem definition [3]

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, 28]. 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, 26], 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, 26]. More discussion on the mesh representation is presented in §2.2.

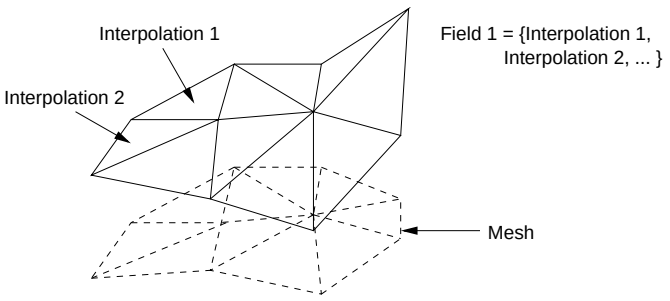


Figure 4: Representation of a field defined over a mesh [3]

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, 6, 11, 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* [13, 17].

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:

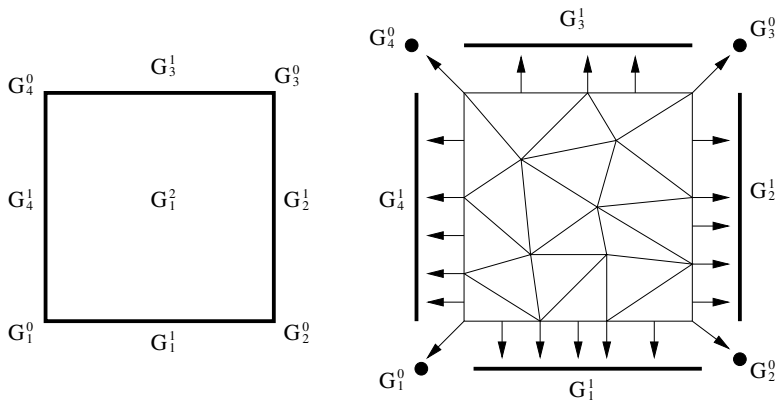


Figure 5: Example of simple model(left) and mesh(right) showing their association via geometric classification [28]

- 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, 26].

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.

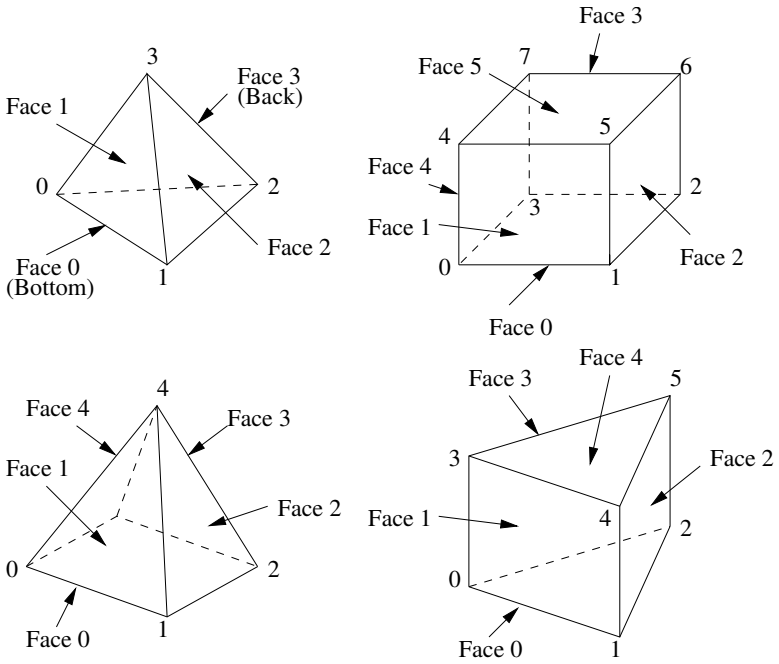


Figure 6: Vertex and face order on a region [28]

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. In the adjacency graph, a solid box denotes that entities of the level are explicitly stored, and a solid arrow denotes that adjacencies from an outgoing level to an incoming level are maintained for the level of entities.

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).

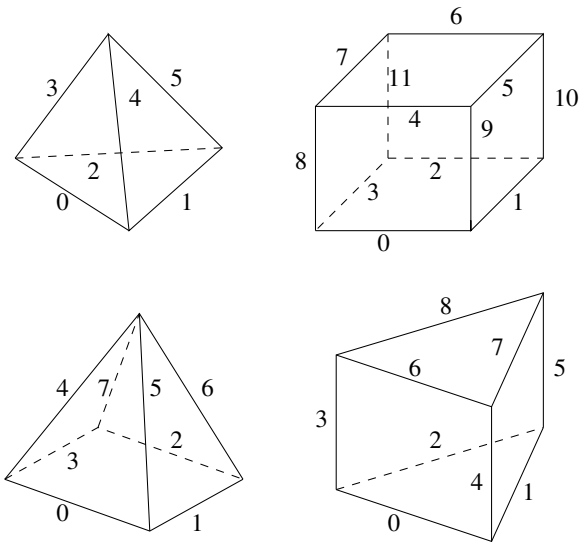


Figure 7: Edge order on a region [28]

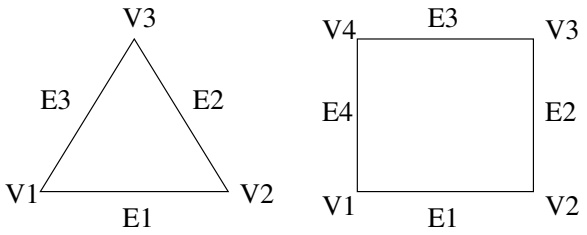


Figure 8: Edge order on a face [28]

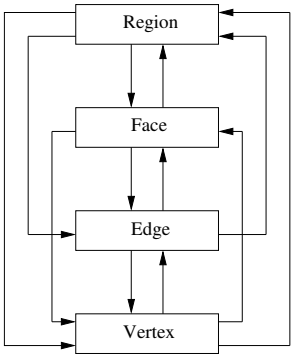


Figure 9: 12 adjacencies possible in the mesh representation [11]

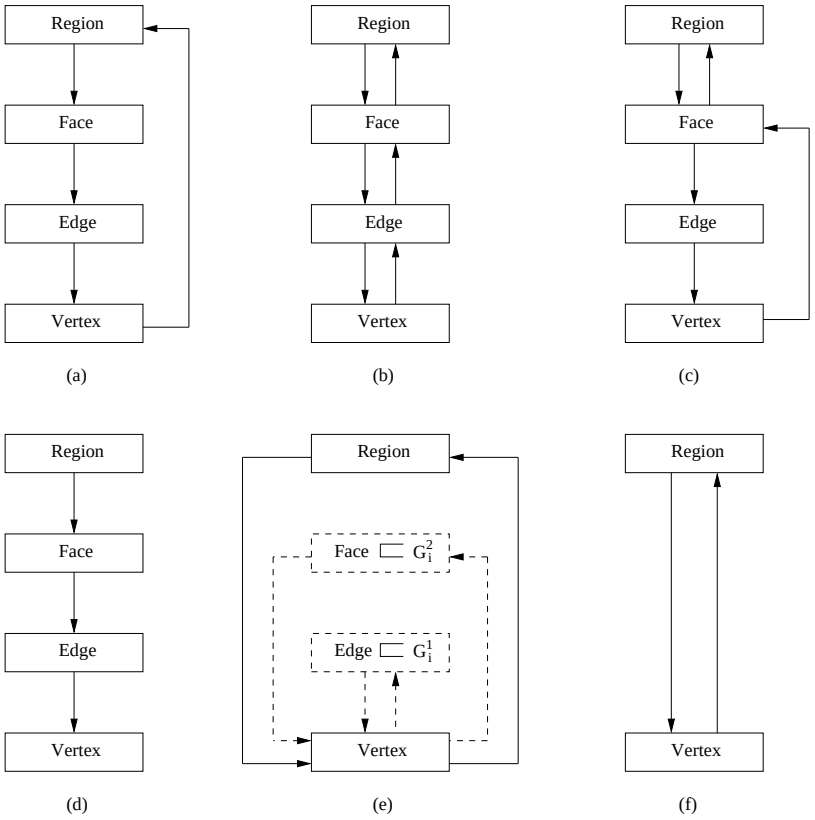


Figure 10: Example of 3D mesh representations

2.2.4 Mesh Representation

The mesh representation can be categorized with two criteria (i) full vs. reduced and (ii) complete vs. incomplete, resulting in 4 different groups [23].

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

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

In the mesh representation graph, a dotted box denotes that among entities of the level, only equally classified ones are explicitly stored, and a dotted arrow denotes that adjacencies from an outgoing level to an incoming level are maintained only for the stored entities.

In Figure 10 illustrates 6 mesh representations in 3D. Note that in the adjacency graph, a dotted

box denotes that among entities of the level, only equally classified ones are explicitly stored, and a dotted arrow denotes that adjacencies from an outgoing level to an incoming level are maintained only for the stored entities. (a) to (c) are full and complete due to all 0 to d levels of entities exist and the 12 adjacencies are obtainable in O(1) time either by direct access or local traversal, (d) is full and incomplete since it requires mesh level global search or traversal to get proper adjacencies, (e) is reduced and complete, and (f) is reduced and incomplete. Representation (b) is *the one-level adjacency representation* as it maintains adjacencies between entities one dimension apart. Representation (e) is *the complete minimum sufficient representation* that stores the minimum sufficient representation plus upward adjacencies from vertices to their bounding entities of level > 0. Representation (f) is the classic mesh connectivity structure that describes the mesh only in terms of elements and nodes, and also has been used for several finite element applications [4].

For more discussions on mesh representations, see Reference [23].

2.2.5 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 functionalities of entity set to effectively support the application needs [13, 17].

- 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 ithout higher order adjacency can be inserted.

Please be noted that entity set is not supported in the current PUMI release.

2.2.6 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.

2.2.7 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 [13, 17].

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

2.3 S/W Structure

In development, geometry-based analysis s/w is modularized based on the features and goals as component. Figure 11 illustrates the software structure of PUMI consisting of the following seven components.

- The *common utility* component provides common utilities and services used in multiple other components such as iterator, set, and tag.
- The *parallel control* component provides parallel-specific utilities and services such as communications and architecture-aware operations.
- The *geometric model* component provides a uniform interface for querying geometric model representations. It uses common utility and parallel control component.

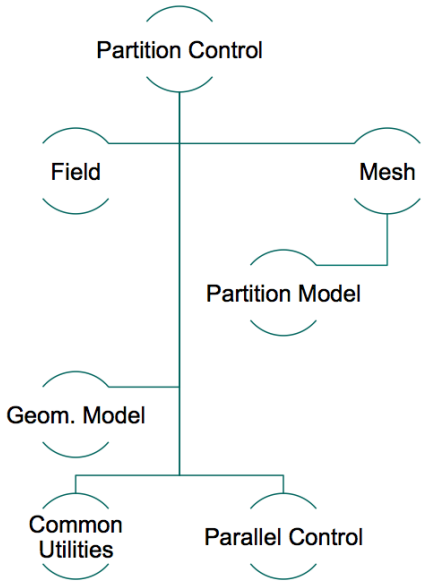


Figure 11: Software structure of PUMI

- The *partition model* component is constructed based on the mesh distribution and provides mesh partitioning representation in topology to the mesh component for the support for efficient update/manipulation of mesh with respect to partitioning.
- The *mesh* component provides the storage and management of distributed unstructured meshes. It uses all components except for the field component.
- The *field* component provides the services for storage and management of solution information on the mesh. It uses common utility, parallel control and mesh components.
- The *partition control* component provides the services for improving mesh partitioning via graph partitioner or existing mesh information such as adajacencies.

PUMI consists of multiple libraries which are modularized based on supported features.

- `pumi`: a library with API hearder file and PUMI error codes. PUMI error codes are defined in the file `pumi_errorcode.h` and the API's are defined in the file `pumi.h`.
- `pcu`: a library with parallel control and comminication. The header file is obtained in `PCU.h`.
- `gmi`: a library with SCOREC-formatted geometric model implementation.
- `mds`: a library with SCOREC-formatted mesh implementation.
- `apf`: a library with field, common mesh interface, common geometric model interface, interations between geometric model and mesh implmentation, etc. The header file is obtained in `apf.h`.
- `parma`: a library with adjacency-based mesh partitioning functions. The header file is obtained in `parma.h`.

- `zoltan`: a library to support interaction with Zoltan [5, 9] graph partitioning S/W.

3 Parallel Control and Communication

The PCU (Parallel Control Utility) is a library for parallel computation based on MPI with additional support for hybrid MPI/thread environments. PCU is included in PUMI to support needed parallel controls and communication for operations with distributed meshes and model.

The PCU provides three things to users:

1. A phased message passing API
2. A thread management API
3. Support for phased message passing between threads instead of processes

Phased message passing is similar to Bulk Synchronous Parallel, but is implemented more efficiently. All messages are exchanged in a phase, which is a collective operation involving all threads in the parallel program. During a phase, the following events happen in sequence:

1. All threads send non-blocking messages to other threads
2. All threads receive all messages sent to them during this phase

PCU provides termination detection, which is the ability to detect when all messages have been received without prior knowledge of which threads are sending to which.

To write hybrid MPI/thread programs, PCU provides a function that creates threads within an MPI process, similar to the way mpirun creates multiple processes. PCU assigns ranks to these threads and has them each run the same function, with thread-specific input arguments to the function.

Once a program has created threads using PCU, it can call the phased message passing API from within threads, which will behave as if each thread were an MPI process. Threads have unique ranks and can send messages to one another, regardless of which process they are in.

4 Geometric Model

PUMI geometric model interface supports the ability to interrogate solid models for topological adjacency and geometric shape information.

The geometric model representation used by PUMI is a boundary representation based on the Radial Edge Data Structure [31]. In this representation the model is a hierarchy of topological entities called regions, shells, faces, loops, edges and vertices. This representation is general and is capable of representing non-manifold models that are common in engineering analyses. The use of a boundary representation is convenient for the association of problem attributes (e.g., loads, material properties and boundary conditions) and mesh generation control information since the entities defining the model are explicitly represented.

The classes used to represent the geometric model support operations to find the various model entities that make up a model and to find which model entities are adjacent to a given entity. Other operations relating to performing geometric queries are also supported.

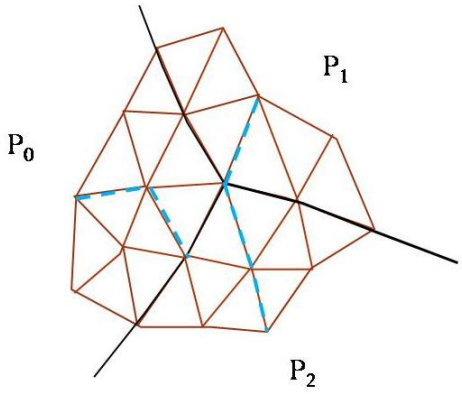


Figure 12: Distributed mesh on three processes P_0, P_1 and P_2 with two parts per each process

5 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. §5.3 describes a partition model that is developed in FMDB for the purpose of effectively meeting the specific functionalities of distributed meshes [23, 24]. The readers not interested in the internal design and implementation of the distributed meshes in FMDB might skip §5.3

5.1 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 12 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 13, 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.

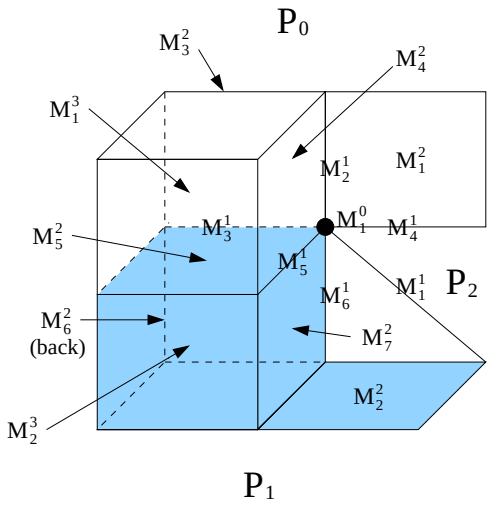


Figure 13: Example 3D mesh distributed on three parts

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 13 are M_1^1, M_1^2, M_2^2, M_1^3, and M_2^3.

5.2 Functional Requirements

5.2.1 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.

5.2.2 Ownership

In parallel adaptive analysis, the mesh and its partitioning can change thousands of time during the simulation [1, 8, 16, 27]. 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

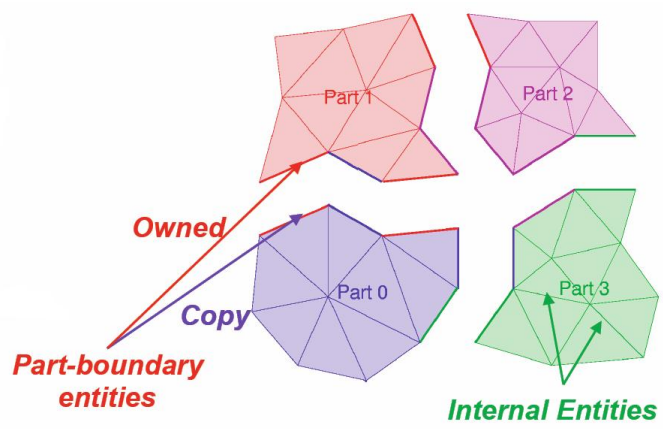


Figure 14: A distributed mesh on four processes with one part per process [13]

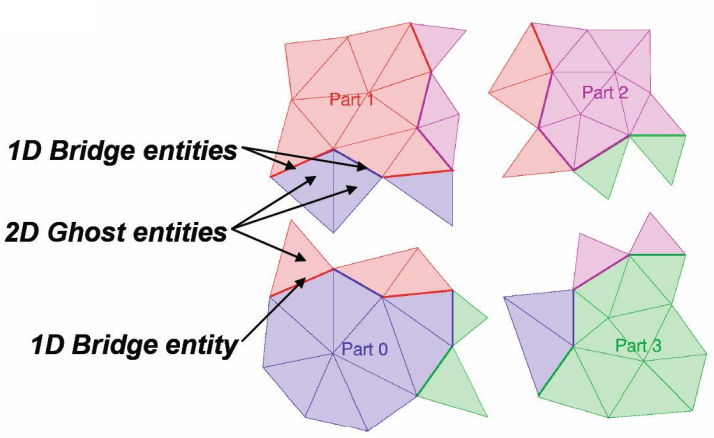


Figure 15: A distributed mesh on four parts with ghost entities [13]

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 14 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.

5.2.3 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* [13].

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

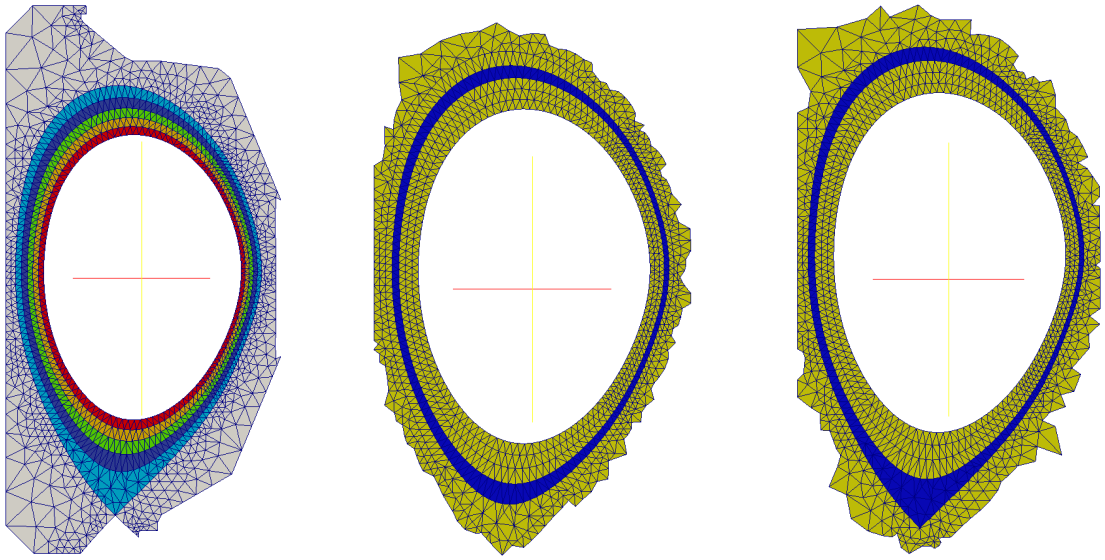


Figure 16: 3-layer ghosting: (left to right) initial 6-part mesh, part 3 with 3 ghost layers, and part 4 with 3 ghost layers

Figure 15 depicts a distributed mesh on four parts with ghost entities along the part boundaries. Similar to ownership of part boundary entities, the original owner entity is designated as *owner* of all ghost copies.

To perform a layer-based ghosting, four input parameters are needed:

- bridge (entity) dimension
- ghost (entity) dimension
- the number of layers
- a true/false flag indicating whether to include non-owned part boundary entities will be included for bridge entities. If true, all part boundary entities of bridge dimension are considered to construct ghost layer(s). If false, only *owned* part boundary entities of bridge dimension are considered.

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, bridge dimension, ghost dimention, and the number of layers shall be, respectively, 2, 3, 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 15, The input parameters of ghosting (bridge dimension, ghost dimension, the number of layers and a flag) are $[1, 2, 1, true]$.

The left figure in Figure 16 depicts 6-part distributed mesh. The parts 0 to 5 are colored in red, orange, green, blue, turquoise, and gray respectively. The middle and right figure in Figure 16 illustrate the part 3 and part 4 with 3 ghost layers, where the original mesh entities are colored in

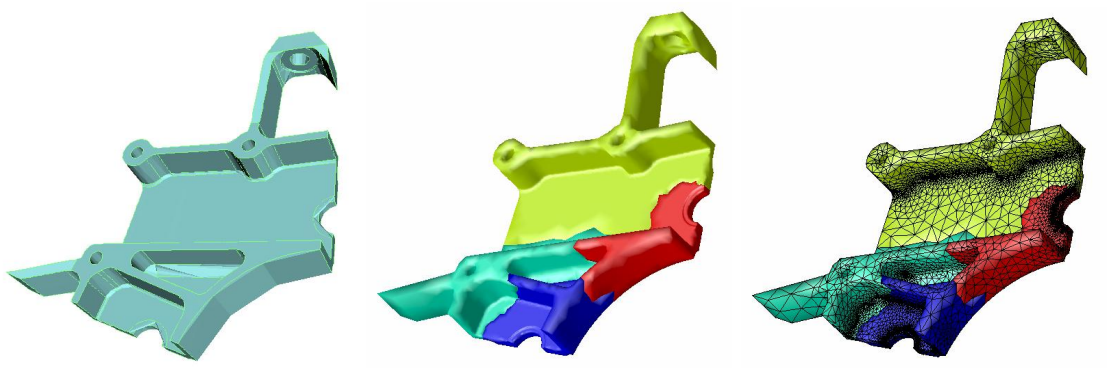


Figure 17: Hierarchy of domain decomposition: (left to right) geometry model, partition model, and distributed mesh on 4 processes

blue and the ghost copies are colored in yellow. The input parameters of ghosting procedure are $[0, 2, 3, true]$.

5.2.4 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.

5.3 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 17, 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 unstructured mesh representation, such that standard mesh 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 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 [13].

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 2.8 *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.

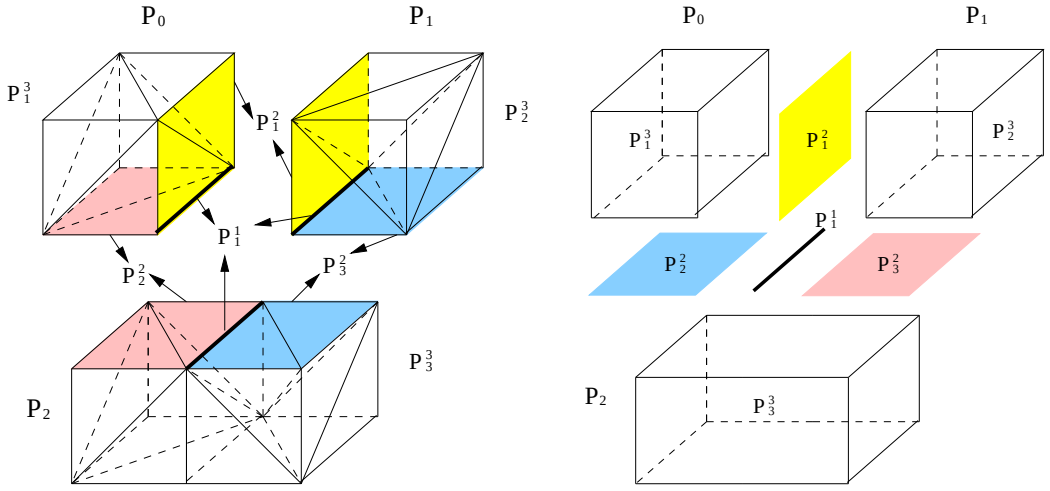
Definition 2.9 *Partition classification*

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 2.10 *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 18 illustrates a 3D distributed mesh with mesh entities labeled with arrows indicating the partition classification of the entities onto the partition model entities and its associated partition model. The mesh vertices and edges on the thick black lines are classified on partition edge P_1^1. The mesh vertices, edges and faces on the shaded planes are classified on the partition faces pointed with each arrow. The remaining mesh entities are non-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.



(a) distributed mesh (b) partition model

Figure 18: Distributed mesh and its association with the partition model via partition classifications

6 Interface Structure

The PUMI API's are provided in the file ‘‘`pumi.h`’’. Please be noted that in the current PUMI release, an entity set and multiple parts per process are not supported. Herein, with a single part per process, a mesh instance and a part handle are interchangeable.

6.1 API naming convention

PUMI API function name consists of two or three words connected with ‘_’.

- the first word is “pumi”
- the second word is an operation target. If the operation target is system-wide, the operation target is ommited. For instance, the function that initializes the PUMI service is *pumi_start*.
- the third word is the operation description starting with a verb. For example, the function *pumi_mesh_getNumEnt* returns the number of mesh entities with specific type. The function *pumi_gent_getID* returns the global ID of geometric model entity.

The following are operation targets used in the second word.

- *geom*: the api is performed on a geometric model
- *gent*: the api is performed on a geometric model entity
- *mesh*: the api is performed on a local mesh or global mesh
- *ghost*: the api performs ghosting functions
- *ment*: the api is performed on a mesh entity
- *tag*: the api is performed on a tag handle
- *field*: the api is performed on a field object
- *shape*: the api is performed on a field shape object
- *node*: the api is performed on a field node
- *numbering*: the api is performed on a numbering object

6.2 Abbreviation

Abbreviations may be used in API naming. See *http://scorec.rpi.edu/wiki/Abbriviations* for more information.

6.3 Data Types and Classes

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 mesh entities 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.

pGeom	geometric model instance
pGeomEnt	geometric entity handle
pMesh	mesh instance
pMeshEnt	mesh entity handle (i.e., vertex, edge, face, region)
pOwnership	an ownership handle to allow user-defined ownership rule for part boundary entities
Vector3	an array of three doubles to hold the coordinate information of node
Adjacent	an array of entities to hold the adjacency
EntityVector	std::vector of mesh entity handle (pMeshEnt)
Copies	std::map of part ID (int) and mesh entity handle (pMeshEnt) to hold remote or ghost copies
Parts	std::set of part ID (int)
pTag	tag handle for geometric model
pMeshTag	tag handle for mesh
pGeomIter	iterator traversing over global geometric model entities
pMeshIter	iterator traversing over local mesh entities
pCopyIter	iterator traversing `Copies`
pShape	shape function handle to define how the field nodes are distributed
pField	field handle
pNumbering	numbering handle to assign local numbering to multiple degree of freedoms in field nodes
pGlobalNumbering	global numbering handle to assign global numbering to multiple degree of freedoms in field nodes

The following classes are defined to support mesh re-distribution and ghosting.

Migration	a class to define migration plan
Distribution	a class to define distribution plan
Ghosting	a class to define ghosting plan

6.4 Enumeration Types

The enumeration type for tag data type is:

```
enum PUMI_TagType {
    PUMI_DBL       = 0 /* double */,
    PUMI_INT,      /* 1 - integer */
    PUMI_LONG,     /* 2 - long integer */
    PUMI_ENT,      /* 3 - entity handle. Only for geometric model*/
    PUMI_SET,      /* 4 - set handle. NOT SUPPORTED */
    PUMI_PTR,      /* 5 - opaque pointer. Only for geometric model*/
    PUMI_STR,      /* 6 - string. NOT SUPPORTED */
    PUMI_BYTE      /* 7 - 1 byte character. Only for geometric model */
}
```

For geometric model, the supported tag types are *PUMI_DBL*, *PUMI_INT*, *PUMI_LONG*, *PUMI_ENT*, *PUMI_PTR*, and *PUMI_BYTE*. For mesh, the supported tag types are *PUMI_DBL*, *PUMI_INT*, and *PUMI_LONG*.

The enumeration type for entity topology is:

```
enum PUMI_EntTopology {
  PUMI_VERTEX,  // 0
  PUMI_EDGE,    // 1
  PUMI_TRIANGLE,  // 2
  PUMI_QUAD,// 3
  PUMI_TET,   // 4
  PUMI_HEX,   // 5
  PUMI_PRISM,  // 6
  PUMI_PYRAMID  // 7
};
```

The enumeration type for field data type is:

```
enum PUMI_FieldType {
  PUMI_SCALAR,  //  a single scalar value
  PUMI_VECTOR,  // a 3D vector
  PUMI_MATRIX,  // a 3x3 matrix
  PUMI_PACKED  // a user-defined set of components
};
```

6.5 Error Codes

If the API function returns an error code and the function succeeds, it returns 0, otherwise, it returns a positive integer representing a type of error. The error codes are defined in *pumi_errorcode.h*.

```
 enum PUMI_ErrCode {
  PUMI_SUCCESS = 0, // no error
  PUMI_MESH_ALREADY_LOADED,
  PUMI_FILE_NOT_FOUND,
  PUMI_FILE_WRITE_ERROR,
  PUMI_NULL_ARRAY,
  PUMI_BAD_ARRAY_SIZE,
  PUMI_BAD_ARRAY_DIMENSION,
  PUMI_INVALID_ENTITY_HANDLE,
  PUMI_INVALID_ENTITY_COUNT,
  PUMI_INVALID_ENTITY_TYPE,
  PUMI_INVALID_ENTITY_TOPO,
  PUMI_BAD_TYPE_AND_TOPO,
  PUMI_ENTITY_CREATION_ERROR,
  PUMI_INVALID_TAG_HANDLE,
  PUMI_TAG_NOT_FOUND,
  PUMI_TAG_ALREADY_EXISTS,
  PUMI_TAG_IN_USE, // try to delete a tag that is in use
  PUMI_INVALID_SET_HANDLE,
  PUMI_INVALID_ITERATOR,
  PUMI_INVALID_ARGUMENT,
  PUMI_MEMORY_ALLOCATION_FAILED,
  PUMI_INVALID_MESH_INSTANCE,
  PUMI_INVALID_GEOM_MODEL,
  PUMI_INVALID_GEOM_CLAS,
  PUMI_INVALID_PTN_CLAS,
  PUMI_INVALID_REMOTE,
  PUMI_INVALID_MATCH,
  PUMI_INVALID_PART_HANDLE,
  PUMI_INVALID_PART_ID,
  PUMI_INVALID_SET_TYPE,
  PUMI_INVALID_TAG_TYPE,
  PUMI_ENTITY_NOT_FOUND,
  PUMI_ENTITY_ALREADY_EXISTS,
  PUMI_REMOTE_NOT_FOUND,
  PUMI_GHOST_NOT_FOUND,
  PUMI_CB_ERROR,
  PUMI_NOT_SUPPORTED,
  PUMI_FAILURE,
}
```

7 Comminication API's

"PCU.h" provides the API functions for parallel communications including MPI.

8 System-wide API's

This section describes API functions and enumeration types which are not bounded with a specific geometric model or mesh data.

```
void pumi_start()
```

Initialize parallel services pertinent to PUMI.

```
void pumi_finalize(bool /* in */ do_mpi_finalize=false)
```

Finalize parallel services and clean the memory. If the input parameter *do_mpi_finalize* is true, MPI finalization is performed as well. *do_mpi_finalize* is optional (default: *false*).

```
int pumi_size()
```

Return the number of processes.

```
int pumi_rank()
```

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

```
void pumi_sync()
```

Synchronize all processes in communicator

```
void pumi_printSys()
```

Print system information such as host name, processor, operating system, etc.
(Example) Linux node10.borg.scorec.rpi.edu 2.6.9-89.ELsmp SMP Mon Jun 22 12:31:33 EDT 2009 x86_64.

```
double pumi_getMem()
```

Return the heap memory increase (MB) on local process since *pumi_start()*.

```
double pumi_getTime()
```

Return the current time in second.

```
void pumi_printTimeMem(
        const char* /* in */ msg,
        double /* in */ time,
        double /* in */ memory)
```

Display "*msg*: *time* (sec) and *memory* (MB)".

9 Geometric Model API's

This chapter describes geometric model related API functions.

9.1 Model managemant and interrogation

```
pGeom pumi_geom_load(
        const char* /* in */ model_file_name,
        const char* /* in */ model_type="mesh",
        void (*geom_load_fp)(const char*))
```

Given geometric model file name and geometric model type (*"mesh", "null", "analytic"*), create a model, load the model data from the file, and return a geometric model instance. If *model_type* is not provided, the default is *"mesh"*.

In case where a geometric model is *NOT* available, `pumi_geom_load(NULL, ``null")` will generate a "null" geometric model which mimicks the minimal geometric model behavior enough to support the PUMI.

In case of the analytic model, the user can provide a function pointer in the third argument, which creates analytic model entities. If the third argument is not provided for the analytic model, a set of analytic model entities can be created later via functions *gmi_add_analytic*, *gmi_add_analytic_region*, *gmi_add_analytic_cell*, and *gmi_add_analytic_reparam*. For the details of analytic model entity creation, see *top_source_dir/gmi/gmi_analytic.h*.

```
void pumi_geom_delete (pGeom /* in */ g)
```

Given a geometric model instance, delete the model instance and deallocate the memory.

```
void pumi_geom_freeze(pGeom /* in */ g)
```

In some special cases, the model entities can be created by users or derived from the mesh. In such cases, when the model construction is completed, the function *pumi_geom_freeze* has to be called to update the internal data of the geometric model accordingly. This function finalizes the geometric model so no more model entity can be created afterwards.

```
int pumi_geom_getNumEnt(
        pGeom /* in */ g,
        int /* in */ d)
```

Given a geometric model instance and dimension d $(0-3)$, return the number of geometric model entities of the dimension d.

```
pGeomEnt pumi_geom_findEnt(
pGeom /* in */ g,
        int /* in*/ d,
        int /* in */ id)
```

Given a geometric model instance, dimension d $(0-3)$, and a local ID, find the corresponding geometric model entity and return its handle. Otherwise, return *NULL*.

```
void pumi_geom_print (
    pGeom  /* in */  g,
    bool  /* in */  print_entities)
```

Given a mdoel instance and a boolean flag pecifing whether to print the details of model entities, display the model information. The information includes size, tag, and individual entities with global ID. The argument *print_entities* is optional (default: false).

9.2 Geometric model interation

```
pGeom g;
for (pGeomIter it = g->begin(d); it!=g->end(d);++it)
{
  pGeomEnt e = *it;
  ...
}
```

Iterate geometric model entities of dimension d.

9.3 Geometric entity interrogation

```
int pumi_gent_getDim (pGeomEnt  /* in */  geom_ent)
```

Given a geometric model entity, return its dimension $(0-3)$.

```
int pumi_gent_getID (pGeomEnt  /* in */  geom_ent)
```

Given a geometric model entity, return its global ID. Note that, in parallel, each process loads the entire geometric model.

```
void pumi_gent_getRevClas (
        pGeomEnt  /* in */  geom_ent,
        std::vector<pMeshEnt>&  /* inout */  mesh_ents)
```

Given a geometric model entity, get the vector *mesh_ents* filled with its reverse classification (equal-order mesh entities classified on the geometric model entity).

```
int pumi_gent_getNumAdj (
        pGeomEnt  /* in */  geom_ent,
        int /* in */ target_dim)
```

Given a geometric model entity and desired adjacency type *target_dim*, get the number of adjacent entities of the type. Note the following: (i) if the geometric model is constructed by users, the adjacencies shall be constructed as well using `apf::add_adj` (See `gmi.h`), (ii) if the geometric model is a *null* model or driven by the mesh, the adjacencies may not be fully constructed.

```
void pumi_gent_getAdj (
        pGeomEnt  /* in */  geom_ent,
        int /* in */ target_dim,
        std::vector<pGeomEnt>& /* inout*/ adj_ents)
```

Given a geometric model entity and desired adjacency type *target_dim*, get the vector *adj_ents* filled with the adjacent entities of the type. Note the following: (i) if the geometric model is constructed by users, the adjacencies shall be constructed as well using `apf::add_adj` (See `gmi.h`), (ii) if the geometric model is a *null* model or driven by the mesh, the adjacencies may not be fully constructed.

```
void pumi_gent_get2ndAdj (
        pGeomEnt  /* in */  geom_ent,
        int /* in */ brg_dim,
        int /* in */ target_dim,
        std::vector<pGeomEnt>& /* inout*/ adj_ents)
```

Given a geometric model entity handle, bridge type *brg_dim*, and desired adjacency type *target_dim*, get the vector container *adj_ents* filled with 2^{nd} order adjacent entities of type *target_dim* obtained through the bridge type *brg_dim*. *brg_dim* and *target_dim* should not be equal.

9.4 Geometric model tag management

```
pTag pumi_geom_createTag (
        pGeom /* in */ g,
        const char* /* in */ tag_name,
        int /* in */ tag_type,
        int /* in */ tag_size)
```

Given a geometric model instance, tag name, tag data type, and tag data size, create a tag handle in model instance and return the tag handle.

```
void pumi_geom_deleteTag (
        pGeom /* in */ g,
        pTag /* in */ tag,
        bool force_delete=false)
```

Given a geometric model instance and a tag handle, destroy the tag from the model instance. If *force_delete* is *true*, it checks if any tag data associated with the tag exists and delete any existing tag data then deletes the tag handle. If *force_delete* is *false*, the tag is deleted without checking tag data associated with the tag. *force_delete* is optional (default: $false$). if *force_delete* is false and the tag handle is still in use, the function crashes.

Note: Since PUMI doesn't keep track of tag data attachment, forced tag deletion is $O(N)$.

```
pTag pumi_geom_findTag (
        pGeom /* in */ g,
        const char* /* in */ tag_name)
```

Given a geometric model instance and character string, return the handle of an existing tag in the model. If there's no tag with the given name, *NULL* is returned.

```
bool pumi_geom_hasTag (
        pGeom /* in */ g,
        pTag /* in */ tag)
```

Given a geometric model instance and a tag handle, return 1 if the tag exists in the model. Otherwise, 0.

```
void pumi_geom_getTag (
        pGeom /* in */ g,
        std::vector<pTag>& /* inout */ tags)
```

Given a geometric model instance, get the vector *tags* filled with tag handles created in the model.

```
int pumi_tag_getType (const pTag /* in */ tag)
```

Given a tag handle, return tag type.

```
void pumi_tag_getName (
        const pTag /* in */ tag,
        const char** /* out */ name)
```

Given a tag handle, get the tag name.

```
int pumi_tag_getSize (const pTag /* in */ tag)
```

Given a tag handle, get the tag data size.

```
void pumi_tag_getByte (const pTag /* in */ tag)
```

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

9.5 Geometric entity tagging

```
void pumi_gent_deleteTag (
        pGeomEnt /* in */ geom_ent,
        pTag /* in */ tag)
```

Given geometric entity handle and tag handle, delete the tag data from the entity.

```
bool pumi_gent_hasTag (
        pGeomEnt /* in */ geom_ent,
        pTag /* in */ tag)
```

Given geometric entity handle and tag handle, return *true* if the tag data is attached to the entity. Otherwise, return $false$.

```
void pumi_gent_getTag (
        pGeomEnt /* in */ geom_ent,
        std::vector<pTag>& /* inout */  tags)
```

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

```
void pumi_gent_setPtrTag (
        pGeomEnt /* in */ geom_ent,
        pTag /* in */ tag,
        const void*  /* in */  data)
```

Given geometric entity handle, tag handle, and opaque data (void*), set or update the tag value of the entity. It fails if (i) tag type is not *PUMI_PTR*, (ii) tag data size is greater than 1. In some cases, users do not really care about the geometric model at all. For such cases, there is a "null" geometric model that can be used, which just mimicks the minimal necessary behavior to support the rest of the code. You can get a null model as follows:

```
void pumi_gent_getPtrTag (
        pGeomEnt /* in */ geom_ent,
        pTag /* in */ tag,
        void**  /* out */  data)
```

Given geometric entity handle and tag handle, get pointer type data tagged to the entity. It fails if (i) the tag doesn't exist with the entity, (ii) tag type is not *PUMI_PTR*, (iii) tag data size is greater than 1.

```
void pumi_gent_setIntTag (
        pGeomEnt /* in */ geom_ent,
        pTag /* in */ tag,
        const int /* in */ data)
```

Given geometric entity handle, tag handle, and integer data, set or update the tag value of the entity. It fails if (i) tag type is not *PUMI_INT*, (ii) tag data size is greater than 1.

```
void pumi_gent_getIntTag (
        pGeomEnt /* in */ geom_ent,
        pTag /* in */ tag,
        int*  /* out */  data)
```

Given geometric entity handle and tag handle, get integer type data tagged to the entity. It fails if (i) the tag doesn't exist with the entity, (ii) tag type is not *PUMI_INT*, (iii) tag data size is greater than 1.

```
void pumi_gent_setLongTag (
        pGeomEnt /* in */ geom_ent,
        pTag /* in */ tag,
        const long /* in */ data)
```

Given geometric entity handle, tag handle, and long data, set or update the tag value of the entity. It fails if (i) tag type is not *PUMI_LONG*, (ii) tag data size is greater than 1.

```
void pumi_gent_getLongTag (
        pGeomEnt /* in */ geom_ent,
        pTag /* in */ tag,
        long*  /* out */  data)
```

Given geometric entity handle and tag handle, get long type data tagged to the entity. It fails if (i) tag type is not *PUMI_LONG*, (ii) tag data size is greater than 1.(i) the tag doesn't exist with the entity, (ii) tag type is not *PUMI_LONG*, (iii) tag data size is greater than 1.

```
void pumi_gent_setDblTag (
        pGeomEnt /* in */ geom_ent,
        pTag /* in */ tag,
        const double  /* in */  data)
```

Given geometric entity handle, tag handle, and double data, set or update the tag value of the entity. It fails if (i) tag type is not *PUMI_DBL*, (ii) tag data size is greater than 1.

```
void pumi_gent_getDblTag (
        pGeomEnt /* in */ geom_ent,
        pTag /* in */ tag,
        double*  /* out */  data)
```

Given geometric entity handle and tag handle, get double type data tagged to the entity. It fails if (i) the tag doesn't exist with the entity, (ii) tag type is not *PUMI_DBL*, (iii) tag data size is greater than 1.

```
void pumi_gent_setEntTag (
        pGeomEnt /* in */ geom_ent,
        pTag /* in */ tag,
        const pGeomEnt /* in */  data)
```

Given geometric entity handle, tag handle, and another geometric model entity, set or update the tag value of the entity. It fails if (i) tag type is not *PUMI_ENT*, (ii) tag data size is greater than 1.

```
void pumi_gent_getEntTag (
        pGeomEnt /* in */ geom_ent,
        pTag /* in */ tag,
        pGeomEnt*  /* out */  data)
```

Given geometric entity handle, tag handle, get the entity tagged to the geometric model entity. It fails if (i) the tag doesn't exist with the entity, (ii) tag type is not *PUMI_ENT*, (iii) tag data size is greater than 1.

```
void pumi_gent_setPtrArrTag (
        pGeomEnt /* in */ geom_ent,
        pTag /* in */ tag,
        void* const* /* in */ data)
```

Given geometric entity handle, tag handle, and opaque array data (void*), set or update the tag value of the entity. It fails if (i) tag type is not *PUMI_PTR*, (ii) the size of tag data doesn't match that of the data.

```
void pumi_gent_getPtrArrTag (
        pGeomEnt /* in */ geom_ent,
        pTag /* in */ tag,
        void**  /* out */  data)
```

Given geometric entity handle and tag handle, get pointer array data tagged to the entity. It fails if (i) the tag doesn't exist with the entity, (ii) tag type is not *PUMI_PTR*, (iii) the size of tag data doesn't match that of the data.

```
void pumi_gent_setIntArrTag (
        pGeomEnt /* in */ geom_ent,
        pTag /* in */ tag,
        const int*  /* in */  data)
```

Given geometric entity handle, tag handle, and integer array data, set or update the tag value of the entity. It fails if (i) tag type is not *PUMI_INT*, (ii) the size of tag data doesn't match that of the data.

```
void pumi_gent_getIntArrTag (
        pGeomEnt /* in */ geom_ent,
        pTag /* in */ tag,
        int**  /* out */  data,
        int*  /* out */  data_size)
```

Given geometric entity handle and tag handle, get integer array data tagged to the entity. *data_size* is the size of *data*. It fails if (i) the tag doesn't exist with the entity, (ii) tag type is not *PUMI_INT*, (iii) the size of tag data doesn't match that of the data.

```
void pumi_gent_setLongArrTag (
        pGeomEnt /* in */ geom_ent,
        pTag /* in */ tag,
        const int*  /* in */  data)
```

Given geometric entity handle, tag handle, and long array data, set or update the tag value of the entity. It fails if (i) tag type is not *PUMI_LONG*, (ii) the size of tag data doesn't match that of the data.
(NOT SUPPORTED)

```
void pumi_gent_getLongArrTag (
        pGeomEnt /* in */ geom_ent,
        pTag /* in */ tag,
        int**  /* out */  data,
        int*  /* out */  data_size)
```

Given geometric entity handle and tag handle, get long array data tagged to the entity. *data_size* is the size of *data*. It fails if (i) the tag doesn't exist with the entity, (ii) tag type is not *PUMI_LONG*, (iii) the size of tag data doesn't match that of the data.
(NOT SUPPORTED)

```
void pumi_gent_setDblArrTag (
        pGeomEnt /* in */ geom_ent,
        pTag /* in */ tag,
        const double*  /* in */  data)
```

Given geometric entity handle, tag handle, and double array data, set or update the tag value of the entity. It fails if (i) tag type is not *PUMI_DBL*, (ii) the size of tag data doesn't match that of the data.

```
void pumi_gent_getDblArrTag (
        pGeomEnt /* in */ geom_ent,
        pTag /* in */ tag,
        double**  /* out */  data,
        int*  /* out */  data_size)
```

Given geometric entity handle and tag handle, get double array data tagged to the entity. *data_size* is the size of *data*. It fails if (i) the tag doesn't exist with the entity, (ii) tag type is not *PUMI_DBL*, (iii) the size of tag data doesn't match that of the data.

```
void pumi_gent_setEntArrTag (
        pGeomEnt /* in */ geom_ent,
        pTag /* in */ tag,
        const pGeomEnt*  /* in */  data,
        int  /* in */  data_size)
```

Given geometric entity handle, tag handle, and entity array data, set or update the tag value of the entity. It fails if (i) tag type is not *PUMI_ENT*, (ii) the size of tag data doesn't match that of the data.

```
void pumi_gent_getEntArrTag (
```

```
pGeomEnt /* in */ geom_ent,
pTag /* in */ tag,
pGeomEnt**  /* out */  data,
int*  /* out */  data_size)
```

Given geometric entity handle and tag handle, get entity array data tagged to the entity. *data_size* is the size of *data*. It fails if (i) the tag doesn't exist with the entity, (ii) tag type is not *PUMI_ENT*, (iii) the size of tag data doesn't match that of the data.

10 Mesh API's

This chapter describes mesh API functions.

10.1 Mesh Functions

10.1.1 Mesh management

```
pMesh pumi_mesh_create(
        pGeom /* in */ g,
        int /* in */ mesh_dim)
```

Given a geometric model instance and mesh dimension, create an empty mesh instance. Call this function only if mesh entities will be created afterwards. Mesh vertex can be created by *pumi_mesh_createVtx*. Non-vertex mesh entity can be created by *pumi_mesh_createEnt*. Mesh element (faces in 2D or regions in 3D) can be created by either by *pumi_mesh_createEnt* or *pumi_mesh_createElem*. Note the followings: (i) in parallel, part boundaries construction should be set by users (ii) after finishing mesh entity creation, the function *pumi_mesh_freeze* must be called to update internal mesh data accordingly.

```
void pumi_mesh_freeze(pMesh /* in */ m)
```

Given a mesh instance, finalize the mesh entity creation. This function updates the internal geometric model data and partition model accordingly. No more mesh entity creation is allowed afterwards.

```
pMeshEnt pumi_mesh_createVtx(
        pMesh /* in */ m,
        pGeomEnt /* in */ ge,
        double* /* in */ xyz)
```

Given a mesh instance, geometric model entity handle and *xyz* coordinates, create a mesh vertex and return its handle.

```
pMeshEnt pumi_mesh_createEnt(
        pMesh /* in */ m,
        pGeomEnt /* in */ ge,
        int /* in */ target_topology,
        pMeshEnt* /* in */ downward)
```

Given a mesh instance, geometric model entity handle, target entity topology, and an array of one-level downward entities, create a non-vertex mesh entity and return its handle. The supported entity topologies are *PUMI_EDGE*, *PUMI_TRIANGLE*, *PUMI_QUAD*, *PUMI_TET*, *PUMI_HEX*, *PUMI_PRISM*, and *PUMI_PYRAMID*. For instance, to create an edge, *target_topology* is `PUMI_EDGE` and *downward* is an array with two vertices. To create a prism, *target_topology* is `PUMI_PRISM` and *downward* is an array of five faces. In terms of the order of downward entities, see Figures 6 and 8.

```
pMeshEnt pumi_mesh_createElem(
        pMesh /* in */ m,
        pGeomEnt /* in */ ge,
        int /* in */ target_topology,
        pMeshEnt* /* in */ vertices)
```

Given a mesh instance, geometric model entity handle, target entity topology, and an array of vertices, create a mesh element (regions in 3D and faces in 2D) and return its handle. The supported en tity topologies are *PUMI_TRIANGLE* and *PUMI_QUAD* for 2D mesh, *PUMI_TET*, *PUMI_HEX*, *PUMI_PRISM*, and *PUMI_PYRAMID* for 3D mesh. For instance, To create a prism, *target_topology* is `PUMI_PRISM` and *vertices* is an array of six vertices. In terms of the order of downward vertices, see Figures 6 and 8.

```
pMesh pumi_mesh_loadSerial(
        pGeom /* in */ g,
        const char* /* in */ filename,
        const char* /* in */ type="mds")
```

Given a model instance, mesh file name, and mesh type, create a mesh instance, load the mesh data onto the master process, then return the mesh instance. If this function is called in p processes (p>1), only the master process (process 0) loads a mesh from the mesh file and the rest of processes have an empty mesh.

As PUMI is designed to support multiple mesh types, the third argument is used to specify the mesh type in file. If *type* is not specified, the default value is *"mds"* where the file with exension ".smb" is loaded. *"mds"* mesh file contains number i before the file extension (*".smb"*), where i represents the process rank.

For instance, for a four-part distributed mesh, the mesh files are *"filename0.smb"*, *"filename1.smb"*, *"filename2.smb"* and *"filename3.smb"*. For a serial mesh, the mesh file is *"filename0.smb"*.

Note to drop the process rank i in the second argument. For instance, to load a serial mesh in *"filename0.smb"*, the second argument should be *"filename.smb"*.

```
pMesh pumi_mesh_load(
        pGeom /* in */ g,
        const char* /* in */ filename,
        int /* in */ n,
        const char* /* in */ type="mds")
```

Given a model instance, file name, the number of input mesh files and mesh type, create a mesh instance, load the mesh data from the file, then return the mesh instance. The number of files n is 1 and the number of process p is greater then 1, first, the serial mesh is loaded onto the master process then partitioned to p parts.

As PUMI is designed to support multiple mesh types, the third argument is used to specify the mesh type. If *type* is not specified, the default value is *"mds"* where the file with exension ".smb" is loaded. *"mds"* mesh file contains number i before the file extension (*".smb"*), where i represents the process rank. For instance, for a four-part distributed mesh, the mesh files are *"filename0.smb"*,

"filename1.smb", *"filename2.smb"* and *"filename3.smb"*. For a serial mesh, the mesh file is *"filename0.smb"*.

Note to drop the process rank i in the second argument. For instance, to load a serial mesh in *"filename0.smb"* onto p processes (p>1), the second and the third arguments are *"filename.smb"* and 1. In order to load a three-part mesh in *"filename0.smb"*, "filename1.smb", "filename2.smb", the second and the third arguments are *"filename.smb"* and 3.

```
void pumi_mesh_delete (pMesh /* in */ m)
```

Given a mesh instance, delete the mesh instance and deallocate the memory.

```
bool pumi_mesh_hasAdjacency (
        pMesh /* in */ m,
        int /* in */ from_dim,
        int /* in */ to_dim)
```

Given a mesh instance and dimensions *from_dim* and *to_dim*, return *true* if the adjacency from *from_dim* to *to_dim* is explicitly stored and maintained. The default mesh representation is the one-level representation (See Figure 10(b)).

```
void pumi_mesh_createAdjacency(
        pMesh /* in */ m,
        int /* in */ from_dim,
        int /* in */ to_dim)
```

Given a mesh instance and dimensions *from_dim* and *to_dim*, if the adjacency from *from_dim* to *to_dim* is not explicitly stored and maintained, create the explicit adjacency from *from_dim* to *to_dim*. The performance of adjacency queries from *from_dim* to *to_dim* will improve at the cost of storing the adjacency. Please be noted that the development with the user-defined mesh representation is on-going, therefore currently not all PUMI API's are supported with user-defined mesh representation.

```
void pumi_mesh_deleteAdjacency (
        pMesh /* in */ m,
        int /* in */ from_dim,
        int /* in */ to_dim)
```

Given a mesh instance and dimensions *from_dim* and *to_dim*, if the adjacency from *from_dim* to *to_dim* is explicitly stored and maintained, delete the adjacency from *from_dim* to *to_dim*. If *from_dim* is one-level apart from *to_dim*, the function doesn't perform. Please be noted that the development with the user-defined mesh representation is on-going, therefore currently not all PUMI API's are supported with user-defined mesh representation.

```
void pumi_mesh_createFullAdjacency (pMesh /* in */ m)
```

Given a mesh instance, create the full adjacencies. For instance, in 3D mesh, 12 adjacencies are explicitly stored (See Figure 9). Please be noted that the development with the full adjacencies is on-going, therefore currently not all PUMI API's are supported with the full adjacencies.

```
void pumi_mesh_write (
        pMesh /* in */ m,
        const char* /* in */ filename,
        const char* /* in */ type="mds")
```

Given a model instance, write a mesh into mesh file(s). The third argument is used to specify the mesh file type. The supported *type* is *"mds"* and *"vtk"*. If *type* is not specified, the default value is *"mds"*. If the second and third argument are *"filename.smb"* and *"mds"*, each process i writes its mesh data in the file "filename$i$.smb". If the second and third argument are *"output"* and *"vtk"*, all vtk files are created in the directory *output*. See Appendix A for how to visualize the *vtk* mesh files in *Paraview*.

10.1.2 Mesh interrogation

```
pGeom pumi_mesh_getGeom(pMesh /* in */ m)
```

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

```
int pumi_mesh_getDim(pMesh /* in */ m)
```

Given a mesh instance, return the dimension.

```
void pumi_mesh_setCount(
        pMesh /* in */ m,
        pOwnership  /* im */  o=NULL)
```

Given a mesh instance and an ownership rule handle, compute the global/owned entity counts for dimension 0−3 on local process. The ownership rule is an optional argument (default: *NULL*). If *NULL*, the default ownership rule provided by PUMI is used, where an owning part of part boundary entities is a part with the minimum number of elements among residence parts.

This function must be called before `pumi_mesh_getNumGlobalEnt` and `pumi_mesh_getNumOwnEnt`,

```
int pumi_mesh_getNumEnt(
        pMesh /* in */ m,
        int /* in */ d)
```

Given a mesh instance and dimension d (0−3), return the local entity count of the dimension d on local process. When a mesh entity is duplicated (part boundary or ghost) on N processes, each duplicate copy is counted.

```
int pumi_mesh_getNumGlobalEnt(
        pMesh /* in */ m,
        int /* in */ d)
```

Given a mesh instance and dimension d (0−3), return the global entity count of the dimension d on all processes. When a mesh entity is duplicated (part boundary or ghost) on N processes, only the owner copy is counted.

Prerequisite: `pumi_mesh_setCount`

```
int pumi_mesh_getNumOwnEnt(
        pMesh /* in */ m,
        int /* in */ d)
```

Given a mesh instance and dimension d (0−3), return the owned entity count of the dimension d on local process.

Prerequisite: `pumi_mesh_setCount`

```
pMeshEnt pumi_mesh_findEnt(
pMesh /* in */ m,
        int /* in*/ d,
        int /* in */ id)
```

Given a mesh instance, dimension d (0−3), and a local ID, find the corresponding mesh entity and return its handle. If a mesh entity of the ID doesn't exist, return *NULL*. For local ID of mesh entity, see *pumi_ment_getID*.

10.1.3 Mesh iteration

```
pMeshEnt e;
pMeshIter it = m->begin(d);
while ((e = m->iterate(it)))
{ ... }
m->end(it);
```

Iterate mesh entities by dimension d.

10.1.4 Tag management

```
pMeshTag pumi_mesh_createIntTag (
        pMesh /* in */ m,
        const char* /* in */ name,
        int /* in */ size)
```

Given a mesh instance, tag name, and tag data size (¿0), create an integer tag and return its handle. If *size* is 1, the associated tag data is single integer. If *size* is greater than 1, the associated tag data is an integer array.

```
pMeshTag pumi_mesh_createLongTag (
        pMesh /* in */ m,
        const char* /* in */ name,
        int /* in */ size)
```

Given a mesh instance, tag name, and tag data size (¿0), create a long tag and return its handle. If *size* is 1, the associated tag data is single long. If *size* is greater than 1, the associated tag data is a long array.

```
pMeshTag pumi_mesh_createDblTag (
        pMesh /* in */ m,
        const char* /* in */ name,
        int /* in */ size)
```

Given a mesh instance, tag name, and tag data size (¿0), create a double tag and return its handle. If *size* is 1, the associated tag data is single double. If *size* is greater than 1, the associated tag data is a double array.

```
void pumi_mesh_deleteTag(
        pMesh /* in */ m,
        pMeshTag /* in */ tag,
        bool /* in */ force_delete=false)
```

Given a mesh instance and a tag handle, destroy the tag from the mesh instance. If *force_delete* is *true*, delete any existing tag data associated with the tag handle before deleting tag handle. If *force_delete* is *false*, the tag handle is deleted without checking tag data associated with the tag. *force_delete* is optional (default: $false$).

Note: Since PUMI doesn't keep track of tag data attachment, forced tag deletion is $O(N)$.

```
pMeshTag pumi_mesh_findTag (
       pMesh /* in */ m,
       const char* /* in */ name)
```

Given a mesh instance and tag name, return a tag handle with the given name. If no tag handle is found, return *NULL*.

```
bool pumi_mesh_hasTag (
        pMesh /* in */ m,
        const pMeshTag /* in */ tag)
```

Given a mesh instance and a tag handle, return *true* if the tag handle exists in the mesh. Otherwise, return *false*.

```
void pumi_mesh_getTag (
        pMesh /* in */ m,
        std::vector<pMeshTag> /* inout */ tags)
```

Given a mesh instance, get the vector *tags* filled with tag handles created.

10.1.5 Mesh migration

```
class Migration
{
  public:
    Migration(pMesh m);
    ~Migration();
    // assign a destination part ID to an element
    void send(pMeshEnt e, int to);
    // return the destination part ID of an element
    int sending(pMeshEnt e);
    ...
};
```

Mesh migration is a mesh-level procedure to send a local partition object (or element) to a single destination part. Duplicated off-part entities along the part boundaries are accessible through remote copies. The class `Migration` provides the mechanism to register a local element for migration. Using the member function `Migration::send`, an element can be registed to migrate to *at most* one remote part. As the `Migration` object keeps the list of elements to be migrated, it is termed *migration plan*.

```
void pumi_mesh_migrate (
        pMesh  /* in */  m,
        Migration*  /* in */  plan)
```

Given a mesh instance and a migration object *plan*, migrate the elements as registed in *plan*. Tagged data, field, global numbering and global entity ID are maintained during the migration.

10.1.6 Mesh distribution

```
class Distribution
{
  public:
    Distribution(pMesh m);
    ~Distribution();
    // assign a destination part ID to an element
    void send(pMeshEnt e, int to);
    // return destination part ID's of an element
    std::set<int>& sending(pMeshEnt e);
    ...
};
```

Mesh distribution is a mesh-level procedure to send a local partition object (or element) to *multiple* destination parts. Duplicated off-part elements are accessible through remote copies. The class

Distribution provides the mechanism to register a local element for distribution. Using the member function `Distribution::send`, an element can be registed to distribute to other remote part. As the `Migration` object keeps the list of elements to be distributed, it is termed *distribution plan*.

```
void pumi_mesh_distribute (
        pMesh  /* in */  m,
        Distribution*  /* in */  plan)
```

Given a mesh instance and a distribution object *plan*, distribute the elements as registed in *plan*. Tagged data, field, global numbering and global entity ID are maintained during the distribution.

10.1.7 Ghosting

```
class Ghosting
{
  public:
    // create a ghosting object with ghost dimension (1-3)
    Ghosting(pMesh, int d);
    ~Ghosting();
    // assign a destination part ID to an entity of ghost dimension
    void send(pMeshEnt e, int to);
    // assign a destination part ID to all entities of ghost dimension
    void send(int to);
    // return destination part ID's of an entity
    std::set<int>& sending(pMeshEnt e);
    ...
};
```

Mesh ghosting is a mesh-level procedure to create local copy of off-part (*internal* or *part boundary*) entities. The local copy of off-part entities are termed "ghost copy". The ghost copy maintains the link to its original entity copy. If the ghost copy is originated from a part boundary entity, it maintains the link only to the owning copy of part boundary entity. However all duplicate copies of part part entity (owned or not) maintain the link to the ghost copy. The class `Ghosting` provides the mechanism to register a local entity for ghosting. Using the member function `Ghosting::send`, an entity can be registed to be ghosted on *at least* one destination part ID. As the `Ghosting` object keeps the list of entities to be ghosted, it is termed *ghosting plan*. Ghosting procedure is *accumulative*; it can be performed multiple times with different options resulting in adding more ghost copies or ghost layers incrementally.

```
void pumi_ghost_create (
        pMesh  /* in */  m,
        Ghosting*  /* in */  plan)
```

Given a mesh instance and a ghosting object *plan*, create ghost copies as registed in *plan*.

```
void pumi_ghost_createLayer (
        pMesh /* in */ m,
        int /* in */ brg_dim,
        int /* in */ ghost_dim,
        int /* in */ num_layer,
        int /* in */ include_copies)
```

Given a mesh instance, desired bridge entity type (0-2) on part boundary, desired ghost entity type (1-3), the number of ghost layers, and an integer flag indicating whether to include non-owned bridge entity (1: yes, 0: no) in ghosting plan computation, create ghost entities. If *include_copies* equals *0* and part boundary entity of type *brg_dim* is not owned by a local part (shortly, non-owned bridge type entity), *ghost_dim*-dimensional entities adjacent to the non-owned bridge type entity is not ghosted. If *include_copies* is non-zero integer, all *ghost_dim* dimensional entities adjacent to the bridge type entities on part boundaries are ghost copied.

The function fails in the following cases:

1. bridge dimension is greater than or equal to ghost dimension
2. bridge dimension is greater than or equal to mesh dimension
3. ghost dimension is mesh vertex
4. ghost dimension is grester than mesh dimension

Tagged data, field, global numbering and global entity ID are maintained during the ghosting.

```
int pumi_ghost_delete (pMesh  /* in */  m)
```

Given a mesh instance, delete ghost entities.

10.1.8 Miscellaneous

```
void pumi_mesh_createGlobalID(
    pMesh /* in */ m,
    pOwnership  /* im */  o=NULL)
```

Given a mesh instance and an ownership rule handle, generate global entity ID's for each dimension $0-3$. Note that the global entity ID is maintained during mesh re-partitioning (migration and distribution) and ghosting. To retrieve mesh entity's global ID, use *pumi_ment_getGlobalID*.

The ownership rule is an optional argument (default: *NULL*). If *NULL*, the default ownership rule provided by PUMI is used, where an owning part of part boundary entities is a part with the minimum number of elements among residence parts.

```
void pumi_mesh_deleteGlobalID(pMesh /* in */ m)
```

Given a mesh instance, delete global ID's generated by *pumi_mesh_createGlobalID*.

```
void pumi_mesh_verify (
    pMesh  /* in */  m,
    bool abort_on_error=true)
```

Given a mesh instance and a boolean flag specifing whether to abort on error or not, print the details of mesh entities,, check if the mesh is valid or not. Mesh verification with ghosted mesh is not supported. The argument *abort_on_error* is optional (default: true).

```
void pumi_mesh_print (
    pMesh  /* in */  m,
    bool  /* in */  print_entities=false)
```

Given a mesh instance and a boolean flag specifing whether to print the details of mesh entities, display the mesh information. The information includes size, tag, field and individual entities with global ID per part. The argument *print_entities* is optional (default: false).

10.2 Entity Functions

10.2.1 General entity information

```
int pumi_ment_getDim(pMeshEnt /* in */ e)
```

Given a mesh entity handle, return the dimension (or type) of the entity (0-3).

```
int pumi_ment_getTopo(pMeshEnt /* in */ e)
```

Given a mesh entity handle, return the topology of the entity. The entity topology is encoded in a C++ enumeration value,

```
enum PUMI_EntTopology {
  PUMI_VERTEX,   // 0
  PUMI_EDGE,     // 1
  PUMI_TRIANGLE, // 2
  PUMI_QUAD,     // 3
  PUMI_TET,      // 4
  PUMI_HEX,      // 5
  PUMI_PRISM,    // 6
  PUMI_PYRAMID   // 7
);
```

```
int pumi_ment_getID(pMeshEnt /* in */ e)
```

Given a mesh entity handle, return its local ID. The local entity ID is not sequential and not maintained during mesh modification (migration, re-partitioning, and ghosting, etc.).

```
int pumi_ment_getGlobalID(pMeshEnt /* in */ e)
```

Given a mesh entity handle, return its global ID generated. Note that the global entity ID is maintained during mesh migration, re-partitioning, and ghosting.

Prerequisite: *pumi_mesh_createGlobalID*

```
int pumi_ment_getNumAdj(
        pMeshEnt /* in */ e,
        int /* in */ target_dim)
```

Given a mesh entity and desired adjacency type *target_dim*, get the number of adjacent entities of the type.

```
int pumi_ment_getAdjacent(
        pMeshEnt /* in */ e,
        int /* in */ target_dim,
        Adjacent& /* out*/ adj_ents)
```

Given a mesh entity and desired adjacency type *target_dim*, get the *Adjacent* type container *adj_ents* filled with the adjacent entities of the type and return the number of resulting entities. *target_dim* should not be the same as the entity type. The type `Adjacent` is an array of entities. See how to iterate the resulting adjacent entities. This function is faster than `pumi_ment_getAdj`.

```
pMeshEnt vertex = ...;
Adjacent adjacent;
int num_adj=pumi_ment_getAdjacent(vertex, 2, adjacent);
for (int i=0; i<num_adj; ++i)
  pMeshEnt face = adjacent[i];
  ...
```

```
int pumi_ment_get2ndAdjacent(
        pMeshEnt /* in */ e,
        int /* in */ brg_dim,
        int /* in */ target_dim,
        Adjacent& /* out*/ adj_ents)
```

Given a mesh entity handle, bridge type *brg_dim*, and desired adjacency type *target_dim*, get the *Adjacent* type container *adj_ents* filled with 2^{nd} order adjacent entities of type *target_dim* obtained through the bridge type *brg_dim* and return the number of resulting entities. *target_dim* should be greater than *brg_dim*. The type `Adjacent` is an array of entities. This function is faster than `pumi_ment_get2ndAdj`. See how to iterate the resulting adjacent entities.

```
pMeshEnt face = ...;
Adjacent adjacent;
int num_adj=pumi_ment_get2ndAdjacent(face, 0, 2, adjacent);
```

```
for (int i=0; i<num_adj; ++i)
  pMeshEnt neighboring_face = adjacent[i];
  ...
```

```
void pumi_ment_getAdj(
        pMeshEnt /* in */ e,
        int /* in */ target_dim
        std::vector<pMeshEnt>& /* inout */ adj_ents)
```

Given a mesh entity and desired adjacency type *target_dim*, get the vector container *adj_ents* filled with the adjacent entities of the type. *target_dim* should not be the same as the entity type. If *target_dim* is negative, get all downward adjcent entities.

```
void pumi_ment_get2ndAdj (
        pMeshEnt /* in */ e,
        int /* in */ brg_dim,
        int /* in */ target_dim,
        std::vector<pMeshEnt>& /* inout */ adj_ents)
```

Given a mesh entity handle, bridge type *brg_dim*, and desired adjacency type *target_dim*, get the vector container *adj_ents* filled with 2^{nd} order adjacent entities of type *target_dim* obtained through the bridge type *brg_dim*. *brg_dim* and *target_dim* should not be equal.

```
pGeomEnt pumi_ment_getGeomClas(pMeshEnt /* in */ e)
```

Given a mesh entity handle, return the geometric entity classified on.

```
pMeshEnt pumi_medge_getOtherVtx (
        pMeshEnt /* in */ edge,
        pMeshEnt /* in */ vtx)
```

Given a mesh edge and a vertex handle which is adjacent to the edge, return the other vertex handle.

10.2.2 Mesh entity tagging

```
void pumi_ment_deleteTag (
        pMeshEnt /* in */ e,
        pMeshTag /* in */ tag)
```

Given mesh entity handle and tag handle, delete the tag data from the entity.

```
bool pumi_ment_hasTag (
        pMeshEnt /* in */ e,
        pMeshTag /* in */ tag)
```

Given mesh entity handle and tag handle, return *true* if the tag data is attached to the entity. Otherwise, return $false$.

```
void pumi_ment_setIntTag (
        pMeshEnt /* in */ e,
        pMeshTag /* in */ tag,
        int const* /* in */ data)
```

Given mesh entity handle, tag handle, and integer data, set or update the tag value of the entity. It fails if (i) tag type is not *PUMI_INT*, or (ii) the size of data doesn't match that of tag handle.

```
void pumi_ment_getIntTag(pMeshEnt e, pMeshTag tag, int* data)
```

Given mesh entity handle and tag handle, get integer type data tagged to the entity. It fails if (i) the tag doesn't exist with the entity, or (ii) tag type is not *PUMI_INT*.

```
void pumi_ment_setLongTag(pMeshEnt e, pMeshTag tag, long const* data)
```

Given mesh entity handle, tag handle, and long data, set or update the tag value of the entity. It fails if (i) tag type is not *PUMI_LONG*, or (ii) the size of data doesn't match that of tag handle.

```
void pumi_ment_getLongTag(pMeshEnt e, pMeshTag tag, long* data)
```

Given mesh entity handle and tag handle, get long type data tagged to the entity. It fails if (i) the tag doesn't exist with the entity, or (ii) tag type is not *PUMI_LONG*.

```
void pumi_ment_setDblTag(pMeshEnt e, pMeshTag tag, double const* data)
```

Given mesh entity handle, tag handle, and double data, set or update the tag value of the entity. It fails if (i) tag type is not *PUMI_DBL*, or (ii) the size of data doesn't match that of tag handle.

```
void pumi_ment_getDblTag (
        pMeshEnt  /* in */  e,
        pMeshTag  /* in */  tag,
        double*  /* out */  data)
```

Given mesh entity handle and tag handle, get double type data tagged to the entity. It fails if (i) the tag doesn't exist with the entity, or (ii) tag type is not *PUMI_DBL*.

10.2.3 Entity in parallel

```
int pumi_ment_getOwnPID(
        pMeshEnt  /* in */  e,
        pOwnership  /* in */  o=NULL)
```

Given a mesh entity handle and an ownership rule, return owning part ID (part ID where the owning entity exists). The ownership rule is an optional argument (default: *NULL*). If *NULL*, the default ownership rule provided by PUMI is used, where an owning part of part boundary entities is a part with the minimum number of elements among residence parts. Note that the global numbering is maintained during mesh migration, re-partitioning, and ghosting.

```
pMeshEnt pumi_ment_getOwnEnt(
        pMeshEnt  /* in */  e,
        pOwnership  /* in */  o=NULL)
```

Given a mesh entity handle and an ownership rule, return owning entity handle. The ownership rule is an optional argument (default: *NULL*). If *NULL*, the default ownership rule provided by PUMI is used, where an owning part of part boundary entities is a part with the minimum number of elements among residence parts.

```
bool pumi_ment_isOwned(
        pMeshEnt  /* in */  e,
        pOwnership  /* in */  o=NULL)
```

Given a mesh entity handle and an ownership rule, return *true* if the entity is owned by the local process. Otherwise, return $false$. The ownership rule is an optional argument (default: *NULL*). If *NULL*, the default ownership rule provided by PUMI is used, where an owning part of part boundary entities is a part with the minimum number of elements among residence parts.

```
bool pumi_ment_isOnBdry (pMeshEnt /* in */ e)
```

Given a mesh entity handle, return *true* if the entity is on part boundary. Otherwise, return $false$.

```
int pumi_ment_getNumRmt (pMeshEnt /* in */ e)
```

Given a mesh entity handle, return the number of remote copies. If the entity is duplicated on N processes, $N-1$ is returned.

```
void pumi_ment_getAllRmt(
      pMeshEnt /* in */ e,
      Copies& /* out */ copies)
```

Given a mesh entity handle, get *copies* filled with remote part ID and the memory address of the entity on the remote part. `Copies` is a `std::map` whose keys are part ID's and whose values are pointers to entities on those parts. All operations available for `std::map` can be applied to `Copies`. For instance, `copies.size()` indicates the number of remote copies and `copies[i]` is the remote entity copy on part i.

See how to iterate the remote copies using the iterator type `pCopyIter`.

```
pMeshEnt vertex = ...;
Copies remotes;
pumi_ment_getAllRmt(vertex, remotes);
for (pCopyIter it = remotes.begin(); it != remotes.end(); ++it)
  std::cout << "shared with part "<<it->first<<" on address "<<it->second<<"\n";
```

```
pMeshEnt pumi_ment_getRmt(
        pMeshEnt /* in */ e,
        int /* in */ part_id)
```

Given a mesh entity handle and remote part ID, return the memory address of the entity on the part.

```
bool pumi_ment_isGhost(pMeshEnt /* in */ e)
```

Given a mesh entity handle, return *true* if the entity is a ghost copy. Otherwise, return $false$.

```
bool pumi_ment_isGhosted(pMeshEnt /* in */ e)
```

Given a mesh entity handle, return *true* if the entity is ghosted (its ghost copy exists on off-part process). Otherwise, return $false$.

```
int pumi_ment_getNumGhost (pMeshEnt /* in */ e)
```

Given a mesh entity handle, get the number of ghost copies. If the entity is a ghost copy, 0 is retured.

```
void pumi_ment_getAllGhost(
      pMeshEnt /* in */ e,
      Copies& /* out */ copies)
```

Given a mesh entity handle, get *copies* filled with ghost part ID and the memory address of the ghost copy on the part. If e is a ghost copy, the owner part ID and the memory address of the owning copy is returned. `Copies` is a `std::map` whose keys are part IDs and whose values are pointers to entities on those parts. All operations available for `std::map` can be applied to `Copies`

See how to iterate the ghost copies using the iterator type `pCopyIter`.

```
pMeshEnt vertex = ...;
Copies ghosts;
pumi_ment_getAllGhost(vertex, ghosts);
for (pCopyIter it = ghosts.begin(); it != ghosts.end(); ++it)
  std::cout << "ghosted on part "<<it->first<<" on address "<<it->second<<"\n";
```

```
pMeshEnt pumi_ment_getGhost(
        pMeshEnt /* in */ e,
        int /* in */ part_id)
```

Given a mesh entity handle and ghost part ID, return the memory address of the ghost copy on the part.

```
bool pumi_ment_isOn (
      pMeshEnt /* in */ e,
      int /* in */ partID)
```

Given a mesh entity handle and part ID, return *true* if the entity exists on the part as a remote copy or ghost copy. Otherwise, return $false$.

```
void pumi_ment_getResidence(
        pMeshEnt /* in */ e,
        Parts& /* inout */ residence)
```

Given a mesh entity handle, get *residence* filled with residence part ID's where the entity physically exists as a remote copy or a ghost copy. `Parts` is a `std::set` whose values are integers indicating part ID's. All operations available for `std::set` can be applied to `Parts`. For instance, `insert`, `size`, `union`, etc.

```
void pumi_ment_getClosureResidence(
        pMeshEnt /* in */ e,
        Parts& /* inout */ residence)
```

Given a mesh entity handle, get *residence* filled with residence part ID's where the entity and its downward adjacent entities physically exist as for a remote copy or a ghost copy. `Parts` is a `std::set` whose values are integers indicating part ID's. All operations available for `std::set` can be applied to `Parts`. All operations available for `std::set` can be applied to `Parts`. For instance, `insert`, `size`, `union`, etc.

11 Field API's

This chapter describes API functions for field shape, field node and field management.

11.1 Field Shape

A field shape defines the node distribution where the coordinates and field values (DOF's) are stored. PUMI supports the following field shapes:

- Linear: The default field shape. Nodes are associated only with vertices and there is no node associated with other entities.
- Lagrange: Lagrangian shape function of some polynomial order (first and second order only)
- Serendipity: Serendipity shape functions of second order
- Constant: Constant shape function over a specific dimension. Shape function places a node on every element of the given dimension up to 3
- Integration Point (IP): Shape function over the integration points of elements. Orders 1 to 3 for dimension 2 or 3 are available.
- Voronoi: Equivalent to the Integration Point except that it is capable of evaluating as a shape function whose value at any point in the element is the value of the closest integration point in that element.
- Integration Point Fit (IPF): equivalent to the Integration Point except that it is capable of evaluating as a shape function whose value at any point in the element is a polynomial fit to the integration point data in that element.
- Hierarchic: Quadratic hierarchic shape function (first and second order only)

```
pShape pumi_mesh_getShape (pMesh  /* in */  m)
```

Given a mesh instance, return its shape handle. The default field shape of the mesh is a linear field shape, where *xyz* coordinates and DOF's are created only over the mesh vertices. Use *pumi_mesh_setShape* to change the default field shape.

```
void pumi_mesh_setShape (
        pMesh  /* in */  m,
        pShape  /* in */  s,
        bool  /* in */  project=true)
```

Given a mesh instance and field shape handle, set the field shape associated with the mesh. Mesh's existing coordinate field is replaced with a new fresh coordinate field. If *project* is *true*, project coordinate values from the old coordinate field to the new coordinate field. If *project* to *false* and coordinate field is not manually set, the file I/O with the new shape will fail. If *project* is not provided, the default is *true*.

```
int pumi_shape_getNumNode (
        pShape  /* in */  s,
        int  /* in */  t)
```

Given a field shape handle and entity topology, return the number of nodes associated with the entity topology.

```
int pumi_shape_hasNode (
        pShape  /* in */  s,
        int  /* in */  topology)
```

Given a field shape handle and an entity topology, return `true` if nodes exist for the topology. Otherwise, return `false`. The entity topology is encoded in a C++ enumeration type.

```
enum PUMI_EntTopology {
  PUMI_VERTEX,   // 0
  PUMI_EDGE,     // 1
  PUMI_TRIANGLE, // 2
  PUMI_QUAD,     // 3
  PUMI_TET,      // 4
  PUMI_HEX,      // 5
  PUMI_PRISM,    // 6
  PUMI_PYRAMID   // 7
);
```

```
pShape pumi_shape_getLagrange (int  /* in */  order)
```

Given a polynomial order, get the Lagrangian shape function of the order (first and second order only).

```
pShape pumi_shape_getSerendipity ()
```

Get the Serendipity shape functions of second order.

```
pShape pumi_shape_getConstant (int  /* in */ dimension)
```

Given an entity dimension (0-3), get the constant shape function over the specific dimension. The constant shape function places a node on every element of the given dimension.

```
pShape pumi_shape_getIP (
        int  /* in */  dimension,
        int  /* in */  order)
```

Given a dimension and order, get the Integration Point distribution. *dimension* is the dimensionality of the elements order. *order* is the order of accuracy, which determines the integration points. This allows users to create a field which has values at the integration points of elements. Orders 1 to 3 for dimension 2 or 3 are supported.

```
pShape pumi_shape_getVoronoi (
        int  /* in */  dimension,
        int  /* in */  order)
```

Given a dimension and order, get the Voronoi shape function. The Voronoi FieldShape is equivalent to the IPShape except that it is capable of evaluating as a shape function whose value at any point in the element is the value of the closest integration point in that element.

```
pShape pumi_shape_getIPFit (
        int  /* in */  dimension,
        int  /* in */  order)
```

Given a dimension and order, get the IP Fit shape function. The IP Fit FieldShape is equivalent to the IPShape except that it is capable of evaluating as a shape function whose value at any point in the element is a polynomial fit to the integration point data in that element.

```
pShape pumi_shape_getHierarchic (int  /* in */  order)
```

Given an order, get the quadratic hierarchic shape function (only first and second order are supported).

11.2 Field Node

11.2.1 Node Coordinates

```
void pumi_node_setCoord (
        pMeshEnt  /* in */  e,
        int  /* in */  n,
        double*  /* in */  coord)
```

Given a mesh entity handle, node order n and xyz coordinates, set or update the coordinates of the n^{th} node of the entity. n starts from 0 and must be less than the number of nodes associated with the entity type. If the entity type is vertex, the order n is 0.

```
void pumi_node_getCoord (
        pMeshEnt  /* in */  e,
        int  /* in */  n,
        double*  /* out */  coord)
```

Given a vertex handle and node order n, get xyz coordinates of the n^{th} node of the entity. n starts from 0 and must be less than the number of nodes associated with the entity type. If the entity type is vertex, the order n is 0.

```
void pumi_node_setCoordVector (
        pMeshEnt  /* in */  e,
        int  /* in */  n,
        Vector3&  /* out */  coord)
```

Given a mesh entity handle, node order n and xyz coordinates in data type `Vector3`, set or update the coordinates of the n^{th} node of the entity. n starts from 0 and must be less than the number of nodes associated with the entity type. If the entity type is vertex, the order n is 0. The data type `Vector3` is a vector of three doubles. `Vector3` can be treated like an array, but users can use a C++ stream output operator (`std::cout`) and mathematical operators like addition, subtraction, multiplication by a scalar, cross production, etc.

```
void pumi_node_getCoordVector (
        pMeshEnt  /* in */  e,
        int  /* in */  n,
        Vector3& /* out */  coord)
```

Given a vertex handle and node order n, get xyz coordinates of the n^{th} node of the entity in `Vector3` object. n starts from 0 and must be less than the number of nodes associated with the entity type. If the entity type is vertex, the order i is always 0. `Vector3` can be treated like an array, but users can use a C++ stream output operator (`std::cout`) and mathematical operators like addition, subtraction, multiplication by a scalar, cross production, etc.

```
Vector3 pumi_vector3_cross (
        Vector3 const& /* in */ a,
        Vector3 const& /* in */ b)
```

Given two Vector3 objects, return cross production in `Vector3` object.

11.2.2 Node Numbering

```
pNumbering pumi_numbering_create(
        pMesh  /* in */  m,
        const char*  /* in */  name,
        pShape  /* in */  shape=NULL,
        int  /* in */  num_component=1)
```

Given a mesh instance, character string name, a field shape handle, and the number of component (degree of freedom) per node, *(i)* create a numbering object and *(ii)* return the numbering handle. Note that no number is assigned to DOF's. `pumi_node_setNumber` and `pumi_node_getNumber` allow the user to set/get the number of node.

The argument *shape* is optional (*NULL*). If *NULL*, the field shape of the mesh (a field shape returned by `pumi_mesh_getShape`) is used. See Section 11.1 for the details of field shape.

The argument *num_component* is also optional (default: 1).

```
pNumbering pumi_numbering_createLocal(
        pMesh  /* in */  m,
        const char*  /* in */  name,
        pShape  /* in */  shape=NULL)
```

Given a mesh instance, character string name and a field shape handle, *(i)* create a numbering object for nodes, *(ii)* assign a local number to *all nodes*, and *(iii)* return the numbering handle. The number begins from 0 on each process. Call `pumi_node_getNumber` to retrieve the number.

The argument *shape* is optional (*NULL*). If *NULL*, the field shape of the mesh (a field shape returned by `pumi_mesh_getShape`) is used. See Section 11.1 for the details of field shape.

```
pNumbering pumi_numbering_createGlobal (
        pMesh  /* in */  m,
        const char*  /* in */  name,
        pShape  /* in */  shape=NULL,
        pOwnership  /* in */  own=NULL)
```

Given a mesh instance, character string name, a field shape handle and an ownership rule, *(i)* create a numbering object for *owned entities*, *(ii)* assign a global number to *owned nodes*, and *(iii)* return the numbering handle. Call `pumi_node_getNumber` to retrieve the number.

The argument *shape* is optional (*NULL*). If *NULL*, the field shape of the mesh (a field shape returned by `pumi_mesh_getShape`) is used. See Section 11.1 for the details of field shape

The ownership rule is optional (default: *NULL*). If *NULL*, the default ownership rule provided by PUMI is used, where an owning part of part boundary entities is a part with the minimum part ID among residence parts.

```
pNumbering pumi_numbering_createOwn(
        pMesh  /* in */  m,
        const char*  /* in */  name,
        pShape  /* in */  shape=NULL,
        pOwnership  /* in */  own=NULL)
```

Given a mesh instance, character string name, a field shape handle and an ownership rule, *(i)* create a numbering object for owned nodes, *(ii)* assign a local number to *owned nodes*, and *(iii)* return the numbering handle. The number begins from 0 on each process. Call `pumi_node_getNumber` to retrieve the number.

The argument *shape* is optional (*NULL*). If *NULL*, the field shape of the mesh (a field shape returned by `pumi_mesh_getShape`) is used. See Section 11.1 for the details of field shape

The ownership rule is optional (default: *NULL*). If *NULL*, the default ownership rule provided by PUMI is used, where an owning part of part boundary entities is a part with the minimum part ID among residence parts.

```
pNumbering pumi_numbering_createOwnDim(
        pMesh  /* in */  m,
        const char*  /* in */  name,
        int  /* in */  dimension,
        pOwnership  /* in */  own=NULL)
```

Given a mesh instance, character string name, an entity dimension (0-3) and an ownership rule, *(i)* create a numbering object for owned entities of the dimension, *(ii)* assign a local number to *owned nodes*, and *(iii)* return the numbering handle. The number begins from 0 on each process. Call `pumi_node_getNumber` to retrieve the number.

The ownership rule is optional (default: *NULL*). If *NULL*, the default ownership rule provided by PUMI is used, where an owning part of part boundary entities is a part with the minimum part ID among residence parts.

```
pNumbering pumi_numbering_createProcGrp (
        pMesh  /* in */  m,
        const char*  /* in */  name,
        int  /* in */  n_pgrp,
        int  /* in */  dimension,
        pOwnership  /* in */  own=NULL)
```

Given a mesh instance, character string name, the number of process group *n_pgrp*, and entity dimension, *(i)* create a numbering object for all entities of the dimension, *(ii)* assign a *intra-plane* global number to nodes of entities, and *(iii)* return the numbering handle. For 16 processes, if *n_pgrp* is 4, the processes 0 to 3 are process group 0, and the processes 4 to 7 are process group 1, and so on. Therefore, the *sequential and unique* numbers are assined to nodes in each process group (start number is 0). Call `pumi_node_getNumber` to retrieve the number.

The ownership rule is optional (default: *NULL*). If *NULL*, the default ownership rule provided by PUMI is used, where an owning part of part boundary entities is a part with the minimum part ID among residence parts.

Prerequisite 1: # total processes modulo $n\_pgrp = 0$
Prerequisite 2: any mesh entity in process group p is owned by a process belonging to the group p

```
void pumi_numbering_delete (pNumbering  /* in */  nb)
```

Given a numbering handle, delete the numbering object.

```
void pumi_numbering_print (
    pNumbering  /* in */  nb,
    int  /* in */  p=-1)
```

Given a numbering handle and a process rank p, print the node numbering of the process p. The process rank is an optional argument (default: -1). If -1, the numbering of all processes will be printed.

```
int pumi_numbering_getNumNode (pNumbering  /* in */ nb)
```

Given a numbering handle, return the number of nodes numbered in the numbering object.

```
void pumi_node_setNumber (
```

```
        pNumbering  /* in */  nb,
        pMeshEnt  /* in */  e,
        int  /* in */  n,
        int  /* in */  c,
        int  /* in */  number)
```

Given a numbering handle, a mesh entity handle, node order (n), component (DOF) order (c), and an integer number, number the c^{th} component of n^{th} node of the entity. The optional arguments n and c specify a high-order field node and degree of freedom per node (default: 0). In case the mesh entity has only one node and one component, n and c are 0.

```
int pumi_node_getNumber (
        pNumbering  /* in */ nb,
        pMeshEnt /* in */ e,
        int  /* in */  n=0,
        int  /* in */  c=0,
        int  /* in */  number)
```

Given a numbering handle, a mesh entity handle, node order (n), component (DOF) order (c), return the number of the c^{th} component of n^{th} node of the entity. The optional arguments n and c specify a high-order field node and degree of freedom per node (default: 0). In case the mesh entity has only one node and one component, n and c are 0.

```
bool pumi_node_isNumbered (
        pNumbering  /* in */  nb,
        pMeshEnt  /* in */  e,
        int  /* in */  n=0,
        int  /* in */  c=0)
```

Given a numbering handle, a mesh entity handle, node order (n), component (DOF) order (c), return *true* if the c^{th} component of n^{th} node of the entity is numbered. Otherwise, return $false$. The optional arguments n and c specify a high-order field node and degree of freedom per node (default: 0). In case the mesh entity has only one node and one component, n and c are 0.

11.3 Field Management

```
pField pumi_field_create (
        pMesh  /* in */  m,
        const char*  /* in */  name,
        int  /* in */  num_component,
        int  /* in */  type=PUMI_PACKED,
        pShape  /* in */  shape = NULL)
```

Given a mesh handle, name, size (the number of DOF's per node), field type and field shape, create a field and return the field handle. The supported field type is encoded in a C++ enumeration type.

- PUMI_SCALAR: a single scalar value (size=1)
- PUMI_VECTOR: a 3D vector (size=3)
- PUMI_MATRIX: a 3x3 matrix (size=9)
- PUMI_PACKED: a user-defined set of field data of any size

The argument *type* is optional (default: PUMI_PACKED). The argument *num_component* is used only if the field type is `PUMI_PACKED`.

The argument *shape* is optional (*NULL*). If *NULL*, the field shape of the mesh (a field shape returned by `pumi_mesh_getShape`) is used. See Section 11.1 for the details of field shape.

There are two options to store the field data: (i) a double *tag* per entity and (ii) a *contiguous* double *array* over the entire mesh. By default, the field data is stored in a double tag for each entity. `pumi_field_freeze` and `pumi_field_unfreeze` allow the user to change the field data structure from *tag* to *array*, and vice versa.

```
void pumi_node_setField (
        pField  /* in */  f,
        pMeshEnt /* in */ e,
        int  /* in */  n,
        double* /* in */ dof_data)
```

Given a field handle, a mesh entity handle, node order n, and double array, set or update the field data of n^{th} node of the entity. n starts from 0 and must be less than the number of nodes associated with the entity type. If the entity type is vertex, the order i is 0.

```
void pumi_node_getField (
        pField  /* in */  f,
        pMeshEnt  /* in */  e,
        int  /* in */  n,
        double*  /* out*/  dof_data)
```

Given a field handle, a mesh entity handle, and node order n, get field data of n^{th} node of the entity. n starts from 0 and must be less than the number of nodes associated with the entity type. If the entity type is vertex, the order i is 0.

```
int pumi_field_getSize (pField  /* in */  f)
```

Give a field handle, return the number of components (DOF's) per node.

```
int pumi_field_getType (pField  /* in */  f)
```

Give a field handle, return the field type.

```
std::string pumi_field_getName (pField  /* in*/  f)
```

Give a field handle, return the field name.

```
pShape pumi_field_getShape (pField  /* in */  f)
```

Give a field handle, return the field shape handle.

```
pNumbering pumi_field_getNumbering (pField  /* in */  f)
```

Given a field handle, return a numbering object associated with the field shape where local numbers for all nodes are assigned.

```
void pumi_field_delete (pField  /* in */  f)
```

Given a field handle, delete the field.

```
void pumi_field_synchronize (
        pField  /* in */  f,
        pOwnership  /* in */  own=NULL)
```

Given a field handle and an ownership rule, synchronize field data between remote, ghost and matched copies. The owner copy's field data is copied to the rest of copies. The ownership rule is an optional argument and the default is *NULL*. If *NULL*, the default ownership rule provided by PUMI is used, where an owning part of part boundary entities is a part with the minimum part ID among residence parts.

```
void pumi_field_accumulate (pField  /* in */  f)
```

Given a field handle, add up field data between remote copies then synchronize.

```
void pumi_field_freeze (pField  /* in */  f)
```

Given a field handle, turn the data structure of field data storage from a contiguous double array tag per entity to a double array over the entire mesh.

```
void pumi_field_unfreeze (pField  /* in */  f)
```

Given a field handle, turn the data structure of field data storage from a contiguous double array over the entire mesh to double array tag per entity.

```
pField pumi_mesh_findField (
        pMesh  /* in */  m,
        const char* /* in */ name)
```

Given a mesh instance and field name, return a field handle with the given name. If no field handle is found, return *NULL*.

```
int pumi_mesh_getNumField (pMesh  /* in */  m)
```

Given a mesh instance, return the number of fields created in the mesh.

```
pField pumi_mesh_getField (
        pMesh  /* in */  m,
        int /* in */ i)
```

Given a mesh instance and integer i, return the i^{th} field handle of the mesh.

```
void pumi_field_copy (
    pField  /* in */  f,
    pField  /* out */  r)
```

Copy the field data from field f to field r.

```
void pumi_field_add (
    pField  /* in */  f1,
    pField  /* in */  f2,
    pField  /* out */  r)
```

Add the field data in field $f1$ and field $f2$ and write the result to field r.

```
void pumi_field_multiply (
    pField  /* in */  f,
    double  /* in */  d,
    pField  /* out */  r)
```

Multiply the field data in field f by a double d, and write the result to field r.

```
void pumi_field_verify (
        pMesh /* in */ m,
        pField  /* in */  f=NULL,
        pOwnership  /* in */  own=NULL)
```

Given a mesh instance, a field handle and an ownership rule, verify the field data between remote and ghost copies. The field handle is an optional argument. If it is *NULL*, all field handles in the mesh are verified. If field data between copies don't match, warning messages are displayed.

The ownership rule is optional (default: *NULL*). If *NULL*, the default ownership rule provided by PUMI is used, where an owning part of part boundary entities is a part with the minimum part ID among residence parts.

```
void pumi_field_print (pField  /* in */  f)
```

Given a field handle, print the field data for all nodes.

12 Example Programs

This chapter presents example programs with PUMI API's.

12.1 *Model/Mesh loading*

```
// [INPUT]
// argv[1]: geometric model file
// argv[2]: mesh file
// argv[3]: # parts in mesh file
//
// 1. display system information, # processes running
// 2. load geometric model and mesh
// 3. display the elapsted time and heap memory increase

#include <mpi.h>
#include <pumi.h>
#include <pumi_errorcode.h>

const char* modelFile = 0;
const char* meshFile = 0;
int num_in_part = 0;

void getConfig(int argc, char** argv)
{
  if ( argc < 4 ) {
    if ( !pumi_rank() )
      printf("Usage: %s <model> <mesh> <num_part_in_mesh>\n", argv[0]);
    MPI_Finalize();
    exit(EXIT_FAILURE);
  }

  modelFile = argv[1];
  meshFile = argv[2];
  num_in_part = atoi(argv[3]);
}

int main(int argc, char** argv)
{
  MPI_Init(&argc,&argv);
  pumi_start();
  pumi_printSys();
  cout<<"Running Test on "<<pumi_size()<<" processes\n";

  // read input args - in-model-file in-mesh-file num-in-part
  getConfig(argc,argv);

  double begin_mem = pumi_getMem(), begin_time=pumi_getTime();

  pGeom g = pumi_geom_load(modelFile);
  pMesh m = pumi_mesh_load(g, meshFile, num_in_part);
```

```
  // print elapsed time and increased heap memory
  pumi_printTimeMem("elapsed time and increased heap memory:",
              pumi_getTime()-begin_time, pumi_getMem()-begin_mem);

  // clean-up
  pumi_mesh_delete(m);
  pumi_finalize();
  MPI_Finalize();

  return PUMI_SUCCESS;
}
```

12.2 *Communication with PCU.h*

```
// 1. iterate over mesh faces
// 2. if mesh face is ghosted, send the first ghost copy and the mesh face to its first ghost part

#include <mpi.h>
#include <PCU.h>
#include <pumi.h>
#include <pumi_errorcode.h>

int main(int argc, char** argv)
{
  char message[50];
  MPI_Init(&argc,&argv);
  pumi_start();

  ...
  // begin message exchange
  PCU_Comm_Begin();

  pMeshEnt e;
  pMeshIter it = m->begin(2);
  while ((e = m->iterate(it))) // iterate mesh faces
  {
    if (!pumi_ment_isGhosted(e)) continue; // skip if not ghosted
    Copies temp;
    pumi_ment_getAllGhost(e,temp);
    int to = temp.begin()->first;
    PCU_COMM_PACK(to,temp.begin()->second); // sender
    PCU_COMM_PACK(to,e);
  }
  PCU_Comm_Send();
  while (PCU_Comm_Receive())
  {
    int from = PCU_Comm_Sender();
    pMeshEnt g;
    PCU_COMM_UNPACK(g);
    pMeshEnt s;
    PCU_COMM_UNPACK(s);
```

```
  }

  pumi_finalize();
  MPI_Finalize();

  return PUMI_SUCCESS;
}
```

12.3 *Tagging with geometric model*

```
#include <pumi.h>
#include <string.h>
#include <cstring>

template <class T>
void TEST_TAG (pTag tag, char* in_name, int name_len, int in_type, int in_size)
{
  const char* tag_name;
  // verifying byte tag info
  pumi_tag_getName (tag, &tag_name);
  int tag_type= pumi_tag_getType (tag);
  int tag_size = pumi_tag_getSize (tag);
  int tag_byte= pumi_tag_getByte (tag);
  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);
  assert(!strcmp(tag_name, in_name)
         && tag_type == in_type && tag_size == in_size
         && ((size_t)tag_byte)==sizeof(T)*tag_size);
}

void TEST_GENT_SETGET_TAG (pGeom g, pGeomEnt ent)
{
  char data[] = "abcdefg";

  pTag pointer_tag=pumi_geom_findTag(g, "pointer");
  pTag int_tag=pumi_geom_findTag(g, "integer");
  pTag long_tag = pumi_geom_findTag(g, "long");
  pTag dbl_tag = pumi_geom_findTag(g, "double");
  pTag ent_tag = pumi_geom_findTag(g, "entity");
  pTag intarr_tag = pumi_geom_findTag(g, "integer array");
  pTag longarr_tag = pumi_geom_findTag(g, "integer array");
  pTag dblarr_tag = pumi_geom_findTag(g, "double array");
  pTag entarr_tag = pumi_geom_findTag(g, "entity array");

  // get the first geometric vertex to use as tag data
  pGeomEnt ent_tag_data=*(g->begin(0));

  // pumi_gent_set/getPtrTag
  pumi_gent_setPtrTag (ent, pointer_tag, (void*)(data));
  void* void_data = (void*)calloc(strlen(data), sizeof(char));
```

```
 pumi_gent_getPtrTag (ent, pointer_tag, &void_data);
 assert(!strcmp((char*)void_data, data));

 // pumi_gent_set/getIntTag
 pumi_gent_setIntTag(ent, int_tag, 1000);
 int int_data;
 pumi_gent_getIntTag (ent, int_tag, &int_data);
 assert(int_data == 1000);

 // pumi_gent_set/getLongTag
 pumi_gent_setLongTag(ent, long_tag, 3000);
 long long_data;
 pumi_gent_getLongTag (ent, long_tag, &long_data);
 assert(long_data==3000);

 // pumi_gent_set/getDblTag
 pumi_gent_setDblTag (ent, dbl_tag, 1000.37);
 double dbl_data;
 pumi_gent_getDblTag (ent, dbl_tag, &dbl_data);
 assert(dbl_data == 1000.37);

 // pumi_gent_set/getEntTag
 pumi_gent_setEntTag (ent, ent_tag, ent_tag_data);
 pGeomEnt ent_data;
 pumi_gent_getEntTag (ent, ent_tag, &ent_data);
 assert(ent_data == ent_tag_data);


// pumi_gent_set/GetIntArrTag with integer arr tag
 int int_arr[] = {4,8,12};
 int arr_size;
 pumi_gent_setIntArrTag (ent, intarr_tag, int_arr);
 int* int_arr_back = new int[4];
 pumi_gent_getIntArrTag (ent, intarr_tag, &int_arr_back, &arr_size);
 assert(arr_size==3 && int_arr_back[0] == 4 && int_arr_back[1] == 8 && int_arr_back[2] == 12);

  // pumi_gent_set/getDblArrTag
 double dbl_arr[] = {4.1,8.2,12.3};
 pumi_gent_setDblArrTag (ent, dblarr_tag, dbl_arr);
 double* dbl_arr_back = new double[4];
 pumi_gent_getDblArrTag (ent, dblarr_tag, &dbl_arr_back, &arr_size);
 assert(arr_size==3 && dbl_arr_back[0] == 4.1 && dbl_arr_back[1] == 8.2 &&
        dbl_arr_back[2] == 12.3);

 // pumi_gent_set/getEntArrTag
 pGeomEnt* ent_arr = new pGeomEnt[3];
 ent_arr[0] = ent_arr[1] = ent_arr[2] = ent_tag_data;
 pumi_gent_setEntArrTag (ent, entarr_tag, ent_arr);
 pGeomEnt* ent_arr_back = new pGeomEnt[4];
 pumi_gent_getEntArrTag (ent, entarr_tag, &ent_arr_back, &arr_size);
 assert(arr_size==3 && ent_arr_back[0] == ent_tag_data && ent_arr_back[1] ==
          ent_tag_data && ent_arr_back[2] == ent_tag_data
          && ent_arr[0]==ent_arr_back[0] && ent_arr[1]==ent_arr_back[1] &&
          ent_arr[2] == ent_arr_back[2]);
```

```
  delete [] int_arr_back;
  delete [] long_arr_back;
  delete [] dbl_arr_back;
  delete [] ent_arr;
  delete [] ent_arr_back;
}

void TEST_GENT_DEL_TAG (pGeom g, pGeomEnt ent)
{
  pTag pointer_tag=pumi_geom_findTag(g, "pointer");
  pTag int_tag=pumi_geom_findTag(g, "integer");
  pTag long_tag = pumi_geom_findTag(g, "long");
  pTag dbl_tag = pumi_geom_findTag(g, "double");
  pTag ent_tag = pumi_geom_findTag(g, "entity");
  pTag intarr_tag = pumi_geom_findTag(g, "integer array");
  pTag longarr_tag = pumi_geom_findTag(g, "long array");
  pTag dblarr_tag = pumi_geom_findTag(g, "double array");
  pTag entarr_tag = pumi_geom_findTag(g, "entity array");

  pumi_gent_deleteTag(ent, pointer_tag);
  pumi_gent_deleteTag(ent, int_tag);
  pumi_gent_deleteTag(ent, long_tag);
  pumi_gent_deleteTag(ent, dbl_tag);
  pumi_gent_deleteTag(ent, ent_tag);
  pumi_gent_deleteTag(ent, intarr_tag);
  pumi_gent_deleteTag(ent, longarr_tag);
  pumi_gent_deleteTag(ent, dblarr_tag);
  pumi_gent_deleteTag(ent, entarr_tag);

  assert(!pumi_gent_hasTag(ent, pointer_tag));
  assert(!pumi_gent_hasTag(ent, int_tag));
  assert(!pumi_gent_hasTag(ent, long_tag));
  assert(!pumi_gent_hasTag(ent, dbl_tag));
  assert(!pumi_gent_hasTag(ent, ent_tag));
  assert(!pumi_gent_hasTag(ent, intarr_tag));
  assert(!pumi_gent_hasTag(ent, longarr_tag));
  assert(!pumi_gent_hasTag(ent, dblarr_tag));
  assert(!pumi_gent_hasTag(ent, entarr_tag));
}

void TEST_GEOM_TAG(pGeom g)
{
  pTag pointer_tag = pumi_geom_createTag(g, "pointer", PUMI_PTR, 1);
  pTag int_tag = pumi_geom_createTag(g, "integer", PUMI_INT, 1);
  pTag long_tag = pumi_geom_createTag(g, "long", PUMI_LONG, 1);
  pTag dbl_tag = pumi_geom_createTag(g, "double", PUMI_DBL, 1);
  pTag ent_tag = pumi_geom_createTag(g, "entity", PUMI_ENT, 1);

  pTag intarr_tag=pumi_geom_createTag(g, "integer array", PUMI_INT, 3);
  pTag longarr_tag=pumi_geom_createTag(g, "long array", PUMI_LONG, 3);
  pTag dblarr_tag = pumi_geom_createTag(g, "double array", PUMI_DBL, 3);
  pTag entarr_tag = pumi_geom_createTag(g, "entity array", PUMI_ENT, 3);
```

```
  // verifying tag info
  TEST_TAG<void*>(pointer_tag, "pointer", strlen("pointer"), PUMI_PTR, 1);
  TEST_TAG<int>(int_tag, "integer", strlen("integer"), PUMI_INT, 1);
  TEST_TAG<long>(long_tag, "long", strlen("long"), PUMI_LONG, 1);
  TEST_TAG<double>(dbl_tag, "double", strlen("double"), PUMI_DBL, 1);
  TEST_TAG<pMeshEnt>(ent_tag, "entity", strlen("entity"), PUMI_ENT, 1);

  TEST_TAG<int>(intarr_tag, "integer array", strlen("integer array"), PUMI_INT, 3);
  TEST_TAG<long>(longarr_tag, "long array", strlen("long array"), PUMI_LONG, 3);
  TEST_TAG<double>(dblarr_tag, "double array", strlen("double array"), PUMI_DBL, 3);
  TEST_TAG<pMeshEnt>(entarr_tag, "entity array", strlen("entity array"), PUMI_ENT, 3);

  assert(pumi_geom_hasTag(g, int_tag));
  pTag cloneTag = pumi_geom_findTag(g, "pointer");
  assert(cloneTag);
  std::vector<pTag> tags;
  pumi_geom_getTag(g, tags);
  assert(cloneTag == pointer_tag && tags.size()==9);

  for (gGeomIter gent_it = g->begin(0); gent_it!=g->end(0);++gent_it)
  {
    TEST_GENT_SETGET_TAG(g, *gent_it);
    TEST_GENT_DEL_TAG(g, *gent_it);
  }

  // delete tags
  for (std::vector<pTag>::iterator tag_it=tags.begin(); tag_it!=tags.end(); ++tag_it)
    pumi_geom_deleteTag(g, *tag_it);

  tags.clear();
  pumi_geom_getTag(g, tags);

  assert(!tags.size());
}

int main(int argc, char** argv)
{
  MPI_Init(&argc,&argv);
  pumi_start();
  ...
  getConfig(argc,argv);
  pGeom g = pumi_geom_load(modelFile);
  TEST_GEOM_TAG(g);
  ...
  return PUMI_SUCCESS;
}
```

12.4 Mesh/Entity information

```
int main(int argc, char** argv)
{
  ...
```

```
  pMeshEnt e;
  vector<pMeshEnt> adj_ents;
  int mesh_dim=pumi_mesh_getDim(m);
  pMeshIter mit = m->begin(mesh_dim);
  while ((e = m->iterate(mit)))
  {
    assert(pumi_ment_getDim(e)==mesh_dim);
    // check # one-level up adjancent entities
    assert(pumi_ment_getNumAdj(e, mesh_dim+1)==0);
    // check # one-level down adjancent entities
    adj_ents.clear();
    pumi_ment_getAdj(e, mesh_dim-1, adj_ents);
    assert(adj_ents.size()==pumi_ment_getNumAdj (e, mesh_dim-1));
    if (!pumi_ment_isOnBdry(e)) continue; // skip internal entity
    // if entity is on part boundary, count remote copies
    Copies copies;
    pumi_ment_getAllRmt(e,copies);
    // check #remotes
    assert (pumi_ment_getNumRmt(e)==copies.size() && copies.size()>0);
    // check the entity is not ghost or ghosted
    assert(!pumi_ment_isGhost(e) && !pumi_ment_isGhosted(e));
  }
  m->end(mit);
  ...
}
```

12.5 2D mesh construction

```
  // create an empty model and 2D mesh
  pGeom g = pumi_geom_load (NULL, "null");
  pMesh m = pumi_mesh_create(g, 2);

  double xyz[3];
  std::vector<pMeshEnt> new_vertices;
  std::vector<pMeshEnt> new_edges;
  pMeshEnt vertices[2];
  pMeshEnt edges[3];

  for (int i=0; i<num_new_vertices; ++i)
  {
    //create a new vertex and store in vector "new_vertices"
    new_vertices.push_back (pumi_mesh_createVtx(m, NULL, xyz));
  }

  for (size_t i=0; i< new_vertices.size()/2-1; ++i)
  {
    vertices[0] = new_vertices[i];
    vertices[1] = new_vertices[i+1];
    // create a new edge and store in vector "new_edges"
    new_edges.push_back(pumi_mesh_createEnt(m, NULL, 1, vertices));
  }
```

```
  // note face-edge order is ignored in this example
  for (size_t i=0; i< new_edges.size()/3-1; ++i)
  {
    for (int k=0; k<3; ++k)
      edges[k] = new_edges[i+k];
    pumi_mesh_createEnt(m, NULL, 2, edges);
  }

  // finalize mesh entity creation
  pumi_mesh_freeze(m);
```

12.6 *Mesh tagging*

```
#include "pumi.h"

int main(int argc, char** argv)
{
  ...
  pMesh m;
  pMeshEnt e;
  ...
  pMeshTag int_tag = pumi_mesh_findTag(m, "integer");
  if (!int_tag) // int_tag doesn't exist
    int_tag = pumi_mesh_createIntTag(m, "integer", 1);
  pMeshTag long_tag = pumi_mesh_createLongTag(m, "long", 1);
  pMeshTag dbl_tag = pumi_mesh_createDblTag(m, "double", 1);

  pMeshTag intarr_tag=pumi_mesh_createIntTag(m, "integer array", 3);
  pMeshTag longarr_tag=pumi_mesh_createLongTag(m, "long array", 3);
  pMeshTag dblarr_tag = pumi_mesh_createDblTag(m, "double array", 3);


  // set/get integer tag
  int int_value=pumi_ment_getLocalID(ent), int_data;
  pumi_ment_setIntTag(ent, int_tag, &int_value);
  pumi_ment_getIntTag (ent, int_tag, &int_data);
  assert(int_data == int_value);
  ...
  // set/get double array tag
  double dbl_arr[] = {4.1,8.2,12.3};
  pumi_ment_setDblTag (ent, dblarr_tag, dbl_arr);
  double* dbl_arr_back = new double[3];
  pumi_ment_getDblTag (ent, dblarr_tag, dbl_arr_back);
  ...
  // retrieve all tags on mesh
  std::vector<pMeshTag> tags;
  pumi_mesh_getTag(m, tags);
  ...
  // delete tags
  for (std::vector<pMeshTag>::iterator tag_it=tags.begin(); tag_it!=tags.end(); ++tag_it)
    pumi_mesh_deleteTag(m, *tag_it);
  ...
```

```
}
```

12.7 Mesh partitioning

```
// 1. load 2D serial mesh
// 2. create a migration plan to partition the mesh as per model faces.
//    (if mesh face is classified on model face i (i>0), send the mesh face to part i.
// 3. check mesh validity

Migration* get_plan_per_model(pGeom g, pMesh m)
{
  int dim = pumi_mesh_getDim(m);
  int num_peers = pumi_size();
  Migration* plan = new Migration(m);
  if (!pumi_rank()) return plan; // only master process construct the plan

  pMeshEnt e;
  int num_gface = pumi_geom_getNumEnt(g, dim);
  assert(num_gface==pumi_size());
  int gface_id;
  int dest_pid;
  pMeshIter it = m->begin(2); // face
  while ((e = m->iterate(it)))
  {
    pGeomEnt gface = pumi_ment_getGeomClas(e); // get the classification
    gface_id = pumi_gent_getID(gface); // get the geom face id
    dest_pid = gface_id-1;
    plan->send(e, dest_pid);
  }
  m->end(it);
  return plan;
}

int main(int argc, char** argv)
{
  ...
  pGeom g = pumi_geom_load(modelFile);
  pMesh m = pumi_mesh_loadSerial(g, meshFile);
  Migration* plan = get_plan_per_model(g, m);
  pumi_mesh_migrate(m, plan);
  pumi_mesh_verify(m);
  ...
}
```

12.8 Mesh distribution

```
// 1. load 2D serial mesh
// 2. send 1/5 of mesh elements to three parts i-1, i, i+i.
//    i is modulo(element's local ID,# processes)
// 3. write the distributed mesh into file "part.smb"
```

```
int main(int argc, char** argv)
{
  ...
  pGeom g = pumi_geom_load(modelFile);
  pMesh m = pumi_mesh_loadSerial(g, meshFile);

  Distribution* plan = new Distribution(m);

  int dim=pumi_mesh_getDim(m), count=0, pid;
  pMeshIter it = m->begin(dim);
  while ((e = m->iterate(it)))
  {
    pid=pumi_ment_getLocalID(e)%pumi_size();
    plan->send(e, pid);
    if (pid-1>=0) plan->send(e, pid-1);
    if (pid+1<pumi_size()) plan->send(e, pid+1);
    if (count==pumi_mesh_getNumEnt(m, dim)/5) break;
    ++count;
  }
  m->end(it);

  pumi_mesh_distribute(m, plan);

  // write mesh in part.smb
  pumi_mesh_write(m,"part.smb");
}
```

12.9 Computing the area of adjacent faces

```
// loop over mesh vertices and calculate the area of adjacent faces
// if a mesh vertex is on part boundary, do communications to add
// the area of remote faces
  pMeshEnt e;
  pMeshTag area_tag = pumi_mesh_createDblTag(m, "area", 1);

  PCU_Comm_Begin();  // start communication

  pMeshIter it = m->begin(0);
  while ((e = m->iterate(it)))
  {
    double area = 0.;
    std::vector<pMeshEnt> faces;
    pumi_ment_getAdj(e, 2, faces);
    for (std::vector<pMeshEnt>::iterator fit=faces.begin(); fit!=faces.end(); ++fit)
      area += get_face_area(m, *fit); // get_face_area computes the face area
    pumi_ment_setDblTag(e, area_tag, &area);
    if (!pumi_ment_isOnBdry(e)) continue;
    Copies remotes;
    pumi_ment_getAllRmt(e,remotes);
    // loop over remote copies and send message to remote copies
    for (pCopyIter rit = remotes.begin(); rit != remotes.end(); ++rit)
```

```
    {
      PCU_COMM_PACK(rit->first, rit->second);
      PCU_Comm_Pack(rit->first, &area, sizeof(double));
    }
  }
  m->end(it);
  PCU_Comm_Send(); // send messages collected in buffers

  // part boundary vertices receive area from the remote copies and add them up
  while (PCU_Comm_Listen())
    while (!PCU_Comm_Unpacked())
    {
      pMeshEnt rmt_e;
      PCU_COMM_UNPACK(rmt_e);
      double rmt_area;
      PCU_Comm_Unpack(&rmt_area, sizeof(double));
      double area;
      pumi_ment_getDblTag (rmt_e, area_tag, &area);
      area+=rmt_area;
      pumi_ment_setDblTag (rmt_e, area_tag, &area);
    }
```

12.10 Entity-wise ghosting

```
Ghosting* getGhostingPlan(pMesh m)
{
  int mesh_dim=pumi_mesh_getDim(m);
  pMeshEnt e;
  Ghosting* plan = new Ghosting(m, mesh_dim);
  {
    pMeshIter it = m->begin(mesh_dim);
    int count=0, pid;
    while ((e = m->iterate(it)))
    {
      for (int i=0; i<pumi_size()/2; ++i)
      {
        pid = (pumi_ment_getGlobalID(e)+rand())%pumi_size();
        plan->send(e, pid);
      }
      ++count;
     if (count==pumi_mesh_getNumEnt(m, mesh_dim)/3) break;
    }
    m->end(it);
  }
  return plan;
}

int main(int argc, char** argv)
{
  ...
  int mesh_dim=pumi_mesh_getDim(m);
  pMeshEnt e;
```

```
  int* org_mcount=new int[4];
  for (int i=0; i<4; ++i)
    org_mcount[i] = pumi_mesh_getNumEnt(m, i);

  Ghosting* ghosting_plan = get_ghosting_plan(m);
  pumi_ghost_create(m, ghosting_plan);

  int num_ghost_vtx=0;
  pMeshIter mit = m->begin(0);
  while ((e = m->iterate(mit)))
  {
    if (pumi_ment_isGhost(e))
    {
      ++num_ghost_vtx;
     assert(pumi_ment_getOwnPID(e)!=pumi_rank());
    }
  }
  m->end(mit);
  assert(num_ghost_vtx+org_mcount[0]==pumi_mesh_getNumEnt(m,0));
  pumi_mesh_verify(m); // this should throw an error message
  pumi_ghost_delete(m);
  for (int i=0; i<4; ++i)
    assert(org_mcount[i] == pumi_mesh_getNumEnt(m, i));
  ...
}
```

12.11 Layer-wise ghosting

```
int main(int argc, char** argv)
{
  ...
  int mesh_dim=pumi_mesh_getDim(m);

  int* org_mcount=new int[4];
  for (int i=0; i<4; ++i)
    org_mcount[i] = pumi_mesh_getNumEnt(m, i);

  for (int brg_dim=mesh_dim-1; brg_dim>=0; --brg_dim)
    for (int num_layer=1; num_layer<=3; ++num_layer)
      for (int include_copy=0; include_copy<=1; ++include_copy)
      {
        pumi_ghost_createLayer (m, brg_dim, mesh_dim, num_layer, include_copy);
        pumi_ghost_delete(m);
        pumi_mesh_verify(m);
        for (int i=0; i<4; ++i)
          assert(org_mcount[i] == pumi_mesh_getNumEnt(m, i));
      }
  ...
}
```

12.12 Accumulative layer ghosting

```
int main(int argc, char** argv)
{
  ...
  int mesh_dim=pumi_mesh_getDim(m);

  int* org_mcount=new int[4];
  for (int i=0; i<4; ++i)
    org_mcount[i] = pumi_mesh_getNumEnt(m, i);

  for (int brg_dim=mesh_dim-1; brg_dim>=0; --brg_dim)
    for (int num_layer=1; num_layer<=3; ++num_layer)
      for (int include_copy=0; include_copy<=1; ++include_copy)
        pumi_ghost_createLayer (m, brg_dim, mesh_dim, num_layer, include_copy);

  pumi_ghost_delete(m);
  pumi_mesh_verify(m);
  for (int i=0; i<4; ++i)
    assert(org_mcount[i] == pumi_mesh_getNumEnt(m, i));
  ...
}
```

12.13 Field shape

```
  // get the default shape of the mesh - only vertex has a node
  pShape s = pumi_mesh_getShape(m);
  assert(pumi_shape_getNumNode(s, 1)==0);

  // change shape to Lagrange with order 2 - edge has a node
  pumi_mesh_setShape(m, pumi_shape_getLagrange(2));
  assert(pumi_shape_getNumNode(pumi_mesh_getShape(m), 1)==1);

  double xyz[3];
  pMeshIter it = m->begin(1);
  pMeshEnt e;
  // loop over edges and get and set the node coordinates
  while ((e = m->iterate(it)))
  {
    pumi_node_getCoord(e, 0, xyz);
    for (int i=0; i<3; ++i) xyz[i] += 0.5;
    pumi_node_setCoord(e, 0, xyz);
  }
  m->end(it);
```

12.14 Field manipulation

```
  int num_dofs_per_node=3;
  pField f =pumi_field_create(m, "xyz_field", num_dofs_per_node);
  assert(pumi_field_getName(f)==std::string("xyz_field"));
```

```
assert(pumi_field_getType(f)==PUMI_PACKED);
assert(pumi_field_getSize(f)==num_dofs_per_node);

// create global numbering for field node
pGlobalNumbering gn = pumi_mesh_createNumbering(m, "xyz_numbering", pumi_field_getShape(f));

// fill the dof data
double xyz[3];
pMeshIter it = m->begin(0);
pMeshEnt e;
while ((e = m->iterate(it)))
{
  if (!pumi_ment_isOwned(e)) continue;
  pumi_node_getCoord(e, 0, xyz);
  if (pumi_ment_isOnBdry(e))
    for (int i=0; i<3;++i)
      xyz[i] *= pumi_ment_getLocalID(e);
  pumi_ment_setField(e, f, 0, xyz);
}
m->end(it);

pumi_field_synchronize(f); // synchronize field value between remote copies

double data[3];
it = m->begin(0);
while ((e = m->iterate(it)))
{
  if (!pumi_ment_isOwned(e)) continue;
  pumi_node_getCoord(e, 0, xyz);
  pumi_ment_getField(e, f, 0, data);
  for (int i=0; i<3;++i)
    if (pumi_ment_isOnBdry(e))
      assert(data[i] == xyz[i]*pumi_ment_getLocalID(e));
}
m->end(it);

// delete global numbering
pumi_mesh_deleteNumbering(gn);
```

13 Installation

PUMI is a free open source software available in `https://github.com/SCOREC/core`. This section discuss the S/W requirements and compilation briefly. The detailed discussion on how to build PUMI can be found in *https://github.com/SCOREC/core/wiki/General-Build-instructions*.

13.1 S/W Requirements

At a minumum, the following softwares are required to install PUMI.

- `cmake` - v3.0 or higher
- `MPI`
- `METIS/ParMETIS` [21]
- `Zoltan` [9]

13.2 Compilation

To build PUMI libraries, run a cmake configuration file and do *"make install"*. Three example cmake configuration files are available in the top source folder; *'example_config.sh"*, *'mpich3-gcc4.4.5-config.sh"*, *'mpich3-gcc4.9.2-config.sh"*.

The essential configuration options include:

- `ZOLTAN_LIBRARY`: path and file name of Zoltan library
- `PARMETIS_LIBRARY`: path and file name of ParMETIS library
- `METIS_LIBRARY`: path and file name of METIS library
- `SCOREC_INCLUDE_DIR`: path to PUMI header files
- `SCOREC_LIB_DIR`: path and file name of PUMI libraries
- `ENABLE_PETSC`: set `ON` to link MSI with PETSc solver
- `PETSC_INCLUDE_DIR`: path to PETSc header files
- `PETSC_LIB_DIR`: path to PETSc libraries
- `ENABLE_TRILINOS`: set `ON` to link MSI with Trilinos solver
- `TRILINOS_INCLUDE_DIR`: path to Trilinos header files
- `TRILINOS_LIB_DIR`: path to Trilinos libraries
- `DENABLE_COMPLEX`: set `ON` to build MSI complex value

- `CMAKE_INSTALL_PREFIX`: path to install MSI header file and library

In the current version, build with Trilinos is not supported.

For a complete list of configuration options, see *"CMakeLists.txt"* in the top source folder.

13.3 Test Program

A test program with PETSc is available in test/petsc/main.cc in the top source folder.

To compile *"MSI API"* test program, do *"make test_pumi"* in your build directory. The input arguments of the executable *"test_pumi"* are the following:

- `argv[1]` - input model file (.dmg)
- `argv[2]` - input mesh file (.smb)
- `argv[3]` - output mesh file (.smb)
- `argv[4]` - the number of parts in input mesh

How to generate PUMI-readable model and mesh files is beyond the scope of this document.

Many other test programs are available in *"test"* folder. See *"test/CMakeLists.txt"* for a complete list of available test programs. The test programs are good start to learn how to use PUMI.

14 Closing Remark

Support for PUMI was provided through the Department of Energy (DOE) office of Science's Scientific Discovery through Advanced Computing (SciDAC) institute as part of the Frameworks, Algorithms, and Scalable Technologies for Mathematics (FASTMath) program, under grant DE-SC0006617.

The latest source is downloadable from `https://github.com/SCOREC/core` and the user's guide is available from `http://www.scorec.rpi.edu/~seol/PUMI.pdf`.

For all inquiries on PUMI including how to generate PUMI-readable model and mesh files, email to *shephard@rpi.edu*.

References

[1] Frédérik Alauzet, Xiangrong Li, E. Seegyoung Seol, and Mark S. Shephard. Parallel anisotropic 3d mesh adaptation by mesh modification. *Engineering with Computers*, 21(3):247–258, jan 2006.

[2] C. Ozturan amd 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.

[3] Mark W. Beall. *An object-oriented framework for the reliable automated solution of problems in mathematical physics.* PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, 1999.

[4] Mark W. Beall and Mark S. Shephard. A general topology-based mesh data structure. *International Journal for Numerical Methods in Engineering*, 40(9):1573–1596, may 1997.

[5] Erik G. Boman, Ümit V. ¸Catalyürek, Cédric Chevalier, and Karen D. Devine. The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and coloring. *Scientific Programming*, 20(2):129–150, 2012.

[6] Waldemar Celes, Glaucio H. Paulino, and Rodrigo Espinha. A compact adjacency-based topological data structure for finite element mesh representation. *International Journal for Numerical Methods in Engineering*, 64(11):1529–1556, nov 2005.

[7] 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.

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

[9] Karen D. Devine, Erik G. Boman, Robert Heaphy, Bruce Hendrickson, and Courtenay Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, 2002.

[10] 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.

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

[12] D.A. Ibanez, E.S. Seol, C.W. Smith, and M.S.Shephard. Pumi: Parallel unstructured mesh infrastructure. *ACM Transaction on Mathematical Software*, 42(3):17:1–17:28, 2016.

[13] ITAPS: Interoperable technologies for advanced petascale simulations of department of energy's scientific discovery through advanced computing (scidac), 2016. http://www.itaps.org.

[14] Benjamin S. Kirk, John W. Peterson, Roy H. Stogner, and Graham F. Carey. libmesh: a c++ library for parallel adaptive mesh refinement/coarsening simulations. *Engineering with Computers*, 22(3–4):237–254, dec 2006.

[15] M. Mantyla. In *An Introduction to Solid Modeling.* Computer Science Press, Rockville Maryland, 1988.

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

[17] C. Ollivier-Gooch, K. Chand, T. Dahlgren, L.F. Diachin, B. Fix, J. Kraftcheck, X. Li, E.S. Seol, M.S. Shephard, T. Tautges, and H. Trease. The tstt mesh interface. *44th AIAA Aerospace Sciences Meeting and Exhibit (AIAA)*, 2006.

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

[20] Jean-François Remacle and Mark S. Shephard. An algorithm oriented mesh database. *International Journal for Numerical Methods in Engineering*, 58(2):349–374, sep 2003.

[21] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience*, 14(3):219–240, mar 2002.

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

[23] E. Seegyoung Seol. *FMDB: flexible distributed mesh database for parallel automated adaptive analysis*. PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, 2005.

[24] E. Seegyoung Seol and Mark S. Shephard. Efficient distributed mesh data structure for parallel automated adaptive analysis. *Engineering with Computers*, 22(3):197–213, dec 2006.

[25] 2016. http://www.sgi.com/tech/stl/stl_index.html.

[26] Mark S. Shephard. Meshing environment for geometry-based analysis. *International Journal for Numerical Methods in Engineering*, 47(1–3):169–190, jan 2000.

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

[28] Simmetrix: Simulation modeling and application suite, 2016. http://www.simmetrix.com.

[29] 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.

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

[31] K.J. Weiler. The radial-edge structure: a topological representation for non-manifold geometric boundary representations. In M.J. Wozny, H.W. McLaughlin, and Jose L. Encarnacao, editors, *Geometric Modeling for CAD Applications: Selected and Expanded Papers from the Ifip Wg 5.2 Working Conference*, pages 3–36. Elsevier Science Ltd., may 1988.

A Mesh Visualization

Paraview is program created by Kitware, Inc. which can visualize meshes and fields on meshes. It is the program of choice for viewing meshes created by the PUMI libraries. PUMI API provides a function *pumi_mesh_write* and if this is executed with mesh type "vtk" for a mesh distributed over two processes:

```
pumi_mesh_write(mesh, "output", "vtk");
```

It would create the files `output0.vtu`, `output1.vtu`, and `output.pvtu`. Opening the `output.pvtu` file in Paraview will show users the mesh.

By default, Paraview will just render the mesh in "Surface" mode. Changing this to "Surface with Edges" will outline each visible element, actually making the decomposition visible.

Also, the mesh by default is rendered in one "Solid Color". There should be other options corresponding to the fields and numberings that were on this mesh at the time of file writing. There is usually an "apf_part" alternative for files written by APF, which allows users to see the parallel partitioning of the mesh in color.

When vertices are numbered, it may be useful to display their numbers. See Section 10.1.8 for information on creating numberings. Right above the mesh viewing area there is a button to select nodes, you may also press the "D" key.

Click and drag to select all the nodes you want to display. Then go to `View->Selection Display Inspector` in the menu and click on the Point Labels options. There you can choose what to display. If you used `numbering` properly, there should be an option with the same name that you gave to the numbering. Note that in some later versions of Paraview, there is a bug which displays all values as floating point numbers by default. If you are trying to show `numbering` values, you may see strange scientific notation instead. Click the following icon in the Selection Display Inspector:

There you will find a format string option, which you can change to "`%d`" in order to show integers.