PUMI: Parallel Unstructured Mesh Infrastructure

Daniel A. Ibanez, Rensselaer Polytechnic Institute
E. Seegyoung Seol, Rensselaer Polytechnic Institute
Cameron W. Smith, Rensselaer Polytechnic Institute
Mark S. Shephard, Rensselaer Polytechnic Institute

The Parallel Unstructured Mesh Infrastructure (PUMI) is designed to support the representation of, and operations on, unstructured meshes as needed for the execution of mesh-based simulations on massively parallel computers. In PUMI, the mesh representation is *complete* in the sense of being able to provide any adjacency of mesh entities of multiple topologies in O(1) time, and *fully distributed* to support relationships of mesh entities on across multiple memory spaces in a manner consistent with supporting massively parallel simulation workflows. PUMI's mesh maintains links to the high-level model definition in terms of a model topology as produced by CAD systems, and is specifically designed to efficiently support evolving meshes as required for mesh generation and adaptation. To support the needs of parallel unstructured mesh simulations, PUMI also supports a specific set of services such as the migration of mesh entities between parts while maintaining the mesh adjacencies, maintaining read-only mesh entity copies from neighboring parts (ghosting), repartitioning parts as the mesh evolves, and dynamic mesh load balancing.

Here we present the overall design, software structures, example programs and performance results. The effectiveness of PUMI is demonstrated by its applications to massively parallel adaptive simulation workflows.

CCS Concepts:•**Mathematics of computing** → *Discretization; Partial differential equations;*•**Computing methodologies** → **Massively parallel and high-performance simulations;**•**Software and its engineering** → **Massively parallel systems;**

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: unstructured mesh, partial differential equation simulation, hybrid MPI/thread, massively parallel

ACM Reference Format:
Daniel A. Ibanez, E. Seegyoung Seol, Cameron W. Smith, and Mark S. Shephard, 2014. PUMI: Parallel Unstructured Mesh Infrastructure. *ACM Trans. Math. Softw.* V, N, Article 0 (2015), 28 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Many areas of application in science and engineering benefit greatly by the application of reliable and accurate mesh based simulations solving appropriate sets of partial differential equations (PDE's) over general domains. Unstructured mesh finite volume and finite element methods have the advantage of being able to solve problems over geometrically complex physical domains using meshes that can be automatically generated, and anisotropically adapted, and to effectively provide the level of solution accuracy desired with two or more orders of magnitude fewer unknowns than uniform

This work is supported by the Department of Energy (DOE) office of Science's Scientific Discovery through Advanced Computing (SciDAC) institute as part of Frameworks, Algorithms, and Scalable Technologies for Mathematics (FASTMath) program under grant DE-SC0006617.
Author's addresses: D.A. Ibanez, E.S. Seol, C.W. Smith and M.S. Shephard, Scientific Computation Research Center, Rensselaer Polytechnic Institute, 110 8th Street, Troy, NY 12180, USA.

DOI: http://dx.doi.org/10.1145/0000000.0000000

mesh techniques. However, the gains in efficiency (many fewer unknowns) and generality (fully automatic mesh generation and control) come at the cost of more complex data structures and algorithms, particularly when considering meshes with billions of entities solved on massively parallel computers. Although the calculation time is dominated by solving the discretized equation (the classic analysis code), the total simulation time and expense is dominated by the creation of the spatial discretization (meshes) and linkage of mesh-based simulation data between coupled analyses. For example, a recent study of the needs of nuclear reactor simulations indicated that only 4 percent of the time was spent in running the simulation while the execution of geometry and meshing issues took 73 percent of the time [Hansen and Owen 2008].

The bottlenecks caused by the generation and control of meshes as well as the interactions between meshes and simulation data only increase when efforts to ensure the simulation reliability, through the application of adaptive control and uncertainty quantification, are applied. Two requirements must be met to eliminate these bottlenecks. The first is full automation of all steps in going from the original problem definition to the final results since any step that is not automated is doomed to be a bottleneck due to the combination of high latency, slow data transfer rate and serial processing that is inevitable when a human is in the loop. The second is that all steps be executed in parallel on the same massively parallel computer that the finite element or finite volume mesh is solved on to avoid the bottlenecks of data transfer through files.

There exist several libraries capable of representing unstructured finite element meshes. STK is a parallel array-based mesh library developed at Sandia National Laboratories with some modifiability but as yet no general adaptive capability [Edwards et al. 2010]. MOAB is another parallel array-based mesh library developed at Argonne National Laboratory which uses a comparatively small memory with limited modifiability [Tautges et al. 2004a]. Work by Celes, Paulino, and Espinha includes a modifiable, adaptive structure in which elements point to their vertices and to adjacent elements, while vertices point to one adjacent element [Celes et al. 2005a]. In collaboration with MOAB, a more modifiable structure called AHF was developed which can be thought of as using the scheme of Celes et al. with additional orientation information between adjacent entities [Dyedov et al. 2014]. Ollivier-Gooch et al. have implemented the serial simplex mesh library GRUMMP, which is capable of adaptivity [GRUMMP Web 2015]. Finally, a predecessor to the libraries presented here is FMDB, a parallel C++ object-based mesh library well suited to general adaptivity [Seol and Shephard 2006a].

Parallel automated adaptive unstructured mesh simulations go from design models directly to fully parallel mesh generation, to a loop of unstructured mesh-based analysis, error estimation and mesh adaptation This paper presents the Parallel Unstructured Mesh Infrastructure (PUMI) that supports parallel automated adaptive unstructured mesh simulations with reliable simulation results. PUMI is being developed as part of the DOE SciDAC FASTMath institute [FASTMath DOE SciDAC Web 2015] to support a full range of operations on adaptively evolving unstructured meshes on massively parallel computers [Seol and Shephard 2006b; Seol et al. 2012; Zhou et al. 2012b; Xie et al. 2014]. When coupled with dynamic load balancing procedures [Zhou et al. 2012a; Boman et al. 2012; Devine et al. 2002], PUMI provides an infrastructure to support parallel automated adaptive simulations as indicated in Figure 1. Functions supported by PUMI include: *(i)* a complete mesh topology, linked back to the original domain definition that ensures the ability to support any mesh-based application including fully automatic mesh generation and mesh adaptation, *(ii)* a partition model to coordinate the interactions and communications of a mesh distributed in parts over the nodes of a parallel computer, *(iii)* utilities to support changing the mesh partitioning

to maintain load balance for various operations such as a posteriori error estimation and mesh-based analysis. As shown in Figure 1, other components required for automated adaptive simulations include a complete domain definition attributed with the required physical attributes (for instance, loads, material properties and boundary conditions) [O'Bara et al. 2002], parallel mesh generation and adaptation, mesh-based analysis, correction indication to drive mesh adaptation, and visualization. A set of the parallel adaptive simulation workflows that have been developed following the structure of Figure 1 includes *(i)* combined FEM/PIC modeling of electromagnetics in particle accelerators [Luo et al. 2011], *(ii)* modeling blood flow in the human arterial system [Zhou et al. 2010b], *(iii)* two-phase simulation of jets [Galimov et al. 2010], and *(iv)* industrial flow problems [Tendulkar et al. 2011; Shephard et al. 2013]. These simulation workflows have been executed on various AMD and Intel clusters, Cray XT5 and XE6 systems, and/or the three generations of IBM Blue Gene systems (L, P and Q). PUMI has recently supported construction of a 92 billion element mesh solved on 3/4 million compute cores [Rasquin et al. 2014].

This paper provides an overview of PUMI, focusing on its design, software structures, examples and applications on massively parallel computers. The fundamental concepts, definitions and design are presented in Section 2. Section 3 and 4 describe the software aspects and example programs, respectively. Section 5 provides the performance and scaling results. Section 6 presents two adaptive simulation applications to produce complete parallel simulation workflows. Finally, Section 7 discusses future directions of this work. For more interested readers, the Online Appendix presents algorithms of the core parallel functionalities of mesh migration and ghosting.

1.1. Basic Notations

V	the model, $V \in \{G, P, M\}$ where G signifies the geometric model, P signifies the partition model, and M signifies the mesh.
$\{V\{V^d\}\}$	a set of topological entities of dimension d in model V, $V \in \{G, P, M\}$. $d = 0$ for vertex, $d = 1$ for edge, $d = 2$ for face, and $d = 3$ for region. For instance, $\{M\{M^2\}\}$ is the set of all the faces in the mesh.
V_i^d	the i^{th} entity of dimension d in model V, $V \in \{G, P, M\}$.
$\{\partial(M_i^d)\}$	a set of mesh entities on the boundary of M_i^d.
$\{M_i^d\{M^q\}\}$	a set of mesh entities of dimension q that are adjacent to M_i^d. For instance, $\{M_3^1\{M^3\}\}$ is a set of mesh regions adjacent to mesh edge M_3^1.
$M_i^d \sqsubset G_j^q$	the geometric classification indicating the unique association of mesh entity M_i^d with geometric model entity G_j^q, $d \leq q$.
$M_i^d \sqsubset P_j^q$	the partition classification indicating the unique association of mesh entity M_i^d with partition model entity P_j^q, $d \leq q$.
$\mathscr{P}[M_i^d]$	a set of part ID(s) where mesh entity M_i^d exists.
$M_i^d@P_j$	mesh entity M_i^d located in part P_j.

2. DEFINITIONS AND DESIGN

The structures used to support the problem definition, the discretization of the model and their interactions are central to mesh-based analysis methods. A geometry-based analysis environment consists of four sets of problem specification information: *the geometric model* which houses the topological and shape description of the domain of the problem, *the attributes* describing the loads, material properties, and boundary conditions on the geometric model needed to define the physical problem, *the mesh* which

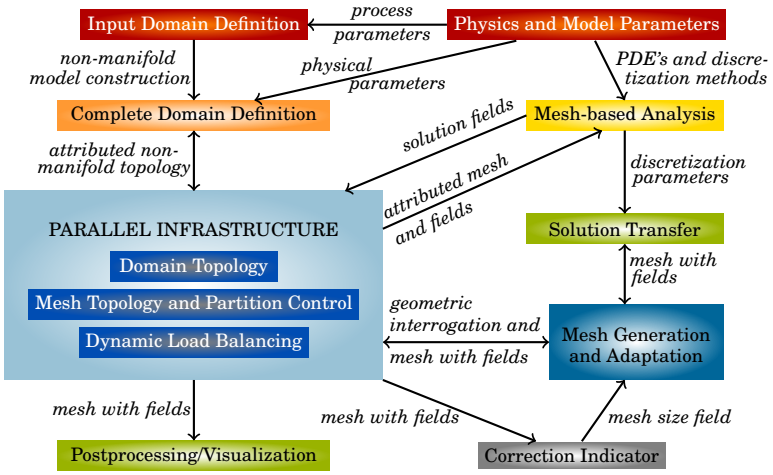


Fig. 1. Simulation infrastructure

describes the discretized representation of the domain used by the analysis method, and *the fields* which describe the distribution of input and solution tensors over the mesh entities [Beall 1999; Simmetrix Web 2015].

These four structures support the information flow between the functional components in a parallel adaptive simulation workflow (Figure 1). The mesh structure is at the core of the workflow since all the functional components of mesh generation, mesh adaptation, mesh-based analysis, correction indication and solution transfer must interact with the mesh. In case of parallel adaptive simulations, the mesh data must meet the following criteria: *(i)* it is distributed in a manner consistent with the needs of the mesh-based analysis, *(ii)* it is flexible enough to support parallel mesh generation and adaptation processes, and *(iii)* it supports the transfer of geometry-based attributes to the mesh to define input fields. As the fields are directly related to the mesh, their parallel distribution is directly related to the distribution of the mesh.

2.1. Geometric Model and Mesh

The most common geometric representation is a boundary representation. A general representation of general non-manifold domains is the Radial Edge Data Structure [Weiler 1988]. Non-manifold models are common in engineering analyses. Simply speaking, non-manifold models consist of general combinations of solids, surfaces, and wires. In the boundary representation, the model is a hierarchy of topological entities of regions, shells, faces, loops, edges, vertices, and in case of non-manifold models, use entities for vertices, edges, loops, and faces are introduced to support the full range of entity adjacencies.

A mesh is a geometric discretization of a domain. With restrictions on the mesh entity topology [Beall and Shephard 1997], the mesh consists of a collection of mesh entities of controlled size, shape, and distribution which are regions (3D), faces (2D), edges (1D) and vertices (0D) [Beall and Shephard 1997; Garimella 2002; Remacle and Shephard 2003; Celes et al. 2005b].

2.2. Classification

Each mesh entity M_i^d maintains a relation, called geometric classification, to a geometric model entity G_j^q that it was created to partially represent. Geometric classification

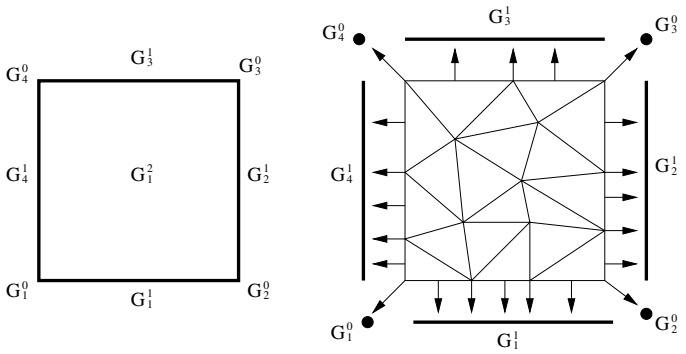


Fig. 2. Example of simple model(left) and mesh(right) showing their association via geometric classification [Beall and Shephard 1997; Beall 1999]

is critical in mesh generation and adaptation [Beall and Shephard 1997; Beall 1999; Shephard 2000].

Definition 2.1. Geometric classification
The unique association of a mesh entity of dimension d, M_i^d, to a geometric model entity of dimension q, G_j^q, on which it lies is termed geometric classification and is denoted $M_i^d \sqsubset G_j^q$, ($d \leq q$). The classification symbol, $\sqsubset$, indicates that the left hand entity, or a set of entities, is classified on the right hand entity.

Figure 2 illustrates a simple square model (left) and the mesh (right). In the right figure, the arrows from mesh entities to model entities indicate their geometric classification. All interior mesh entities without arrows are classified on the model face G_1^2.

Definition 2.2. Reverse geometric classification
For each geometric entity, the set of equal order mesh entities classified on that model entity defines the reverse geometric classification. The reverse geometric classification is denoted as RC(G_j^q) = $\{M_i^q \mid M_i^q \sqsubset G_j^q\}$.

Reverse geometric classification provides an understanding of which attributes (for instance, boundary conditions or material properties) are related to the mesh entities and how the solution relates back to the original problem description.

Since the amount of information defining the geometric model and attributes is small compared to the other data, they are usually fully represented on each node/-core. However, as the level of geometric model complexity increases, it becomes more important to consider methods to distribute the geometric model in parallel. Since the mesh is the key parallel structure, one approach being taken to distribute the geometric model is to have a copy of a model entity and its adjacencies represented on each part which has mesh entities classified on it [Tendulkar et al. 2011].

2.3. Adjacencies

The relationships of the entities are well described by topological adjacencies. For a mesh entity of dimension d, a 1^{st}-order adjacency returns all of the 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 ($d<q$). The notation $\{M_i^d\{M^q\}\}$ indicates a set of entities of dimension q in mesh that are adjacent to M_i^d. Ordering conventions can be used to enforce the specific downward 1^{st}-order adjacent

entity. The notation $M_i^d\{M^q\}_j$ indicates the j^{th} entity in the set of mesh entities of dimension q that are adjacent to mesh entity M_i^d [Beall and Shephard 1997; Beall 1999].

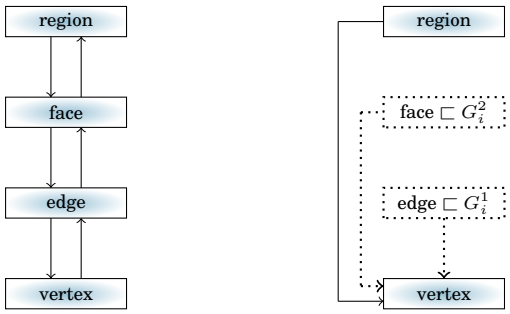
2.4. Mesh Representation

Important factors in designing a mesh data structure are *storage* and *computational efficiency*, which are mainly dominated by the entities and adjacencies present in the mesh representation. The analysis of mesh data structures of various representations suggests how the mesh representation option and intelligent mesh algorithms are important to achieve efficiency with mesh applications [Beall and Shephard 1997; Garimella 2002; Seol 2005; Ollivier-Gooch et al. 2010]. Depending on the dimensions of entities and adjacencies explicitly stored in the mesh, the mesh representation can be categorized with two criteria - *(i)* full vs. reduced - if all 0 to d dimensional entities are explicitly stored, the mesh representation is *full*, otherwise, it is *reduced (ii)* complete vs. incomplete - if all adjacency information is obtainable in O(1) such that the number of operations is bounded by the constant, the representation is *complete*, otherwise, it is *incomplete*.

Provided sufficient links are stored, the number of mesh entities on the boundary of another entity is a known small constant. Therefore computing downward adjacencies is O(1). However, in the most relaxed definition of a mesh, the size of upward adjacency can grow arbitrarily. A simple example is a disk that is meshed with a fan of triangles sharing the center point. Note that as the number of elements around a vertex increases, their average shape quality must decrease. Therefore, with finite element meshes, there is a constant upper bound on the number of upward adjacent entities. If the physical analysis demands a lower bound on element shape quality, a corresponding upper bound exists on upward adjacencies.

A *general* topology-based mesh data structure must satisfy completeness of adjacencies to support adaptive analysis efficiently. It doesn't necessarily mean that all 0 to d dimensional entities and adjacencies need be explicitly stored in the representation. There are many representation options in the design of general topology-based mesh data structure. Figure 3 illustrates two complete representations - *one-level* (full) [Beall and Shephard 1997] and *minimum sufficient* (reduced) [Remacle and Shephard 2003]. A solid box and a solid arrow denote, respectively, explicitly stored dimension of entities and explicitly stored adjacencies from outgoing dimension to incoming dimension. A dotted box denotes that among entities of the dimension, only equally classified ones are explicitly stored, and a dotted arrow denotes that adjacencies from an outgoing dimension to an incoming dimension are maintained only for the stored entities. Note that completeness for reduced representations can be defined as having O(1) retrieval time of adjacencies for both implicit (unrepresented) and explicit (represented) entities. Seol and Shephard presented the detailed discussions on how to manipulate the implicit entities for adjacencies and migration as well as the performance comparison of the one-level representation and the minimum sufficient representation in distributed meshes [Seol 2005; Shephard and Seol 2009].

PUMI stores the one-level representation that is full and complete, so it maintains adjacencies between entities one dimension apart [Beall and Shephard 1997]. A complete mesh representation is necessary to be able to carry out fast local modifications and diffusive load balancing. Using a full representation greatly simplifies the algorithms involved in mesh modification, which make use of the full mapping from mesh topology to geometric topology to preserve the mesh boundary and other properties.



(a) one-level representation (b) minimum sufficient representation

Fig. 3. Two complete mesh representations [Beall and Shephard 1997; Remacle and Shephard 2003]

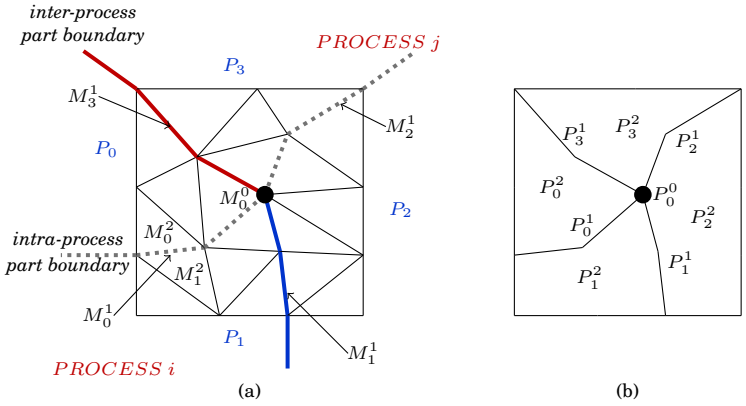


Fig. 4. (a) Distributed mesh on two processes with two parts per process. The thick solid line is inter-process part boundary and the thick dotted line is intra-process part boundary (b) Partition model

2.5. Distributed Mesh

A *distributed* mesh is a mesh divided into *parts* for distribution over a set of processes for specific reasons, for example, parallel computation. A *part* consists of a set of mesh entities that is assigned to a process. Therefore, each part can have *(i)* a unique global part ID within an entire system and *(ii)* a local part ID within a process. *Part boundaries* describe groups of mesh entities that are on inter-part boundaries. With the addition of part boundaries, each part can be treated as a serial mesh. In PUMI, mesh entities on part boundaries (shortly *part boundary entities*) 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 are called *internal entities*.

Figure 4(a) illustrates a distributed mesh on two processes where the mesh on each process has two parts. The dotted lines are *intra-process part boundaries* within a process and thick solid lines are *inter-process part boundaries* between the processes.

Residence part set operator $\mathscr{P}[M_i^d]$ returns a set of global part ID(s) where mesh entity M_i^d exists. For instance, $\mathscr{P}[M_0^0]$ is $\{P_0, P_1, P_2, P_3\}$ and $\mathscr{P}[M_0^1]$ is $\{P_0, P_1\}$. The

two parts, P_i and P_j, *neighbor* over entity type [1] d if they share d-dimensional mesh entities on part boundary.

Definition 2.3. Residence part set equation of M_i^d
For a mesh entity M_i^d existing in part p, if $\{M_i^d\{M^q\}\} = \emptyset$ (d<q), $\mathscr{P}[M_i^d]$ = $\{p\}$. If $\{M_i^d\{M^q\}\} \neq \emptyset$ (d<q), $\mathscr{P}[M_i^d]$ = $\cup\, \mathscr{P}[M_j^q \mid M_i^d \in \{\partial(M_j^q)\}]$.

The residence part set of mesh entity M_i^d is the union of residence part sets of all entities that it bounds. For a mesh topology where the entities of dimension d are bounded by entities of dimension d-1 (d>0), if a mesh entity M_i^d in part p is not bounded by any upward adjacent entity such that $\{M_i^d\{M^{d+1}\}\} = \emptyset$, $\mathscr{P}[M_i^d]$ is determined to be {p}. If a mesh entity M_i^d is bounded by upward adjacent entities such that $\{M_i^d\{M^{d+1}\}\} \neq \emptyset$, $\mathscr{P}[M_i^d]$ is $\cup\, \mathscr{P}[M_j^{d+1} \mid M_i^d \in \{\partial(M_j^{d+1})\}]$. For instance, in Figure 4(a), $\mathscr{P}[M_0^1]$ = $\mathscr{P}[M_0^2]$ $\cup\, \mathscr{P}[M_1^2] = \{P_0\} \cup \{P_1\}$ = $\{P_0, P_1\}$.

2.6. Partition Model

When a mesh is distributed into parts, each part can be viewed as an abstract unit of domain decomposition by partitioning. Putting all abstract units together, we get a conceptual model that represents partitioning, which is another kind of domain decomposition. For the distributed mesh in Figure 4(a), Figure 4(b) illustrates its conceptual model that represents partitioning with 4 abstract faces of domain decomposition. Such a conceptual model exists between the mesh and the geometric model as a part of hierarchical domain decomposition and is termed *partition model*. The two purposes of the partition model are: *(i)* representing mesh partitioning in topology and *(ii)* supporting mesh-level parallel operations through inter-part boundary links.

In the parallel extension of unstructured mesh representation, the partition model consists of a set of topological entities that represent collections of mesh entities based on their partitioning. Therefore, the standard mesh entities and adjacencies on each process can be used with only the addition of the partition entity information needed to support all operations across multiple processes.

Grouping mesh entities to define a partition model entity can be done with multiple criteria based on the functionalities and needs of distributed meshes. At a minimum, mesh entities should be grouped by residence part set to support a inter-part communication. They could further be grouped by adjacencies within the same part, which would diagnose partitioning problems (i.e. a large number of disconnected components), and inform algorithms that deal with connected components only. In PUMI, for efficiency, only the residence part set is used for this grouping.

Definition 2.4. Partition (model) entity
A topological entity in the partition model, P_i^d, represents a group of mesh entities of dimension d and their downward adjacent entities, that have the same residence part set. Each partition model entity is uniquely determined by the residence part set.

The partition model in Figure 4(b) consists of *(i)* partition vertex P_0^0 representing mesh vertex duplicated on all four parts, *(ii)* partition edges P_0^1 representing mesh vertices and edges duplicated on P_0 and P_1, P_1^1 representing mesh vertices and edges duplicated on P_1 and P_2, P_2^1 representing mesh vertices and edges duplicated on P_2 and P_3, P_3^1 representing mesh vertices and edges duplicated on P_0 and P_3, *(iii)* partition faces P_i^2 representing internal mesh vertices, edges and faces on part P_i, $0 \leq i \leq 3$.

[1] dimension and type are interchangeable

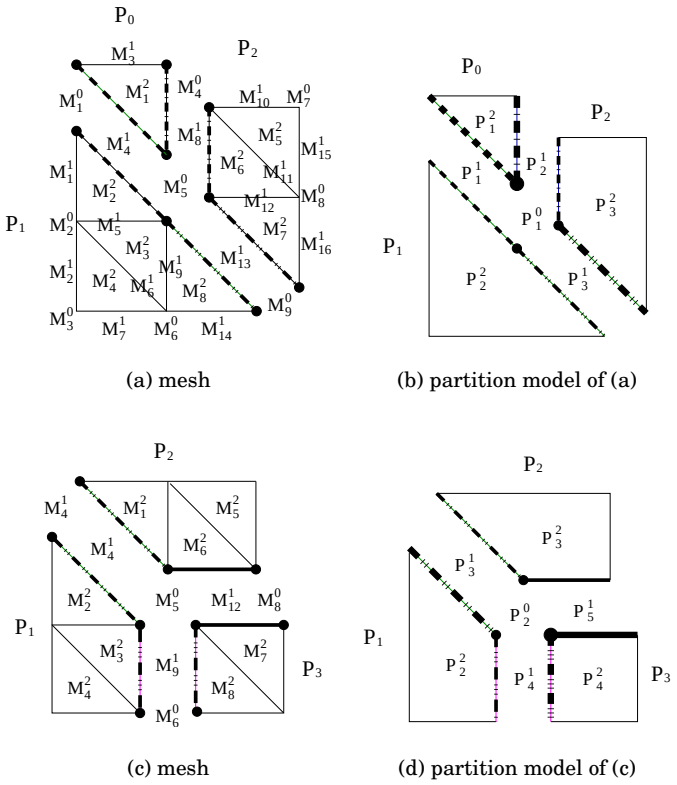


Fig. 5. Distributed mesh and owning part stored in partition model entities based on poor-to-rich rule [Seol 2005; Seol and Shephard 2006b]

Definition 2.5. Partition classification
The unique association of mesh entity of dimension d, M_i^d, to partition model entity of dimension q, P_j^q, on which it lies is termed partition classification and is denoted $M_i^d \sqsubset P_j^q$, $(d \leq q)$.

Each partition model entity stores dimension, residence part set and owning part ID, and each mesh entity keeps a pointer to the partition model entity based on the partitioning. Therefore at the mesh entity level, the residence part set and owning part information are maintained through partition classification. In Figure 4, mesh vertex M_0^0 is classified on partition vertex P_0^0, mesh edges M_i^1 are classified on partition edges P_i^1, $0 \leq i \leq 3$. Internal mesh entities (vertices, edges and faces) on P_i are classified on partition face P_i^2, $0 \leq i \leq 3$.

2.7. Mesh Migration

In adaptive simulations on a distributed mesh, the mesh entities need to be migrated from part to part frequently in order to facilitate the local mesh modification and computations, or to re-gain mesh load balance for other steps in the simulation workflow. Figure 5 illustrates an example of mesh migration. Figure 5(a) and (b) are the initial mesh and partition model and Figure 5(c) and (d) are the mesh and partition model after migrating mesh faces M_5^2 and M_6^2 from part P_2 to part P_0 and M_8^2 from part P_1 to part P_2.

To initiate the migration of mesh entities, the destination part ID's of mesh entities must be determined. The residence part set equation dictates that if a destination part ID of mesh entity M_i^d, which is not on the boundary of any other mesh entities, is determined, its downward adjacent entities, $\{M_i^d\{M^q\}\}$ ($d>q$), should be migrated to the same destination part as well. Thus, a mesh entity that is not on the boundary of any upward adjacent entities is the basic unit to assign the destination part ID in the migration procedure.

Definition 2.6. Partition object
The basic unit to assign a destination part ID in migration. A mesh entity not being on the boundary of any upward adjacent entities (termed *mesh element*) can be a partition object. In a 3D mesh, partition objects are all mesh regions, the mesh faces not bounding any mesh regions, the mesh edges not bounding any mesh faces or regions, and mesh vertices not bounding any mesh edges, faces or regions. An ordered set of unique mesh elements confined to a part can also be a partition object if designated to be migrated as a unit.

An ordered set of unique mesh elements confined to a part is referred as *p-set* [Xie et al. 2014]. A p-set is useful in representing boundary layer stacks [Sahni et al. 2008; Ovcharenko et al. 2013].

In case of a 3D (resp. 2D) manifold model, partition objects are p-sets consisiting of regions (resp. faces) and mesh regions (resp. faces) not contained in a p-set. In case of a non-manifold model, a lookup for mesh entities not bounding any upward adjacent entities is required for each dimension d, $0\leq d\leq 3$. In Figure 4(a), partition objects are all mesh faces.

If a mesh entity is duplicated, it must be aware of where it is duplicated. Among multiple duplicate copies, an entity in a specific part is assigned as the owner with charge of modification, communication or computation of the copies. For the purpose of simple denotation, the entity on the owning part is called the *owner* of all its copies. Note that the owning part ID for part boundary entities is maintained at the partition model entity and the mesh entity's owning part is retrieved through partition classification. In PUMI, the owning part is determined based on *poor-to-rich rule* - the owning part is a part with the least number of partition object mesh entities among all residence parts [Seol 2005; Seol and Shephard 2006b]. And if residence parts have the same least number of partition object entities, the part with smaller ID is the owning part. In Figure 5, the bigger circle and thicker lines denote the owning part of partition model entities. For instance, the owner of mesh entity M_5^0 in Figure 5(a) (resp. (c)) is $M_5^0@P_0$ (resp. $M_5^0@P_3$). The owner of M_4^1 in Figure 5(a) (resp. (c)) is $M_4^1@P_0$ (resp. $M_4^1@P_1$).

2.8. Mesh Ghosting

For the support for specific mesh operations requiring data from mesh entities on remote processes, the ghosting procedure localizes internal mesh entities and their member data on neighboring parts for the purpose of minimizing inter-process communications [Lawlor et al. 2006; iMeshP Web 2015]. The inputs to the ghosting procedure are *(i)* ghost type g ($0<g\leq$*mesh dimension*), which is an entity type to be ghosted, *(ii)* bridge type b ($0\leq b<g$ and $b\neq g$), which is an entity type to be used to compute entities to be ghosted based on adjacency, *(iii)* the number of ghost layers n measured from inter-part boundary up to the number with which whole part can be ghosted, *(iv)* an integer *inc_copy* to indicate whether to include non-owned bridge type entities (1) or not (0) in computating entities to be ghosted [iMeshP Web 2015].

Given input parameters, the ghosting procedure first iterates all part boundary entities of type b and collect entities to be ghosted (shortly, ghosting candidates) as the following: Note that a ghosting candidate is created in a neighboring residence part of bridge entity only if the ghost candidate doesn't exist in the part as a remote copy or ghost copy.

(1) if $n = 1$, for each part boundary entity of type b, M_i^b, ghosting candidates are upward adjacent entities of type g, $\{M_i^b\{M^g\}\}$. If the input *inc_copy* is 0 and M_i^b is not the owner copy, M_i^b is not considered for the ghosting candidate computation. If the input *inc_copy* is 1, M_i^b is considered for the ghosting candidate computation no matter it is the owner copy or not.
(2) if $n>1$, for each mesh entity M_j^g in ghosting candidates of $n-1$ layer ghosting, ghosting candidates are a set of entities $\{M_j^g\{M^b\}\{M^g\}\}$. The notation $\{M_j^g\{M^b\}\}$ denotes a set of mesh entities of type b adjacent to mesh entity M_j^g. The notation $\{M_j^g\{M^b\}\{M^g\}\}$ denotes a 2^{nd}-order adjacency meaning a set of mesh entities of type g adjacent to mesh entity M_i^b, where $M_i^b \in \{M_j^g\{M^b\}\}$.

As parallel finite element analysis (FEA) needs remote vertices and finite volume method (FVM) needs remote regions for analysis, the ghosting procedure is an essential functionality for analysis. In case of a 3D analysis, FEA may use 1-layer ghosting with bridge type 0 (vertex) and ghost type 3 (region). Low-order FVM may use 1-layer ghosting with bridge type 2 (face) and ghost type 3 (region). High-order FVM may use 2 or more layer ghosting with bridge type 2 (face) and ghost type 3 (region).

2.9. Graph-based Mesh Partitioning

Graph-based n-to-m mesh partitioning re-distributes n-part mesh into m parts ($n \neq m$, $n<m$). The procedure consists of four steps: *(i)* transforming the unstructured mesh data structures to the graph vertex and edge structures needed by Zoltan [Boman et al. 2012; Devine et al. 2002], *(ii)* running Zoltan, *(iii)* transforming the output of Zoltan (graph vertices and destination part ID's) to STL maps of partition object (mesh element and/or p-set) and part ID required by the migration procedure, and *(iv)* running the migration. Graph vertices are defined by partition objects (mesh elements and/or p-sets), and graph edges by the mesh faces (3D) or edges (2D) shared by adjacent elements and p-sets, either through topology or through periodicity. This graph definition supports the natural division of the mesh into non-overlapping parts P_i of dimension d such that $M = \bigcup P_i$. In a 3D mesh, this uniquely assigns each mesh region to a single part. Creating graph vertices from p-sets conforms to this approach and ensures that user-defined groups of mesh elements will be accessible within the same part after partitioning. Zoltan's interface requires that each graph vertex is defined by a unique user-defined object. Likewise a graph edge is defined by two graph vertex objects. The most efficient definition of the graph vertex object is an integer [Boman et al. 2012; Devine et al. 2002].

3. SOFTWARE STRUCTURE

PUMI consists of six software components (Figure 6) [Seol et al. 2012; PUMI Web 2015]: *(i)* the Common Utility for common tools and services used in multiple other components, *(ii)* the Parallel Control component for parallel-specific tools and services, *(iii)* the Geometric Model component for interfacing with geometric model kernels, *(iv)* the Mesh component to provide the distributed mesh representation and manipulations, *(v)* the Partition Model component for partition model representation and manipulations, and *(vi)* the Field component for field storage and manipulations.

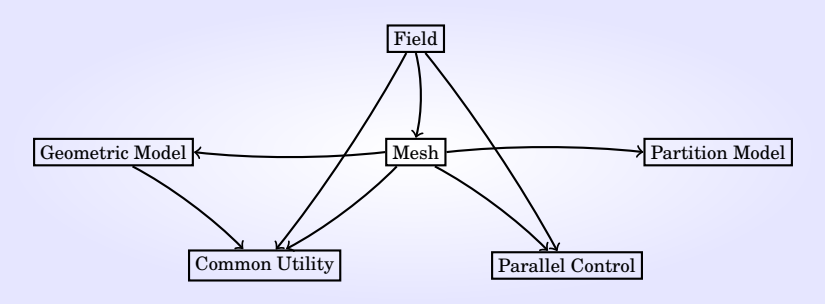


Fig. 6. PUMI components: an arrow from component A to component B indicates that the component A is dependent on the component B

As a partition model is constructed based on the mesh distribution, it is automatically updated when mesh partitioning is changed. Therefore its functionality is embedded in mesh implementation and furthermore, access/modification by the users is not needed. Except for the Partition Model component, each component is a self-contained, independently usable library with its own set of requirements and well-defined API.

3.1. pcu - Parallel Control Utility Library

pcu is a library that provides a parallel programming model including parallel control functions. Its two major functionalities are *message passing* that allows parallel tasks to coordinate and *thread management* that extends the MPI programming model into a hybrid MPI/thread system.

The foundation of *pcu* is its point-to-point message passing routines where non-blocking synchronous message passing primitives are defined. There are two versions, one of which is a direct interface to MPI, and the second supports message passing between threads [Ibanez et al. 2014]. The two versions are interchangeable, and *pcu* can change which set of them is being used at run-time without affecting the rest of software components.

Building on the point-to-point primitives, *pcu* has an extensible framework for collective operations such as reduction, broadcast, scan, and barrier. Any collective whose communication pattern can be encoded as some kind of tree is supported, and the most common ones come built-in to *pcu*. These collectives are directly available to users.

Using both point-to-point and collective communication, *pcu* provides a message passing algorithm for general unstructured communication. The *send* phase allows tasks to send any messages out to neighbors and the *receive* phase ensures that neighbors receive all the messages they have been sent. This *phased* communication algorithm is equivalent to the non-blocking consensus algorithm for sparse data exchange [Hoefler et al. 2010].

Finally, *pcu* has a system for creating a pool of threads within each process and assigning them ranks in a way that MPI does to processes. Users can call this API to enter a hybrid MPI/thread mode in which all the communication APIs (point-to-point, collective, and phased) work between threads. These capabilities support a hybrid MPI/thread operation.

3.2. gmi - Geometric Model Interface Library

The problem domain is the basis for generating meshes on which the analysis is performed. Commercial geometric modeling kernels provide the description of the problem domain.

The geometric model library provides modeling kernel-independent geometry access by polymorphism and replicating the topological information in modeling kernels. Therefore geometry-based applications (including mesh) can interface with various modeling kernels through geometric model library [Panthaki et al. 1997; Tautges 2001; Beall et al. 2004]. In *gmi*, the class hierarchy of geometric model is derived for implementation with specific commercial geometric modeling kernels such as Acis [ACIS Web 2015], GeomSim [GeomSim Web 2015], and Parasolid [Parasolid Web 2015]. In case of no commercial modeling kernel available, *gmi* supports a geometric model constructed from mesh, called *mesh model* [Beall et al. 2004]. The following core functionalities are provided at the high-level API regardless of underlying modeling kernel.

- *modeling kernel registration* - establishing the relationship between the high-level API and the modeling kernel specific API and importing the geometric model information into topological representation.
- *geometric model representation* - maintaining pointers to topological model entities. In boundary representation, they are regions, shells, faces, loops, edges, vertices and *use* entities for vertices, edges, loops, and faces with a non-manifold model [Weiler 1988]. The information stored in these data structures provide topological definition of the geometric model, so the mesh structure can always be correctly classified and the topological similarity between the mesh and the model can be maintained during modifications.
- *interrogations* - adjacency, tolerance, and shape information, etc. Model entity objects themselves contain boundary-representation adjacency structures to other model entities as well as declaring virtual methods for modeler queries such as evaluating a point of a parametric surface [Beall et al. 2004].

Recently, models with thousands to millions of model entities are being constructed in *gmi*, which prompted the addition of fast lookup structure to the model object since retrieval of a model entity from its integer identifier is a common operation during message passing and file reading.

3.3. mds - Mesh Data Structure Library

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 parts, *(ii)* migration of entities or group of entities between parts, *(iii)* dynamic load balancing. *mds* provides the storage and management of distributed unstructured meshes and partition model. It supports all the mesh-level services to interrogate/modify the mesh data needed by parallel adaptive analysis. Core *mds* functionalities include:

- *interrogations* - 1^{st} and 2^{nd}-order adjacency, owning part ID, status (internal, part boundary, matched, ghost or ghosted), classification (geometric and partition model), etc.
- *modification* - entity and entity set [2] creation/deletion, migrating entity and p-set from part to part including tagged data, and a capability to dynamically change the number of parts per process.
- *load-balancing* - a capability to balance the mesh load on each part predictively [Flaherty et al. 1997; Zhou et al. 2012b] or as post-processing with the help of

[2]grouping of arbitrary entities from multiple parts or a single part [Tautges et al. 2004b; Ollivier-Gooch et al. 2010; ITAPS Web 2015].

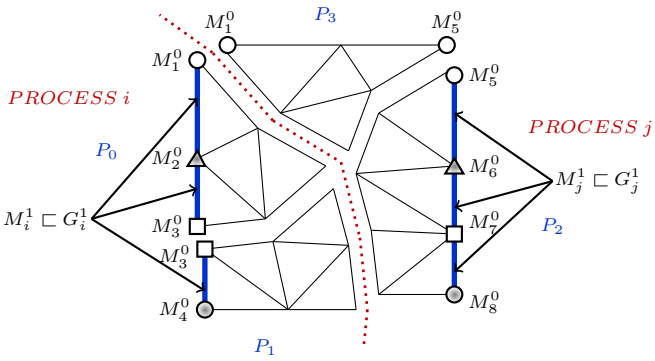


Fig. 7. 2D distributed mesh with periodic model edges G_i^1 and G_j^1

mesh migration and partitioning libraries such as Zoltan, ParMETIS, and ParMA. Zoltan is a toolkit for scientific applications that provides graph-based load balancing and partitioning algorithms [Boman et al. 2012; Devine et al. 2002]. ParMETIS provides algorithms for partitioning unstructured graphs, meshes, and for computing fill-reducing orderings of sparse matrices [Schloegel et al. 2002]. ParMA is a partitioning library developed at SCOREC without using the classical graph data structure. Three ParMA procedures are *(i)* multi-criteria diffusive partition improvement [Zhou et al. 2010a; Zhou et al. 2012a], *(ii)* predicted load balancing [Flaherty et al. 1997; Zhou et al. 2012b], and *(iii)* heavy part splitting [Zhou et al. 2012b; Seol et al. 2012].

- *matching* - a capability to maintain mesh entities on matched, periodic model boundaries [Karanam et al. 2008; MeshSim Web 2015]. Figure 7 illustrates four-part distributed mesh with periodic model edges G_i^1 and G_j^1 (thick solid lines). The mesh entities classified on G_i^1 are matched to mesh entities classified on G_j^1, and vice versa. For instance, $M_1^0@P_0$ is matched to three mesh vertices, $M_1^0@P_3$, $M_5^0@P_3$, and $M_5^0@P_2$. $M_3^0@P_0$ is matched to two mesh vertices, $M_3^0@P_1$, and $M_7^0@P_2$. When a mesh entity is modified or migrated, all the matched copies should be updated as well to keep the identity.
- *file I/O* - ascii or binary file I/O in various mesh formats (NetCDF [NetCDF Web 2015], Exodus [Schoof and Yarberry 1994], VTK [VTK Web 2015], Simmetrix [Simmetrix Web 2015], etc.) for partition model and mesh entities with auxiliary data including geometric classification, partition classification, tagged data, entity set, matched copies, etc.

The term *instance* is used to indicate an object of model data existing on each process. For example, a mesh instance on a process means a pointer to a mesh data structure on the process where all parts on the process are maintained by, and accessible through. The term *handle* is used to indicate a pointer to other types of data object such as part, entity and entity set. For example, a mesh entity handle means a pointer to the mesh entity data [Seol and Shephard 2006b; Ollivier-Gooch et al. 2010; Seol et al. 2012; ITAPS Web 2015].

Figure 8 illustrates a mesh and related data in each process. The mesh instance maintains *(i)* one or more part handles, and *(ii)* zero or more entity set handles [ITAPS Web 2015]. Each part handle maintains *(i)* a link to a geometric model instance, *(ii)* a link to a partition model instance, *(iii)* zero or more entity handles per dimension (0 to 3), *(iv)* zero or more p-set handles.

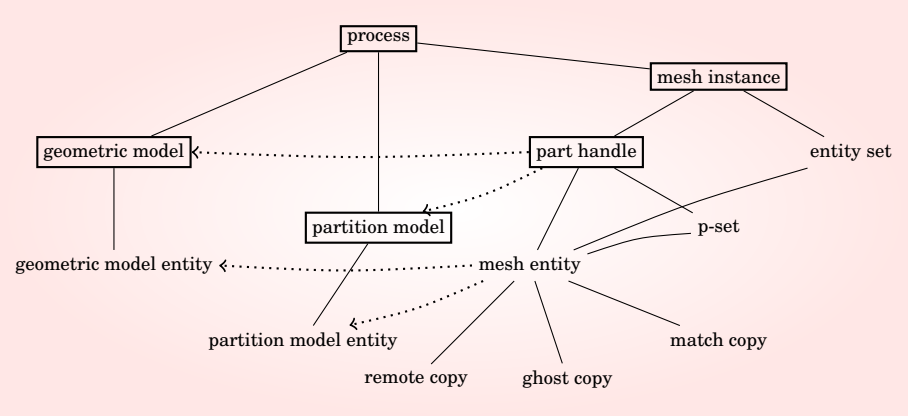


Fig. 8. Mesh data in process: A solid line from upper positioned component U to lower positioned component L indicates that the component U contains 0 or more of component L. A dotted arrow from component A to component B indicates that the component A maintains a link to component B. A data drawn with rectangle indicates that at least one copy of the data should exist on each process.

Each mesh entity maintains the following information:

- *dimension or type* - 0 for vertex, 1 for edge, 2 for face and 3 for region.
- *topology* - for a region, PUMI supports tetrahedron, hexahedron, prism (wedge) and pyramid. For a face, PUMI supports triangle and quadrilateral.
- *geometric classification* - a link to a geometric model entity handle where it's discretized.
- *partition classification* - a link to a partition model entity handle representing residence part set and owning part ID of the mesh entity. When a residence part set of mesh entity is changed, the mesh entity searches a partition model entity that has the same residence part set and updates the partition classification. If a partition model entity with the same residence part set is not found, a new partition model entity is created and the new partition model entity is used to set the partition classification of the mesh entity.
- *adjacency* - i^{th} dimensional mesh entity maintains the links to $(i-1)^{th}$ and $(i+1)^{th}$ dimensional adjacent entities ($0 \leq i \leq 3$) if applicable.
- *remote copy* - a memory location of a part boundary entity on another part. To reduce the communication time, every part boundary entity stores an STL map of *(remote part ID, remote copy)* no matter if it is a master (owner copy) or a slave (non-owner copy). Note that the performance of mesh migration dictates the overall performance of simulation workflow due to its high frequency. In our experiment, the migration time of only owner copy having remote copy information was more than that of every part boundary entity storing remote copy information.
- *ghost copy* - a memory location of duplicate copy of internal entity in neighboring parts [Lawlor et al. 2006; iMeshP Web 2015]. For efficient communications, the original internal entity is designated to be an owner copy and maintains an STL map of *(ghost part ID, ghost copy)*. The ghost copy maintains owning part ID and the memory location of owner copy. Since ghost copies are stored explicitly in the mesh as regular entities, all entity-level functions can be used with ghost copies. For instance, traversal, entity set, tagging [3], and various interrogations such as

[3]attaching a user data of various types (single or array of integer, pointer, floating point, binary, entity set, entity) to a part, entity set or entity [Ollivier-Gooch et al. 2010; Seol et al. 2012; ITAPS Web 2015]

geometric classification, adjacencies, remote copies, residence part set, owning part ID, partition classification, etc.
- *match copy* - a memory location of a matched entity handle. Note that an entity can have one or more match copies per part. Therefore, when a mesh entity is matched to other entity, it maintains an STL multimap of *(match part ID, match copy)*. If an entity is matched to a part boundary entity, it must be matched to all the remote copies.

In less flexible applications, it would be possible to globally number the mesh entities such that duplicate copies have the same integer ID. Then, each part could store its neighbors and entities would store nothing but their global number. It is then possible to send data across parts along with a global number, and search for the corresponding number after receiving the data. However, supporting this search is expensive, and more importantly, a global number would be invalidated by mesh modification unless expensive methods to maintain unique labeling of entities being created across the cores of the parallel system were used. Therefore, the remote copies are carefully tracked pointers as opposed to a global numbering.

Even with this requirement, there are three options as to how the remote copies could be stored: *(i)* all duplicate copies store the remote copy information, *(ii)* each non-owner copy stores a single pointer to its owner copy and the owner copy stores nothing, and *(iii)* only owner copy stores the remote copy information. In case of the option 2 (resp. 3), an algorithm that broadcasts data from the owner to non-owners (resp. non-owners to owner), a common operation in finite element codes, would have to use an "echo" system where non-owners (resp. owner) would request the information from the owner (resp. non-owners) first, since the owner (resp. non-owners) does (resp. do) not yet know where to send it. This would double the number of communication rounds, which is worse than the slight increased memory cost of having part boundary entities know all their copies.

For mesh entities migrated to other part(s), new partition classification can be derived from the remote copy information (it provides the new residence part set). However, it can be useful during the migration algorithm to first update the partition classifications based on where mesh entities are migrated, and then actually move the entities and update the remote copies based on the new partition classification. This justifies that storing partition classification per entity benefits the mesh migration.

3.4. apf - Attached Parallel Field and Mesh Interface Library

The Attached Parallel Field library provides the finite element field manipulation. It also provides interfacing between the field and multiple databases including *mds*, MeshSim [Simmetrix Web 2015] and STK [APF Web 2015]. The core functionalities are *(i)* common finite element field operations such as numbering and synchronization over part boundaries, *(ii)* a wide variety of finite element basis functions and numerical integration rules, and *(iii)* array-based field storage for multiple mesh databases, including support for frequent migration and modification of the mesh. The field and numbering data are stored in separate arrays and keyed by the local ID of mesh entities.

3.5. Array-based Data Structure

The previous PUMI implementation was based on an object-oriented data structure [Seol and Shephard 2006b; Seol et al. 2012]. In the recent improvement, the one-level adjacency information, vertex coordinates, simulations fields, and the classification links have been compressed into an array structure. The difficulty involved

in an array structure is preserving fast mesh modification without increasing the runtime of existing applications and workflows.

Although PUMI stores the mesh representation in arrays, it still retains the ability to make modifications at the single-entity level without restrictions. Aside from locality and memory access patterns, the storage of data in arrays has resulted in a significant reduced memory usage with respect to other flexible and scalable mesh implementations. For instance, using array-based data structure, the memory usage of PUMI went down to 25% of the previous data structure. A full description of the array scheme is beyond the scope of this paper and will be published in the near future.

4. EXAMPLES

This section presents three example programs to show how users can query and modify a distributed mesh through the C++ API functions. Note that the PUMI API aims to be abstract over several different mesh implementations so the API uses C-style idioms. For instance, the iterator API does not follow the usual C++ STL iterator convention because not all mesh implementations interoperable with PUMI support separate increment and dereference.

Listing 1 illustrates a program that loads a mesh (Line 15) and identifies every triangle classified on a geometric model face (Lines 23-25). The program establishes a local vertex numbering (Line 16), and prints the three vertex numbers for each triangle classified on a geometric model face (Line 33). It also prints the unique tag associated with the model face (Line 34). Lines 28-29 illustrate face-to-vertex derivation. The while loop (Lines 21-36) can easily be extended to apply boundary conditions to the vertices, or integrate a quantity of interest over the face and accumulate a total integral over the mesh surface.

Listing 2 loads a partitioned mesh (Line 15), establishes a globally unique numbering of the vertices (Line 16), and then prints the inter-part sharing of vertices (Lines 24-31). Coupled with element-to-vertex information, the inter-part sharing information of vertices is sufficient information to construct a partitioning graph. To construct a partitioning graph in reasonable time, Interested readers can find an example program advanced use of sparse data exchange API is needed to avoid local search process based on global numbers. Given geometric model file name, input mesh file name, output mesh file name, and an integer n, Listing 3 loads a mesh (Line 53), partition it to n parts per process (Line 54) using Zoltan, and writes out the resulting mesh into the PUMI format (Line 28).

Although not described in this paper, PUMI provides API's to individually add and remove entities, which would allow interested users to develop their own mesh adaptation codes. The authors provide robust, scalable, state of the art mesh adaptation library, MeshAdapt, which is built on PUMI tools and services [Li et al. 2005; Alauzet et al. 2006; MeshAdapt Web 2015].

5. RESULTS

In the performance studies, the IBM BlueGene/Q at the Center for Computational Innovations [CCI Web 2015] in Rensselaer Polytechnic Institute is used. The CCI BlueGene/Q is a 64-bit supercomputer with 5-rack custom-connected 5,120 nodes reaching LINPACK peak performance 894.4 TFlop/s and theoretical peak 1,048.6 TFlop/s. Each compute node has a 16-core 1.6 GHz PowerPC A2 processor and 16GB DDRS memory.

5.1. Parallel Mesh Construction from Node-Element Information

PUMI provides a function to construct a distributed mesh and partition model using the minimum node-element information for each part. The minimum required information is, for each part, five arrays with global node ID, node coordinates, element's

Listing 1. Boundary triangle example

```
#include <apf.h>
#include <gmi_mesh.h>
#include <apfMDS.h>
#include <apfMesh2.h>
#include <apfNumbering.h>
#include <PCU.h>

using namespace apf;

int main(int argc, char** argv)
{
  MPI_Init(&argc, &argv);
  PCU_Comm_Init();
  gmi_register_mesh();
  Mesh2* mesh = loadMdsMesh("model.dmg", "mesh.smb");
  Numbering* n = numberOverlapNodes(mesh, "ID");
  int faceDimension = 2;
  int vertexDimension = 0;
  MeshIterator* it = mesh->begin(faceDimension);
  MeshEntity* face;
  while ((face = mesh->iterate(it)))
  {
    ModelEntity* me = mesh->toModel(face);
    if ((mesh->getType(face) == Mesh::TRIANGLE) &&
        (mesh->getModelType(me) == faceDimension))
    {
      int modelFaceTag = mesh->getModelTag(me);
      Downward vertices;
      mesh->getDownward(face, vertexDimension, vertices);
      int ids[3];
      for (int i = 0; i < 3; ++i)
        ids[i] = getNumber(n, vertices[i], 0, 0);
      std::cout << "triangle "<< ids[0]<<" "<<ids[1]<<" "<<ids[2];
      std::cout << " is on model face "<< modelFaceTag << '\n';
    }
  }
  mesh->end(it);
  mesh->destroyNative();
  destroyMesh(mesh);
  PCU_Comm_Free();
  MPI_Finalize();
}
```

part ID, element's topology, and consisting nodes' global ID's for each element. Using the node-element information on 256 parts, it took 12 seconds to construct 25.6 million element mesh and partition model.

5.2. Migration

The migration procedure has been designed for per-element speed as well as parallel scalability. One stress test is to have each part migrate 10 thousand elements to the neighboring part. Running this test with an input mesh of 1.6 billion elements in 16 thousand parts takes 12 seconds when using 16 thousand cores of the CCI BlueGene/Q.

Listing 2. Vertex sharing example

```
#include <apf.h>
#include <gmi_mesh.h>
#include <apfMDS.h>
#include <apfMesh2.h>
#include <apfNumbering.h>
#include <PCU.h>

using namespace apf;

int main(int argc, char** argv)
{
  MPI_Init(&argc, &argv);
  PCU_Comm_Init();
  gmi_register_mesh();
  Mesh2* mesh = loadMdsMesh("model.dmg", "mesh.smb");
  GlobalNumbering* gn = makeGlobal(numberOwnedNodes(mesh, "ID"));
  int vertexDimension = 0;
  int self = PCU_Comm_Self();
  MeshIterator* it = mesh->begin(vertexDimension);
  MeshEntity* vertex;
  while ((vertex = mesh->iterate(it)))
  {
    long id = getNumber(gn, Node(vertex, 0));
    Parts ps;
    mesh->getResidence(vertex, ps);
    for (Parts::iterator it = ps.begin(); it != ps.end(); ++it)
    {
      int sharedWith = *it;
      std::cout << "part " << self << " shares vertex ";
      std::cout << id << " with part " << sharedWith << '\n';
    }
  }
  mesh->end(it);
  mesh->destroyNative();
  destroyMesh(mesh);
  PCU_Comm_Free();
  MPI_Finalize();
}
```

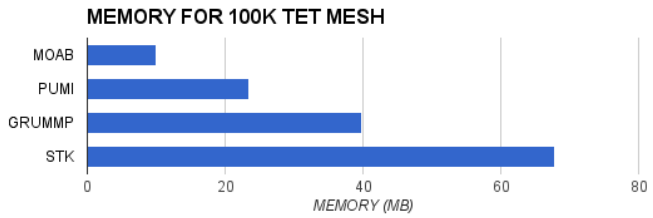


Fig. 9. Memory cost comparison to MOAB, GRUMMP and STK

5.3. Memory Cost

Figure 9 shows the memory used to store an identical 100K [4] element tetrahedral mesh using MOAB, GRUMMP, STK, and PUMI. PUMI uses less memory than other

[4] 1K = 1,024

Listing 3. Mesh partitioning example

```
#include <gmi_mesh.h>
#include <apf.h>
#include <apfMesh2.h>
#include <apfMDS.h>
#include <PCU.h>
#include <apfZoltan.h>
#include <parma.h>

const char* modelFile = 0;
const char* meshFile = 0;
const char* outFile = 0;
int partitionFactor = 1;

apf::Migration* getMigrationPlan(apf::Mesh* m)
{
  apf::Splitter* splitter = apf::makeZoltanSplitter(
      m, apf::GRAPH, apf::PARTITION, false);
  apf::MeshTag* weights = Parma_WeighByMemory(m);
  apf::Migration* plan = splitter->split(weights, 1.05, partitionFactor);
  apf::removeTagFromDimension(m, weights, m->getDimension());
  m->destroyTag(weights);
  delete splitter;
  return plan;
}

void cleanup(apf::Mesh2* m)
{
  m->writeNative(outFile);
  m->destroyNative();
  apf::destroyMesh(m);
}

void getConfig(int argc, char** argv)
{
  if (argc != 5 ) {
    if (!PCU_Comm_Self())
      printf("Usage: %s <model> <mesh> <outMesh> <factor>\n", argv[0]);
    MPI_Finalize();
    exit(EXIT_FAILURE);
  }
  modelFile = argv[1]; meshFile = argv[2];  outFile = argv[3];
  partitionFactor = atoi(argv[4]);
}

int main(int argc, char** argv)
{
  int provided;
  MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
  assert(provided==MPI_THREAD_MULTIPLE);
  PCU_Comm_Init();
  gmi_register_mesh();
  getConfig(argc, argv);
  apf::Mesh2* m = apf::loadMdsMesh(modelFile, meshFile);
  splitMdsMesh(m, getMigrationPlan(m), partitionFactor, cleanup);
  PCU_Comm_Free();
  MPI_Finalize();
}
```

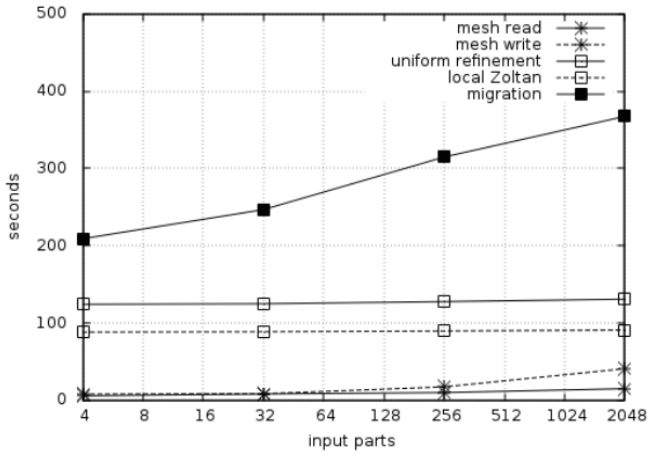


Fig. 10. Scaling results with threads up to 1.6 billion elements and 16K parts

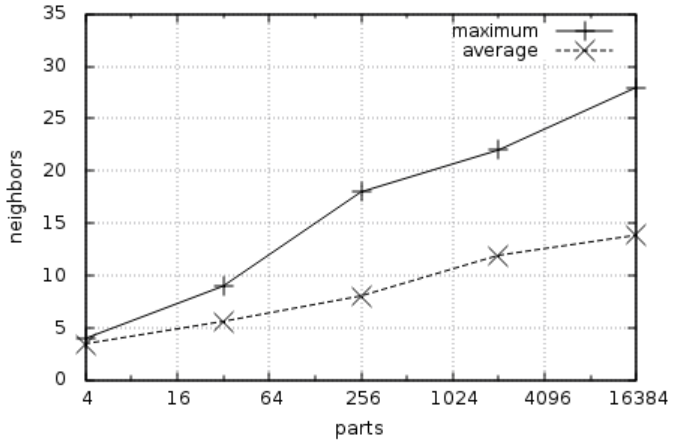


Fig. 11. Part neighborhood size increase during scaling

modifiable structures, sometimes up to 3 times less. Furthermore, PUMI uses only 2 times more memory than the simplest non-modifiable representation.

5.4. Full Stack Scaling

In order to test the capability of the hybrid MPI-thread system, *pcu*, an 1.6 billion element mesh is created using up to 16K cores of the Blue Gene/Q. Mesh generation begins with a 4-part, 400K element tetrahedral mesh and proceeds so as to maintain a part density of 100K elements per part. Each up-scaling repartitioning step begins with uniform mesh refinement, which multiplies the element count by 8. Following that, the PUMI-Zoltan interface is applied locally to each part, splitting it into 8 new parts. For each part, 7 new threads are created, and inter-thread migration is used to distribute the elements among the 8 threads according to the Zoltan output. During each step, we start with 2 processes per node, which results in 16 threads per node or one thread per core at the end. Figure 10 shows the time consumed by each step of this

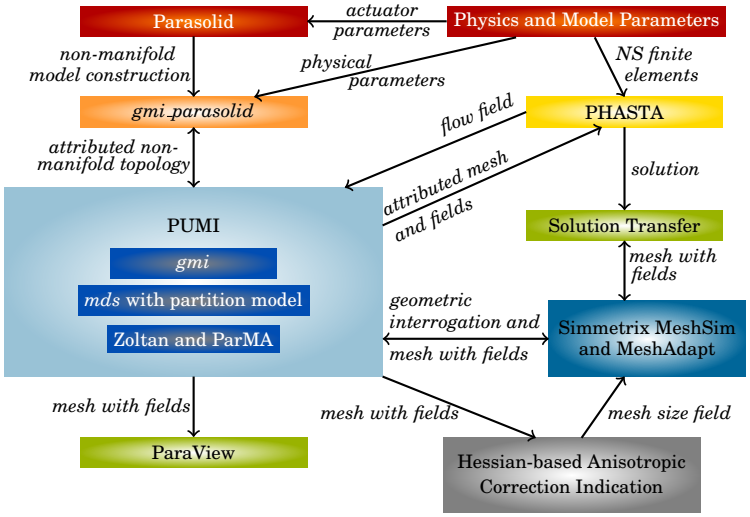


Fig. 12. Workflow of parallel PHASTA adaptive loop

workflow as the number of parts is increased from 4 to 16K. The final step converts 2K parts into 16K parts.

Note that the increase in migration time in this scaling study can most likely be attributed to the fact that as the mesh is further refined and partitioned, the average and maximum number of parts adjacent to each part increases. This is illustrated in Figure 11. This in turn affects the number of messages that have to be exchanged, and migration time is dominated by waiting for messages to exchange. For good, contiguous 3D partitions, the maximum number of neighbors should have an asymptotic limit, although this scaling study did not reach this theoretical ceiling.

6. APPLICATIONS

6.1. Active Flow Control

PHASTA [Jansen et al. 2000; Whiting et al. 2003] is an effective implicit finite element based CFD code for bridging a broad range of time and length scales in various flows including turbulent ones (based on URANSS, DES, LES, DNS). It has been applied with anisotropic adaptive algorithms [Sahni et al. 2006; Sahni et al. 2008; Sahni et al. 2009; Ovcharenko et al. 2013] along with advanced numerical models of flow physics [Hughes et al. 2000; Tejada-Martínez and Jansen 2005; 2006]. Modeling large-scale aerodynamic problems and active flow control's effects on large-scale flow changes (for instance, re-attachment of separated flow or virtual aerodynamic shaping of lifting surfaces) from micro-scale input [Amitay et al. 1998; Glezer and Amitay 2002; Sahni et al. 2011] requires an efficient parallel adaptive workflow.

A workflow supporting parallel adaptive PHASTA flow simulations based on the component workflow in Figure 1 is given in Figure 12. Figure 13 shows the initial and adapted mesh near the leading edge of TrapWing NASA test case [Chitale et al. 2014]. In this test, the Argonne Leadership Computing Facility's IBM BlueGene Q Mira system. Mira system provides 4 hardware threads per core. Running this workflow in Mira system with 4 MPI processes per core, PHASTA achieved strong scaling in mesh-based computations on up to 786,432 cores using 3,145,728 MPI processes with a 92 billion element mesh [Rasquin et al. 2014].

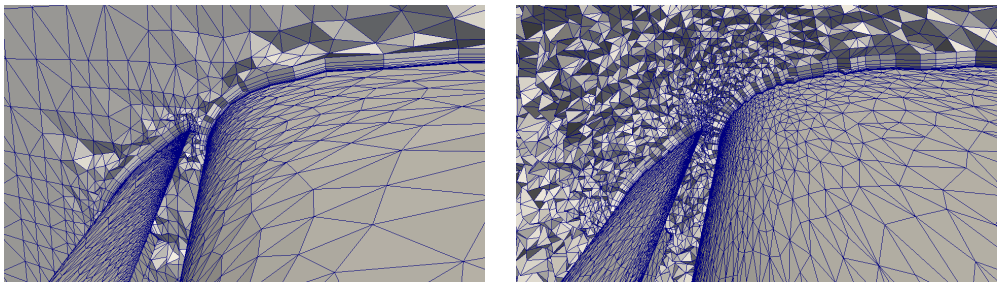


Fig. 13. Cut views of the initial (left) and adapted (right) anisotropic boundary layer meshes for NASA TrapWing [Chitale et al. 2014]

Key to this scaling and the efficiency of the workflow is controlling the load balance through PUMI interfaces to Zoltan and ParMA load balancing and partitioning tools. The workflow invokes load balancing after parallel mesh generation, during general unstructured mesh adaptation and before execution of PHASTA. Dynamic partitioning using a combination of ParMA and Zoltan is executed after parallel mesh generation to reach the partition sizes needed by mesh adaptation and PHASTA. During mesh adaptation, ParMA predictive load balancing procedures are used to ensure system memory is not exhausted and the resulting mesh is balanced. Lastly, before PHASTA execution, ParMA multi-criteria diffusive procedures are performed to reduce both the mesh element and mesh vertex imbalance. During each of these stages the association of PHASTA solution data with mesh entities is maintained via field migration and local solution transfer procedures.

An in-memory coupling supporting a parallel adaptive PHASTA analysis loop [Smith et al. 2015] using the components depicted in Figure 12 is enabled through a functional interface to the FORTRAN 77/90 based flow solver. Through the use of FORTRAN 2003 *iso_c_bindings*, this interface supports interoperability with C/C++ components, the control of solver execution, and the interrogation and management of solver data structures.

6.2. Albany Adaptive Loop

Albany [Salinger et al. 2014] is a general-purpose finite element code built on the Trilinos framework [Heroux et al. 2005; Trilinos Web 2015], both of which are developed primarily at Sandia National Laboratories. This code is highly extensible, allowing the creation of new finite element numerical methods, which makes it an ideal platform for research in finite elements. The design of Albany is parallel from the start, and also includes an abstract interface for discretization storage, i.e. a mesh database, as well as various adaptivity codes.

As illustrated in Figure 14, PUMI was used to form a parallel adaptive loop using Albany and MeshAdapt [Li et al. 2005; Alauzet et al. 2006; Smith et al. 2015]. This is entirely an in-memory coupling: the mesh database provides simple connectivity arrays and field data arrays to Albany for analysis, which Albany returns after a specified number of analysis steps on an unchanging mesh.

Once the data is back in PUMI structures, mesh adaptivity can be invoked on them to produce a new mesh, and solution transfer of key solution variables allows this new state to be sent back to Albany for further analysis, resulting in a self-contained, in-memory adaptive finite element code. The rich encoding of the PUMI mesh means that it is almost always a superset of the mesh information required for an analysis code. As such, we were able to convert not only to connectivity structures used internally by Albany, but also to another mesh data structure known as STK, which is a part of

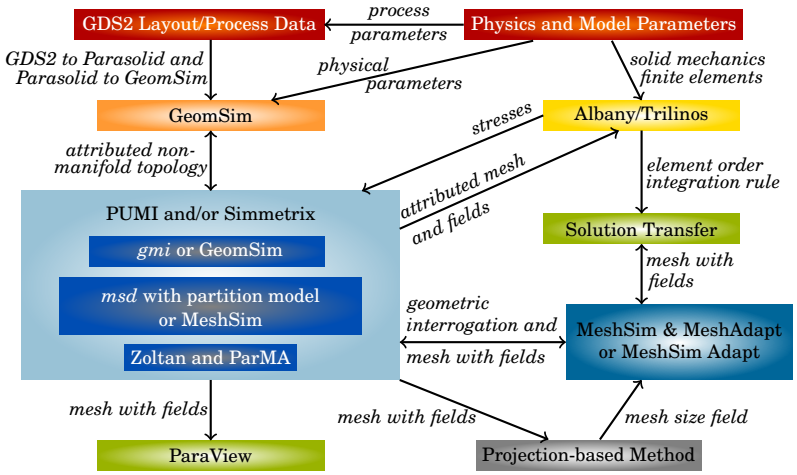


Fig. 14. Workflow of parallel Albany adaptive loop

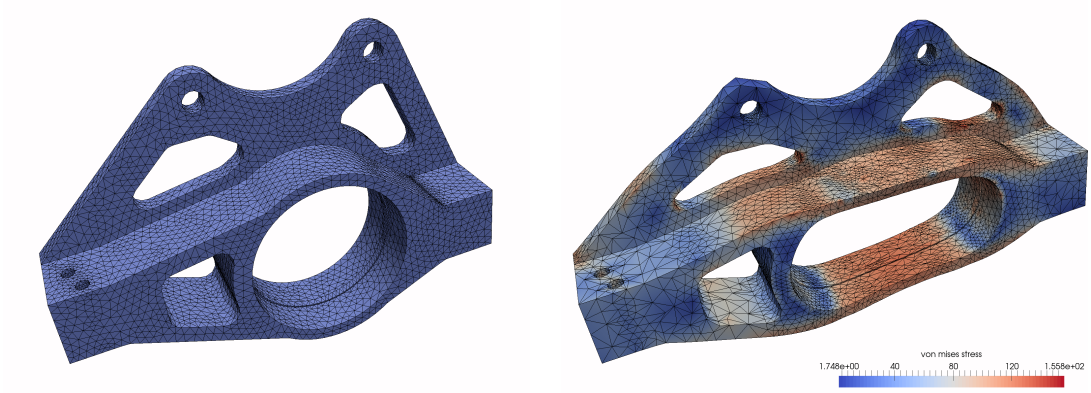


Fig. 15. Initial (left) and adapted mesh showing the Von Mises stress field that guides adaptivity (right)

the Trilinos framework. This makes PUMI more interoperable with any finite element codes involving the Trilinos framework. Figure 15 illustrates the initial and adapted mesh in large deformation analysis with Albany adaptive loop.

7. CLOSING REMARKS

PUMI is a parallel unstructured mesh infrastructure designed to support adaptive analysis simulations on massively parallel computers with an enriched set of distributed mesh control routines. In this paper, we discussed a rich design, software structure, example programs, performance results and two applications. The in-depth study toward extreme-scale performance continues since it requires an optimized orchestration of a complicated interplay of the problem statement, programming techniques, architectures knowledge (processor, memory, I/O, and network interconnections), and balance between computational and communication loads [ASCAC Web 2015; LLNL HPC Web 2015].

Except for *gmi_acis*, *gmi_geomsim*, and *gmi_parasolid* which directly interact with commercial modeling kernels, all PUMI libraries are open source programs download-

able at *https://github.com/SCOREC/core.git*. In the same source repository, the users can download MeshAdapt, ParMA, and many useful example programs. For more information on PUMI, visit *http://www.scorec.rpi.edu/pumi*.

REFERENCES

ACIS Web 2015. 3D ACIS Modeling. (2015). http://spatial.com/products/3d-acis-modeling.

Frédérik Alauzet, Xiangrong Li, E. Seegyoung Seol, and Mark S. Shephard. 2006. Parallel anisotropic 3D mesh adaptation by mesh modification. *Engineering with Computers* 21, 3 (jan 2006), 247–258. DOI:http://dx.doi.org/10.1007/s00366-005-0009-3

Micheal Amitay, Barton L. Smith, and Ari Glezer. 1998. Aerodynamic flow control using synthetic jet technology. In *36th AIAA Aerospace Sciences Meeting and Exhibit*. American Institute of Aeronautics and Astronautics. DOI:http://dx.doi.org/10.2514/6.1998-208

APF Web 2015. APF: A Parallel Field Library. (2015). http://www.scorec.rpi.edu/apf.

ASCAC Web 2015. ASCAC: Advanced Scientific Commuting Advisory Committee of U.S. Department of Energy. (2015). http://science.energy.gov/ascr/ascac.

Mark W. Beall. 1999. *An object-oriented framework for the reliable automated solution of problems in mathematical physics*. Ph.D. Dissertation. Rensselaer Polytechnic Institute, Troy, NY.

Mark W. Beall and Mark S. Shephard. 1997. A general topology-based mesh data structure. *Internat. J. Numer. Methods Engrg.* 40, 9 (may 1997), 1573–1596. DOI:http://dx.doi.org/10.1002/(SICI)1097-0207(19970515)40:9⟨1573::AID-NME128⟩3.0.CO;2-9

Mark W. Beall, Joe Walsh, and Mark S. Shephard. 2004. A comparison of techniques for geometry access related to mesh generation. *Engineering with Computers* 20, 3 (sep 2004), 210–221. DOI:http://dx.doi.org/10.1007/s00366-004-0289-z

E. G. Boman, U. V. Catalyurek, C. Chevalier, and K. D. Devine. 2012. The Zoltan and Isorropia Parallel Toolkits for Combinatorial Scientific Computing: Partitioning, Ordering, and Coloring. *Scientific Programming* 20, 2 (2012), 129–150.

CCI Web 2015. CCI: Center for Computational Innovations, Rensselaer Polytechnic Institute. (2015). http://cci.rpi.edu.

Waldemar Celes, Glaucio H Paulino, and Rodrigo Espinha. 2005a. A compact adjacency-based topological data structure for finite element mesh representation. *Internat. J. Numer. Methods Engrg.* 64, 11 (2005), 1529–1556.

Waldemar Celes, Glaucio H. Paulino, and Rodrigo Espinha. 2005b. A compact adjacency-based topological data structure for finite element mesh representation. *Internat. J. Numer. Methods Engrg.* 64, 11 (nov 2005), 1529–1556. DOI:http://dx.doi.org/10.1002/nme.1440

Kedar C. Chitale, Onkar Sahni, Mark S. Shephard, Saurabh Tendulkar, and Kenneth E. Jansen. 2014. Anisotropic adaptation for transonic flows with turbulent boundary layers. *AIAA Journal* 53, 2 (2014), 367–378. DOI:http://dx.doi.org/10.2514/1.J053159

Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson, and Courtenay Vaughan. 2002. Zoltan Data Management Services for Parallel Dynamic Applications. *Computing in Science and Engineering* 4, 2 (2002), 90–97.

Vladimir Dyedov, Navamita Ray, Daniel Einstein, Xiangmin Jiao, and Timothy J Tautges. 2014. AHF: Array-Based Half-Facet Data Structure for Mixed-Dimensional and Non-manifold Meshes. In *Proceedings of the 22nd International Meshing Roundtable*. Springer, 445–464.

H Carter Edwards, Alan B Williams, Gregory D Sjaardema, David G Baur, and William K Cochran. 2010. SIERRA toolkit computational mesh conceptual model. *Sandia National Laboratories SAND Series, SAND2010-1192* (2010).

FASTMath DOE SciDAC Web 2015. FASTMath: Applied Mathematics Algorithms, Tools, and Software for HPC Applications. (2015). http://www.fastmath-scidac.org.

Joe E. Flaherty, Raymond M. Loy, Mark S. Shephard, Boleslaw K. Szymanski, James D. Teresco, and Louis H. Ziantz. 1997. Predictive load balancing for parallel adaptive finite element computation. In *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, Xiangmin Jiao and Jean-Christophe Weill (Eds.), Vol. 1. 460–469. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.11.4649

Azat Yu. Galimov, Onkar Sahni, Jr. Richard T. Lahey, Mark S. Shephard, Donald A. Drew, and Kenneth E. Jansen. 2010. Parallel adaptive simulation of a plunging liquid jet. *Acta Mathematica Scientia* 30, 2 (mar 2010), 522–538. DOI:http://dx.doi.org/10.1016/S0252-9602(10)60060-4

Rao V. Garimella. 2002. Mesh data structure selection for mesh generation and FEA applications. *Internat. J. Numer. Methods Engrg.* 55 (2002), 451–478. DOI:http://dx.doi.org/10.1002/nme.509

GeomSim Web 2015. GeomSim: Direct Geometry Access. (2015). http://www.simmetrix.com/products/SimulationModelingSuite/geomsim/geomsim.html.

Ari Glezer and Micheal Amitay. 2002. Synthetic jets. *Annual Review of Fluid Mechanics* 34 (jan 2002), 503–529. DOI:http://dx.doi.org/10.1146/annurev.fluid.34.090501.094913

GRUMMP Web 2015. Generation and Refinement of Unstructured, Mixed-Element Meshes. (2015). http://tetra.mech.ubc.ca/GRUMMP.

Glen Hansen and Steve Owen. 2008. Mesh generation technology for nuclear reactor simulation; Barriers and opportunities. *Journal of Nuclear Engineering and Design* 238, 10 (oct 2008), 2590–2605. DOI:http://dx.doi.org/10.1016/j.nucengdes.2008.05.016

Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. 2005. An overview of the Trilinos project. *ACM Transaction on Mathematical Software (TOMS) - Special issue on the Advanced ComputaTional Software (ACTS) Collection* 31, 3 (sep 2005), 397–423. DOI:http://dx.doi.org/10.1145/1089014.1089021

Torsten Hoefler, Christian Siebert, and Andrew Lumsdaine. 2010. Scalable communication protocols for dynamic sparse data exchange. *ACM Sigplan Notices* 45, 5 (2010), 159–168.

Thomas J.R. Hughes, Luca Mazzei, and Kenneth E. Jansen. 2000. Large-eddy simulation and the variational multiscale method. *Computing and Visualization in Science* 3, 1–2 (may 2000), 47–59. DOI:http://dx.doi.org/10.1007/s007910050051

Daniel A. Ibanez, Ian Dunn, and Mark S. Shephard. 2014. Hybrid MPI-thread parallelization of adaptive mesh operations. *Parallel Comput.* (2014). submitted.

iMeshP Web 2015. iMeshP: SciDAC ITAPS Parallel Mesh Interface. (2015). http://www.itaps.org/software/iMeshP_html.

ITAPS Web 2015. ITAPS: Interoperable Technologies for Advanced Petascale Simulations of Department of Energy's Scientific Discovery through Advanced Computing (SciDAC). (2015). http://www.itaps.org.

Kenneth E. Jansen, Christian H. Whiting, and Gregory M. Hulbert. 2000. A generalized-α method for integrating the filtered Navier-Stokes equations with a stabilized finite element method. *Computer Methods in Applied Mechanics and Engineering* 190, 3–4 (oct 2000), 305–319. DOI:http://dx.doi.org/10.1016/S0045-7825(00)00203-6

Anil K. Karanam, Kenneth E. Jansen, and Christian H. Whiting. 2008. Geometry based pre-processor for parallel fluid dynamic simulations using a hierarchical basis. *Engineering with Computers* 24, 1 (jan 2008), 17–26. DOI:http://dx.doi.org/10.1007/s00366-007-0063-0

Orion S. Lawlor, Sayantan Chakravorty, Terry L. Wilmarth, Nilesh Choudhury, Issac Dooley, Gengbin Zheng, and Laxmikant V. Kalé. 2006. Parfum: A parallel framework for unstructured meshes for scalable dynamic physics applications. *Engineering with Computers* 22, 3 (dec 2006), 215–235. DOI:http://dx.doi.org/10.1007/s00366-006-0039-5

Xiangrong Li, Mark S. Shephard, and Mark W. Beall. 2005. 3D anisotropic mesh adaptation by mesh modification. *Computer Methods in Applied Mechanics and Engineering* 194, 48–49 (nov 2005), 4915–4950. DOI:http://dx.doi.org/10.1016/j.cma.2004.11.019

LLNL HPC Web 2015. High Performance Computing Training, Lawrence Livermore National Laboratory. (2015). http://computing.llnl.gov/training.

Xiao-Juan Luo, Mark S. Shephard, Lie-Quan Lee, Lixin Ge, and Cho Ng. 2011. Moving curved mesh adaption for higher-order finite element simulations. *Engineering with Computers* 27 (2011), 41–50. DOI:http://dx.doi.org/10.1007/s00366-010-0179-5

MeshAdapt Web 2015. MeshAdapt: Parallel Unstructured Mesh Adaptation Library. (2015). http://www.scorec.rpi.edu/meshadapt.

MeshSim Web 2015. MeshSim: Mesh Matching. (2015). http://www.simmetrix.com/products/SimulationModelingSuite/MeshSim/MeshMatching/MeshMatching.html.

NetCDF Web 2015. NetCDF: Network Common Data Form. (2015). http://www.unidata.ucar.edu/software/netcdf.

Robert M. O'Bara, Mark W. Beall, and Mark S. Shephard. 2002. Attribute management system for engineering analysis. *Engineering with Computers* 18 (2002), 339–351. DOI:http://dx.doi.org/10.1007/s00366020030

Carl Ollivier-Gooch, Lori F. Diachin, Mark S. Shephard, Tim J. Tautges, Jason A. Kraftcheck, Vitus Leung, Xiaojuan Luo, and Mark Miller. 2010. An interoperable, data-structure-neutral

component for mesh query and manipulation. *ACM Trans. Math. Software* 37, 3 (sep 2010). DOI:http://dx.doi.org/10.1145/1824801.1864430

Aleksandr Ovcharenko, Kedar C. Chitale, Onkar Sahni, Kenneth E. Jansen, and Mark S. Shephard. 2013. Parallel adaptive boundary layer meshing for CFD analysis. In *Proceedings of the 21st International Meshing Roundtable*, Xiangmin Jiao and Jean-Christophe Weill (Eds.). Springer Berlin Heidelberg, 437–455. DOI:http://dx.doi.org/10.1007/978-3-642-33573-0_26

Malcolm J. Panthaki, Raikanta Sahu, and Walter H. Gerstle. 1997. An object-oriented virtual geometry interface. In *Proceedings of the 6th International Meshing Roundtable*. Springer Berlin Heidelberg, Park City, Utah, USA, 67–82. http://www.imr.sandia.gov/papers/abstracts/Pa54.html

Parasolid Web 2015. Parasolid: 3D Geometric Modeling Engine. (2015). http://www.plm.automation.siemens.com/en_us/products/open/parasolid.

PUMI Web 2015. PUMI: Parallel Unstructured Mesh Infrastructure. (2015). http://www.scorec.rpi.edu/pumi.

Michel Rasquin, Cameron W. Smith, Kedar Chitale, E. Seegyoung Seol, Ben Matthews, J. Martin, Onkar Sahni, Raymond Loy, Mark S. Shephard, and Kenneth E. Jansen. 2014. Scalable fully implicit finite element flow solver with application to high-fidelity flow control simulations on a realistic wind design. *Computing in Science and Engineering* (2014). submitted.

Jean-François Remacle and Mark S. Shephard. 2003. An algorithm oriented mesh database. *Internat. J. Numer. Methods Engrg.* 58, 2 (sep 2003), 349–374. DOI:http://dx.doi.org/10.1002/nme.774

Onkar Sahni, Kenneth E. Jansen, Mark S. Shephard, Charles A. Taylor, and Mark W. Beall. 2008. Adaptive boundary layer meshing for viscous flow simulations. *Engineering with Computers* 24, 3 (sep 2008), 267–285. DOI:http://dx.doi.org/10.1007/s00366-008-0095-0

Onkar Sahni, Kenneth E. Jansen, Charles A. Taylor, and Mark S. Shephard. 2009. Automated adaptive cardiovascular flow simulations. *Engineering with Computers* 25, 1 (2009), 25–36. DOI:http://dx.doi.org/10.1007/s00366-008-0110-5

Onkar Sahni, Jens Müller, Kenneth E. Jansen, Mark S. Shephard, and Charles A. Taylor. 2006. Efficient anisotropic adaptive discretization of cardiovascular system. *Computer Methods in Applied Mechanics and Engineering* 195, 41–43 (aug 2006), 5634–5655. DOI:http://dx.doi.org/10.1016/j.cma.2005.10.018

Onkar Sahni, Joshua Wood, Kenneth E. Jansen, and Michael Amitay. 2011. Three-dimensional interactions between a finite-span synthetic jet and a crossflow. *Journal of Fluid Mechanics* 671 (2011), 254 – 287. http://dx.doi.org/10.1017/S0022112010005604

Andrew G. Salinger, Roscoe A. Bartlett, Qiushi Chen, Xujiao Gao, Glen A. Hansen, Irina Kalashnikova, Alejandro Mota, Richard P. Muller, Erik Nielsen, Jakob T. Ostien, Roger P. Pawlowski, Eric T. Phipps, and Waiching Sun. 2014. Albany: a component-based partial differential equation code built on Trilinos. *ACM Transaction on Mathematical Software (TOMS)* (2014). under review.

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

Larry A. Schoof and Victor R. Yarberry. 1994. *EXODUS II: a finite element data model*. Technical Report SAND92-2137. Sandia National Laboratories, Albuquerque, NM 87158 and Livermore, CA 94550.

E. Seegyoung Seol. 2005. *FMDB: flexible distributed mesh database for parallel automated adaptive analysis*. Ph.D. Dissertation. Rensselaer Polytechnic Institute, Troy, NY.

E Seegyoung Seol and Mark S Shephard. 2006a. Efficient distributed mesh data structure for parallel automated adaptive analysis. *Engineering with Computers* 22, 3-4 (2006), 197–213.

E. Seegyoung Seol and Mark S. Shephard. 2006b. Efficient distributed mesh data structure for parallel automated adaptive analysis. *Engineering with Computers* 22, 3 (dec 2006), 197–213. DOI:http://dx.doi.org/10.1007/s00366-006-0048-4

E. Seegyoung Seol, Cameron W. Smith, Daniel A. Ibanez, and Mark S. Shephard. 2012. A parallel unstructured mesh infrastructure. *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:* (nov 2012), 1124–1132. DOI:http://dx.doi.org/10.1109/SC.Companion.2012.135

Mark S. Shephard. 2000. Meshing environment for geometry-based analysis. *Internat. J. Numer. Methods Engrg.* 47, 1–3 (jan 2000), 169–190.

Mark S. Shephard and E. Seegyoung Seol. 2009. Flexible distributed mesh data structure for parallel adaptive analysis. In *Advanced Computational Infrastructures for Parallel and Distributed Applications*, Manish Parashar and Xiaolin Li (Eds.). Wiley, Chapter 19, 407–435. DOI:http://dx.doi.org/10.1002/9780470558027.ch19

Mark S. Shephard, Cameron Smith, E. Seegyoung Seol, and Onkar Sahni. 2013. Methods and tools for parallel anisotropic mesh adaptation and analysis. In *VI International Conference on Adaptive Modeling and Simulation (ADMOS 2013)*, J.P. Moitinho de Almeida, P. Díez, C. Tiago, and N. Parés (Eds.). the European Community in Computational Methods in Applied Sciencies (ECCOMAS), Lisbon, Portugal. http://www.lacan.upc.edu/admos2013/proceedings/a69.pdf

Simmetrix Web 2015. Simmetrix: Simulation Modeling and Application Suite. (2015). http://www.simmetrix.com.

Cameron W. Smith, Kedar Chitale, Dan A. Ibanez, Brian Orecchio, E. Seegyoung Seol, Onkar Sahni, Kenneth E. Jansen, and Mark S. Shephard. 2015. Building effective parallel unstructured adaptive simulations by in-memory integration of existing software components. *SIAM Journal on Scientific Computing* (2015). In Preparation.

Tim J. Tautges. 2001. CGM: a geometry interface for mesh generation, analysis and other applications. *Engineering with Computers* 17, 3 (oct 2001), 299–314. DOI:http://dx.doi.org/10.1007/PL00013387

T. J. Tautges, R. Meyers, K. Merkley, C. Stimpson, and C. Ernst. 2004a. *MOAB: A Mesh-Oriented Database*. SAND2004-1592. Sandia National Laboratories. Report.

Tim J. Tautges, Ray Meyers, Karl Merkley, Clint Stimpson, and Corey Ernst. 2004b. *MOAB: a mesh-oriented database*. Technical Report SAND2004-1592. Sandia National Laboratories, Albuquerque, NM 87158 and Livermore, CA 94550.

Andrés E. Tejada-Martínez and Kenneth E. Jansen. 2005. On the interaction between dynamic model dissipation and numerical dissipation due to streamline upwind/Petrov-Galerkin stabilization. *Computer Methods in Applied Mechanics and Engineering* 194, 9–11 (mar 2005), 1225–1248. DOI:http://dx.doi.org/10.1016/j.cma.2004.06.037

Andrés E. Tejada-Martínez and Kenneth E. Jansen. 2006. A parameter-free dynamic subgrid-scale model for large-eddy simulation. *Computer Methods in Applied Mechanics and Engineering* 195, 23 (apr 2006), 2919–2938. DOI:http://dx.doi.org/10.1016/j.cma.2004.09.016

Saurabh Tendulkar, Mark W. Beall, Mark S. Shephard, and Kenneth E. Jansen. 2011. Parallel mesh generation and adaptation for CAD geometries. In *Proc. NAFEMS World Congress*. Boston, MA.

Trilinos Web 2015. The Trilinos Project: Sandia National Laboratories. (2015). http://trilinos.sandia.gov.

VTK Web 2015. VTK: Visualization Toolkit. (2015). http://www.vtk.org.

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

Christian H. Whiting, Kenneth E. Jansen, and Saikat Dey. 2003. Hierarchical basis for stabilized finite element methods for compressible flows. *Computer Methods in Applied Mechanics and Engineering* 192, 47-48 (2003), 5167 – 5185. http://dx.doi.org/10.1016/j.cma.2003.07.011

Ting Xie, E. Seegyoung Seol, and Mark S. Shephard. 2014. Generic components for petascale adaptive unstructured mesh-based simulations. *Engineering with Computers* 30, 1 (jan 2014), 79–95. DOI:http://dx.doi.org/10.1007/s00366-012-0288-4

Min Zhou, Onkar Sahni, Karen D. Devine, Mark S. Shephard, and Kenneth E. Jansen. 2010a. Controlling Unstructured Mesh Partitions for Massively Parallel Simulations. *SIAM Journal on Scientific Computing* 32, 6 (nov 2010), 3201–3227. DOI:http://dx.doi.org/10.1137/090777323

Min Zhou, Onkar Sahni, H. Jin Kim, C. Alberto Figueroa, Charles A. Taylor, Mark S. Shephard, and Kenneth E. Jansen. 2010b. Cardiovascular flow simulation at extreme scale. *Computational Mechanics* 46, 1 (2010), 71–82. DOI:http://dx.doi.org/10.1007/s00466-009-0450-z

Min Zhou, Onkar Sahni, Ting Xie, Mark S. Shephard, and Kenneth E. Jansen. 2012a. Unstructured mesh partition improvement for implicit finite element at extreme scale. *Journal of Supercomputing* 59, 3 (mar 2012), 1218–1228. DOI:http://dx.doi.org/10.1007/s11227-010-0521-0

Min Zhou, Ting Xie, E. Seegyoung Seol, Mark S. Shephard, Onkar Sahni, and Kenneth E. Jansen. 2012b. Tools to support mesh adaptation on massively parallel computers. *Engineering with Computers* 28, 3 (jul 2012), 287–301. DOI:http://dx.doi.org/10.1007/s00366-011-0218-x

Online Appendix to: PUMI: Parallel Unstructured Mesh Infrastructure

Daniel A. Ibanez, Rensselaer Polytechnic Institute
E. Seegyoung Seol, Rensselaer Polytechnic Institute
Cameron W. Smith, Rensselaer Polytechnic Institute
Mark S. Shephard, Rensselaer Polytechnic Institute

A. ALGORITHMS FOR MESH MIGRATION

For d dimensional mesh, the input to the migration procedure is two arrays $pset\_to\_migr$ and ent_to_migr, containing partition object (p-sets or d dimensional mesh entities), source part (a part where the partition object exists) ID, and destination part ID. Based on the residence part equation, the migration procedure computes all 0 to $d-1$ dimensional mesh entities to migrate and migrates all subsidiary data (for instance, tagged data to mesh entities and p-set) to designated destination part. Based on the new partitioning topology, the procedure also updates the following:

- for each partition model per part, partition model entity with residence part set and owning part ID
- for each p-set in $pset\_to\_migr$, partition object entity handles contained in the p-set
- for each mesh entity influenced by migration, remote copies, match copies, and partition classification (a link to partition model entity). Mesh entities influenced by migration are *(i)* all entities contained in a p-set$\in pset_to_migr$, *(ii)* all entities in ent_to_migr, *(iii)* all downward entities of (i) and (ii), *(iv)* all remote copies of (iii), *(v)* all match copies of (i)-(iv).

The overall steps of migration procedure are the following (Algorithm 2):

(1) *Migrate p-sets*: for each $[S_i, P_{src}, P_{dest}] \in pset\_to\_migr$, copy partition object entities in S_i into ent_to_migr and create a new p-set S_i' on a destination part P_{dest}. The part P_{src} stores pairs of S_i and S_i' in an STL map container $pset\_map$ to use in step 4.

(2) *Collect entities to exchange and update residence part set*: collect mesh entities of which internal data (remote copy, partition classification, match copy, etc.) should be updated based on the new partitioning topology. This step collects entities to be updated, determines new residence part set of them and reset their partition classification. The entities collected are *(i)* entities in ent_to_migr, (ii) downward adjacent entities of entities in ent_to_migr, *(iii)* remote copies of (ii), and stored in array $ent\_to\_exchg[i]$ per type, $0 \leq i \leq d$. Algorithm 3 presents the pseudo code of this step.

(3) *Update partition classification and collect entities to remove*: based on new residence part set computed in step 2, this step updates partition classification and determines entities to remove after migration. The entities to remove are stored in 2D array ent_to_remove where entities of type i are stored in $ent\_to\_remove[i]$, $0 \leq i \leq d$.

(4) *Exchange entities*: Based on new residence part set determined in step 2, this step is performed for each entity type from low to high dimension. If a mesh entity M_i^d is on part boundary, the entity on part with the minimum part ID is in charge of sending the message to the new residence part P_{dest} where the entity doesn't

DOI: http://dx.doi.org/10.1145/0000000.0000000

exist. A new entity created in P_{dest}, $M_i^d@P_{dest}$, sends its address to the original mesh entity, $M_i^d@P_{src}$, then $M_i^d@P_{src}$ sends $M_i^d@P_{dest}$ to all its remote copies so all remote and *local* match copies of M_i^d are properly updated to have $M_i^d@P_{dest}$ as remote or match copies. If M_i^d is in a p-set S_i, S_i' is sent along. Algorithm 4 presents the pseudo-code of this step. Note that if the mesh is matched, the match copy should be also updated based on new partitioning topology. In the last of this step, Algorithm 5 is performed to update the *remote* match copies using the new remote copy information on part boundary.

(5) *Remove entities*: this step removes unnecessary entities collected in step 3 from high to low dimension. If the mesh entity to remove, $M_i^d@P_{src}$, is on part boundary, its remote copies and match copies also get updated to remove $M_i^d@P_{src}$. Algorithm 6 illustrates the pseudo-code of this step.
(6) *Remove p-sets*: remove p-sets in $pset\_to\_migr$.
(7) *Update owning part in partition model*: update the owning part ID of each partition model entity based on poor-to-rich rule.

B. ALGORITHMS FOR MESH GHOSTING

The input arguments of a ghosting procedure are ghost type g, bridge type b, the number of ghost layers n, an integer flag inc_copy to indicate whether to include non-owned remote copies in ghost computation or not. The procedure is designed to be *cumulative* such that if the procedure is performed multiple times with different input arguments, it constructs more ghost layers and/or copies in addition to existing ghost layers and/or copies.

The overall steps of cumulative *n*-layer ghosting procedure are the following (Algorithm 7):

(1) *Collect ghost candidates*: This step computes the entities and their destination part ID's to be ghosted and store the entities in array $ent\_to\_ghost[i]$ per type, $0 \leq i \leq g$. Algorithm 8 presents the pseudo code to compute the entities of type g. Algorithm 9 illustrates the psueco code to compute the entitites of type 0 to $g-1$. In Algorithm 8, if the input argument inc_copy is 0, only owner copies of part boundary entity of type b are considered in computation. Otherwise, all part boundary entities of type b are considered in computation.
(2) *Owner copies send messages to construct ghost layer(s)*: this step is performed for each entity type d from low to high dimension, $0 \leq d \leq g$. The owner copy of M_i^d in part P_{owner} sends a message to part P_{ghost} where the entity M_i^d doesn't exists as remote copy or ghost copy. Upon receiving the message in part P_{ghost}, a ghost copy M_i^d is created.
(3) *Ghost copies send messages to owner copies*: The ghost copy $M_i^d@P_{ghost}$ sends its address to the owner copy $M_i^d@P_{owner}$. Upon receiving the message in part P_{owner}, the owner copy $M_i^d@P_{owner}$ stores $M_i^d@P_{ghost}$ in its internal data structure to keep track of ghost copies.

ALGORITHM 2: Mesh migration

Input: mesh instance M of dimension d, 1D array $pset\_to\_migr$ containing $[S_i, P_{src}, P_{dest}]$, 1D array ent_to_migr containing $[M_i^d, P_{src}, P_{dest}]$ where S_i is p-set, M_i^d is partition object entity, P_{src} is source part ID, P_{dest} is destination part ID
Output: Mesh and partition model updated with new partitioning

```
// step 1: migrate p-sets
for each [S_i, P_src, P_dest] ∈ pset_to_migr do
    for each M_i^d ∈ S_i do
        insert [M_i^d, P_src, P_dest] into ent_to_migr;
    end
    if P_dest is local part handle then
        create a new p-set S_i'@P_dest;
        insert [S_i, S_i'] into pset_map;
    else
        send a message [S_i, P_src, P_dest] to P_dest;
    end
end
for each message [S_i, P_src, P_dest] received on P_dest do
    create a new p-set S_i'@P_dest;
    send a message [S_i, S_i'] to P_src;
end
for each message [S_i, S_i'] received on P_src do
    insert [S_i, S_i'] into pset_map;
end
// step 2: collect entitites to exchage between parts
// ent_to_exchg is 2D array of entities such that ent_to_exchg[i] contains entities of type i
collect_ent_to_exchange(M, ent_to_migr, ent_to_exchg);
// step 3: collect entitites to remove and update partition classification for i ← 0, d do
    for each M_i^d@P_src ∈ ent_to_exchg[d] do
        if P_src ∉ 𝒫[M_i^d] then
            insert M_i^d into ent_to_remove[d];
        else
            set partition classification based on 𝒫[M_i^d];
        end
    end
end
// step 4: exchange entities between parts from low to high dimension
for i ← 0, d do
    exchange_ent(ent_to_exchg[i], pset_map);
end
// step 5: remove unnecessary entities from high to low dimension
for i ← d, 0 do
    remove_ent(M, ent_to_remove[i]);
end
// step 6: remove p-sets
for each [S_i, P_src, P_dest] ∈ pset_to_migr do
    delete S_i from P_src;
end
// step 7: update owning part information based on new partitioning
for partition model P of each part in M do
    for each P_i^d of P do
        P_i^d's owning part ← a part with the least # partition object entities among 𝒫[P_i^d];
    end
end
```

ALGORITHM 3: *collect_ent_to_exchange*(M, ent_to_migr, ent_to_exchg);

Input: mesh instance M, 1D array ent_to_migr containing $[M_i^d, P_{src}, P_{dest}]$
Output: $ent\_to\_exchg[i]$ filled with $[M_j^i, P_{src}, P_{dest}]$, $0 \leq i \leq d$
for *each* $[M_i^d, P_{src}, P_{dest}] \in ent\_to\_migr$ **do**
  clear partition classification and $\mathscr{P}[M_i^d]$;
  insert P_{dest} into $\mathscr{P}[M_i^d]$;
  insert M_i^d into $ent\_to\_exchg[d]$;
end
for *each* $M_j^q @ P_{src} \in \{\partial(M_i^d)\}$ **do**
  if $M_j^q \notin ent\_to\_exchg[q]$ **then**
    clear partition classification and $\mathscr{P}[M_j^q]$;
    insert M_j^q into $ent\_to\_exchg[q]$;
  end
  if $\mathscr{P}[M_j^d]=\emptyset$ *where* $M_j^d \in \{M_j^q\{M^d\}\}$ **then**
    insert P_{src} into $\mathscr{P}[M_j^q]$;
  end
end
for $i \leftarrow 0$, d-1 **do**
  for *each* $M_i^d \in ent\_to\_exchg[d]$ **do**
    if M_i^d *is not on part boundary* **then**
      continue;
    end
    send $\mathscr{P}[M_j^d]$ to each remote copy;
  end
end
for *each message* $\mathscr{P}[M_j^q]$ *received from* P_{src} *to* P_{dest} **do**
  if $M_j^q \notin ent\_to\_exchg[q]$ **then**
    clear partition classification and $\mathscr{P}[M_j^q]$;
  end
  insert M_j^q into $ent\_to\_exchg[q]$;
  if $\mathscr{P}[M_i^d]=\emptyset$ *where* $M_i^d \in \{M_j^q\{M^d\}\}$ **then**
    insert P_{dest} into $\mathscr{P}[M_j^q]$;
    send P_{dest} to all remote copies of M_j^q;
  end
end
for *each message* P_i *received at* $M_i^d @ P_{dest}$ **do**
  insert P_i into $\mathscr{P}[M_i^d]$;
end

ALGORITHM 4: *exchange_ent*(ent_to_exchg, $pset\_map$);

Input: mesh instance M, ent_to_exchg, $pset\_map$
Output: M with new entities created
for *each* $[M_i^d, P_{src}] \in ent\_to\_exchg$ **do**
  if P_{src}=*min(*$\mathscr{P}[M_i^d]$*)* **then**
    for *each* $P_i \in \mathscr{P}[M_i^d]$, $P_i \neq P_{src}$ **do**
      pack message for M_i^d and send to P_i; // message includes $\{M_i^d\{M^{d-1}\}\}@P_i$,
      // geometric classification, $\mathscr{P}$, remote copies, *pset_map*$[S_i]$ (if any)
      // and tagged data (if any)
    end
  end
end
for *each message received on* P_{new} **do**
  unpack message and create M_i^d using $\{M_i^d\{M^{d-1}\}\}@P_i$;
  if *p-set* S_i *is found in the message* **then**
    insert M_i^d into S_i';
  end
  insert $M_i^d@P_{new}$ into list ent_to_bounce;
end
for *each* $M_i^d@P_{new} \in ent\_to\_bounce$, $d<mesh\_dimension$ **do**
  send $[M_i^d, P_{new}]$ to $M_i^d@P_{src}$;
end
for *each message* $[M_i^d, P_{new}]$ *received on* $M_i^d@P_{src}$ **do**
  $M_i^d@P_{src}$→add_remote($M_i^d@P_{new}$); // add $M_i^d@P_{new}$ to remote copy
  if $M_i^d@P_{src}$ *is matched* **then**
    $M_i^d@P_{src}$→add_match($M_i^d@P_{new}$); // add $M_i^d@P_{new}$ to match copy
    for *each match copy* $M_j^q@P_{match}$ **do**
      if P_{match} *is local part handle* **then**
        $M_j^q \rightarrow$ add_match($M_i^d@P_{new}$); // add $M_i^d@P_{new}$ to match copy
      end
    end
  end
  insert $[M_i^d@P_{src}, M_i^d@P_{new}]$ into list $ent\_to\_broadcast$;
end
for *each* $[M_i^d@P_{src}, M_i^d@P_{new}] \in ent\_to\_broadcast$ **do**
  send $[M_i^d, P_{new}]$ to all remote copies $M_i^d@P_{remote}$, $P_{remote} \neq P_{new}$;
end
for *each message* $[M_i^d, P_{new}]$ *received on* $[M_i^d, P_{remote}]$ **do**
  M_i^d, P_{remote}→add_remote(M_i^d, P_{new});
  if $M_i^d@P_{remote}$ *is matched* **then**
    $M_i^d@P_{remote}$→add_match($M_i^d@P_{new}$); // add $M_i^d@P_{new}$ to match copy
    for *each match copy* $M_j^q@P_{match}$ **do**
      if P_{match} *is local part handle* **then**
        $M_j^q \rightarrow$ add_match($M_i^d@P_{new}$); // add $M_i^d@P_{new}$ to match copy
      end
    end
  end
end
// update remote match copies
update_match(ent_to_exchg);

ALGORITHM 5: *update_match*(M, ent_to_exchg);

Input: mesh instance M, ent_to_exchg containing $[M_i^d, P_{src}]$
Output: M with match copy unified between remote matching entities
for *each* $[M_i^d, P_{src}] \in ent\_to\_exchg$ **do**
  if M_i^d *doesn't have match copies* **then**
    continue;
  end
  pack message with all match copies;
  for *each match copy* $M_j^q@P_i$ **do**
    if P_i *is local part handle* **then**
      continue;
    end
    if M_j^q *is remote copy of* M_i^d **then**
      continue;
    end
    send message to $M_j^q@P_i$;
  end
end
for *each message received from* $M_i^d@P_{src}$ *on* $M_j^q@P_i$ **do**
  for *each match copy* $M_k^p@P_{match}$∈*message* **do**
    if M_j^q = M_k^p **then**
      $M_j^q@P_i$→*add_match*($M_i^d@P_{src}$);
    else
      if $M_j^q@P_i$ *doesn't have match copy* $M_k^p@P_{match}$ **then**
        $M_j^q@P_i$→*add_match*($M_k^p@P_{match}$);
      end
    end
  end
end

ALGORITHM 6: *remove_ent*(M, ent_to_remove);

Input: mesh instance M, ent_to_remove
Output: M with unnecessary entities removed
for *each* $[M_i^d, P_{src}] \in ent\_to\_remove$ **do**
  if M_i^d *is on part boundary* **then**
    send $[M_i^d, P_{src}]$ to all remote copies $M_i^d@P_{remote}$;
  end
  if M_i^d *doesn't have match copies* **then**
    continue;
  end
  for *each match copy* $M_j^q@P_i$ **do**
    if P_i *is local part handle* **then**
      $M_j^q \rightarrow$*delete_match*($M_i^d@P_{src}$); // remove M_i^d from match copy
    else
      send $[M_i^d, P_{src}]$ to all match copies $M_j^q@P_{match}$;
    end
  end
end
for *each message* $[M_i^d, P_{src}]$ *received on* $M_i^d@P_{remote}$ **do**
  $M_i^d@P_{remote} \rightarrow$*delete_remote*($M_i^d@P_{src}$); // remove M_i^d from remote copy
end
for *each message* $[M_i^d, P_{src}]$ *received on* $M_j^q@P_{match}$ **do**
  $M_j^q@P_{match} \rightarrow$*delete_match*($M_i^d@P_{src}$); // remove M_i^d from match copy
end
for *each* $[M_i^d, P_{src}] \in ent\_to\_remove$ **do**
  if M_i^d *is in p-set* S_i **then**
    $S_i \rightarrow$*delete_ent*(M_i^d); // remove M_i^d from p-set
  end
  $P_{src} \rightarrow$*delete_ent*(M_i^d); // remove M_i^d from source part
end

ALGORITHM 7: Cumulative N-layer ghosting

Input: mesh instance M, ghost type g, bridge type b, the number of ghost layer n, a flag to indicate whether to include non-owned remote copies in ghost computation or not inc_copy
Output: M with n-layer ghost copies
// step 1: compute entities be ghosted
// ent_to_ghost is 2D array of entities such that $ent\_to\_ghost[i]$ contains entities of type i
compute_ent_to_ghost(M, g, b, n, inc_copy, $ent\_to\_ghost[g]$); // collect entities of type g
collect_ent_to_ghost(M, ent_to_ghost); // collect entities of type 0 to $g-1$
// step 2: owner copies send message to construct n-layer ghost copies
for $d \leftarrow 0$, g **do**
  for *each* $M_i^d \in ent\_to\_ghost[d]$ **do**
    if M_i^d *is owned by local part* P_{src} **then**
      for *each* $P_i \in \mathscr{P}[M_i^d]$ **do**
        // if M_i^d exists on P_i as remote copy or ghost copy, skip
        if $P_i \neq P_{src}$ & $P_i \neq P_{remote}$ & $P_i \neq P_{ghost}$ **then**
          pack message for M_i^d and send to P_i; //message includes $\{M_i^d\{M^{d-1}\}\}@P_i$,
          // geometric classification, $\mathscr{P}$, remote copies, match copies, and tagged data
        end
      end
    end
  end
end
for *each message received from* $M_i^d@P_{owner}$ *to* P_{ghost} **do**
  unpack message and create M_i^d using $\{M_i^d\{M^{d-1}\}\}@P_{ghost}$;
  // ent_to_bounce is 1D array to store a pair of $M_i^d@P_{owner}$ and $M_i^d@P_{ghost}$
  insert $[M_i^d@P_{owner}, M_i^d@P_{ghost}]$ into ent_to_bounce;
end
// step 3: send ghost copy to owner copy
for *each* $[M_i^d@P_{owner}, M_i^d@P_{ghost}] \in ent\_to\_bounce$ **do**
  send $M_i^d@P_{ghost}$ to $M_i^d@P_{owner}$;
end
for *each message received from* $M_i^d@P_{ghost}$ *to* $M_i^d@P_{owner}$ **do**
  $M_i^d@P_{owner} \rightarrow$*add_ghost*($M_i^d@P_{ghost}$); // owner copy keeps track of ghost copies
end

ALGORITHM 8: *compute_ent_to_ghost*(M, g, b, n, inc_copy, ent_to_ghost)

Input: M, g, b, n, inc_copy
Output: get 1D array ent_to_ghost filled with entities of type g to be ghosted
for *each* $M_i^b@P_{src}$ *on part boundary* **do**
  if inc_copy*=0 &* M_i^b *is not owned by* P_{src} **then**
    continue;
  end
  $\mathscr{P}[M_i^b] \leftarrow \emptyset$;
  // compute 1^{st} layer entities to be ghosted
  for *each* $M_j^g \in \{M_i^b\{M^g\}\}$ **do**
    if M_j^g *is ghost copy* **then**
      continue;
    end
    if M_j^g *is not found in* ent_to_ghost **then**
      $\mathscr{P}[M_j^g] \leftarrow \emptyset$;
      insert M_j^g into ent_to_ghost;
    end
    insert M_i^b's remote part ID's into $\mathscr{P}[M_j^g]$;
    mark M_j^g as $visited$;
  end
  // compute $2-n^{th}$ layer entities to be ghosted (n>1)
  for *layer*$\leftarrow 2, n$ **do**
    for *each* $M_k^g \in$ *ent_to_ghost added in* $(layer-1)^{th}$ *computing step* **do**
      for *each* $M_l^g \in \{M_k^g\{M^b\}\{M^g\}\}$ **do**
        if M_l^g *is ghost copy or marked as visited* **then**
          continue;
        end
        if M_l^g *is not found in* ent_to_ghost **then**
          $\mathscr{P}[M_l^g] \leftarrow \emptyset$;
          insert M_l^g into ent_to_ghost;
        end
        insert M_i^b's remote part ID's into $\mathscr{P}[M_l^g]$;
        mark M_l^g as $visited$;
      end
    end
  end
end

ALGORITHM 9: *collect_ent_to_ghost*(M, ent_to_ghost);

Input: mesh instance M, 2D array ent_to_ghost
Output: for each d, $0 \leq d < g$, get $ent\_to\_ghost[d]$ filled with M_i^d to be ghosted
for *each* $M_i^g @ P_{src} \in ent\_to\_ghost[g]$ **do**
  for *each* $M_j^d \in \{M_i^g\{M^d\}\}$, $0 \leq d < g$ **do**
    if M_j^d *is not found in ent_to_ghost*$[d]$ **then**
      $\mathscr{P}[M_j^d] \leftarrow \emptyset$;
      insert M_j^d into $ent\_to\_ghost[d]$;
    end
    $\mathscr{P}[M_j^d] \leftarrow \mathscr{P}[M_i^g] \cup \mathscr{P}[M_j^d]$;
  end
end
for $d \leftarrow 0$, g **do**
  for *each* $M_i^d \in ent\_to\_ghost[d]$ **do**
    send $\mathscr{P}[M_i^d]$ to all remote copies;
  end
end
for *each message* $\mathscr{P}[M_i^d @ P_{src}]$ *received from* $M_i^d @ P_{src}$ *to* $M_k^d @ P_{remote}$ **do**
  if $M_k^d \notin ent\_to\_ghost[d]$ **then**
    $\mathscr{P}[M_k^d] \leftarrow \emptyset$;
    insert M_k^d into $ent\_to\_ghost[d]$;
  end
  $\mathscr{P}[M_k^d] \leftarrow \mathscr{P}[M_k^d] \cup \mathscr{P}[M_i^d @ P_{src}]$;
end