A

PUMI: Parallel Unstructured Mesh Infrastructure

E. Seegyoung Seol, Rensselaer Polytechnic Institute
Daniel A. Ibanez, 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, data structures, algorithms, and API of MPI-based PUMI. The effectiveness of PUMI is demonstrated by its performance results and applications to massively parallel adaptive simulation workflows.

Categories and Subject Descriptors: D.1 [**Programming Techniques**]: Parallel programming; Object-oriented programming; E.1 [**Data Structures**]: Distributed data structures; G.1.8 [**Numerical Analysis**]: Partial Differential Equations - Finite element methods; Finite volume methods; H.3.4 [**Information Storage and Retrieval**]: Systems and Software - Distributed systems

General Terms: Algorithms, Design, Performance

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

ACM Reference Format:
E. Seegyoung Seol, Daniel A. Ibanez, Cameron W. Smith, and Mark S. Shephard, 2014. PUMI: Parallel Unstructured Mesh Infrastructure. *ACM Trans. Math. Softw.* V, N, Article A (January YYYY), 42 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 (PDEs) over general domains. Unstructured mesh finite volume and finite element methods have the advantage of being able to solve problems over

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: E.S. Seol, D.A. Ibanez, 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

geometrically complex physical domains using meshes that can be automatically generated, and anisotropically adapted, to effectively provide the level of solution accuracy desired with two, or more, orders of magnitude fewer unknowns than uniform mesh techniques and many unstructured mesh data structures have been introduced to date such as AHF [Dyedov et al. 2014], libMesh [Kirk et al. 2006], MOAB [Tautges 2004], OpenVolumeMesh [Kremer et al. 2013], etc. 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.

This paper presents a Parallel Unstructured Mesh Infrastructure (PUMI) that supports parallel automated adaptive unstructured mesh simulations that go from design models directly to fully parallel mesh generation, to a loop of unstructured mesh-based analysis, error estimation and mesh adaptation, to provide reliable simulation results. PUMI has recently supported construction of a 92 billion element mesh solved on 3/4 million compute cores [Rasquin et al. 2014]. PUMI is being developed as part of the DOE SciDAC FASTMath institute [FASTMath DOE SciDAC Web 2014] to support a full range of operations on adaptively evolving unstructured meshes on massively parallel computers [Seol and Shephard 2006; Seol et al. 2012; Zhou et al. 2012b; Xie et al. 2014]. When coupled with dynamic load balancing procedures [Zhou et al. 2012a; Zoltan Web 2014] PUMI provides the parallel 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 (e.g., 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 for combined FEM/PIC modeling of electromagnetics in particle accelerators [Luo et al. 2011], modeling blood flow in the human arterial system [Zhou et al. 2010], and two-

phase simulation of jets [Galimov et al. 2010] and 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).

This paper provides an overview of PUMI, focusing on its design, algorithms of core parallel functionalities, API's and applications on massively parallel computers. The fundamental concepts and definitions are presented in Section 2. Section 3 and 4 discuss the s/w aspects and core parallel algorithms, respectively. The design philosophy and the specification of the API's are presented in Section 5 and the example programs are illustrated in Section 6. Section 7 provides the results of three parallel services and Section 8 provides two adaptive simulation applications using PUMI to produce complete parallel simulation workflows. Finally, Section 9 discusses future directions of this work.

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

2. DEFINITIONS

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, *attributes* describing the loads, material properties, and boundary conditions on the geometric model needed to define the problem, *the mesh* which 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 2014].

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 the case of parallel adaptive simulations, the mesh data must be distributed in a manner consistent with the needs of the mesh-based analysis and

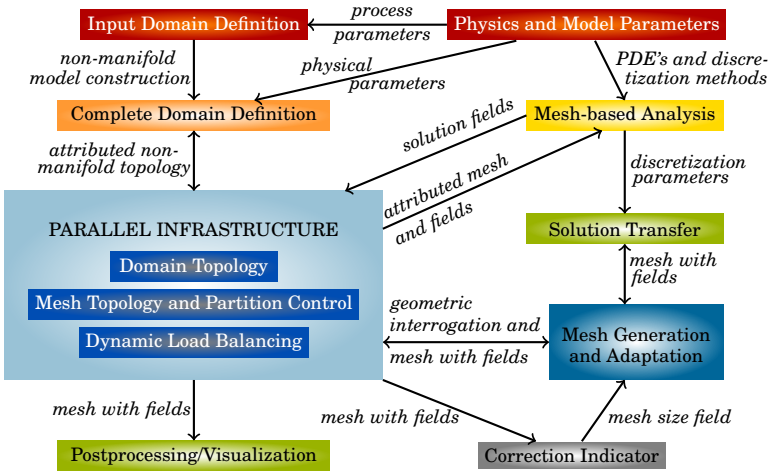


Fig. 1. Simulation infrastructure

must be flexible enough to support parallel mesh generation and adaptation processes. The mesh must also support the transfer of geometry-based attributes to the mesh to define input fields. Since the fields are directly related to the mesh, their parallel distribution is directly related to the distribution of the mesh. Since the amount of information defining the geometric model and attributes is small compared to the other data, they are most commonly fully represented on each node/core. However, as the level of geometric model complexity increases it is becoming 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 have a copy of the model entities and their adjacencies represented on each part for which a mesh entity that is classified on that model entity is stored [Tendulkar et al. 2011].

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. 2005].

2.2. Classification

Each mesh entity $M_i^{d_i}$ maintains a relation, called geometric classification, to the geometric model entity $G_j^{d_j}$ that it was created to partially represent.

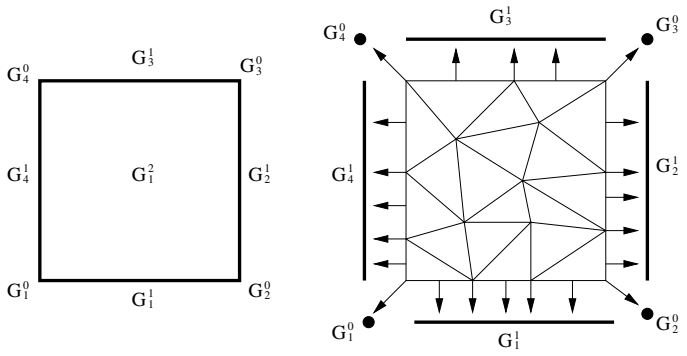


Fig. 2. Example of simple model(left) and mesh(right) showing their association via geometric classification [Beall and Shephard 1997; Beall 1999]

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

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 partition classification is denoted as RC(G_j^d) = $\{M_i^d \mid M_i^d \sqsubset G_j^d\}$.

Geometric classification allows an understanding of which attributes (e.g. boundary conditions or material properties) are related to the mesh entities and the how the solution relates back to the original problem description, and is critical in mesh generation and adaptation [Beall and Shephard 1997; Beall 1999; Shephard 2000]. In Figure 2, a mesh of simple square model with entities labeled is shown with arrows indicating the classification of the mesh entities onto the model entities. All of the interior mesh faces, mesh edges, and mesh vertices are classified on the model face G_1^2.

2.3. Adjacencies

The relationships of the entities are well described by topological adjacencies. For an entity of dimension d, first-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 ($d < q$). $\{M_i^d\{M^q\}\}$ indicates a set of entities of dimension q in mesh that are adjacent to M_i^d. As illustrated in Figure 3, ordering conventions can be used to enforce the specific downward first-order adjacent entity and $M_i^d\{M^q\}_j$ indicates the j^{th} entity in the set of mesh entities of dimension q that are adjacent to M_i^d.

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 levels of en-

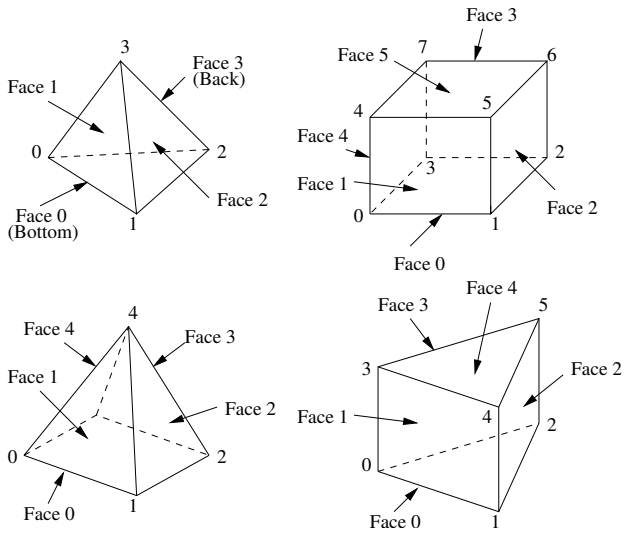


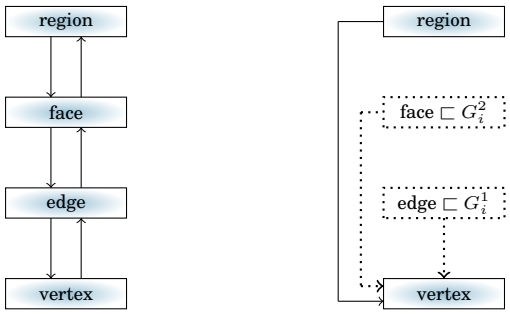
Fig. 3. Vertex and face order on a region [Beall and Shephard 1997; Beall 1999]

tities 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 level entities explicitly stored, the mesh representation is *full*, otherwise, *reduced (ii)* complete vs. incomplete - if all adjacency information is obtainable in O(1) time, the representation is *complete*, otherwise it is *incomplete*.

The *general* topology-based mesh data structures must satisfy completeness of adjacencies to support adaptive analysis efficiently. It doesn't necessarily mean that all d level entities and adjacencies need be explicitly stored in the representation. There are many representation options in the design of general topology-based mesh data structure. Figure 4 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 level of entities and explicitly stored adjacencies from outgoing level to incoming level. A dotted box denotes that among entities of the level, only equally classified ones are explicitly stored, and a dotted arrow denotes that adjacencies from an outgoing level to an incoming level are maintained only for the stored entities.

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, where *a part* consists of the set of mesh entities assigned to a process. For each part, a unique global part id within an entire system and a local part id within a process can be given. Each part will be treated as a serial mesh with the addition of *part boundaries* to describe groups of mesh entities that are on inter-part boundaries. In the MPI-based PUMI, mesh entities on part boundaries are duplicated on all parts on which they are used in adjacency relations. Mesh entities not on the part boundary exist on only one part and referred as *internal entities*. In implementation, for effective manipulation of multiple parts on each process, a single mesh data is defined on each process so multiple parts are contained in the mesh data where the mesh data is assigned to a process. Figure 5(a) illustrates a distributed mesh on two processes where the mesh on each process has two part handles. The dotted lines are *intra-process part boundaries* and thick solid



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

Fig. 4. Two complete mesh representations [Beall and Shephard 1997; Remacle and Shephard 2003]

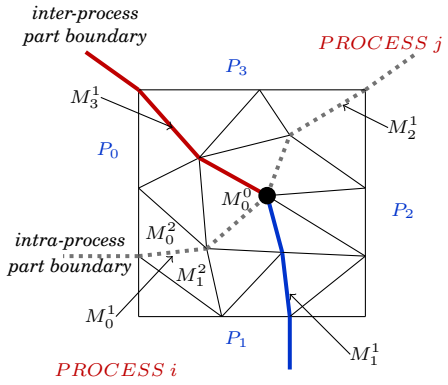


Fig. 5. Distributed mesh on two processes with two parts per process. The thick solid line is inter-process part boundary and the thick dotted line is intra-process part boundary.

lines are *inter-process part boundaries* between the processes representing mesh entities duplicated on multiple parts. *Residence part set operator* $\mathscr{P}[M_i^d]$ returns a set of global part id(s) where M_i^d exists. For instance, $\mathscr{P}[M_0^0]$ is $\{P_0, P_1, P_2, P_3\}$ and $\mathscr{P}[M_i^1]$ is $\{P_0, P_1\}$. The two parts, P_i and P_j, are *neighbor* over entity type d if they share d-dimensional mesh entities on part boundary.

Definition 2.3. Residence part equation of M_i^d
If $\{M_i^d\{M^q\}\} = \emptyset$, $d < q$, $\mathscr{P}[M_i^d] = \{p\}$ where p is the part id where M_i^d exists. Otherwise, $\mathscr{P}[M_i^d] = \cup\, \mathscr{P}[M_j^q \mid M_i^d \in \{\partial(M_j^q)\}]$.

The residence part set of M_i^d is the union of residence part set of all entities that it bounds. For a mesh topology where the entities of order $d > 0$ are bounded by entities of order d-1, $\mathscr{P}[M_i^d]$ is determined to be {p} if $\{M_i^d\{M_k^{d+1}\}\} = \emptyset$. Otherwise, $\mathscr{P}[M_i^d]$ is $\cup\, \mathscr{P}[M_k^{d+1} \mid M_i^d \in \{\partial(M_k^{d+1})\}]$. For instance, in Figure 5, $\mathscr{P}[M_0^1] = \mathscr{P}[M_0^2] \cup \mathscr{P}[M_1^2] = \{P_0\} \cup \{P_1\} = \{P_0, P_1\}$.

2.6. Partition Model

In a distributed mesh environment, a partition model exists between the mesh and the geometric model as a part of hierarchical domain decomposition for the purpose of rep-

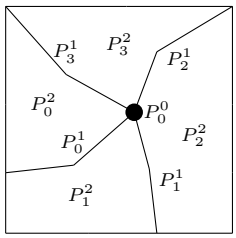


Fig. 6. Partition model of mesh in Figure 5

resenting mesh partitioning in topology and supporting mesh-level parallel operations through inter-part boundary links.

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

The partition model introduces a set of topological entities that represents the collections of mesh entities based on their location with respect to the partitioning. Grouping mesh entities to define a partition entity can be done with multiple criteria based on the level of functionalities and needs of distributed meshes. These constructs are consistent with the ITAPS iMeshP's parallel mesh requirements [iMeshP Web 2014].

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

Definition 2.4. Partition (model) entity
A topological entity in the partition model, P_i^d, represents a group of mesh entities of dimension d, that have the same residence part set $\mathscr{P}$. Each partition model entity is uniquely determined by $\mathscr{P}$.

Figure 6 illustrates the partition model of the mesh in Figure 5 representing the partitioning topology that 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 entities duplicated on P_0 and P_3, *(iii)* partition faces P_i^2 representing internal mesh entities (vertices, edges and faces) on part P_i.

Definition 2.5. Partition classification
The unique association of mesh topological entities of dimension d_i, $M_i^{d_i}$, to the topological entity of the partition model of dimension d_j, $P_j^{d_j}$ where $d_i \leq d_j$, on which it lies is termed partition classification and is denoted $M_i^{d_i} \sqsubset P_j^{d_j}$.

Each partition model entity stores dimension, id, residence part set, and the owning part. From a mesh entity level, by keeping proper relation to the partition model entity, all needed services to represent mesh partitioning (residence part set, and the owning

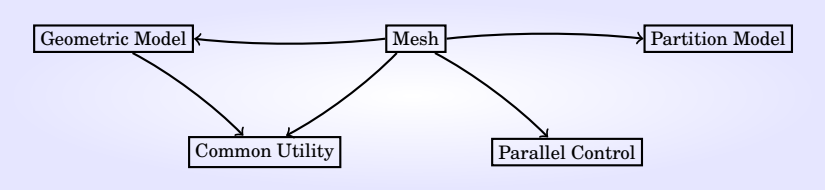


Fig. 7. PUMI components: an arrow from component A to component B indicates that the component A is dependent on the component B.

part) and support inter-part communications are easily supported. In Figure 5, mesh vertex M_0^0 is classified on partition vertex P_0^0, mesh edges M_i^1 is 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, $0 \leq i \leq 3$. In the implementation, maintaining partition classification is not required for internal mesh entities.

3. SOFTWARE STRUCTURE

PUMI consists of five s/w components where each component is a self-contained, independently usable piece of software with its own set of requirements and well-defined API (Figure 7). The Geometric Model component uses the Common Utility, and the Mesh component uses the Geometric Model, Partition Model, Common Utility and Parallel Control [Seol et al. 2012]. As the partition model is constructed based on the mesh distribution for the support of efficient update/manipulation of mesh with respect to partitioning and it's automatically updated while mesh partitioning is changed, its functionality is embedded in mesh implementation and doesn't provide a library nor allow the user to access/modify.

3.1. pumi_util - Common Utility Library

This library provides common utilities and services re-usable in multiple other components. It includes the three utilities *(i)* iterator for iterating over a range of data, *(ii)* set for grouping arbitrary data with common set requirements, *(iii)* tag for attaching arbitrary user data to arbitrary data or set with common tagging requirements [Tautges et al. 2004; Ollivier-Gooch et al. 2010; ITAPS Web 2014], that have been developed using a generic methodology so they are easily extendable in other components to meet individual needs [Xie et al. 2014]. For instance, the tag utility is used for tagging for geometric model entity, mesh entity, part handle and three different kinds of entity sets.

3.2. pcu - Parallel Control Utility Library

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

The foundation of PCU is its point-to-point message passing routines. Non-blocking synchronous message passing primitives are defined here. There are two versions, one of which is a direct interface to MPI, and the other which allows 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 runtime without affecting the rest of s/w 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 collectives and point-to-point communication, PCU provides a message passing algorithm for very general unstructured communication, allowing tasks to send any messages out to neighbors and ensuring that neighbors receive all the messages they have been sent. This *phased communication* API is the part of PCU used most heavily by the mesh database and other users.

Finally, PCU has a system for creating a pool of threads within each process and assigning them ranks the way 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.

3.3. pumi_geom - Geometric Model Library

Commercial geometric modeling kernels such as Acis [ACIS Web 2014], Parasolid [Parasolid Web 2014] and GeomSim [GeomSim Web 2014], provides initial description of the problem domain which is the basis for generating meshes on which the analysis is performed. The geometric model library between geometry-based applications and modeling kernels provides modeling kernel-independent geometry access by replicating the topological information stored in modeling kernels and polymorphism to enable the applications (including mesh) to interface with various modeling kernels [Panthaki et al. 1997; Tautges 2001; Beall et al. 2004]. In *pumi_geom*, the geometric model class hierarchy is derived for implementation with specific commercial geometric modeling kernels and the following functionalities are provided at the high-level API level regardless underlying modeling kernel. In the cases of no commercial modeling kernel available, *pumi_geom* supports geometric model constructed from mesh, called *mesh model* [Beall et al. 2004].

- *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.
- *file* I/O - ascii or binary in four geometric model formats (mesh model [Beall et al. 2004], Acis [ACIS Web 2014], GeomSim [GeomSim Web 2014], and Parasolid [Parasolid Web 2014]). A function to register a modeling kernel should be called to establish the relationship between the high level model loading API and the modeler specific API for loading the model file.
- *tagging* - attaching user data of various types (single or array of integer, pointer, floating point, binary, set, entity) to model entity
- *traversal* - model entity traversal by dimension
- *entity set* - grouping arbitrary set of model entities
- *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 *pumi_geom*, 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.

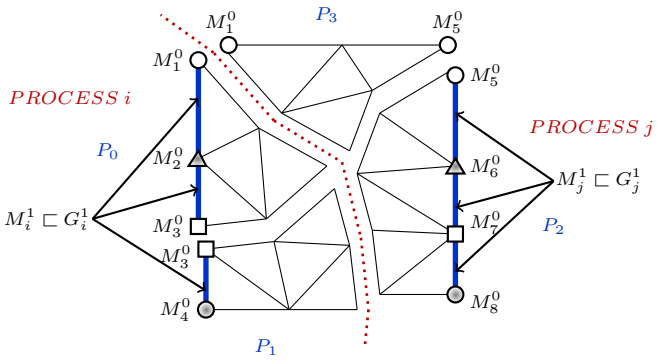


Fig. 8. 2D distributed mesh with periodic model edges G_i^1 and G_j^1

3.4. pumi_mesh - Mesh and Partition Model Library

pumi_mesh provides the storage and management of distributed unstructured meshes including all the mesh-level services to interrogate/modify the mesh data. A distributed mesh data structure provides all parallel mesh-based operations needed to support parallel adaptive analysis. An efficient and scalable distributed mesh data structure is mandatory to achieve performance since it strongly influences the overall performance of adaptive mesh-based simulations. In addition to the general mesh-based operations, the distributed mesh data structure must support *(i)* efficient communication between entities duplicated over multiple processes, *(ii)* migration of entities or group of entities between parts, *(iii)* dynamic load balancing. The core functionalities are the following:

- *file* I/O - ascii or binary in various mesh formats (NetCDF [NetCDF Web 2014], Exodus [Schoof and Yarberry 1994], VTK [VTK Web 2014], Simmetrix [Simmetrix Web 2014], etc.) for partition model and mesh entities with auxiliary data including geometric classification, partition classification, tagged data, sets, matched copies, etc.
- *tagging* - attaching user data of various types (single or array of integer, pointer, floating point, binary, set, entity) to part, entity and set handle
- *traversal* - entity traversal by part, dimension, shape, status (part boundary, ghost), and reverse geometric classification
- *entity set* - grouping arbitrary set of entities from multiple parts or a single part [Tautges et al. 2004; Ollivier-Gooch et al. 2010]. Especially, the entity set belonging to a part (termed *p-set*) is useful in representing mesh boundary layer [Sahni et al. 2008; Ovcharenko et al. 2013] and its requirements are *(i)* the consisting entity do not bound any higher-order mesh entities *(ii)* the consisting entities are in the same part handle *(iii)* the consisting entities are unique *(iv)* insertion order is preserved, *(v)* if *p-set* is designated to migrate, all entities in *p-set* are migrated as a unit.
- *modification* - entity and set creation/deletion, migrating entity and p-set from part to part including tagged data [Alauzet et al. 2006; Seol and Shephard 2006], and a capability to dynamically change the number of parts per process.
- *ghosting* - a procedure to localize off-part mesh entity and p-set to avoid inter-process communications for computations. A ghost is a read-only, duplicated, off-part internal entity copy including tag data [Lawlor et al. 2006; iMeshP Web 2014].

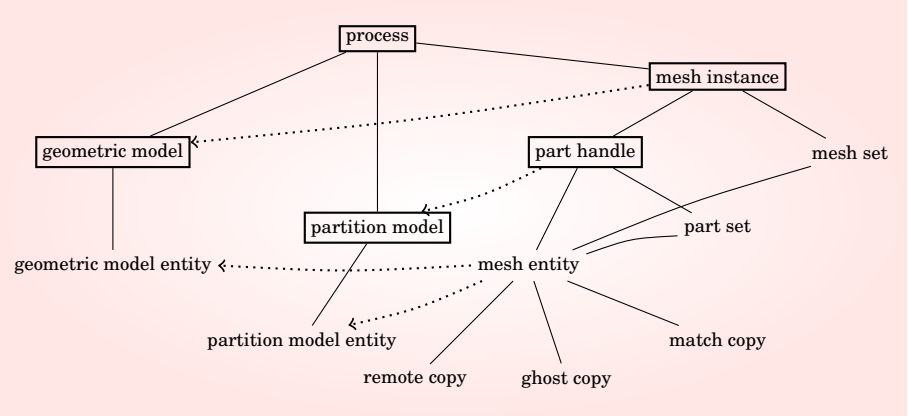


Fig. 9. Data model in process: A solid line from upper positioned component U to lower positioned component L indicates that the component U contains 0 or more component L. A dotted arrow from component A to component B indicates that the component A maintains a link to component B. A data drawn with rectangle indicates that at least one copy of the data should exist on each process.

- *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 mesh migration and partitioning libraries such as Zoltan [Zoltan Web 2014], ParMETIS [ParMETIS Web 2014], ParMA [ParMA Web 2014].
- *matching* - a capability to maintain mesh entities on matched, periodic model boundaries [Karanam et al. 2008; MeshSim Web 2014]. Figure 8 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_i^1, 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.
- *interrogations* - 1^{st} and 2^{nd}-order adjacency availed by complete representation, owning part, status (internal, part boundary, matched, ghost or ghosted), classification (geometric and partition model), etc.

In the data model on each process (Figure 9), the term *instance* is used to indicate the 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 and all parts on the process are maintained by and accessible through the mesh instance. The term *handle* is used to indicate the pointer to the other data such as part, entity and entity set. For example, a mesh entity handle means a pointer to the mesh entity data [Seol and Shephard 2006; Ollivier-Gooch et al. 2010; Seol et al. 2012; ITAPS Web 2014]. illustrates a data model in each process. Each mesh instance maintains *(i)* a link to a geometric model instance, *(ii)* one or more part handles, *(iii)* zero or more entity set handles where each set is either l-set (list-type set) or s-set (set-type set) [ITAPS Web 2014]]. The mesh stores the one-level adjacency representation which maintains adjacencies between entities one dimension apart [Beall and Shephard 1997]. Each part handle maintains *(i)* a link to a partition model instance, *(ii)* zero or more entity handles per dimension (0 to 3), *(iii)* zero or more p-set handles.

If a mesh entity is on the part boundaries, ghosted or matched, it must be aware of where they are duplicated, ghosted and matched. For entities duplicated or ghosted, a

specific part is assigned as the owner with charge of modification, communication or computation of the copies. For the purpose of simple denotation, an entity on owning part is referred as *owner* of all duplicate copies. Note 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. The owning part is determined based on *poor-to-rich rule* - owing part is a part with the least number of partition object entities among all residence parts [Seol 2005; Seol and Shephard 2006]. And if multiple residence parts have the same least number of partition object entities, the part with smaller id is the owning part. For 2D meshes and partition models illustrated in Figure 10, the bigger circle and thicker lines denote the owning part of partition model entities. The owner of M_5^0 of Figure 10(a) (resp. (c)) is $M_5^0@P_0$ (resp. $M_5^0@P_3$). The owner of M_4^1 of in Figure 10(a) (resp. (c)) is $M_4^1@P_0$ (resp. $M_4^1@P_1$). 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 tetrahedral, quadrilateral, hexahedral, prism (wedge) and pyramid. For a face, PUMI supports triangle and quadratic.
- *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.
- *adjacency* - PUMI supports one-level mesh representation where 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* - for a part boundary entity, *a remote copy* is the memory location of a mesh entity duplicated on non-local part. Part boundary entity maintains a map of *(remote part id, remote copy)* along with partition classification.
- *ghost copy* - when an owned mesh entity is ghosted on other parts, the memory location of non-owned duplicate copy is termed *ghost copy*. The owner copy maintains a map of *(ghost part id, ghost copy)* and a ghost copy maintains owning part id and the memory address of owner copy.
- *match copy* - when a mesh entity is matched to other entity, it maintains a multimap of *(match part id, match copy)* where *a match copy* is the memory location of a matched entity handle. Note an entity can have one or more match copies per part. If an entity is matched to a part boundary entity, it must be matched to all the remote copies.

4. PARALLEL MESH SERVICE ALGORITHMS

This section presents algorithms for the parallel mesh services of mesh migration, n-to-m partitioning and cumulative n-layer ghosting.

4.1. 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. To migrate entities to other parts, the destination part set of mesh entities must be specified before migrating the mesh entities. The residence part set equation implies that once the destination part id of a M_i^d that is not on its boundary of any other mesh entities is set, the other entities needed to move are determined by the entities it bounds. Thus, a mesh entity that is not on the boundary of any higher order mesh entities is the basic unit to assign the destination part id in migrating mesh entities.

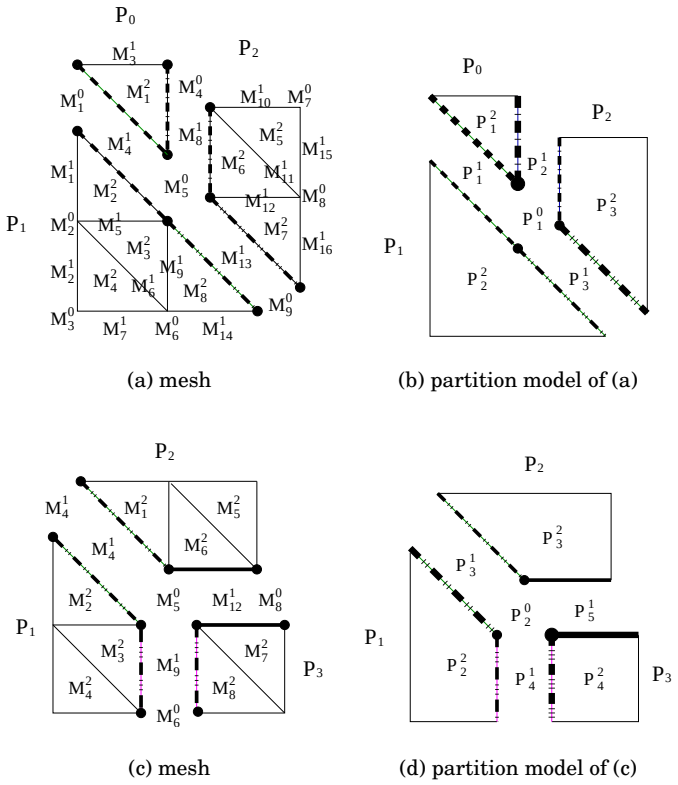


Fig. 10. Distributed mesh and owning part stored in partition model entities based on poor-to-rich rule [Seol 2005; Seol and Shephard 2006]

Definition 4.1. Partition object
The basic unit to which a destination part id is assigned in migration that is a mesh entity not being part of the boundary of any higher order mesh entities. In a 3D mesh, this includes all mesh regions, the mesh faces not bounded by any mesh regions, the mesh edges not bounded by any mesh faces or regions, and mesh vertices not bounded by any mesh edges, faces or regions. A set of unique mesh entities referred as a *p-set* can also be a partition object if designated to be always migrated as a unit.

In case of a manifold model, partition objects are all mesh regions in 3D and all mesh faces in 2D. In case of a non-manifold model, the careful lookup for entities not being bounded is required over the entities of one specific dimension. In Figure 5(a), all mesh faces are partition object.

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 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, Algorithm 2, are the following:

(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 a 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.
(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 performs 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 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 the mesh is matched, the match copy should be also updated based on new partitioning topology. So as illustrated in Algorithm 5, the *remote* match copies are updated in the last stage 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 shows 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.

4.2. Graph-based Mesh Partitioning

Graph-based mesh partitioning consists of four steps *(i)* transformating the unstructured mesh data structures discussed in Section 3.4 to the graph vertex and edge structures needed by Zoltan [Devine et al. 2014; Zoltan Web 2014] *(ii)* running Zoltan *(iii)* transforming the output of Zoltan (graph vertices and destination part ids) to partition object (mesh element and p-set) and part id maps required by the migration procedure

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

```
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
// ent_to_exchg is 2D array of entities where ent_to_exchg[i] contains entities of type i
collect_ent_to_exchange(M, ent_to_migr, ent_to_exchg); // see Algorithm 3
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
for i ← 0, d do
    exchange_ent(ent_to_exchg[i], pset_map); // see Algorithm 4
end
for i ← d, 0 do
    remove_ent(M, ent_to_remove[i]); // see Algorithm 6
end
for each [S_i, P_src, P_dest] ∈ pset_to_migr do
    delete S_i from P_src;
end
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
```

discussed in Section 4.1 *(iv)* running mesh migration. Graph vertices are defined by partition objects (mesh elements or and 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

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 $\mathscr{P}[M_j^q]$=$\emptyset$ *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*;
    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

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 [Devine et al. 2014].

The pseudo-code for running Zoltan graph-based partitioning is listed in Algorithm 7.

(1) A global numbering of partition object entities and p-sets is created by *set_gid*(M). The function *get_gid* returns the global id generated. For meshes containing over two billion elements an eight byte integer is used to prevent overflow.

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_match(ent_to_exchg); // See Algorithm 5

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*;
  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*;
    if M_j^q *is remote copy of* M_i^d **then** *continue*;
    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* $M_k^p@P_{match} \in$ *message* **do**
    if $M_j^q = M_k^p$ **then**
      $M_j^q@P_i \rightarrow$ *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**
        $M_j^q@P_i \rightarrow$ *add_match*($M_k^p@P_{match}$);
      end
    end
  end
end

(2) Using global numbering, *setup_elem_edge*() and *setup_pset_remote_edge*(), illustrated in Algorithms 8, and 9 respectively, prepare for the creation of graph edges that span inter-part boundaries by localizing off-part global ids. *setup_elem_edge*() iterates over elements bounded by a d-1 partition model entity and sends the global id, and the pointer to the dimension d-1 mesh entity that is shared by both elements, the remote copy, to the neighboring part via a call to *pack_remote_copy_gid*(), presented in Algorithm 10. Likewise *setup_pset_remote_edge*() iterates over the faces on the exterior of the p-set and calls *pack_remote_copy_gid*(). In both procedures, the receiving part calls *unpack_remote_copy_gid*() listed in Algorithm 11 to setup an edge to a p-set or an edge to an element. An edge to a p-set is created if the mesh element bounded by the dimension d-1 mesh entity, the remote copy $remote\_copy$, belongs to a p-set. If the bounded element does not belong to a p-set an edge to an element is created.
(3) Before Zoltan runs, the Zoltan graph edge and vertex creation callbacks are registered and various Zoltan control parameters are set in *setup_zoltan*(M). Algorithms 12 and 13 are the respective callbacks for creating the contiguous array of vertex and edge objects. The array of vertex objects, vtx_gid in Algorithm 12, is populated by simply writing the global ids of on-part mesh elements not belonging to p-sets, and then on-part p-sets. Populating the $edge\_gid$ in Algorithm 13 follows the iteration order of Algorithm 12. For each mesh element that does not belong to a p-set the global ids of on-part adjacent elements, followed by off-part adjacent elements and p-sets, are written to $edge\_gid$. To write p-set edges' global id and weight, the $neighbor$ map is iterated over where $neighbor[i]$ contains the integral weight value of edge with global id i. Before looping over the $neighbor$ map, local p-set adjacencies are added via the call to *add_pset_local_edge*(), presented in Algorithm 14. Additional callbacks are required by Zoltan to determine the length of

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*;
  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

the arrays, and assignment of graph vertices to MPI process ids, but the details of their implementation are not discussed here.

(4) The requested Zoltan partitioning procedure is executed with the call to *run_zoltan*(ztn) and the registered callback functions invoked.
(5) The output of the partitioning is an assignment of graph vertices to part ids. The conversion of this information is by *get_migr_input*(M, ztn, ent_to_migr, $pset\_to\_migr$) to the element and p-set maps is required by the migration procedure discussed in Section 4.1.
(6) After the migration input is created, the global ids and tags are no longer needed and are destroyed with their respective destructor calls in.
(7) Lastly, the migration is performed with the partition object and destination part id maps, ent_to_migr and $pset\_to\_migr$.

4.3. Mesh Ghosting

Ghosting procedure localizes the mesh entities and data on neighboring remote part for the purpose of minimizing inter-process communications in specific mesh operations requiring data from mesh entities on remote processes [Lawlor et al. 2006; iMeshP Web 2014]. The inputs to the ghosting procedure are *(i)* ghost type g ($0<g\leq$*mesh dimension*) for entity type to be ghosted, *(ii)* bridge type b ($b < g$ and $b \neq g$), *(iii)* the number of ghost layers n measured from inter-process part boundary up to the number with which whole part can be ghosted, *(iv)* an integer *inc_copy* to indicate whether to include

ALGORITHM 7: Graph-based Mesh Partitioning

Input: Mesh instance M
Output: Mesh and partition model updated with new partitioning based on Zoltan output
set_gid(M); // create gloabal id for partition object entities and p-sets
$elem\_tag \leftarrow$ *create_int_tag*(M,"elem_neighbor") ;
setup_elem_edge(M, $elem_neighbor$); // see Algorithm 8
$pset\_tag \leftarrow$ *create_ptr_tag*(M,"pset_neighbor");
setup_pset_local_edge(M, $pset\_tag$);
setup_pset_remote_edge(M, $pset\_tag$, $elem\_tag$); // see Algorithm 9
ztn = *setup_zoltan*(M);
run_zoltan(ztn);
get_migr_input(M, ztn, ent_to_migr, $pset\_to\_migr$);
destroy_tag($elem\_tag$);
destroy_tag($pset\_tag$);
destroy_gid(M);
run_migration(M, ent_to_migr, $pset\_to\_migr$); // see Algorithm 2

ALGORITHM 8: *setup_elem_edge*(M, $elem_neighbor$)

Input: Mesh instance M of dimension d, element graph edge tag $elem\_tag$
Output: Set mesh element graph edge inter-process global ids.
for *each* M_i^{d-1} *on part part boundary* **do**
  if $M_i^{d-1}\{M^d\}_0 \in$ *p-set* **then** *continue*;
  for *each remote copy* $M_i^{d-1}@P_{remote}$ **do**
    $message \leftarrow$ *pack_remote_copy_gid*(M_i^{d-1}, P_{remote}); // see Algorithm 10
    send $message$ to $M_i^{d-1}@P_{remote}$;
  end
end
for *each* $message$ *received on* $M_i^{d-1}@P_{remote}$ **do**
  unpack_remote_copy_gid($message$, $pset\_tag$, $elem\_tag$); // see Algorithm 11
end

ALGORITHM 9: *setup_pset_remote_edge*(M, $pset\_tag$, $elem\_tag$)

Input: Mesh Instance M of dimension d, p-set graph edge tag $pset\_tag$, element graph edge tag $elem\_tag$
for *each* $S_i \in M$ **do**
  for *each* M_i^{d-1} *on the exterior of* S_i **do**
    if M_i^{d-1} *is on part boundary* **then**
      for *each remote copy* $M_i^{d-1}@P_{remote}$ **do**
        $message \leftarrow$ *pack_remote_copy_gid*(M_i^{d-1}, P_{remote}); // see Algorithm 10
        send $message$ to $M_i^{d-1}@P_{remote}$;
      end
    end
  end
end
for *each* $message$ *received on* $M_i^{d-1}@P_{remote}$ **do**
  unpack_remote_copy_gid($message$, $pset\_tag$, $elem\_tag$); // see Algorithm 11
end

ALGORITHM 10: *pack_remote_copy_gid*(M_i^{d-1}, P_{remote})

Input: Mesh entity M_i^{d-1}, remote part id P_{remote}
$gid \leftarrow$ ***get_gid***($M_i^{d-1}\{M^d\}_0$);
pack($M_i^{d-1}@P_{remote}$, gid);

ALGORITHM 11: *unpack_remote_copy_gid*($message$, $pset\_ntag$, $elem\_tag$)

Input: Message $message$, p-set graph edge tag $pset\_tag$, element graph edge tag $elem\_tag$
$M_i^{d-1} \leftarrow$ ***unpack***($message$);
$gid \leftarrow$ ***unpack***($message$);
if $M_i^d \in S_i$ *where* $M_i^d \in M_i^{d-1}\{M^d\}$ **then**
  $neighbor \leftarrow$ ***get_tag_data***($pset\_tag$, S_i); // get map<int,int> attached to S_i
  ++$neighbor$[gid];
else
  set_int_tag($elem\_tag$, M_i^{d-1}, gid); // attach gid to M_i^{d-1}
end

ALGORITHM 12: *construct_vtx_graph*(M, $pset\_tag$, $elem\_tag$, vtx_gid)

Input: Mesh Instance M of dimension d, p-set graph edge tag $pset\_tag$, element graph edge tag $elem\_tag$
Output: 1D array vtx_gid filled with vertex global id
$i \leftarrow 0$;
for *each* $M_i^d \in M$ **do**
  if $M_i^d \in$ ***p-set*** **then** *continue*;
  $vtx\_gid[i{+}{+}] \leftarrow$ ***get_gid***(M_i^d);
end
for *each* $S_i \in M$ **do**
  $vtx\_gid[i{+}{+}] \leftarrow$ ***get_gid***(S_i);
end

non-owned bridge entities (1) or not (0) in ghosting candidate computation [iMeshP Web 2014]. If $n = 1$, the entities to be ghosted (shortly, *ghosting candidate*) are a set of entities $\{M_i^b\}\{M^g\}$ where M_i^b is a part boundary entity. If $n > 1$, ghosting candidates are computed similar to 2^{nd}-order adjacencies, i.e. $\{M_i^d\{M^b\}\{M^g\}\}$ where M_i^d is a ghosting candidate in $n-1$ layer ghosting. n

For a four-part distributed 2D mesh shown in Figure 11(a), Figure 11(b) illustrates the mesh after one-layer ghosting with bridge edges and ghost faces. Since ghost entities are stored explicitly in the mesh, all mesh entity functions can be used with ghost copy. This includes traversal, tagging, entity set, and various interrogations such as geometric classification, adjacencies, remote copies, residence part set, owning part id, partition classification, etc..

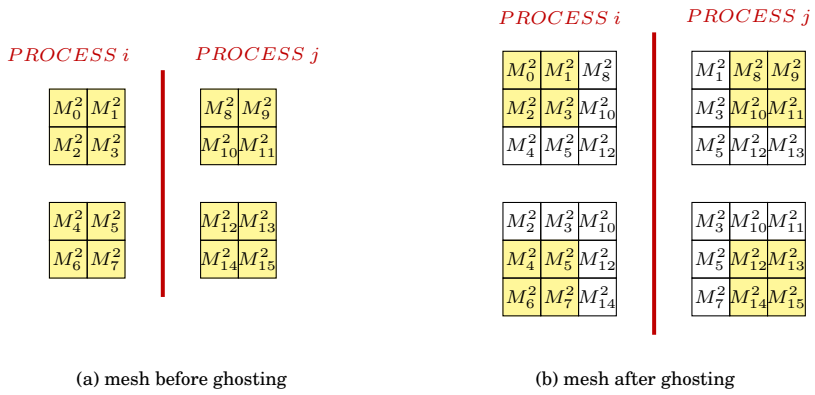
Algorithms 15, 16 and 17 describe general *n*-layer ghosting procedure designed to construct ghost layers cumulatively. Therefore multiple calls build more ghost copies in addition to existing ghost copies. Although not covered in this section, ghost copy traversal and ghost deletion are also supported in the API.

ALGORITHM 13: *construct_edge_gid*(M, $pset\_tag$, $elem\_tag$, $edge\_gid$, $edge\_weight$)

Input: Mesh Instance M of dimension d, p-set graph edge tag $pset\_tag$, element graph edge tag $elem\_tag$
Output: 1D arrays $edge\_gid$ and $edge\_weight$ filled with edge global id and weight, respectively
$i \leftarrow 0$;
for *each* $M_i^d \in M$ **do**
  if $M_i^d \in$ *p-set* **then** *continue*;
  // on-part adjacencies
  for *each* $M_j^d \in \{M_i^d\{M^{d-1}\{M^d\}\}\}, M_j^d \neq M_i^d$ **do**
    if $M_j^d \in S_i$ **then**
      $edge\_gid[i] \leftarrow$ *get_gid*(S_i);
    else
      $edge\_gid[i] \leftarrow$ *get_gid*(M_j^d);
    end
    $edge\_weight[i] \leftarrow 1$;
    ++i;
  end
  // off-part adjacencies
  for *each* M_j^{d-1} *on part boundary,* $M_j^{d-1} \in \{M_i^d\{M^{d-1}\}\}$ **do**
    $edge\_gid[i] \leftarrow$ *get_int_tag*($elem\_tag$, M_j^{d-1});
    $edge\_weight[i] \leftarrow 1$;
    ++i;
  end
end
for *each* $S_i \in M$ **do**
  $neighbor \leftarrow$ *get_tag_data*($pset\_tag$, S_i); // get map<int,int> attached to S_i
  add_pset_local_edge(S_i, $neighbor$); // see Algorithm 14
  for *each* $\{gid,weight\} \in neighbor$ **do**
    $edge\_gid[i] \leftarrow gid$;
    $edge\_weight[i] \leftarrow weight$;
    ++i;
  end
end

ALGORITHM 14: *add_pset_local_edge*(S_i, $neighbor$)

Input: p-set S_i, map<int,int> $neighbor$
Output: $neighbor$ updated with on-part p-set adjacencies
for *each* M_i^{d-1} *on the exterior of* S_i **do**
  if M_i^{d-1} *is not on part boundary* **then**
    for *each* $M_j^d \leftarrow M_i^{d-1}\{M^d\}, M_j^d \notin S_i$ **do**
      $adj\_gid \leftarrow$ *get_gid*(M_j^d);
      if $M_j^d \in S_j \& S_j \neq S_i$ **then**
        $adj\_gid \leftarrow$ *get_gid*(S_j);
      end
      ++$neighbor$[adj_gid];
    end
  end
end



(a) mesh before ghosting (b) mesh after ghosting

Fig. 11. Four-part distributed 2D mesh before/after 1-layer ghosting (*ghost type*=2, *bridge type*=1, *inc_copy*=1)

5. API

The PUMI libraries interact with each other and other application programs through well-defined C++ API consisting of predefined data types for class definitions (Table I), enumerated types for named values, and routines for predefined data types.

- *pumi.h*: pre-defined and enumerated data types shared by all libraries are provided.
- *pumi_errorcode.h*: ITAPS-compliant error codes [ITAPS Web 2014; Ollivier-Gooch et al. 2010].
- *pumi_comm.h*: parallel control utility functions defined in *pcu* library.
- *pumi_util.h*: common utility functions defined in *pumi_util* library.
- *pumi_geom.h*: geometric model and mesh model related functions defined in *pumi_geom* library.
- *pumi_geom_acis.h*: Acis-specific geometric model functions defined in *pumi_geom_acis* library.
- *pumi_geom_geomsim.h*: GeomSim-specific geometric model functions defined in *pumi_geom_geomsim* library.
- *pumi_geom_parasolid.h*: Parasolid-specific geometric model functions defined in *pumi_geom_parasolid* library.
- *pumi_mesh.h*: mesh and partition model related functions defined in *pumi_mesh* library.

The API naming conventions to enhance readability and self-descriptability are *(i)* function name consists of three words connected with '_', *(ii)* the first word is "`PUMI`", *(iii)* the second word represents the data object where the API function performs. Table II illustrates some of data objects used as the second word, *(iv)* the first letter of a word is upper case and the rest is lower case, *(v)* the second word is omitted for system-wide function, *(vi)* the third word describes the operation performed. If the API function returns an error code indicating the operation succeeded (zero) or not (positive), the third word starts with a verb. Otherwise, the third word starts with a noun. The PUMI user's guide [PUMI Web 2014] provides detailed discussions on each data type and routine in API.

ALGORITHM 15: 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
// ent_to_ghost is 2D array of entities where $ent\_to\_ghost[i]$ contains entities of type i
// step 1: compute entities to ghost of type g
compute_ent_to_ghost(M, g, b, n, inc_copy, $ent\_to\_ghost[g]$); // see Algorithm 16
// step 2: collect entities to ghost of 0 to $g-1$ type
collect_ent_to_ghost(M, ent_to_ghost); // see Algorithm 17
// step 3: 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 4: 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

6. EXAMPLE PROGRAMS

6.1. Parallel model/mesh file I/O

This program loads mesh-based geometric model and mesh files

```
int main(int argc, char *argv[])
{
  char in_mesh_file[1024], out_mesh_file [1024];

  PUMI_Init(MPI_COMM_WORLD); // initialize services including MPI

  pGeomMdl model; // create geometric model instance
  PUMI_Geom_RegisterMesh(); // register mesh model

  PUMI_Geom_CreateModel(PUMI_MESHMODEL, model);
  PUMI_Geom_LoadFromFile(model, "geom.dmg"); // load mesh model file geom.dmg
```

ALGORITHM 16: *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*;
  $\mathscr{P}[M_i^b] \leftarrow \emptyset$;
  // computing 1^{st} layer entities to ghost
  for *each* $M_j^g \in \{M_i^b\{M^g\}\}$ **do**
    if M_j^g *is ghost copy* **then** *continue*;
    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
  // computing $2-n^{th}$ layer entities to ghost ($n > 1$)
  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*;
        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

```
pMeshMdl mesh;
PUMI_Mesh_Create(model, mesh); // create mesh instance

int num_proc=PUMI_CommSize();
// if num_proc>1, load distributed mesh in Simmetrix format,
// otherwise, load serial mesh
PUMI_Mesh_LoadFromFile(mesh, argv[1], num_proc-1);

PUMI_Geom_DspStat(model); // display model statistics
PUMI_Mesh_DspSize(mesh);  // display mesh size information per process

int isValid;
PUMI_Mesh_Verify(mesh, &isValid); // verify mesh and partition model

// export mesh into exodusii format
PUMI_Mesh_WriteToFile(mesh, argv[2], PUMI_CommSize()-1, "exodusii");

PUMI_Geom_Del(model); // delete model instance
PUMI_Mesh_Del(mesh);  // delete mesh instance
```

ALGORITHM 17: *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
    $\mathscr{P}[M_j^d] = \mathscr{P}[M_i^g] \cup \mathscr{P}[M_j^d]$;
  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
  $\mathscr{P}[M_k^d] = \mathscr{P}[M_k^d] \cup \mathscr{P}[M_i^d @ P_{src}]$;
end

Table I. Some of Pre-defined Data Types

pGeomMdl	geometric model instance
pGeomEnt	geometric entity handle
pGeomVtx	geometric vertex handle
pGeomEdge	geometric edge handle
pGeomFace	geometric face handle
pGeomRgn	geometric region handle
pGeomEntUse	geometric entity use handle
pMeshMdl	mesh instance
pPart	part handle
pMeshEnt	mesh entity handle
pMeshVtx	mesh vertex handle
pMeshEdge	mesh edge handle
pMeshFace	mesh face handle
pMeshRgn	mesh region handle
pNode	higher order node
pEntSet	entity set handle (l-set, s-set or p-set)
pPartSet	part set handle
pTag	tag handle
pGeomEntIter	iterator traversing over geometric entities in geometric model
pMeshSetIter	iterator traversing over entity sets in mesh
pPartSetIter	iterator traversing over entity sets in part
pPartEntIter	iterator traversing over mesh entities in part
pSetEntIter	iterator traversing over mesh entities in set (mesh set or part set)

Note: a prefix *p* to indicate the pointer data type.

```
  PUMI_Finalize(); // finalize services
  return PUMI_SUCCESS;
}
```

Table II. Operation Object in Second Word of API Name

$Thrd$	the API performs thread management
$Comm$	the API performs MPI communications
Tag	the API performs on a tag
$Geom$	the API is performed on a geometric model
$Mesh$	the API is performed on a mesh or all parts in the mesh
$Part$	the API is performed on a part
$GeomEnt$	the API is performed on a geometric model entity. If the API performs on a specific entity type, it uses one of four - $GeomVtx$, $GeomEdge$, $GeomFace$, $GeomRgn$
$MeshEnt$	the API is performed on a mesh entity. If the API performs on a specific entity type, it uses one of the following four - $MeshVtx$, $MeshEdge$, $MeshFace$, $MeshRgn$
$Node$	the API is performed on a higher-order node
$EntSet$	the API is performed on entity set (l-set, s-set, p-set)
$PartSet$	the API is performed on p-set
$Iter$	the API is performed on an iterator with designated condition such as entity type, topology, etc. The iterator types available in PUMI are *(i)* $GeomEntIter$: geometric entity traversal, *(ii)* $PartEntIter$: mesh entity traversal in part, *(iii)* $SetEntIter$: mesh entity traversal in entity set, *(iv)* $MeshSetIter$: $l-$ or $s-set$ traversal in mesh, *(v)* $PartSetIter$: *p-set* traversal in part

Note: see *http://scorec.rpi.edu/wiki/Abbriviations* for all abbreviations.

6.2. Tag management

This program demonstrates various tag handle management such as creation, attribute (size, type, name) retrieval, deletion, etc. Section 6.9 illustrates a program to attach various tag data to a part handle. A function to turn on/off automatic tag data migration with mesh entities is presented in Section 6.16.

```
template <class T>
void TEST_TAG(pMeshMdl mesh, pTag tag, char* in_name,
              int in_type, int in_size)
{
  const char *tag_name;
  int tag_type, tag_size, tag_byte;
  PUMI_Tag_GetName(mesh, tag, &tag_name);
  PUMI_Tag_GetType(mesh, tag, &tag_type);
  PUMI_Tag_GetSize(mesh, tag, &tag_size);
  PUMI_Tag_GetByte(mesh, tag, &tag_byte);
  assert(!strcmp(tag_name, in_name) && tag_type==in_type
         && tag_size==in_size && tag_byte==sizeof(T)*tag_size);
}

int main()
{
   ...
  pTag int_tag, dbl_tag, ent_tag, ptrarr_tag, clone_tag;
  int tag_type, tag_size, exist, in_use;
  char tag_name[256];

  PUMI_Mesh_CreateTag(mesh, "pointer array", PUMI_PTR, 7,  ptrarr_tag);
  PUMI_Mesh_CreateTag(mesh, "integer", PUMI_INT, 1, int_tag);
  PUMI_Mesh_CreateTag(mesh, "double", PUMI_DBL, 1, dbl_tag);
  PUMI_Mesh_CreateTag(mesh, "entity", PUMI_ENT, 1, ent_tag);

  // verifying tag info
  TEST_TAG<char>(mesh, ptrarr_tag, "pointer array", PUMI_PTR, 7);
  TEST_TAG<int>(mesh, int_tag, "integer", PUMI_INT, 1);
  TEST_TAG<double>(mesh, dbl_tag, "double", PUMI_DBL, 1);
  TEST_TAG<pMeshEnt>(mesh, ent_tag, "entity", PUMI_ENT, 1);
```

```
  PUMI_Mesh_HasTag(mesh, int_tag, &exist);
  PUMI_Mesh_FindTag(mesh, "pointer array", clone_tag);
  std::vector<pTag> tag_handles;
  PUMI_Mesh_GetTag(mesh, tag_handles); // get all tag handles created in mesh
  // tag handle is in use if tag data is attached to part/entity/set with it
  PUMI_Mesh_IsTagInUse(mesh, ptrarr_tag, &in_use);
  assert(exist && clone_tag==ptrarr_tag && tag_handles.size()==4 && !in_use);

  // if tag handle is not in use, delete tag handle from mesh instance
  PUMI_Mesh_DelTag(mesh, ptrarr_tag, 0);

  // forcibly delete tag data associated with tag handle, then delete tag handle
  PUMI_Mesh_DelTag(mesh, int_tag, 1);
  ...
}
```

6.3. Part information

This program verifies the number of entities per type in the first part handle of the mesh.

```
  int partDim, numEnt, numVt, numEg, numFc, numRg;
  pPart part;
  PUMI_Mesh_GetPart(mesh, 0, part); // get the first part handle in the mesh

  PUMI_Part_GetNumEnt(part, PUMI_VERTEX, PUMI_ALLTOPO, &numVt);
  PUMI_Part_GetNumEnt(part, PUMI_EDGE, PUMI_ALLTOPO, &numEg);
  PUMI_Part_GetNumEnt(part, PUMI_FACE, PUMI_ALLTOPO, &numFc);
  PUMI_Part_GetNumEnt(part, PUMI_REGION, PUMI_ALLTOPO, &numRg);

  PUMI_Part_GetNumEnt(part, PUMI_ALLTYPE,  PUMI_ALLTOPO, &numEnt);
  assert(numEnt==numVt+numEg+numFc+numRg);
```

6.4. Part iteration for entities

This program traverses mesh regions in a part.

```
  pMeshEnt ent;
  pPartEntIter entIter;

  PUMI_PartEntIter_Init(part, PUMI_REGION, PUMI_ALLTOPO, entIter);
  while (PUMI_PartEntIter_GetNext(entIter, ent)==PUMI_SUCCESS)
  {
    ...
  } // while
  PUMI_PartEntIter_Del(entIter);
```

6.5. Mesh entity adjacency

This program compares the output of *PUMI_MeshEnt_GetAdj* and *PUMI_MeshEnt_GetNumAdj*.

```
  pPartEntIter entIter;
  vector<pMeshEnt> adjEnts;
  for (int type=PUMI_VERTEX; i<=PUMI_REGION; ++i)
  {
    PUMI_PartEntIter_Init(part, type,  PUMI_ALLTOPO, entIter);
    while (PUMI_PartEntIter_GetNext(entIter, ent)==PUMI_SUCCESS)
    {
```

```
      for (int j=PUMI_VERTEX; j<PUMI_REGION; ++j)
      {
        if (i==j) continue;
        adjEnts.clear();
        PUMI_MeshEnt_GetAdj(ent, j, 1, adjEnts);
        PUMI_MeshEnt_GetNumAdj(ent, j, &numAdj);
        assert(adjEnts.size()==((size_t)numAdj));
      }
    }
    PUMI_PartEntIter_Del(entIter);
  }
```

6.6. Mesh entity creation/deletion

Mesh entity M_i^d is created in terms of the lower order mesh entities of dimension that bound it. Since the current PUMI maintains one-level representation where M_i^d maintains the link to one-level upward and downward adjacent entities, any non-existing $\{M_i^d\{M_j\}\}$ is created automatically on M_i^d creation (d>0). For mesh entity deletion, M_i^d can be removed only if no higher order adjacent mesh entity exists. Therefore, the mesh entities should be removed in $region \rightarrow face \rightarrow edge \rightarrow vertex$ order. In modifying distributed mesh using individual entity creation/deletion, the remote copies and partition model should be modified properly as well by the application.

The following code demonstrates the steps to modify 3D distributed mesh - *(i)* mesh entity creation, *(ii)* remote copy update, *(iii)* partition model and tagged data update.

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

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

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

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

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

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

pMeshEnt new_tet;
if (PUMI_MeshEnt_Find(PUMI_TET, new_face, 4, new_tet)==PUMI_MeshEntITY_NOT_FOUND)
  PUMI_MeshRgn_Create(part, (pGeomEnt)NULL, PUMI_TET, new_face, 4, new_tet);

// for any part boundary entity, update remote copies
PUMI_MeshEnt_SetRmt(part_bdry_ent, part_id, remote_copy_ent);
...
// based on new partitioning topology, update partition classification
PUMI_Mesh_SyncPtn(mesh);
// for tagged data on part boundary entity, unify its value with owner copy's data
PUMI_Tag_SyncPtn(mesh, tag_handle, entity_type);
```

6.7. Reverse geometric clssification retrieval

This program retrieves the mesh entities classified on geometric vertices.

```
pGeomModel geomModel;
PUMI_Mesh_GetGeomMdl(mesh, geomModel); // get geometric model associated with mesh

pGeomVtx gVtx;
pMeshEnt mEnt;
pGeomEntIter gVtxIter;
PUMI_GeomEntIter_Init(geomModel, gVtxIter);
while (PUMI_GeomEntIter_GetNext(gVtxIter, gVtx)==PUMI_SUCCESS)
{
  pPartEntIter revIter;
  PUMI_PartEntIter_InitRevClas(part, (pGEntity)gVtx, revIter);
  while (PUMI_PartEntIter_GetNext(revIter, mEnt)==PUMI_SUCCESS)
  {
    ...
  }
  PUMI_PartEntIter_Del(revIter);
}
PUMI_GeomEntIter_Del(gVtxIter);
```

6.8. Part boundary iteration and interrogation

This program traverses part boundary entities per dimension (vertex to face).

```
pMeshEnt ent;
int partDim;
PUMI_Part_GetDim(part, &partDim);  // get part dimension

for (int type = 0; type < partDim; ++type)
{
  pPartEntIter pbiter;
  PUMI_PartEntIter_InitPartBdry(part, -1, type, pbIter);
  while (PUMI_PartEntIter_GetNext(pbIter, ent)==PUMI_SUCCESS)
  {
    ...
  }
  PUMI_PartEntIter_Del(pbIter);
}
```

6.9. Part tagging

This program illustrates tagging routines for part handles such as setting, getting and deleting tag data associated with a part handle. The tagging routines for entity set and entity handles are named alike.

```
pMeshEnt ent;
// get the first vertex in the part
PUMI_PartEntIter_Init(part, PUMI_VERTEX, PUMI_POINT, iter);
PUMI_PartEntIter_GetNext(iter, ent);
PUMI_PartEntIter_Del(iter);

PUMI_Part_SetIntTag(part, int_tag, 1000);
PUMI_Part_SetDblTag(part, dbl_tag, 1000.37);
PUMI_Part_SetEntTag(part, ent_tag, ent);
int tag_size;
PUMI_Tag_GetSize(ptrarr_tag,&tag_size);
```

```
void** ptr_data = new void*[tag_size];
// fill ptr_data
PUMI_Part_SetPtrArrTag(part, ptrarr_tag, ptr_data);


PUMI_Part_GetIntTag(part, int_tag, &int_data);
PUMI_Part_GetDblTag(part, dbl_tag, &dbl_data);
PUMI_Part_GetEntTag(part, ent_tag, &ent_data);
void** ptr_data_back = new void*[tag_size];
PUMI_Part_GetPtrArrTag(part, ptrarr_tag, ptr_data_back);

PUMI_Part_DelTag(part, int_tag);
PUMI_Part_DelTag(part, dbl_tag);
PUMI_Part_DelTag(part, ent_tag);
PUMI_Part_DelTag(part, ptrarr_tag);
```

6.10. Entity set traversal

This program creates a p-set and traverses entities in the set.

```
pEntSet mySet, set;
PUMI_Set_Create(mesh, part, PUMI_PSET, mySet);
PUMI_PartSetIter_Init(part, psetIter);
while (PUMI_PartSetIter_GetNext(part, psetIter, set)==PUMI_SUCCESS)
{
  ...
}
iterEnd = PUMI_PartSetIter_Reset(part, psetIter);
PUMI_Set_Del(mesh, part, mySet);
```

6.11. Entity set management

This program illustrates various set manipulation operations such as adding entities, getting the size (the number of entities in the set), clearing (removing all entities) and checking emptiness.

```
PUMI_PartEntIter_Reset(iter);
while (PUMI_PartEntIter_GetNext(iter, ent)==PUMI_SUCCESS)
{
  PUMI_Set_AddEnt(l_set, ent);
  PUMI_Set_AddEnt(l_set, ent);
  PUMI_Set_AddEnt(s_set, ent);
  PUMI_Set_AddEnt(s_set, ent);
}
PUMI_Set_GetNumEnt (l_set, &l_num);
PUMI_Set_GetNumEnt (s_set, &s_num);

PUMI_Set_Clr(l_set);
PUMI_Set_Clr(s_set);
PUMI_Set_IsEmpty(l_set, &l_set_empty);
PUMI_Set_IsEmpty(s_set, &s_set_empty);
assert(l_num==num_type[0]*2 && s_num==num_type[0] && l_set_empty && s_set_empty);
```

6.12. Entity set iterator

This program traverses mesh vertices in a set.

```
pSetEntIter esIter;
PUMI_SetEntIter_Init(listEntSet, PUMI_ALLTYPE, PUMI_POINT, esIter);
while (PUMI_SetEntIter_GetNext(esIter, ent)==PUMI_SUCCESS)
```

```
{
  ...
}
PUMI_SetEntIter_Reset(esIter);
```

6.13. Entity's owning part and owner copy

This program retrieves the owning part id and owner copy of part boundary entities.

```
pMeshEnt ent, ownerEnt;
int ownerPartID;

pPartEntIter iter;
PUMI_PartEntIter_InitPartBdry(part, PUMI_NONE, PUMI_ALLTYPE, PUMI_ALLTOPO, iter);
while (PUMI_PartEntIter_GetNext(iter, ent)==PUMI_SUCCESS)
{
  // get entity's owning part id,
  PUMI_MeshEnt_GetOwnPartID(ent, part, &ownerPartID);
  // get owner copy
  PUMI_MeshEnt_GetOwnEnt(ent, part, ownerEnt);
}
// delete part boundary iterator
PUMI_PartEntIter_Del(iter);
```

6.14. Remote copies

This program performs various remote copy operators such as retrieval, modification, and getting the size.

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

PUMI_PartEntIter_InitPartBdry(part,-1, 1, iter);
while(PUMI_PartEntIter_GetNext(iter, ent)==PUMI_SUCCESS)
{
  // get all remote copies of the part boundary entity
  vecRmtCpy.clear();
  PUMI_MeshEnt_GetAllRmt(ent, vecRmtCpy);

  // iterate through all remote copies
  for (itVec = vecRmtCpy.begin(); itVec != vecRmtCpy.end(); ++itVec)
  {
    pid = itVec->first;
    rmt = itVec->second;
    // there should exist a remote copy on remote part then, crosscheck with function
    // GetRmt operation for getting remote copy on a target part id
    PUMI_MeshEnt_GetRmt(ent, pid, rmtCopy);
    assert(rmtCopy==rmt);
  }

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

  // clear all remote copies
  PUMI_MeshEnt_ClrRmt(ent);
```

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

  // get the number of remote copies again
  PUMI_MeshEnt_GetNumRmt(ent, &numCopiesNew);
  assert(numCopiesNew==numCopies);
}
PUMI_PartEntIter_Del(iter);   // delete part boundary iterator
```

6.15. Global mesh re-distribution on multiple parts

This program re-distributes the mesh with increasing number of parts per process (from 2 to 20) and export distributed mesh into files.

```
for (int numPart=2; numPart<20; ++numPart)
{
  PUMI_Mesh_SetNumPart(mesh, numPart);  // set #parts per mesh instance
  PUMI_Mesh_GlobPtn(mesh); // do mesh re-partitioning

  char out_file[256];
  sprintf(out_file,"%dpart_p.sms", numPart, "pumi");  // export each part in file
}
```

6.16. Mesh migration

This program picks partition object entities randomly and migrate them to random destination part.

```
pMeshEnt ent;
pPartEntIter iter;
int numProc, procID, meshDim, isValid, numPart;
pPart part;

PUMI_Mesh_GetDim(mesh, &meshDim);
PUMI_Mesh_GetNumPart(mesh, &numPart);
std::map<pMeshEnt, std::pair<int, int> > EntToMigr;
std::map<pEntSet, std::pair<int, int> > SetToMigr;

for (int i=0; i<numPart; ++i)
{
  PUMI_Mesh_GetPart(mesh, i, part); // get i'th local part on mesh instance
  PUMI_PartEntIter_Init(part, meshDim, PUMI_ALLTOPO, iter);
  while (PUMI_PartEntIter_GetNext(iter, ent)==PUMI_SUCCESS)
  {
    if (rand()%3==1)   // Randomly select partition objects for migration
      EntToMigr.insert(std::map<pMeshEnt, std::pair<int, int> >::value_type
       (ent, std::make_pair(PUMI_Part_ID(*part_it), rand()%(PUMI_CommSize()*numPart))));
  }
  PUMI_PartEntIter_Del (iter);
}
// turn on automatic tag migration for tag data attached to regions with "int_tag" handle
PUMI_Mesh_SetAutoTagMigrOn(mesh, int_tag, PUMI_REGION);
```

```
// set "# entity exchange stages" 2 to reduce memory usage during entity exchange
// while sacrificing migration time (default: 1). Set the debug_level to 0.
PUMI_Mesh_SetMigrParam(mesh, 2, 0);

PUMI_Mesh_Migr(meshEntToMigr, SetToMigr); // do migration

// turn off automatic tag migration for "int_tag" handle with regions
PUMI_Mesh_SetAutoTagMigrOff(mesh, int_tag, PUMI_REGION);

PUMI_Mesh_Verify(mesh, &isValid); // verify mesh
```

6.17. Ghosting

This program creates ghost regions up to 3 layers cumulatively and destroys at the end.

```
// cumulatively, construct 3 ghost layers with including copy
int brg_dim=PUMI_VERTEX;
int ghost_dim=PUMI_REGION;
int include_copy=1;
for (int num_layer=1; num_layer<=3; ++num_layer)
  if (PUMI_Mesh_CreateGhost(mesh, brg_dim, ghost_dim, num_layer, include_copy))
    if (!PUMI_CommRank()) // only master process prints the message
      std::cout<<"(brg "<<brg_dim<<", ghost "<<ghost_dim<<") ghosting not performed\n";

// get accumulated values of brg_dim, ghost_dim, num_layer, include_copy
std::vector<int> ghost_info;
PUMI_Mesh_GetGhostInfo(mesh, ghost_info);

// delete all ghost layer(s)
PUMI_Mesh_DelGhost(mesh);
```

7. RESULTS

In the performance studies, the IBM BlueGene/Q at the Center for Computational Innovations [CCI Web 2014] 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.

7.1. Parallel Mesh Construction from Node-Element Information

Using the minimum node-element information for each part, the function *PUMI_Mesh_BuildFromArr* constructs a distributed mesh and partition model. The minimum required information is, for each part, five std vectors with global node id, node coordinates, element's part id, element's topology, and consisting nodes' global id's for each element. Using the minimum node-element information on 256 parts, it took 12 seconds to construct 25.6 million element distributed mesh and partition model. f

7.2. Migration

The migration service provided by PUMI is critical to the performance of user applications including load balancing and other parts of adaptivity. It has been designed for per-element speed as well as parallel scalability. One stress test is to have each part migrate ten 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.

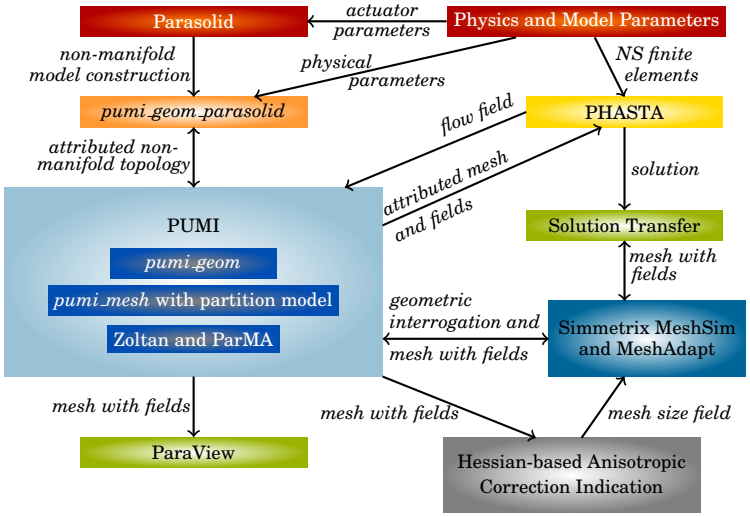


Fig. 12. Workflow of parallel PHASTA adaptive loop

7.3. Ghosting

Running the program in Section 6.17 with 2 million element mesh in 256 parts generated 4.5 million ghosted elements reaching 6.5 million elements in total. It took 6.5 minutes for ghost construction and 5 seconds for ghost destruction.

7.4. File I/O

The philosophy in constructing a workflow or pipeline is to transfer data in-memory within the same process when possible. However, many programs cannot run on the same machine, machine allocations may be too small to accomplish a given task, and for many other reasons workflows usually consist of steps involving state saving to file and reloading that state using a different running program. PUMI's binary file format preserves a parallel partitioned mesh and all its tagged information. It has also been designed with per-element performance and scalability in mind, particularly considering the I/O mechanisms of leading HPC machines. Using the same 1.6 billion element mesh as in Section 7.2, the time to read all 16 thousand binary files in parallel was 40 seconds, and writing them took 22 seconds.

8. APPLICATIONS

8.1. 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 (e.g., 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.

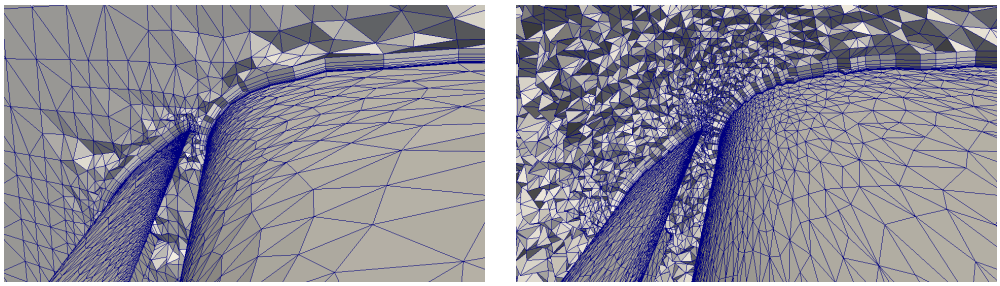


Fig. 13. Cut views of the initial (left) and adapted (right) anisotropic boundary layer meshes for NASA TrapWing [Chitale et al. 2014]

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]. Using this workflow, PHASTA achieved strong scaling in mesh-based computations on up to 786,432 cores using 3,145,728 MPI processes on Mira IBM BlueGene Q with a 92 billion element mesh [Rasquin 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 run 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 APF field migration and local solution transfer procedures [APF Web 2014].

An in-memory coupling supporting a parallel adaptive PHASTA analysis loop [Smith et al. 2014] 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 and supports the control of solver execution, and the interrogation and management of solver data structures.

8.2. 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 2014], 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 [Smith et al. 2014; MeshAdapt Web 2014]. 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.

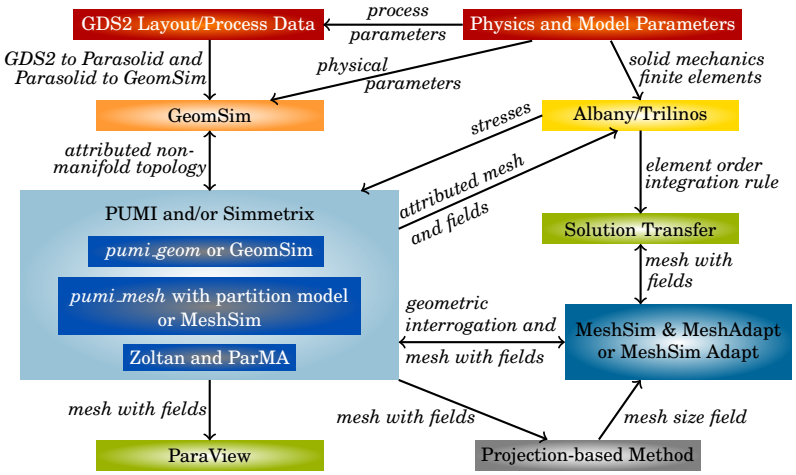


Fig. 14. Workflow of parallel Albany adaptive loop

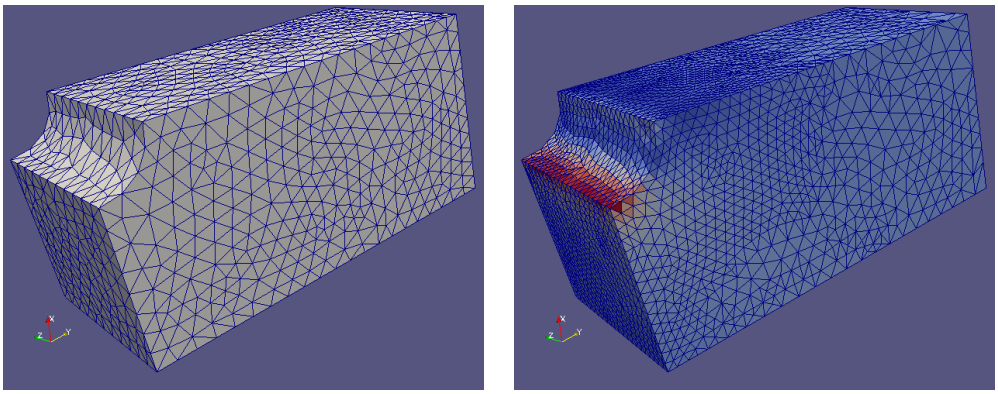


Fig. 15. Initial mesh (left) and adapted mesh showing the stress field that guides adaptivity (right).

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, part of 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.

9. 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 its effective design, data structure, algorithms, API design, example programs, performance results and two applications. The on-going improvement is focused on hybrid programming using

MPI and threading to take full advantage of new generations of high core count nodes [Ibanez et al. 2014]. 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 2014; LLNL HPC Web 2014].

Except for *pumi_geom_acis*, *pumi_geom_geomsim*, and *pumi_geom_parasolid* which directly interact with commercial modeling kernels, all PUMI libraries are open source programs available from the web page *http://www.scorec.rpi.edu/pumi*.

REFERENCES

ACIS Web 2014. 3D ACIS Modeling. (2014). 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 2014. APF: A Parallel Field Library. (2014). http://www.scorec.rpi.edu/apf.

ASCAC Web 2014. ASCAC: Advanced Scientific Commuting Advisory Committee of U.S. Department of Energy. (2014). 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

CCI Web 2014. CCI: Center for Computational Innovations, Rensselaer Polytechnic Institute. (2014). http://cci.rpi.edu.

Waldemar Celes, Glaucio H. Paulino, and Rodrigo Espinha. 2005. 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, Michel Rasquin, Onkar Sahni, Mark S. Shephard, and Kenneth E. Jansen. 2014. Anisotropic Boundary Layer Adaptivity of Multi-Element Wings. In *52nd Aerospace Sciences Meeting (SciTech)*. American Institute of Aeronautics and Astronautics, National Harbor, MD. DOI:http://dx.doi.org/10.2514/6.2014-0117

Karen D. Devine, Vitus Leung, Erik G. Boman, Sivasankaran Rajamanickam, Lee Ann Riesen, and Umit Catalyurek. 2014. Zoltan User's Guide, Version 3.8. (2014). http://www.cs.sandia.gov/Zoltan/ug_html/ug.html.

Vladimir Dyedov, Navamita Ray, Daniel Einstein, Xiangmin Jiao, and Tim 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*, Josep Sarrate and Matthew Staten (Eds.). Springer Berlin Heidelberg, 445–464. DOI:http://dx.doi.org/10.1007/978-3-319-02335-9_25

FASTMath DOE SciDAC Web 2014. FASTMath: Applied Mathematics Algorithms, Tools, and Software for HPC Applications. (2014). http://www.fastmath-scidac.org.

Joe E. Flaherty, Raymond M. Loy, Mark S. Shephard, Boleslow 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 2014. GeomSim: Direct Geometry Access. (2014). 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

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

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 2014. iMeshP: SciDAC ITAPS Parallel Mesh Interface. (2014). http://www.itaps.org/software/iMeshP_html.

ITAPS Web 2014. ITAPS: Interoperable Technologies for Advanced Petascale Simulations of Department of Energy's Scientific Discovery through Advanced Computing (SciDAC). (2014). 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

Benjamin S. Kirk, John W. Peterson, Roy H. Stogner, and Graham F. Carey. 2006. libMesh: a C++ library for parallel adaptive mesh refinement/coarsening simulations. *Engineering with Computers* 22, 3–4 (dec 2006), 237–254. DOI:http://dx.doi.org/10.1007/s00366-006-0049-3

Michael Kremer, David Bommes, and Leif Kobbelt. 2013. OpenVolumeMesh - A versatile index-based data structure for 3D polytopal complexes. In *Proceedings of the 21st International Meshing Roundtable*, Xiangmin Jiao and Jean-Christophe Weill (Eds.). Springer Berlin Heidelberg, 531–548. DOI:http://dx.doi.org/10.1007/978-3-642-33573-0_31

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

LLNL HPC Web 2014. High Performance Computing Training, Lawrence Livermore National Laboratory. (2014). 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 2014. MeshAdapt: Parallel Unstructured Mesh Adaptation Library. (2014). http://www.scorec.rpi.edu/meshadapt.

MeshSim Web 2014. MeshSim: Mesh Matching. (2014). http://www.simmetrix.com/products/SimulationModelingSuite/MeshSim/MeshMatching/MeshMatching.html.

NetCDF Web 2014. NetCDF: Network Common Data Form. (2014). 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 2014. Parasolid: 3D Geometric Modeling Engine. (2014). http://www.plm.automation.siemens.com/en_us/products/open/parasolid.

ParMA Web 2014. ParMA: Diffusive Partition Improvement Library. (2014). http://www.scorec.rpi.edu/parma.

ParMETIS Web 2014. ParMETIS: Parallel Graph Partitioning and Fill-reducing Matrix Ordering. (2014). http://glaros.dtc.umn.edu/gkhome/views/metis.

PUMI Web 2014. PUMI: Parallel Unstructured Mesh Infrastructure. (2014). 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.

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. 2006. 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. DOI: http://dx.doi.org/10.1002/(SICI)1097-0207(20000110/30)47:1/3⟨169::AID-NME766⟩3.0.CO;2-A

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 2014. Simmetrix: Simulation Modeling and Application Suite. (2014). http://www.simmetrix.com.

Cameron W. Smith, Kedar Chitale, Dan Ibanez, Brian Orecchio, E. Seegyoung Seol, Onkar Sahni, Kenneth E. Jansen, and Mark S. Shephard. 2014. Building effective parallel unstructured adaptive simula-

tions by in-memory integration of existing software components. *SIAM Journal on Scientific Computing* (2014). submitted.

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

Tim J. Tautges. 2004. MOAB-SD: integrated structured and unstructured mesh representation. *Engineering with Computers* 20, 3 (aug 2004), 286–293. DOI:http://dx.doi.org/10.1007/s00366-004-0296-0

Tim J. Tautges, Ray Meyers, Karl Merkley, Clint Stimpson, and Corey Ernst. 2004. *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 2014. The Trilinos Project: Sandia National Laboratories. (2014). http://trilinos.sandia.gov.

VTK Web 2014. VTK: Visualization Toolkit. (2014). 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, H. Jin Kim, C. Alberto Figueroa, Charles A. Taylor, Mark S. Shephard, and Kenneth E. Jansen. 2010. 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

Zoltan Web 2014. Zoltan: Parallel Partitioning, Load Balancing and Data-Management Services - User's Guide. (2014). http://www.cs.sandia.gov/Zoltan.

Online Appendix to: PUMI: Parallel Unstructured Mesh Infrastructure

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

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