Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Engineering 00 (2015) 000–000

Procedia Engineering

www.elsevier.com/locate/procedia

24th International Meshing Roundtable (IMR24)

Array-Based Hierarchical Mesh Generation in Parallel

Navamita Ray[a*], Iulian Grindeanu[a], Xinglin Zhao[b], Vijay Mahadevan[a], Xiangmin Jiao[b]

[a]Mathematics and Computer Science, Argonne National Laboratory, Argonne, IL 60439, USA

[b]Department of Applied Mathematics and Statistics, Stony Brook University, Stony Brook, NY 11794, USA.

Abstract

In this paper, we describe an array-based hierarchical mesh generation capability through uniform refinement of unstructured meshes for efficient solution of PDE's using finite element methods and multigrid solvers. A multi-degree, multi-dimensional and multi-level framework is designed to generate the nested hierarchies from an initial mesh that can be used for a number of purposes such as multi-level methods to generating large meshes. The capability is developed under the parallel mesh framework "Mesh Oriented dAtaBase" a.k.a MOAB [16]. We describe the underlying data structures and algorithms to generate such hierarchies and present numerical results for computational efficiency and mesh quality. We also present results to demonstrate the applicability of the developed capability to a multigrid finite-element solver.

© 2015 The Authors. Published by Elsevier Ltd.
Peer-review under responsibility of organizing committee of the 24th International Meshing Roundtable (IMR24).

Keywords: uniform mesh refinement, hierarchical meshes, multigrid methods, parallel computation, half-facet

1. Introduction

In the numerical solution of complex partial differential equations (PDE's) using finite element methods for unstructured meshes, the two most computationally intensive steps are mesh generation and linear solvers. An initial coarse mesh representing the computational domain might not be of sufficient resolution to get meaningful results out of the discretizations for physical scales that might be present. As a result, the capability to refine a mesh is an essential part of any simulation process. On the other hand, it is well known that multi-level methods such as geometric multigrid methods (GMG) can theoretically deliver optimal time complexity for solving sparse linear systems from PDE discretizations. Thus, it would be advantageous to use nested multi-level i.e., hierarchical meshes to achieve accuracy and computational efficiency, especially in the context of large-scale parallel computing, as both the number of processors and the mesh resolution increase. Uniform mesh refinement (UMR) provides a simple and efficient way to generate such hierarchies via successive refinement of the mesh at a previous level. It also provides a natural hierarchy via parent and child type of relationship between entities of meshes at different levels. While UMR is a relatively simple process, it is by no means trivial. Notable issues include mesh quality, multi-level and multi-degree refinement, and data structure and software design.

* Corresponding author. Email: nray@mcs.anl.gov.

In this paper, we develop parallel uniform refinement-based algorithms to generate multi-degree, multi-dimensional and multi-level meshes from coarse unstructured meshes. The generated mesh hierarchies can be used for a variety of purposes such as convergence studies, multilevel methods, generating large meshes in parallel to overcome IO bottlenecks, etc. While the multi-degree refinement allows achieving greater resolution faster, the multi-dimensional refinement preserves the hierarchy over explicit lower dimensional entities such as curves in surfaces, or surfaces embedded in volumes.

The key contributions of the paper include: 1) developing a template-based refinement strategy for subdividing each entity into smaller entities to support multi-degree refinement patterns, 2) extending the array-based half-facet (AHF) data structure [3] to support hierarchy generation and efficient mesh traversals, and 3) developing efficient parallel communication strategies to resolve shared entities along processor boundaries after refinement. We develop the capability under the parallel array-based mesh framework "Mesh Oriented dAtaBase" a.k.a MOAB[16]. The developed mesh hierarchy generation supports 1D (edges), 2D (triangles, quadrilaterals), 3D (tetrahedral, hexahedral) meshes and mixed-dimensional meshes. We present results to demonstrate the computational efficiency, memory requirements and effect on mesh quality due to template-based UMR. We also demonstrate the effectiveness of the hierarchical meshes through a multigrid application.

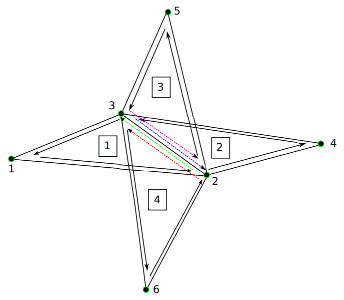
The remainder of the paper is organized as follows. Section 2 reviews some background knowledge and related mesh data structures. Section 3 describes the refinement templates, underlying mesh hierarchy storage and extended half-facet data structures. Section 4 describes the algorithms for the parallel communication and mesh hierarchy generation. Section 5 presents numerical results for computational efficiency, memory requirements, effect on mesh quality and an example multigrid application. Section 6 concludes the paper with a discussion.

2. Background

Mesh data structures are fundamental to meshing algorithms and mesh-based numerical methods. The underlying data structure strongly influences the overall performance of the algorithms or simulations since it is used to perform all the mesh-based combinatorial operations and as a result have been investigated since the inception of mesh generation and computational geometry. We review some terminology before describing our data structures and mesh frameworks. We say a mesh is a *manifold* or *non-manifold* if its geometric realization is a manifold or non-manifold, respectively. In a d-dimensional mesh, we refer to the d-dimensional entities as *elements*, and refer to the $(d-1)$-dimensional sub-entities as its *facets*. Each facet in a 2-D or 3-D mesh has an orientation with respect to the containing element. For example, each edge of a triangle has a direction, and all the edges form an oriented loop. Thus, it makes sense to call the facets *half-facets*. Each facet may have multiple incident elements, especially for non-manifold entities. We refer to all such half-facets as *sibling half-facets*. A mesh is said to be *conformal* if the pairwise intersection of any two entities is either another entity (lower-dimensional) or is empty. In this paper, we consider only conformal meshes, which may be manifold or non-manifold. In some engineering applications, especially in coupled or multi-component systems, the domain of interest may be composed of a union of topologically 1-D, 2-D, and 3-D objects, such as a mixture of cables, thin-shells, and solids. We refer to such a domain and its mesh as *mixed-dimensional*. We refer to a subset of the mesh corresponding to a 1-D, 2-D, and 3-D object in the domain as a *sub-mesh*.

2.1. Array-Based Half-Facet (AHF) Data Structure

There are a number of mesh data structures such as entity-based, boundary representations, corner table, radial-edge, winged, half- edge/face, incidence graphs, etc., that are used for mesh representation and queries. The two data structures that are relevant in our context are the half-edge and half-face data structures. The half-edge data structure is for 2D and surface meshes. It uses edge as the core object where the edge within each face is called a directed or half-edge. Typical implementations (e.g., CGAL [4,9], OpenMesh [2] and Surface_Mesh [15]) store mappings from each half-edge to its opposite half-edge, its previous and next half-edge within its face, its vertices, its incident face, as well as the mapping from each vertex and each face to an incident half-edge. More compact representations, such as [1], can be obtained by storing only the mapping between opposite half-edge, optionally the mapping from each vertex to an incident half-edge, along with the element connectivity. The half-edge concept was generalized to half-faces (e.g.,[1] and [11]) for volume meshes where half-faces refer to the oriented faces within a cell. These basic



element	connectivity		
1	1	2	3
2	2	4	3
3	2	5	3
4	2	3	6

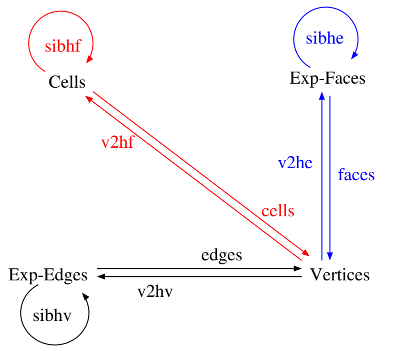
sibling half-edges

element	sibhes		
1	⟨0, 0⟩	⟨2, 2⟩	⟨0, 0⟩
2	⟨0, 0⟩	⟨0, 0⟩	⟨3, 2⟩
3	⟨0, 0⟩	⟨0, 0⟩	⟨4, 0⟩
4	⟨1, 1⟩	⟨0, 0⟩	⟨0, 0⟩

vertex to half-edges

vertices	v2he
1	⟨1, 0⟩
2	⟨3, 0⟩
3	⟨4, 1⟩
4	⟨2, 1⟩
5	⟨3, 1⟩
6	⟨4, 2⟩

(a) AHF maps for a non-manifold mesh.



(b) AHF maps for mixed-dimensional meshes.

Figure 1: (a) An example of the AHF maps stored for a non-manifold triangular mesh. (b) Typical AHF for mixed-dimensional meshes is composed of half-vertex (black, for explicit edges only), half-edge (blue, for explicit faces only), and half-face (red) data structures.

half-edge and half-face data structures are simple and are restricted to oriented, manifold meshes (with or without boundary) in 2-D and 3-D, respectively.

In [3], an efficient, compact and general array-based half-facet (AHF) mesh data structure with support for mixed-dimensional meshes, which may be non-manifold and/or non-oriented was proposed. The core object of AHF is *half-facet* as defined previously and is represented as an *implicit entity.* The concept of *sibling half-facets* unifies the half-vertex, half-edge, and half-face data structures for 1-D, 2-D, and 3-D meshes, which may be manifold or non-manifold with boundary. The AHF data structure consists of two key maps:

- sibling half-facets (*sibhfs*): The mapping between the sibling half-facets using a cyclic linked list.
- vertex to half-facet (*v2hf*): The map of each vertex to its incident half-facet. This map provides an anchor for each vertex to its locality in the mesh.

An example of the AHF maps for a non-manifold mesh is illustrated in Figure 1a. For $d \geq 2$, AHF provides a compact representation, since the intermediate dimensional entities are not stored but referenced implicitly. This data model stores the above two maps for each dimension in a modular and self-contained manner and hence supports mixed-dimensional meshes, which may be composed of sub-meshes of 1-D, 2-D, and 3-D. Figure 1b shows a diagram of a typical half-facet data structure, where the half-vertices and half-edges are only required for explicit edges and faces in the mesh, respectively. In addition, this data model can also be used for meshes with high-order elements (such as six-node triangles or 10-node tetrahedra), where the mid-edge, mid-face or mid-cell nodes do not affect the definition and identification of the half-facets.We use the AHF as the underlying mesh data structure for developing the refinement algorithms.

2.2. *Mesh Oriented datABase (MOAB)*

To be of any practical use to numerical simulation workflows, the mesh framework needs to support a wide range of functionalities such as efficient local mesh traversals for matrix assemblies, boundary extraction for boundary conditions, representation of mesh data, etc., in a parallel setting. Over the past years, a number of such frameworks have been developed e.g., FMDB [13], MSTK [5], libMesh [10], etc. In such implementations, the entities are represented as "objects" explicitly, and pointers (or handles) are used to refer to these explicit objects. In our work, we choose to use array-based, pointer-free implementations for a number of reasons. First, using arrays can lead to faster memory access, fewer cache misses and hence better efficiency. Secondly, in an array-based implementation, we can treat intermediate dimensional entities (such as half-facets) as implicit entities, and reference them without forming explicit objects. This can lead to significant savings in storage, especially on computers with 64-bit pointers. In addition, array-based implementations also offer better interoperability across application codes, different programming languages, and different hardware platforms (such as between GPUs and CPUs).

One such array-based parallel mesh framework is the Mesh Oriented datABase a.k.a **MOAB** [16] which is a mesh data representation designed to support a range of mesh related operations, such as memory efficient mesh representation, mesh querying and representation of application specific data. Internally it uses array-based storage for fine-grained data, which in many cases provides more efficient access, especially for large portions of mesh and associated data. The AHF data structure was implemented in MOAB to support query-intensive algorithms and it was found to be extremely efficient (in some cases over two order of magnitude improvement [3,12]) over the existing data structures. MOAB is also a parallel framework and supports spatial domain-decomposed view of a parallel mesh where each subdomain is assigned to a processor, lower-dimensional entities on interfaces between subdomains are shared between processors, and ghost entities can be exchanged with neighboring processors. We develop our hierarchical mesh generation algorithms on top of MOAB's parallel framework. In the next section, we describe the three key components of the proposed hierarchical mesh generation algorithm.

3. Multi-Degree, Multi-dimensional and Multi-Level Hierarchical Meshes

The uniform refinement based mesh hierarchy generation has three key design components. First, entity type and degree-specific refinement templates are defined that are used to subdivide an entity into its children. The templates are also used to update the underlying data structures for all the new children. The second key design component is the use of array-based half-facet data structures for efficient mesh traversal during refinement. Finally, the newly created meshes are stored in level-wise contiguous array to provide memory compactness.

3.1. Multi-degree Refinement Templates

The standard refinement strategy divides each d dimensional entity to $1 \rightarrow 2^d$ subentites. However, a desired mesh resolution can be reached much faster if a higher degree of refinement is used. The length of the mesh hierarchies is also small in such cases which might be preferable from some multigrid methods such as GMGs. Another motivation to use high degree refinement patterns is that it allows straightforward extension to high-order entities and thus support *hp*-refinements. By multi-degree refinement we mean a refinement pattern following vertex positions analogous to high-order (degree p where $p \geq 2$) Lagrangian elements. Thus, for a d dimensional entity, a degree p refinement divides the entity to $1 \rightarrow p^d$ subentites. We support prime number degrees as any higher degree refinements can be obtained by applying a sequence of such prime degree refinements. Table (1) lists the currently supported degrees for each dimension. We note that in uniform refinement the same modification operation of dividing an entity into a specific number of entities is applied to each entity of the mesh. Thus defining static templates w.r.t a reference entity that contain certain entity type and degree specific information could aid the refinement process. The template mainly stores the following information for each entity type and supported degree of refinement :

- *Numbering convention of new vertices*: The new vertices are assigned local indices through which they are uniquely identified within a reference entity.
- *Connectivity of new child entities*: The connectivity of the new entities using the local indices of the new vertices. Each such child is also uniquely identifiable by a local id w.r.t reference.
- *Half-facet maps for new child entities*: The local sibling half-facet (**sibhfs**) and vertex to incident half-facet (**v2hf**) maps are stored.

Figure 2 shows the templates for triangle and quadrilateral reference entities for degree 2 and 3. The numbering convention of the reference entity follows the MOAB canonical numbering. Apart from these three pieces of data, the template also stores some other auxiliary information to aid tracking of new vertices introduced on the entity boundary

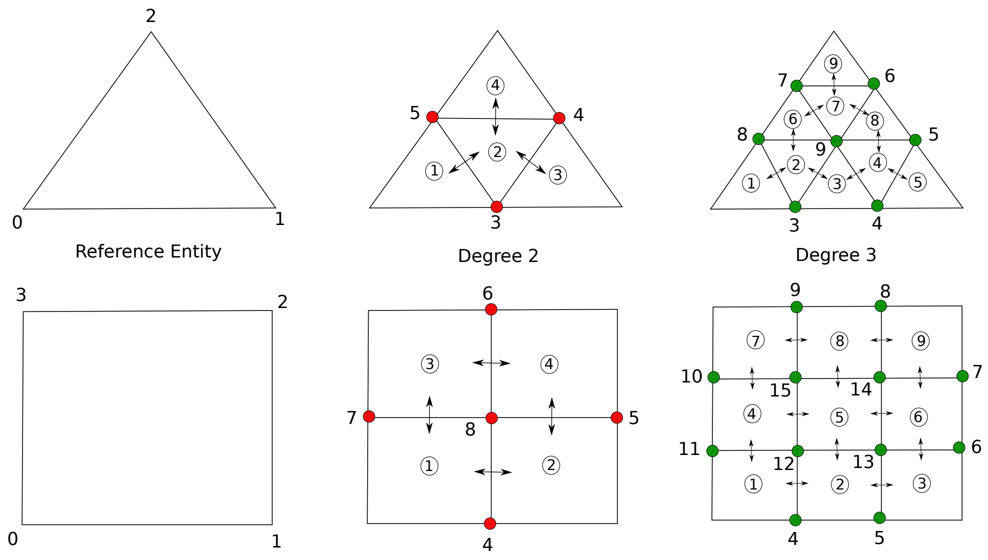


Figure 2: The refinement templates defined for degree 2 and 3 refinements over a reference triangle and quadrilateral entity. The local vertex order of the reference entities follow MOAB conventions[17]. The new vertices to be introduced during refinement are assigned local ids wrt to the reference. Similarly, the children entities are assigned local order. The AHF maps are created for the refined entities in terms of their local ids.

to avoid vertex duplication during refinement. An example of a degree 3 refinement template for a reference triangle is illustrated in the Table (2).

Except for tetrahedron entity, the templates are static for all other entities (triangle, quads, hexes) in the sense that they do not need to refer to the physical entity in order to refine the parent entity. The refinement schemes for a tetrahedron involves division into smaller congruent tetrahedrons and octahedrons which are subsequently divided into four tetrahedrons. Figure 3 shows how refining a tetrahedron leads to congruent smaller tetrahedra and octahedra. The right side of the figure shows the three possible diagonal choices for each such octahedron to be divided into four tetrahedrons. To deliver a good mesh quality, each such octahedra is divided into sub-tetrahedra by connecting the shortest diagonal[19]. Thus, for higher-degree refinements, each octahedron has to find the shortest diagonal using the physical entity. However, we note that though the smaller tetrahedra's obtained by tessellating an octahedron are not congruent to the parent tetrahedron, all the octahedra's are congruent to each other. Therefore, the shortest diagonal is unique and hence each octahedron would be divided using the same pattern. We use this fact to define three static templates for each degree. During refinement, each tetrahedron makes a choice of the appropriate template based on the smallest diagonal of the physical tetrahedron. Currently, the template generation is not automatic. Effort is being made to make it automatic so that higher-degrees such as 7, 11, etc. can also be supported.

3.2. Multi-dimensional Mesh Data Structures

During refinement, one of the key tasks is to avoid introducing duplicate vertices for shared boundaries between entities. This requires frequent calls to adjacency routines to get entities connected through entity boundaries. Since AHF stores the half-facet maps between entities, these types of queries are the most efficient ones it can support. As a result, it is natural to use AHF as the underlying data structure. As each level of the mesh hierarchy is generated by refinement of the previous level, it is necessary to update the appropriate AHF maps for the new level so that the new mesh can be queried later. The process of updating the AHF maps for the new level is aided both by the refinement templates (defined in the previous subsection) and also by the fact that the topology of the domain doesn't change during refinement. Thus, children of non-manifold entities remain so for all levels and no new non-manifold entities are introduced. The AHF maps are updated in two stages:

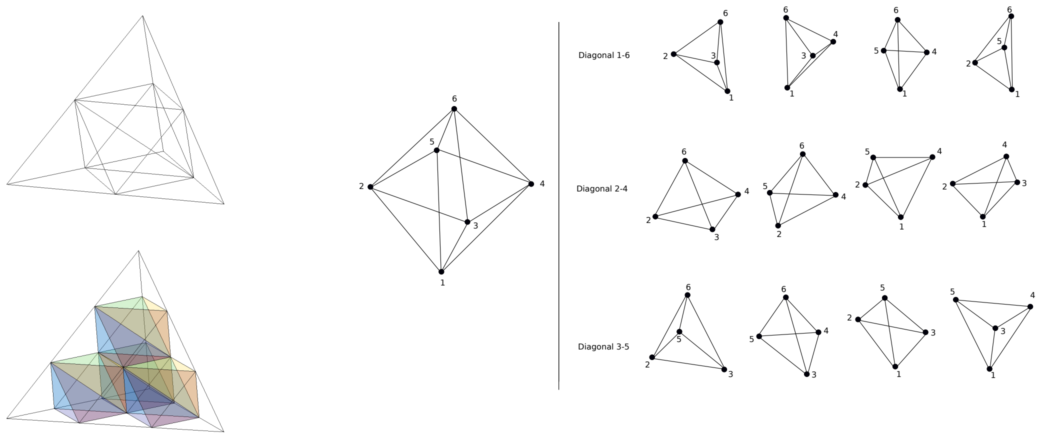


Figure 3: Left: A tetrahedron is divided into congruent smaller tetrahedrons and octahedrons. Middle: A reference octahedron. Right: Each octahedron can be further subdivided into four tetrahedrons in three possible ways depending on which diagonal is chosen.

Table 1: The degrees of refinement currently supported and the corresponding number of children.

Dimension	Degrees	#Children
1	2, 3, 5	2, 3, 5
2	2, 3, 5	4, 9, 25
3	2, 3	8, 27

Table 2: An example for the degree 3 template of a triangle reference entity. Children entity connectivity and the template $v2hf$ and $sibhfs$ maps.

Child Entity Connectivity	
1	⟨0, 3, 8⟩
2	⟨3, 9, 8⟩
3	⟨3, 4, 9⟩
4	⟨4, 5, 9⟩
5	⟨4, 1, 5⟩
6	⟨8, 9, 7⟩
7	⟨9, 6, 7⟩
8	⟨9, 5, 6⟩
9	⟨7, 6, 2⟩

sibhfs		
⟨0, 0⟩	⟨2, 2⟩	⟨0, 0⟩
⟨3, 2⟩	⟨6, 0⟩	⟨1, 1⟩
⟨0, 0⟩	⟨4, 2⟩	⟨2, 1⟩
⟨5, 2⟩	⟨8, 0⟩	⟨3, 1⟩
⟨0, 0⟩	⟨0, 0⟩	⟨4, 0⟩
⟨2, 1⟩	⟨7, 2⟩	⟨0, 0⟩
⟨8, 2⟩	⟨9, 0⟩	⟨6, 1⟩
⟨4, 1⟩	⟨0, 0⟩	⟨7, 0⟩
⟨7, 1⟩	⟨0, 0⟩	⟨0, 0⟩

v	*v2hf*
0	⟨1, 0⟩
1	⟨5, 1⟩
2	⟨9, 2⟩
3	⟨3, 0⟩
4	⟨5, 0⟩
5	⟨8, 1⟩
6	⟨9, 1⟩
7	⟨6, 2⟩
8	⟨1, 2⟩
9	⟨8, 0⟩

1. *Update maps for children of an entity* : After an entity is refined, the sibling half-facet **(sibhfs)** and vertex-to-incident half-facet (**v2hf**) maps are updated only for the child entities of the working entity by using the same from the refinement templates.
2. *Update maps between children of parent siblings* : After all the entities of the mesh have been refined, the maps are now updated to connect the child entities incident on boundaries of the parents.

Figure 4 illustrates the above two steps steps during refinement of a triangle mesh with two entities. Algorithms 1 and 2 outline the procedure for updating local and global AHF maps. For mixed-dimensional meshes, the maps for each lower dimensional submesh are updated in a similar manner.

3.3. Multi-level Mesh Storage

An array-based mesh storage leads to increased efficiency, however it requires careful consideration to maintain it for operations involving a change in the contiguity of the memory space. By virtue of the uniform refinement of the mesh, it is possible to estimate the total number of entities that will be created for a given degree of refinement. Table 3 shows the estimates for new entities created after a single level of refinement for a 3D mesh with embedded curves and surfaces. Once the storage requirement for a new level is estimated, the memory is allocated in contiguous blocks. During refinement, as each entity is subdivided, the new entities are stored according to the local order specified in the

Algorithm 1 Updating AHF maps for children of an entity

Input: : childEnts: ordered child entities of a parent, childVerts: ordered new vertex indices of childEnts, refTemplate: refinement template for entity type and degree
Output: sibhfs: update sibling half-facets for childEnts, v2hf: update vertex to half-facet for childVerts

```
1: for each vertex $v$ in childVerts do
2:     if v2hf($v$) = 0 then
3:         $cid \leftarrow$ child id from refTemplate$\rightarrow$v2hf;
4:         $lid \leftarrow$ half-facet id from refTemplate$\rightarrow$v2hf;
5:         v2hf($v$) = (childEnts[$cid$], $lid$);
6:     end if
7: end for
8: for each child $c$ in childEnts do
9:     for each facet $f$ in $c$ do
10:        if sibhfs($f$ is not set then
11:            $cid \leftarrow$ sibling child id from refTemplate$\rightarrow$sibhfs;
12:            $lid \leftarrow$ sibling half-facet id from refTemplate$\rightarrow$sibhfs;
13:            sibhfs($c$, $f$) = (childEnts[$cid$], $lid$);
14:            sibhfs(childEnts[$cid$], $lid$) = ($c$, $f$);
15:        end if
16:    end for
17: end for
```

Algorithm 2 Updating AHF maps between children of sibling parents sharing a facet.

Input: : PE: entities of previous level, CE: entities of current level, PV: vertex indices of PE, CV: vertex indices of CE, refTemplate: refinement template for entity type and degree entity type and degree
Output: sibhfs: update sibling half-facets for CE, v2hf: update vertex to half-facet for duplicates of PV in CV

```
1: for each vertex $v$ in PV do
2:     $lvid \leftarrow$ local id of v in $pid$;
3:     $pid \leftarrow$ parent entity from incident half-facet $v2hf(v)$;
4:     $cid \leftarrow$ child id from refTemplate$\rightarrow$v2hf;
5:     $lid \leftarrow$ half-facet id from refTemplate$\rightarrow$v2hf;
6:     $curvid \leftarrow$ current id of v in CV;
7:     v2hf($curvid$) = (CE[$cid$], $lid$);
8: end for
9: for each entity $e$ in PE do
10:    for each facet $f$ in $e$ do
11:        $childs \leftarrow$ children entities of $e$ incident on $f$;
12:        $sibeids \leftarrow$ sibling entities from $sibhfs(e, f)$;
13:        $siblids \leftarrow$ sibling facet ids from $sibhfs(e, f)$;
14:        for each sibling $seid$ in $sibeids$ do
15:            $sibchilds \leftarrow$ children entities of $seid$ incident on $siblids[seid]$;
16:            find orientation of $f$ and $siblids[seid]$;
17:            update sibhfs($childs$,$f$) with matching ($sibeids$, $siblids$);
18:        end for
19:    end for
20: end for
```

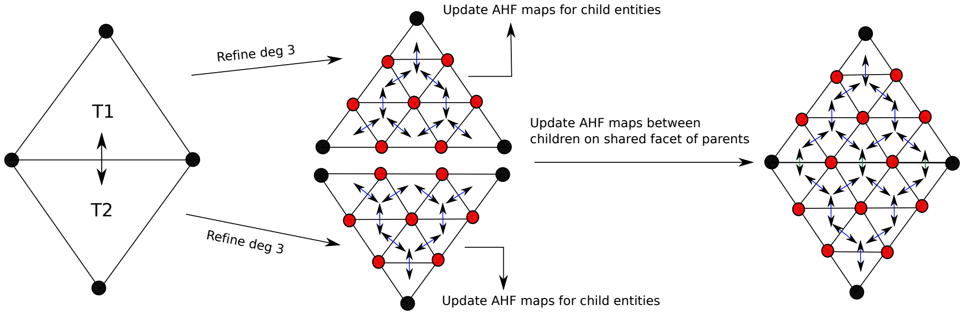

T1
T2
Refine deg 3
Refine deg 3
Update AHF maps for child entities
Update AHF maps between children on shared facet of parents
Update AHF maps for child entities

Figure 4: Updating the AHF maps for the refined mesh takes place in two stages. First, maps for the children of each triangle are updated. After, both the triangles have been refined, the maps are now updated to connect the children entities sharing a parent facet.

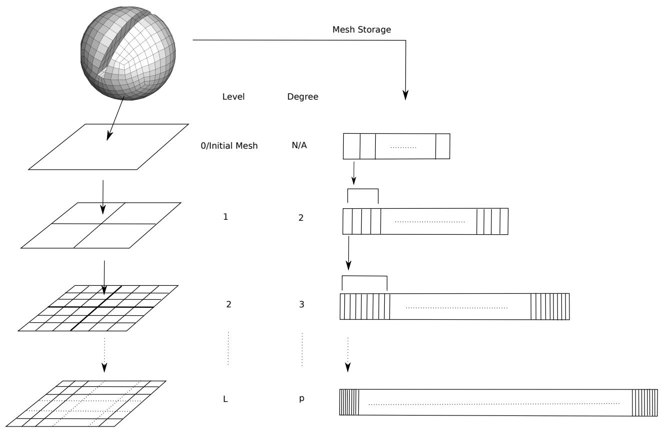


Figure 5: The data layout of meshes at each level. For example, starting with the first quad of the sphere mesh, after each level of refinement, the children are stored in a pre-decided order as defined in the refinement templates. This local order uniquely identifies each child of the parent.

Table 3: Estimates for new entities that will be created after a refinement of degree p. These estimates are for a 3D mesh with explicit curves and surfaces. Here$|\cdot|$ denotes the number of entities in a particular entity set. E^p and F^p are the total number of edges and faces in the 3D mesh at the previous level, respectively.

	#Entities at previous level	#Entities after refinement of degree p
Vertices	$\lvert V^p \rvert$	$\lvert V\rvert = \lvert V^p\rvert + nv_e * \lvert E^p\rvert + nv_f * \lvert F^p\rvert + nv_c * \lvert C\rvert$
Explicit Edges	$\left\lvert E^p_{exp}\right\rvert$	$\left\lvert E_{exp}\right\rvert = p * \left\lvert E^p_{exp}\right\rvert$
Explicit Faces	$\left\lvert F^p_{exp}\right\rvert$	$\left\lvert F_{exp}\right\rvert = p^2 * \left\lvert F^p_{exp}\right\rvert$
Cells	$\lvert C\rvert$	$\lvert C\rvert = p^3 * \lvert C^p\rvert$

refinement template. Thus, children of the first entity in previous level are stored first, then the children of the second and so on. This data layout supports straightforward index based inter-level (i.e., parent-to-child or child-to-parent) queries which are vital to multi-level methods. Figure 5 illustrates this memory layout. To provide mesh independence at each level, we duplicate vertices from the previous level to create the new mesh along with new vertices. As a result, the mesh hierarchy generation would return a sequence of meshes that are independent of each other while providing inter- and intra-level mesh access. If necessary, the vertices at a specific level can be renumbered to maintain small bandwidths for the assembled matrices.

4. Hierarchical Mesh Generation in Parallel

Generating mesh hierarchies using uniform refinement in parallel may seem to be a relatively straightforward process since each processor refines its local mesh. If the initial mesh distributed is balanced, then uniform refinement would not introduce any additional imbalance. Thus, there is no need to move mesh entities between processors. However, generation of each level of the hierarchy also introduces new entities on the shared interface between processors. Unless these new entities are resolved, the generated hierarchies are only useful for local operations and more complicated algorithms requiring shared information such as exchanging ghost-layers would break down. In this section, we discuss a parallel communication algorithm that resolves such new entities on the shared interface between processors. We also describe the refinement algorithm to generate multi-degree, multi-dimensional hierarchies of unstructured meshes in parallel.

4.1. Parallel Communication to Resolve Shared Entities

The MOAB library implemented with array-based data-structures have been designed to be scalable in memory access and there are several optimized one-sided communication algorithms that make use of strategies to minimize total data transferred between processors. Using this parallel framework, the mesh hierarchy generation can be performed in a series of optimal steps. Once each processor loads a part of the distributed coarse mesh, local refinement for all the entities can be performed. Since the refinement is based on pre-defined templates, the co-ordinates for the new vertices and entities on the shared interface between processors will be the same. We utilize this information to design the communication algorithm to resolve the newly created entities in preparation for synchronization of ghost layers and exchange of meta-data (MOAB tags).

This parallel merge of the interface mesh algorithm first matches mesh vertices based on geometric proximity and then uses connectivity matching algorithm to decipher the corresponding entity in the local mesh. The merging algorithm derives its motivation from the vertex-matching algorithm described in [6,18]. The algorithm proceeds by first partitioning the geometric bounding box of all vertices over processors, with each processor responsible for a distinct geometric region (plus a small epsilon layer whose thickness is twice the distance tolerance of the merge). Then, each processor retrieves the vertices on the skin of the local mesh and assembles a tuple list [18] that holds the coordinate positions of the vertices and the destination processor. Finally, the higher-dimensional entities on the skin are resolved using the connectivities.

Alternately, we can design a scheme for assigning consistent GLOBAL_ID's across processors for the newly created entities, which can yield a significantly faster parallel merging algorithm due to lower communication and local search costs. In the current work, we only present the scalability results for the co-ordinate based merging algorithm and the optimized GLOBAL_ID merge algorithm along with comparisons will be pursued separately.

4.2. Refinement Algorithm

Figure 6 shows the flowchart of the refinement algorithm to generate a hierarchy of unstructured meshes in parallel. A key aspect of the refinement algorithm is the positioning of the new vertices as entities are refined. In this work, we use a linear point projection scheme for the new vertices. However, this would compromise the accuracy of the geometry and in turn that of the finite element solver. To address this issue, we are currently exploring integration with geometric models, when available, or to use a polynomial based high-order boundary reconstruction [7,8]. We have also developed a tool on top of this functionality in MOAB which can be used to read in a mesh, generate the hierarchies in parallel for a given number of levels and a sequence of degree of refinements.

5. Numerical Results

We present some numerical results to demonstrate the effectiveness of uniform mesh refinement, in terms of mesh quality, computational efficiency of the parallel framework and its application to multigrid and efficiency of multigrid methods, and most importantly, the accuracy of finite element methods.

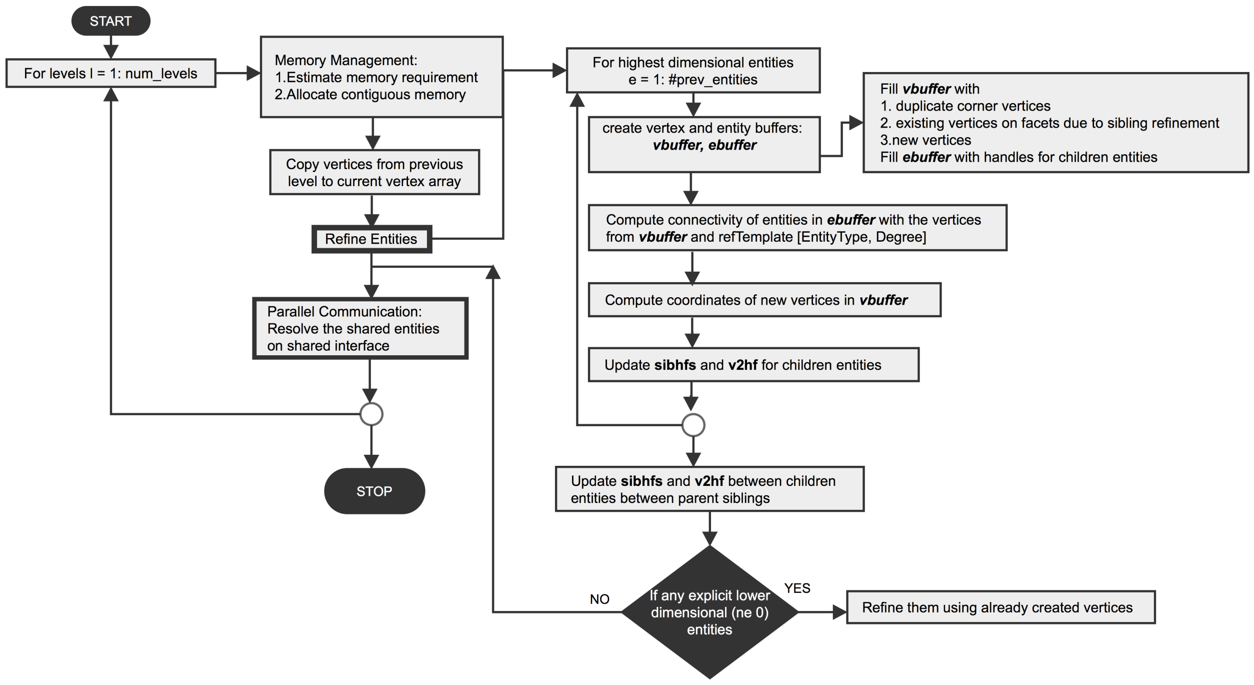

START
For levels l = 1: num_levels
Memory Management:
1.Estimate memory requirement
2.Allocate contiguous memory
Copy vertices from previous level to current vertex array
Refine Entities
Parallel Communication: Resolve the shared entities on shared interface
STOP
For highest dimensional entities e = 1: #prev_entities
create vertex and entity buffers: vbuffer, ebuffer
Fill vbuffer with
1. duplicate corner vertices
2. existing vertices on facets due to sibling refinement
3.new vertices
Fill ebuffer with handles for children entities
Compute connectivity of entities in ebuffer with the vertices from vbuffer and refTemplate [EntityType, Degree]
Compute coordinates of new vertices in vbuffer
Update sibhfs and v2hf for children entities
Update sibhfs and v2hf between children entities between parent siblings
If any explicit lower dimensional (ne 0) entities
NO
YES
Refine them using already created vertices

Figure 6: The flowchart for generation of mesh hierarchy with length num_levels.

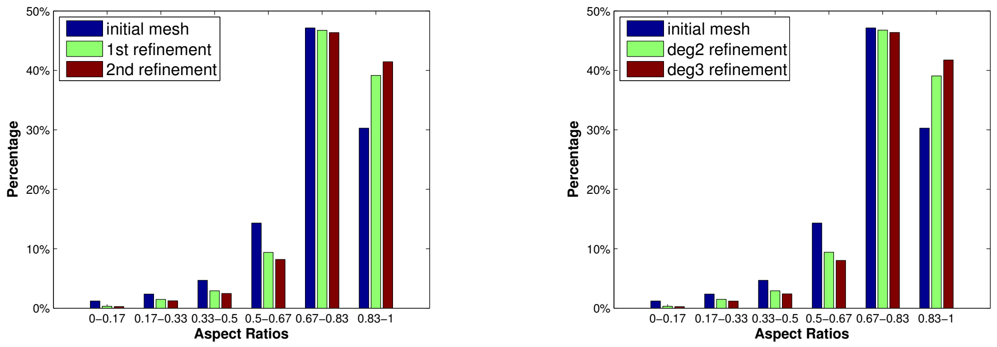
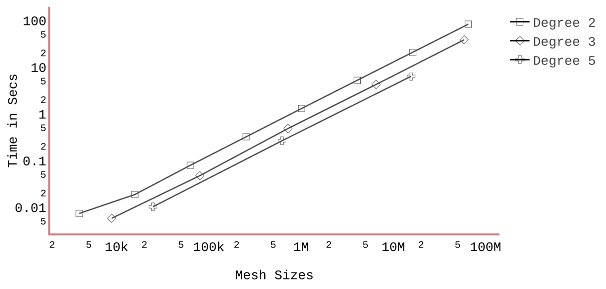


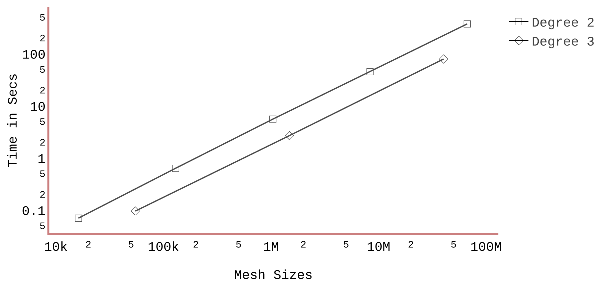
Figure 7: Aspect ratio distributions with two-level degrees 2+2 (left) and degrees 2+3 (right) refinements.

5.1. Mesh Quality for Hierarchical Meshes

For tetrahedral meshes, uniform mesh refinement does not produce congruent sub-tetrahedra. It is interesting to note that the average mesh quality actually improves during UMR with the shortest-diagonal approach[19]. To demonstrate it, we generate an initial coarse tetrahedral mesh using TetGen [14], with approximately 5K vertices and 2.5K tetrahedra. We uniformly refine twice with two strategies. First, we refine it using two degree-2 refinements. The resulting finest mesh has about 280K vertices and 1.6M tetrahedra. Second, we refine it using one degree-2 followed by a degree-3 refinement. We measure the mesh quality using the aspect ratio, computed as three times the inscribed radius (IR) divided by the circumsphere radius (CR), i.e., $3 \times \text{IR}/\text{CR}$. Larger aspect ratio (closer to 1) corresponds to better element quality. The two graphs in Figure 7 show the distributions of the aspect ratios on each level for the two strategies, respectively. In both cases, the overall mesh quality improved, and degree-3 refinement delivers similar and even slightly better quality improvement because there are more intermediate octahedra in degree-3 refinement.



(a) Triangle Meshes



(b) Hexahedral Meshes

Figure 8: Time to generate hierarchies for different degree of refinements.

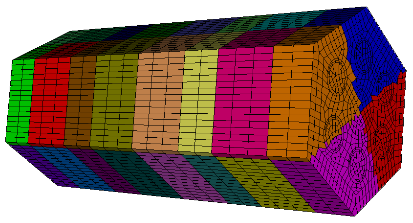
Table 4: Lower and upper bounds for the number of entities of the initial and final level mesh of the generated hierarchy with degree sequence {2, 3, 2}

#processors	Initial Mesh	Last level of Mesh Hierarchy
1	15180	26,231,040
2	7590	13,11,5520
4	3795	6,557,760
8	1895-1900	3,274,560-3,283,200
16	946-950	1,634,688-1,641,600
32	378-570	654,912-984,960

5.2. Computational Efficiency

We now report computational efficiency results of the mesh hierarchy generation algorithm. In Figure 8, the serial run times for generating hierarchies for each supported degree of refinement for two representative meshes for two and three dimensions are shown. In Figure 8a, we start with a triangle mesh with 1000 entities and generate hierarchies with degrees 2, 3 and 5. Clearly, the results show that higher-degree of refinement reach greater resolution much faster. Similarly, in Figure 8b, an initial hexahedral mesh with 2048 entities is used to generate hierarchies with degrees 2 and 3. We conclude that if a deep hierarchy is required, a degree 2 refinement per level would give a gradually increasing mesh with more levels. On the other hand, high resolution can be reached very quickly with small hierarchies using high-degree refinement. These serial tests were performed on a Mac computer with 2.3 GHz Intel Core i7 processor and 16GB RAM.

Next, we present the strong scalability studies for the refinement of a coarse hexahedral mesh of a single assembly in a reactor core with fuel pins and central control rod as shown in Figure 9 (left). The coarse mesh is refined using a degree sequence {2, 3, 2}. The codes were run on a cluster with 310 nodes, 16 cores/node with Intel Sandy Bridge processor and 64GB RAM per node. Figure 9 (right) shows the efficiency as the number of processors is increased for the two main components of the algorithm 1) the refinement and 2) the vertex merge based parallel communication algorithm. We can see from this preliminary study that the algorithm achieves super-linear efficiency which might be due to better cache use for such small number of entities per processors. However more rigorous testing is needed on more number of processors with higher count of mesh entities. Table 4 lists the lower and upper bounds for the number of entities of the initial and the most refined mesh of the hierarchy. Currently, more studies are being performed on larger and more complicated unstructured meshes.



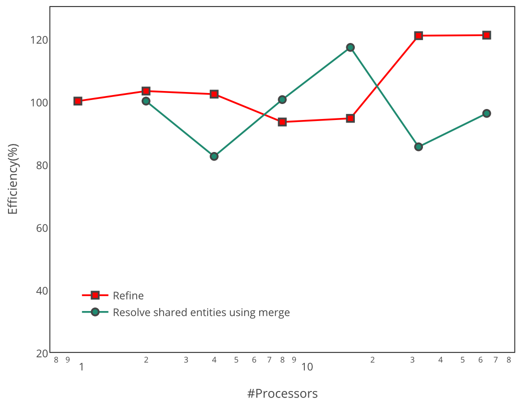


Figure 9: Scalability studies for the mesh hierarchy generation algorithm. The left figure shows the initial coarse mesh with 32 partitions. The right figure shows the time in seconds as the number of processors are increased.

5.3. Demonstration of UMR capabilities in discrete solvers

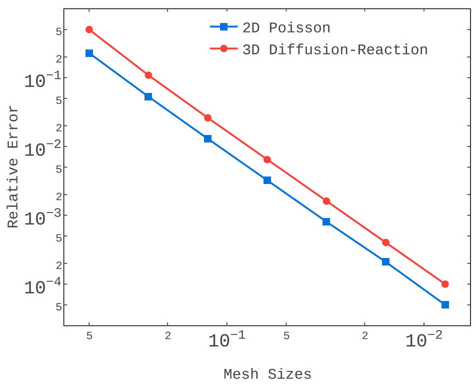
The parallel UMR capability is useful in generating hierarchy of meshes to perform convergence studies and for creating optimal multigrid preconditioners for elliptic PDE solvers. A multigrid Poisson solver written using the PETSc-MOAB (DMMoab) interface that leverages the scalability of both the codes and utilize UMR is presented here for computing order of accuracy efficiently. Figure 10 (a) shows that the Poisson solver with an inhomogeneous source term in 2-D and using a Method of Manufactured Solution (sinusoidal exact solution) yields the expected second order convergence (linear Lagrange continuous Galerkin FEM). These results were generated by successively refining the mesh through UMR and solved with Geometric Multigrid (GMG). The iteration convergence of geometric multigrid-based preconditioner for a Poisson solver using a Generalized Finite Difference (GFD) method is also shown in Figure 10 (b) and comparison to standard black-box algebraic preconditioners including AMG, shows optimal reduction in iteration error independent of the mesh resolution or degrees-of-freedom.

6. Conclusion and Discussion

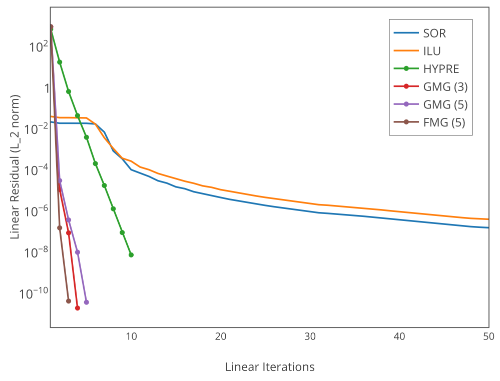
In this paper, we presented a method for generating a hierarchical unstructured meshes in parallel for efficient solution of PDE's using finite element methods and multigrid solvers. A multi-degree, multi-dimensional and multilevel framework is designed to generate the nested hierarchies from an initial mesh that can be used for a number of purposes such as multi-level methods to generating large meshes. We presented results to demonstrate the computational efficiency of the algorithm as well as its application to multilevel and finite element methods. The current method has few shortcomings as it uses a geometric proximity based algorithm for resolving shared entities. This approach may be prone to numerical errors. In future, we plan to optimize the parallel communication algorithm via a vertex GLOBAL_ID based approach.

Acknowledgements

We thank Cao Lu for his implementation of the GFD Poisson solver with multigrid using DMMoab. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program and the FASTMath project. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.



(a) Mesh convergence of a 2D Poisson and 3D Diffusion-Reaction problem.



(b) Comparison of convergence of multigrid and standard black-box preconditioners.

Figure 10: Application of mesh hierarchies in mesh convergence studies and convergence of multigrid methods.

References

[1] T. Alumbaugh and X. Jiao. Compact array-based mesh data structures. In *Proceedings of 14th International Meshing Roundtable*, pages 485–504, 2005.
[2] B. S. Bischoff, M. Botsch, S. Steinberg, S. Bischoff, L. Kobbelt, and R. Aachen. OpenMesh – a generic and efficient polygon mesh data structure. In *In OpenSG Symposium*, 2002.
[3] V. Dyedov, N. Ray, D. Einstein, X. Jiao, and T. Tautges. AHF: Array-based half-facet data structure for mixed-dimensional and non-manifold meshes. In J. Sarrate and M. Staten, editors, *Proceedings of the 22nd International Meshing Roundtable*, pages 445–464. Springer International Publishing, 2014.
[4] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, a computational geometry algorithms library. *Softw. – Pract. Exp.*, 30:1167–1202, 2000. Special Issue on Discrete Algorithm Engineering.
[5] R. V. Garimella. MSTK – a flexible infrastructure library for developing mesh based applications. In *Proceedings of 13th International Meshing Roundtable*, pages 213–220, 2004.
[6] R. Jain and T. J. Tautges. Generating unstructured nuclear reactor core meshes in parallel. *Procedia Engineering*, 82(0):351 – 363, 2014. 23rd International Meshing Roundtable (IMR23).
[7] X. Jiao and D. Wang. Reconstructing High-Order Surfaces for Meshing. In S. Shontz, editor, *Proceedings of the 19th International Meshing Roundtable*, pages 143–160. Springer Berlin Heidelberg, 2010.
[8] X. Jiao and D. Wang. Reconstructing high-order surfaces for meshing. *Engineering with Computers*, 28:361–373, 2012.
[9] L. Kettner. Using generic programming for designing a data structure for polyhedral surfaces. *Comput. Geom. Theo. Appl.*, 13:65–90, 1999.
[10] B. S. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey. libMesh: A c++ library for parallel adaptive mesh refinement/coarsening simulations. *Engineering with Computers*, 22:237–254, 2006.
[11] M. Kremer, D. Bommes, and L. Kobbelt. OpenVolumeMesh – a versatile index based data structure for 3D polytopal complexes. *Proceedings of 21st International Meshing Roundtable*, pages 531–548, 2012.
[12] N. Ray. *High-Order Surface Reconstruction and its Applications to Surface Integrals and Surface Remeshing*. PhD thesis, STATE UNIVERSITY OF NEW YORK AT STONY BROOK, 2013.
[13] E. S. Seol. *FMDB: Flexible Distributed Mesh Database For Parallel Automated Adaptive Analysis*. PhD thesis, Rensselaer Polytechnic Institute, 2005.
[14] H. Si. TetGen, a quality tetrahedral mesh generator and three-dimensional Delaunay triangulator v1.4, 2006.
[15] D. Sieger and M. Botsch. Design, implementation and evaluation of the surface mesh data structure. In *In Proceedings of the 20th International Meshing Roundtable*, 2011.
[16] T. Tautges, R. Meyers, and K. Merkley. MOAB: A mesh-oriented database. Technical report, Sandia National Laboratories, 2004.
[17] T. J. Tautges. Canonical numbering systems for finite-element codes. *International Journal for Numerical Methods in Biomedical Engineering*, 26(12):1559–1572, 2010.
[18] T. J. Tautges, J. A. Kraftcheck, N. Bertram, V. Sachdeva, and J. Magerlein. Mesh interface resolution and ghost exchange in a parallel mesh representation. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW '12, pages 1670–1679, Washington, DC, USA, 2012. IEEE Computer Society.
[19] K. Tim and T. Preusser. Stability of the 8-tetrahedra shortest-interior-edge partitioning method. *Numerische Mathematik*, 109:435–457, 2008.