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)

Conformal and Non-conformal Adaptive Mesh Refinement with Hierarchical Array-based Half-Facet Data Structures

Xinglin Zhao[a], Rebecca Conley[a], Navamita Ray[b], Vijay S. Mahadevan[b], Xiangmin Jiao[a,*]

[a]*Department of Applied Mathematics and Statistics, Stony Brook University, Stony Brook, NY 11794, USA*
[b]*Mathematics and Computer Science, Argonne National Laboratory, Argonne, IL 60439, USA*

Abstract

We present a generalization of the *Array-based Half-Facet (AHF)* mesh data structure, called *Hierarchical AHF*, for hierarchical unstructured meshes generated from adaptive mesh refinement for solving PDEs. This data structure extends the *AHF* data structure (V. Dyedov, et al. AHF: Array-based Half-Facet Data Structure for Mixed-Dimensional and Non-manifold Meshes) to support meshes with hierarchical structure, which often are generated from adaptive mesh refinement (AMR). The design goals of our data structure include generality to support efficient neighborhood queries, refinement and derefinement, and *hp*-FEM with mesh smoothing. Our data structure utilizes the *sibling half-facets* as a core abstraction, coupled with a tree structure for hierarchical information. To facilitate the interoperability of mesh based applications, auxiliary data will be designed on top of Hierarchical AHF. We describe the data structure and software requirements, and present numerical experiments to demonstrate its effectiveness.

Peer-review under responsibility of organizing committee of the 24th International Meshing Roundtable (IMR24).

Keywords: adaptive mesh refinement, hierarchical meshes, mesh generation, data structure, sibling half-facets

1. Introduction

In large scale simulation problems, mesh generation and linear solvers are often the two most expensive steps in the solutions of partial differential equations (PDEs), using finite element methods with unstructured meshes. A mesh with billions of elements will be required. Since, in most cases the criteria for mesh resolution is not known *a priori*, in order to avoid computational overhead, the generated original mesh is often relatively coarse overall with uniform resolution. However, to obtain accuracy some regions need to be refined to reduce discretization errors while other regions require finer models to approximate. Adaptive mesh refinement allows more efficient numerical simulations by increasing the computational effort near interesting features of the solutions [4–6].

AMR has gradually become a vital step in large-scale numerical simulations since it optimizes the relationship between accuracy and computational effort. One aspect of the refinement strategy is whether it requires the refined mesh to be conformal or not. A mesh is said to be *conformal* if the pairwise intersection of any two entities is either a lower-dimensional entity or is empty. Otherwise, a mesh is non-conformal. The conformal requirement will make

* Corresponding author. Tel:+1-631-632-4408 ; fax: +1-631-380-8004.
E-mail address: jiao@ams.sunysb.edu

Peer-review under responsibility of organizing committee of the 24th International Meshing Roundtable (IMR24).

no change to the underlying data structure for the mesh and the formulation of numerical algorithms. A considerable amount of work has been done in this area [7,9,10,25]. However, to preserve conformity, some procedures need to be applied which would probably deliver a finer mesh than needed, or even potentially affect the overall mesh quality which is crucial for the linear system in FEM [26]. This drives the research on refinement strategies allowing non-conformal meshes, i.e. hanging nodes, in [7,15,18,26]. Non-conformal refinement will incur extra work in the PDE solver part, but it will be much easier for the unification of *hp*-adaptivity for finite element method [7,8]. Here *hp*-adaptivity means that both the mesh size h and the degree p of the approximating piecewise polynomials are adapted.

Most of the implementations for mesh adaptation adopt a pointer-based mesh data structure, since they are relatively easy to manipulate. In this paper, we develop an array-based mesh data structure to support adaptive mesh refinement and derefinement.[1] It generalizes AHF [12], which provides efficient mesh queries and modification. The array-based mesh data structures have many advantages in the context of numerical simulations, in terms of more compact memory footprint, better interoperability with simulation codes, better efficiency on modern computer architectures with deep memory hierarchy, and relative simplicity and higher efficiency for parallel implementations. However, it is more challenging to support adaptive mesh refinement with array-based mesh data structures, which require dynamic creation and deletion of entities.

The key contributions of this paper are mainly twofold: First, we introduce a simple data model for meshes with hierarchical structure. Our data model is easy to implement and is efficient in both memory and computational cost. When used for meshes smaller than two billion elements per processor on 64-bit architecture, Hierarchical AHF is particular efficient in terms of memory storage. The data structure facilitates both straightforward refinement and derefinement operations, and also allows both conformal and non-conformal meshes. In addition, a generic adaptive mesh refinement (AMR) framework on top of Hierarchical AHF is developed and a prototype is implemented for both 2D triangular and 3D tetrahedral meshes. Second, as an array-based data structure, AHF facilitates better interoperability across different application codes, different programming languages (such as MATLAB, C/C++, FORTRAN, etc.), and different hardware platforms. Meanwhile, since both the tree hierarchy and mesh data are array based, better memory compactness and computational efficiency could be achieved. Moreover, the data model can be easily integrated with multi-level methods such as multigrid solvers. Efficient intra- and inter-level mesh traversals are supported and the data structure is flexible enough to support both uniform mesh refinement (UMR) and AMR. We developed UMR with surface reconstruction for the multigrid method of the finite element method, which we will report elsewhere. The C++ implementation on top of MOAB [28] will be based on the work of MOAB_AHF [12].

The remainder of the paper is organized as follows. Section 2 reviews some background knowledge and related mesh data structures. Section 3 describes our data model for hierarchical meshes. Section 4 describes the algorithms for the construction, query, mesh modification operations, as well as their implementations in MATLAB. Section 5 presents some numerical results. Section 6 concludes the paper with a discussion.

2. Background and Related Work

In this section, we first briefly review the mesh adaptation methodology for numerical PDEs. Then some terminology for mesh data structures is explained and some existing data structures will be referenced for comparison, which will establish the foundation of our proposed data structure.

2.1. Mesh Adaptation for Numerical PDEs

Adaptive methods for numerical PDEs have been an active research area since the late 1970s [4,5] and are widely used in practice nowadays, to balance accuracy and computational efficiency. In particular, Adaptive FEM (AFEM) based on the local mesh refinement has loops of the following form:

SOLVE → ESTIMATE → MARK → REFINE

[1] We use the term "derefinement" instead of "coarsening" because the algorithm would only undo the refinement selectively, and it would not coarsen beyond the original mesh.

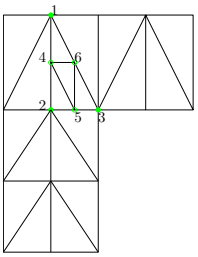


Figure 1: Non-conformal refinement with hanging nodes.

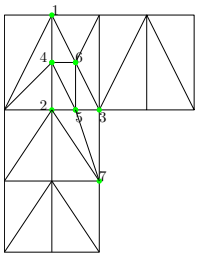


Figure 2: Conformal refinements with transient elements.

to iteratively improve the accuracy of the numerical approximation. Generally, an *a posteriori* error estimator is used to measure the accuracy of obtained numerical solutions, which is exactly the **ESTIMATE** module mentioned above. Elements with large errors are marked and adaptive mesh refinement/coarsening strategies are utilized to minimize the error. Some of the refinement strategies will deliver conformal meshes while others will not. Our goal is to design a data structure that supports both conformal and non-conformal adaptive mesh refinement.

In the process of adaptive mesh refinement, an *a posteriori* error estimator would indicate elements with large error. These elements would be marked and in the *h*-adaptivity, the **REFINE** module refines all these marked elements. Different refinement strategies have been considered. In 2D, during the middle of the 1980s Rivara introduced an effective mesh refinement algorithm based on longest edge bisection [24] while Mitchell developed a recursive algorithm for the newest vertex bisection [21] and Bank adopted regular refinement with selected temporary bisections [2]. Bank's method is known as red-green refinement and was used in the software package PLTMG [9]. In the beginning of the 1990s, Rivara and Levin extended the longest edge refinement algorithm to tetrahedral meshes [25]. However, it is not clear whether this algorithm degrades the mesh quality. Meanwhile, extensions of red-green refinement to 3D were considered in [11] and Bänsch generalized the newest vertex bisection method to 3D [10]. Both of them preserve the mesh quality under refinement. Similar approaches were developed by Liu and Joe [19] and Arnold et al. [3]. Moreover, Kossaczky [16] derived a recursive variant of Bänsch's algorithm, with a bisection rule for tetrahedra using the local order of vertices and element type. This concept is convenient for implementation and generalization to any space dimension. For a more complete discussion of mesh refinement, see [13,22].

Generally, the newest vertex refinement will deliver a conformal mesh, but the minimum angles of the mesh will be degraded. On the other hand, regular refinement would preserve the angles during refinement, but irregular nodes (or hanging nodes) will be created. For instance, see Figure 1; triangle $\Delta 123$ is marked and refined in a regular way. The refinement would result in unbalanced vertices 4, 5, and 6, a.k.a. "hanging nodes." Generally, there are two strategies to deal with hanging nodes in FEM: 1) associate degrees of freedom with the hanging nodes and eliminate them in the linear system according to continuity constraints [8,26]; or 2) convert the neighboring cells of the hanging nodes into transient cells, as shown in Figure 2. The latter is often referred to as red-greed refinement [2].

In terms of implementation, 1-irregularity rule is often applied for non-conformal AMR, which requires that each edge has at most one hanging node. In general, k-irregularity rule [26] could be applied, which means that each edge could have at most k hanging nodes. Then, $k = 0$ means no hanging nodes are allowed and the refined mesh remains conformal, and $k = \infty$ corresponds to adaptivity with arbitrary-level hanging nodes.

2.2. Terminology

In our setting, a *mesh* is a simplicial complex discretely representing a geometric or topological object. Topologically, a d-dimensional object is a *manifold with boundary* if every point in it has a neighborhood homeomorphic to either a d-dimensional ball or half-ball, where the points whose neighborhood is homeomorphic to a half-ball are *boundary* (or *border*) *points*. We say a mesh is 1D, 2D, or 3D if the object that it represents is topologically 1D, 2D, or 3D, respectively. A *mesh* is composed of 0D, 1D, 2D, and 3D entities, which we refer to as *vertices*, *edges*, *faces*, and *cells*, respectively. Typically, a face is either a triangle or quadrilateral, and a cell is a tetrahedron, prism, pyramid, or hexahedron, especially for finite element methods, although general polygons and polyhedra are also often used in finite volume meshes.

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*. More specifically, the facets of a cell are its faces, the facets of a face are its edges, and the facets of an edge are its vertices. Each facet 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 as *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 half-facet without any siblings is a *border half-facet*, and we refer to a vertex incident on a border half-facet as a *border vertex*.

2.3. Half-Facet Data Structure

The *half-facet data structure* is a generalization of the concept of the *doubly-connected edge list* (DCEL) for surface and volume meshes [1,12,17]. In an oriented manifold surface mesh, suppose the edges within each face can be ordered in a counter-clockwise direction with respect to the outward normal (or upward normal for 2D meshes). For a volume mesh, within each cell, suppose the edges of each face are oriented in a counter-clockwise order with respect to the outward normal of the cell. In 2D, the edges within each face are called *directed edges* or *half-edges* while we refer to the oriented faces as *half-faces* in 3D. Each edge has two incident faces, and the two half-edges have opposite orientations and hence are said to be *opposite* or *twin* half-edges of each other. An edge on the boundary does not have a twin half-edge. For typical meshes in engineering applications, each face in the interior of a volume mesh has two corresponding half-faces with opposite orientations, which are said to be *opposite* or *twin* half-faces of each other.

2.4. Pointer Versus Array Based Implementations

A mesh data structure may be implemented using either pointers or arrays. The pointer-based implementations are more common, since they are relatively easy to manipulate. deal.II [7], a C++ finite element library, supports *hp*-FEM in 1D, 2D (quadrilaterals) and 3D (hexahedra), and allows hanging nodes introduced in *hp*-adaptivity. Hanging nodes are eliminated according to continuity constraints [8]. Likewise, libMesh [15] is also a C++ library for serial/parallel adaptive algorithms. Hermes2D [27] supports adaptive FEM in 2D based on the algorithm in [26].

In our work, we choose to use an array-based, pointer-free implementation for a number of reasons. First, 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. Second, using arrays can also lead to faster memory access and hence better efficiency. 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).

3. Data Model for Hierarchical Meshes

In this data model, we assume that each element has a standard numbering convention for its vertices and its facets. For standard elements, we follow the convention of the CGNS (CFD General Notation System) [23,29]. We do not require explicit representation of intermediate dimensional entities between 1 and $d-1$. Instead, we treat the half-facets as *implicit entit*ies, and refer to a half-facet using the element ID and its local ID within the element.

In the process of AMR, to avoid the duplication of new vertices introduced by refinement, efficient adjacency queries are critical. The AHF data structure provides efficient query operations with nice memory performance. A hierarchical structure is generally necessary for multi-level methods for the linear system of numerical PDEs. In our data model the hierarchy is stored in an array-based tree-like structure. We refer to this data model as the *Hierarchical AHF*.

3.1. Hierarchical Structure

The design of the mesh data structure for adaptive mesh refinement assumes that we start with a conformal manifold mesh. An initial conformal mesh is easy to generate and it is natural to form a hierarchical structure by mesh adaptation. The refinement and derefinement require efficient adjacency queries, which are provided by AHF.

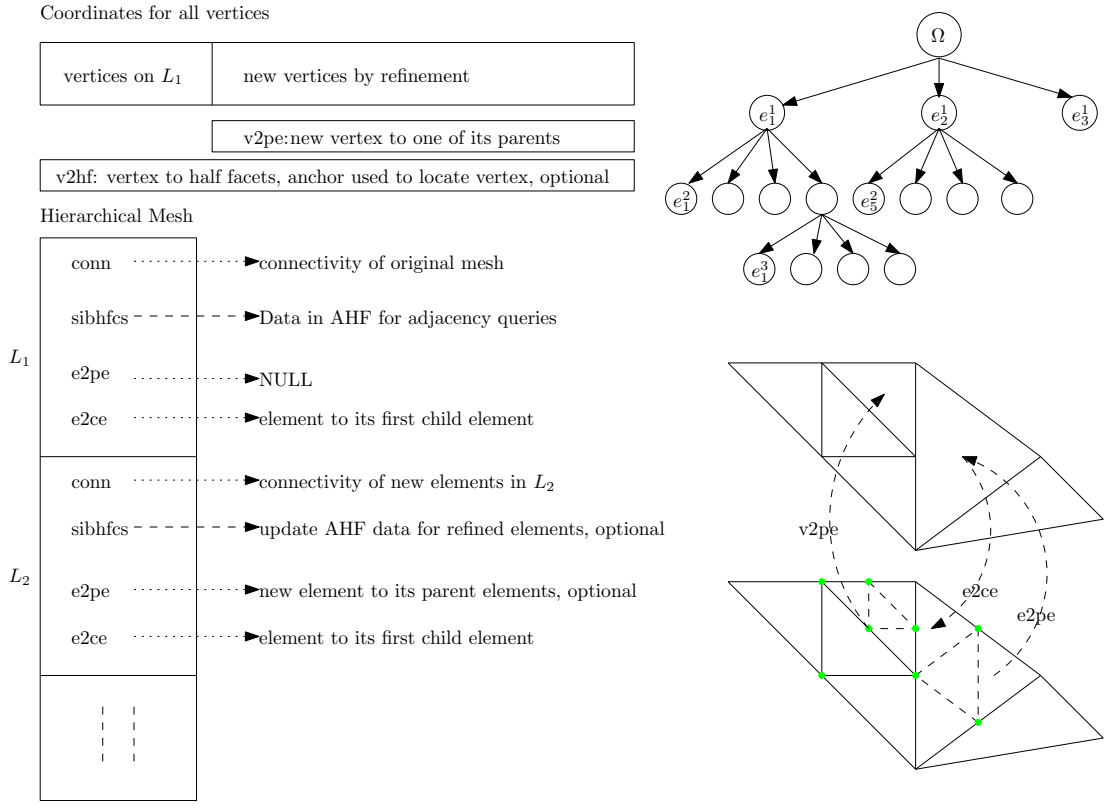


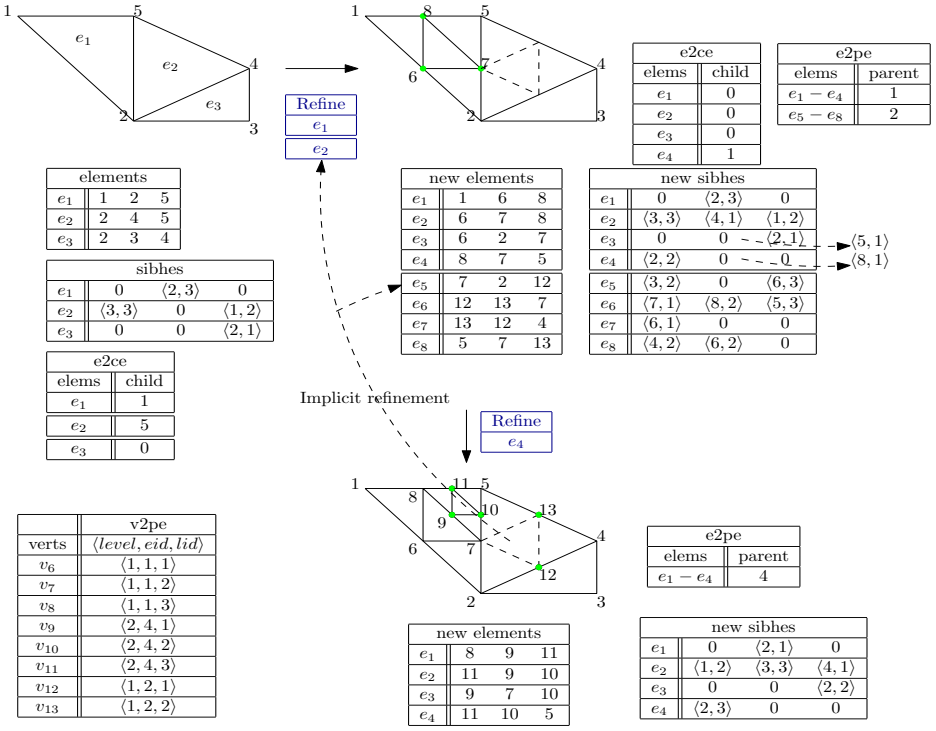
Figure 3: Hierarchical array-based half-facet data structure for a multi-level mesh.

The initial mesh is adaptively refined and the results will be stored in a hierarchical structure. The resulting elements from a subdivision of element K will be referred to as the *child elements* of K, which in turn is called the *parent*. If element K is refined, then it is said to be *inactive*. Elements generated from subsequent refinement of the children of K will be called the *descendants* of K. On the first level, the original mesh is stored. Then some elements of the mesh are marked for refinement. The second level would be the child elements of these elements; see Figure 3. On each level, the child elements of the upper level will form a conformal mesh (which might not be manifold). This is analogous to quad-tree. For instance, in Figure 3, on level 1, element e_1^1, e_2^1, e_3^1 form the initial conformal mesh. On the second level, the children of e_1^1 and e_2^1 , i.e. $e_1^2, \ldots, e_8^2$ will also form a conformal mesh. To traverse the tree, we store *e2ce* for each element, which is the mapping from the elements to the IDs of their child elements on the next level. On each level, *e2ce* is represented as an array. For regular refinement, *e2ce* will only store the ID of the first child element, since all children are stored in consecutive order in an array.

3.2. Hierarchical AHF

In the hierarchical mesh data structure, the topological information, i.e., the connectivity table of elements will be stored for each level of the mesh. The original mesh is treated as the first level of the mesh. During the refinement, some elements of the mesh are marked to be refined. The second level would be the child elements; see Figure 3. On each level, the connectivity will be stored in *conn* of the mesh data structure. Each level of the sub-mesh will contain vertices both from the current level and previous levels, thus storing vertices for each level would be a waste of memory. Therefore we store the geometric data, i.e. coordinates of all vertices, in a separate array. The Hierarchical AHF representation is illustrated in Figure 3.

To support efficient intra- and inter-level queries, auxiliary information is necessary. For the intra-level queries, the neighboring information, i.e. AHF data will be stored in an array for each level sub-mesh. Since the sub-mesh



elements			
e_1	1	2	5
e_2	2	4	5
e_3	2	3	4

sibhes			
e_1	0	$\langle 2,3\rangle$	0
e_2	$\langle 3,3\rangle$	0	$\langle 1,2\rangle$
e_3	0	0	$\langle 2,1\rangle$

e2ce	
elems	child
e_1	1
e_2	5
e_3	0

new elements			
e_1	1	6	8
e_2	6	7	8
e_3	6	2	7
e_4	8	7	5
e_5	7	2	12
e_6	12	13	7
e_7	13	12	4
e_8	5	7	13

new sibhes			
e_1	0	$\langle 2,3\rangle$	0
e_2	$\langle 3,3\rangle$	$\langle 4,1\rangle$	$\langle 1,2\rangle$
e_3	0	0	$\langle 2,1\rangle$
e_4	$\langle 2,2\rangle$	0	0
e_5	$\langle 3,2\rangle$	0	$\langle 6,3\rangle$
e_6	$\langle 7,1\rangle$	$\langle 8,2\rangle$	$\langle 5,3\rangle$
e_7	$\langle 6,1\rangle$	0	0
e_8	$\langle 4,2\rangle$	$\langle 6,2\rangle$	0

$\langle 5,1\rangle$
$\langle 8,1\rangle$

e2ce	
elems	child
e_1	0
e_2	0
e_3	0
e_4	1

e2pe	
elems	parent
$e_1 - e_4$	1
$e_5 - e_8$	2

	v2pe
verts	$\langle level, eid, lid\rangle$
v_6	$\langle 1,1,1\rangle$
v_7	$\langle 1,1,2\rangle$
v_8	$\langle 1,1,3\rangle$
v_9	$\langle 2,4,1\rangle$
v_{10}	$\langle 2,4,2\rangle$
v_{11}	$\langle 2,4,3\rangle$
v_{12}	$\langle 1,2,1\rangle$
v_{13}	$\langle 1,2,2\rangle$

e2pe	
elems	parent
$e_1 - e_4$	4

new elements			
e_1	8	9	11
e_2	11	9	10
e_3	9	7	10
e_4	11	10	5

new sibhes			
e_1	0	$\langle 2,1\rangle$	0
e_2	$\langle 1,2\rangle$	$\langle 3,3\rangle$	$\langle 4,1\rangle$
e_3	0	0	$\langle 2,2\rangle$
e_4	$\langle 2,3\rangle$	0	0

Figure 4: Example of Hierarchical AHF under refinement.

is conformal on each level, AHF data can be built in a natural way and the data is represented as *sibhfcs* (sibling half-facets) in the mesh data structure in Figure 3. In the process of mesh adaptation, the neighbor information *sibhfcs* will be updated incrementally. For inter-level queries, extra information (like *e2pe* and *e2ce)* is stored in arrays. For each element on a certain level, *e2ce* is the ID of the first child element on the next level. On the next level, other child elements of this element will be stored next to the first child element. *e2ce* is stored in an array for each level and it is necessary for inter-level traversals. *e2pe*, element to parent element, is the ID of the parent element, which is on the previous level of this element, and it is optional.

To support efficient queries to the parent element for each new vertex, a separate mapping *v2pe* is stored. For each new vertex, *v2pe* is an array of tuples: *level, eid, lid*, where *level* is the level of its parent element, *eid* is the ID of the parent element in *level*, *lid* is the canonical ID of this vertex in its parent ID. If the 1-irregularity rule is applied, the *lid* would be the same as the local ID of the refined edge. *v2pe* is generally necessary for multi-level methods. Also we could use *v2pe* to determine which vertex is a hanging node on which level. Generally, if the 1-irregularity rule is applied, vertex *v* could only be a hanging node on level *v2pe(v).level+1*. This could be further determined by checking if *v2pe(v).eid*'s sibling elements are refined. If not, then *v* is a hanging node.

In Figure 4, we illustrate the data structure by refining a simple mesh. First, a user specifies refinement of e_1. The new elements will be created and *e2ce* in level 1 will be updated. Then the user specifies e_4 on the second level to be refined. Here the 1-irregularity rule is applied to keep the mesh graded. This will introduce an implicit refinement of e_2 on the first level. Correspondingly, data on level 2 will be updated. *v2pe* will be stored in a separate array.

4. Construction and Modification of Hierarchical AHF

In this section, we describe some detailed algorithms for the construction of Hierarchical AHF, as well as some query and modification operations. Since AHF is array-based, these algorithms can be implemented in many programming languages, including MATLAB, C/C++, FORTRAN, etc. We will also describe our implementation in MATLAB.

4.1. Construction of Hierarchical AHF

In the half-facet data structure, there are two components: *sibhfcs* (sibling half-facets) and *v2hf* (vertex to half-facet). The former is central to AHF, as nearly all adjacency queries require it. These sibling half-facets should map to each other and form a cycle. The latter array, *v2hf*, is optional for many operations; for Hierarchical AHF, it is not built for new vertices. Instead, *v2pe* (vertex to parent element) is constructed to store information of new vertices.

In a hierarchical mesh, the AHF for the initial mesh will be constructed first and then the sub-mesh of each level is constructed incrementally, taking advantage of ancestor information. In general, the refinement and derefinement require different algorithms. In the following, we describe these two parts in a manner independent of the dimension of the mesh.

4.1.1. Hierarchical AHF: Refinement

Algorithm 1 outlines the steps for mesh refinement, which is applicable to half-facets in arbitrary dimensions, and is particularly efficient in 1 to 3 dimensions. New elements are created and appended in corresponding levels. Meanwhile, the adjacency information, *sibhfcs*, is updated. This step requires the input of elements that are marked for refinement:

refTags: arrays stored in a hierarchical structure, which store the elements to be refined on each level.

Algorithm 1 Update Sibling Half-Facets for Refinement.

Input: hielems: hierarchical mesh data, refTags
Output: sibhfcs: cyclic mappings of sibling half-facets for each level of mesh

```
1:  for each level in hielems do
2:    for each element e in refTags(level) do
3:      for each edge in element e do
4:        Loop through elements in level incident to edge to check if edge is refined
5:        if edge is not refined then
6:          Refine edge by inserting vertex v in the middle;
7:          Update v2pe for vertex v, v2pe(v) = ⟨level, e, edge⟩;
8:      Refine element e by predefined strategy and update e2ce(e)
9:      Construct sibhfcs for children of element e;
        {Update sibhfcs for submesh on level+1:}
10:     for each facet in element e do
11:       Check opposite element of facet on level of submesh
12:       if opposite element is refined then
13:         Update sibhfcs for children of element e;
14:         Update sibhfcs for children of opposite element;
15:       else
16:         Update sibhfcs for children of element e;
```

The computational cost of Algorithm 1 is linear, assuming that the number of elements incident on an edge is bounded by a small constant c. For the storage requirement, let $|C_r|$ denote the number of elements to be refined in a certain level of the mesh. The amortized memory storage increased by refinement will be approximately $(2^d v_c + 2^d f_c + 2^d)|C_r|$ integers, for the connectivity, the neighbor information, and inter-level maps, with extra space for new vertices coordinates and $v2pe$ map. Here v_c and f_c are the numbers of vertices per cell and the number of faces per cells, 2^d is the number of children per element.

4.1.2. Hierarchical AHF: Derefinement

During the second step, we update the sibling half-facets during derefinement. This step requires the input of elements that are marked for derefinement:

derefTags: a hierarchical structure which stores elements to be derefined on each level. For each element e in derefTags, we assume that e is refined and the derefinement operation will remove all children of e and set e to be active.

Algorithm 2 outlines the procedure for this stage, which is applicable to half-facets of arbitrary dimensions. Particularly, a vertex is active if and only if it has incident cells. A hanging node will be set as inactive if all incident elements are removed during derefinement.

Algorithm 2 Update Sibling Half-Facets for Derefinement.

Input: hielems: hierarchical mesh data, derefTags
Output: sibhfcs: cyclic mappings of sibling half-facets for each level of mesh

```
1:  for each level in hielems do
2:    for each element e in derefTags(level) do
3:      for each edge in element e do
4:        Loop through elements in level incident to edge;
5:        if none of incident elements is refined then
6:          Derefine edge by removing vertex v which is in the middle;
7:          Update v2pe for vertex v, v2pe(v) = ⟨0, 0, 0⟩;
8:        else
9:          edge cannot be derefined;
10:         vertex v which is in the middle is still active
11:     Derefine element e and set e2ce(e) = 0;
12:     Set all its children mute;
        {Update sibhfcs for submesh on level + 1:}
13:     for each facet in element e do
14:       Check opposite element of facet on level of submesh
15:       if opposite element is refined then
16:         Update sibhfcs for children of opposite element;
17:       Set sibhfcs for children of element e to zeros;
```

Similar to Algorithm 1, the computational cost of Algorithm 2 is also linear, assuming that the number of elements incident on an edge is bounded by a small constant c. To analyze the storage requirement, let $|C_d|$ denote the number of elements to be coarsened in a certain level of the mesh. The update will introduce approximately $(2^d v_c + 2^d f_c + 2^d)|C_d|$ "holes" in the element connectivity, *sibhfcs* arrays and the map $e2pe$. Dynamic memory management could be utilized to reuse such holes, for instance, by building a queue to record the holes in the corresponding array introduced by deletion and having any new insertion reuse the memory.

4.2. Mesh Adaptation

Mesh refinement and derefinement can be implemented relatively easily in AHF. For hierarchical meshes, AHF is particularly attractive because the adaptivity could be performed efficiently and AHF can be modified incrementally. This leads to very modular adaptivity strategies. To avoid excessive memory copying, we expand the array by a small percentage (e.g. by 20%) each reallocation, so that the amortized cost for the local modifications is constant.

The data structure could support a refinement strategy whether the mesh is required to be conformal or not. In our MATLAB implementation, we support both regular refinement and red-green refinement (Figure 5). Particularly, we enforce the 1-irregularity rule for the non-conformal refinement. The Kelly error indicator [14] is utilized for estimating accuracy and marking elements. The AHF code [12] is used to generate sibling half facet data for the initial mesh.

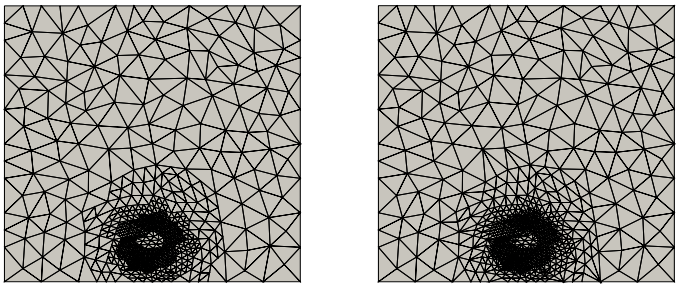
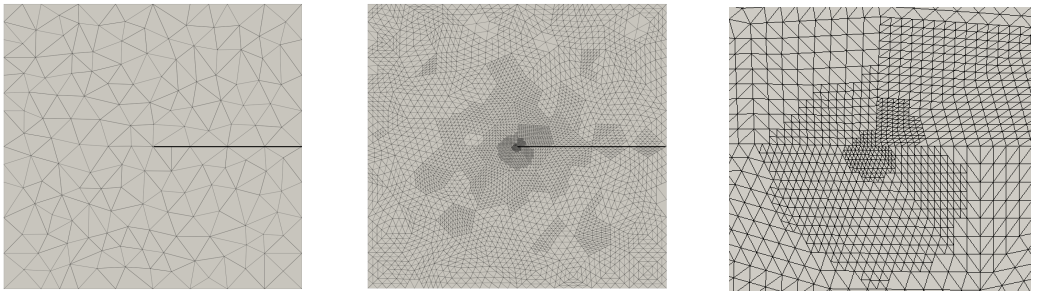


Figure 5: Red-Green Refinement: Before and After Elimination of Hanging Nodes



(a) Original Mesh (b) 6th AMR (c) Zoomed-In Mesh Around Center

Figure 6: AFEM: Mesh adaptation for re-entrant corner problem

5. Numerical Experiments

5.1. Adaptive Finite Element Method (AFEM)

We implemented AMR for the re-entrant corner problem from [20] with the Kelly error indicator. The equation is

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = 0$$

on domain $[-1,1] \times [-1,1] \backslash \{0 \le x \le 1, y = 0\}$. The boundary condition is $u = g$ and the exact solution is

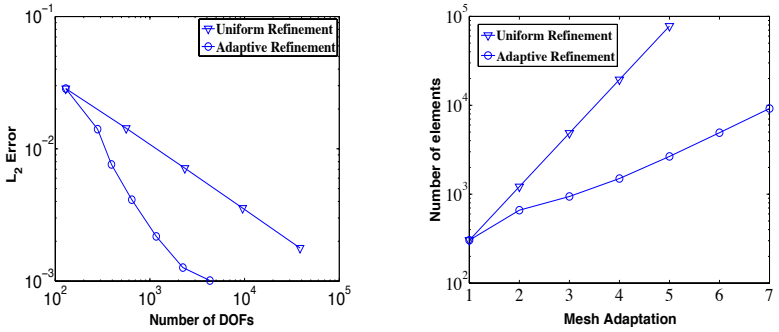
$$u = r^{\frac{1}{2}} \sin\left(\frac{\theta}{2}\right)$$

where $r = \sqrt{x^2 + y^2}$, $\theta = \tan^{-1}(y/x) \in [0, 2\pi)$.

We apply 6 adaptive refinements over the original mesh and compare it with FEM on a mesh with uniform regular refinement. The results are shown in Figure 6, 7. The original mesh has a crack $\{0 \le x \le 1, y = 0\}$ along which the solution is not smooth. The refinement is centralized along the crack due to the non-smoothness of the solution, see Figure 6b. The result in Figure 7a shows that the adaptive FEM approach delivers a better convergence rate than FEM over a uniformly refined mesh. The L_2 error is computed as $\int_\Omega |u - u_h|^2 dA$. The numerical results indicate that the same accuracy could be achieved with many less DOFs or number of elements.

5.2. Comparison with Pointer-Based Data Structure

We will compare the storage, through theoretical analysis, for Hierarchical AHF with the pointer-based data structure in libMesh, as it is the most closely related to our data structure. Let C and V represent the set of cells and vertices of the given mesh, and let $|\cdot|$ denote the size of a set. For Hierarchical AHF, let C_1 and V_1 denote the cells and vertices,



(a) Convergence rate (b) Number of elements

Figure 7: AFEM: Numerical results for re-entrant corner problem

respectively, in the original, unrefined mesh. We will consider an implementation of Hierarchical AHF that includes the element connectivity (*elements*), vertex to parent element mapping (*v2pe*), sibling half-facet mapping (*sibhfcs*), element to parent element mapping (*e2pe*) and element to child element mapping (*e2ce*). Since the vertex to incident half-facet mapping (*v2hf*) is optional, we will not include it in this analysis. Therefore, we have the following five maps which require the following number of entities:

element connectivity: $n_c = v_c\,|C|$
vertex to parent element map: $n_p = |V| - |V_1|$
sibling half-facet map: $n_s = f_c\,|C|$
element to parent map: $n_{ep} = |C| - |C_1|$
element to child map: $n_{ec} = |C|$

where v_c and f_c are the numbers of vertices per cell and the number of faces per cells, respectively. In general, the entities are stored as 32-bit integers. For the half-facet ID $\langle eid,\ lfid\rangle$, we encode it in a 32-bit integer. For the vertex to parent element map we store $\langle level,\ eid,\ lid\rangle$ as two 32-bit integers, one for *level* and one for $\langle eid,\ lfid\rangle$. Thus the storage in bytes is

$$\begin{aligned} S_{\text{AHF32}} &= 4n_c + 8n_p + 4n_s + 4n_{ep} + 4n_{ec} \\ &= 4\,(2 + v_c + f_c)\,|C| - 4\,|C_1| + 8\,|V| - 8\,|V_1| \end{aligned}$$

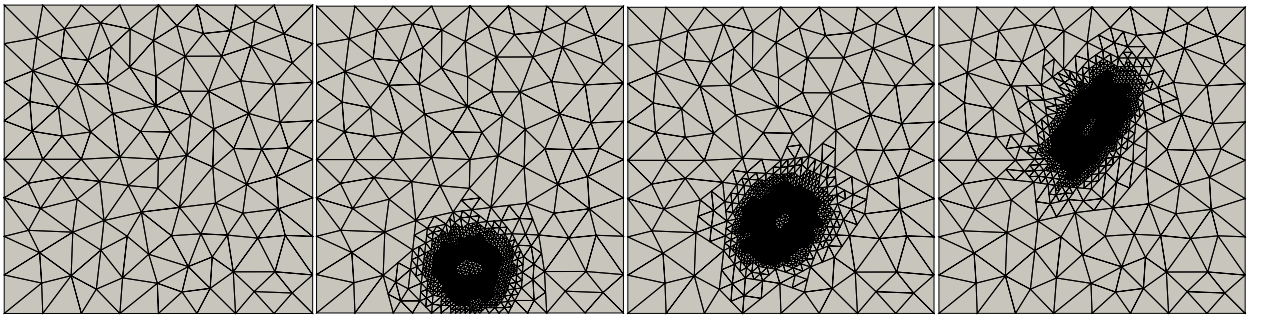
If we were to store the entities as 64-bit integers, the storage would effectively double.

For each cell, libMesh stores the element connectivity and the "face neighbors" of the cells. Two cells are face neighbors if they share a side; in 1D a side is a vertex, in 2D a side is an edge, and in 3D a side is a face. Like Hierarchical AHF, adaptive mesh refinement and coarsening is central to libMesh and hence the cells and their ancestors are stored in a tree. Specifically, a pointer to the parent of an element and an array of pointers to its children (if any) are stored. In general, a d-dimensional element is refined into 2^d children of the same type except when dealing with pyramids, which are refined into pyramids and tetrahedra. For the sake of simplicity, we will use 2^d as the number of children of an element. Note that the level of an element is not stored in libMesh, since this can be found recursively from the parents. To store nodal information, libMesh has a node class. Each object in the node class stores the coordinates of the node, a unique global ID number and the degree of freedom indices. Since we are comparing the storage for the topological information of the mesh, we will consider the storage cost of the global ID number. Therefore we have 4 maps requiring the following number of entities:

element connectivity: $n_c = v_c|C|$
neighboring objects: $n_n = f_c|C|$
hierarchical: $n_h = |C| + 2^d|C_r|$
nodal: $n_v = |V|$

Table 1: Comparison of storage requirements in kilobytes of Hierarchical AHF and libMesh for a mesh in various stages of refinement on 32-bit and 64-bit architectures. On 64-bit architecture, one may store Hierarchical AHF with 32-bit integers or 64-bit integers. This comparison is based on our theoretical analysis.

	32-Bit Architecture		64-Bit Architecture		
	Hier AHF	libMesh	Hier AHF (32-Bit)	Hier AHF (64-Bit)	libMesh
Original Mesh	57.867	59.535	57.867	115.734	119.070
Refinement 1	85.070	86.105	85.070	170.141	172.210
Refinement 2	142.742	142.441	142.742	285.484	284.882
Refinement 3	233.516	231.266	233.516	467.031	462.531
Refinement 4	332.398	328.051	332.398	664.797	656.102
Refinement 5	446.680	440.180	446.680	893.359	880.359



(a) Position 1: Before Adaptation (b) Position 1: After Adaptation (c) Position 2: After Adaptation (d) Position 5: After Adaptation

Figure 8: Example triangular mesh during adaptive mesh refinement.

where C_r is the number of refined cells in the mesh. Since all these mappings are stored as pointers, if we assume 32-bit architecture, then we can estimate the storage in bytes as

$$\begin{aligned} S_{\text{libMesh32}} &= 4n_c + 4n_n + 4n_h \\ &= 4\left(1 + v_c + f_c\right)|C| + 4 \cdot 2^d |C_r| + 4|V| \end{aligned}$$

On 64-bit architectures, the storage would double.

As an example, Table 1 shows the storage required by the first six meshes used in Section 5.3 for Hierarchical AHF and libMesh. It can be seen that the memory cost of the two approaches is comparable if an integer has the same length as a pointer. However, Hierarchical AHF would require about half of the storage on modern 64-bit architectures for meshes with less than two billion elements per processor.

5.3. Mesh Adaptation

Hierarchical AHF supports both efficient refinement and derefinement. Here we illustrate the results in Figure 8. A function from [13], i.e. $x(x-1)y(y-1)e^{-100((x-0.5)^2+(y-0.117)^2)}$ over the domain $[0,1]\times[0,1]$ and its counter-clockwise rotations serve as a series of numerical solutions. The Kelly error estimator is used to drive the AMR algorithm to mark and adapt the mesh. The function is rotated 4 times, thus it has 5 positions, referred to as position 1 (i.e. original function), position 2, and so on. Starting from position 1, the original mesh (Figure 8a) is adapted by the solution at this position. Then the solution is rotated to position 2 and AMR is applied over the mesh in position 1 (Figure 8b), and a new mesh (Figure 8c) is obtained in position 2, so on and so forth. At each position we make 5 adaptations. The number of active elements at each adaptation can be found in Figure 9.

To demonstrate the mesh adaptation in 3D, we define a function $e^{-1000((x-x_c)^2+(y-y_c)^2+(z-z_c)^2)}$ over the unit cube $[0,1]^3$, and rotate its center (x_c, y_c, z_c) along the plane $z_c = 0.5$ five times. We perform AMR based on the approximation errors to this series of functions. Figure 11 shows the cross-sections of the initial mesh and the mesh at three different

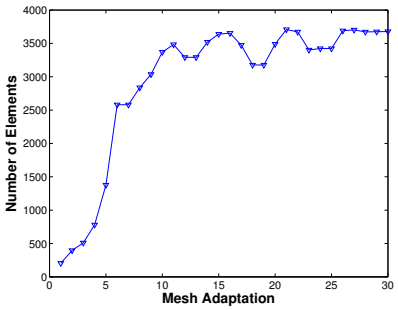


Figure 9: AMR for 2D triangular mesh: number of active cells

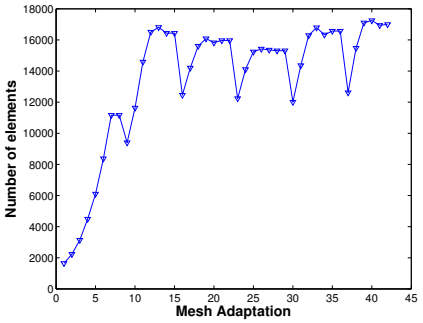
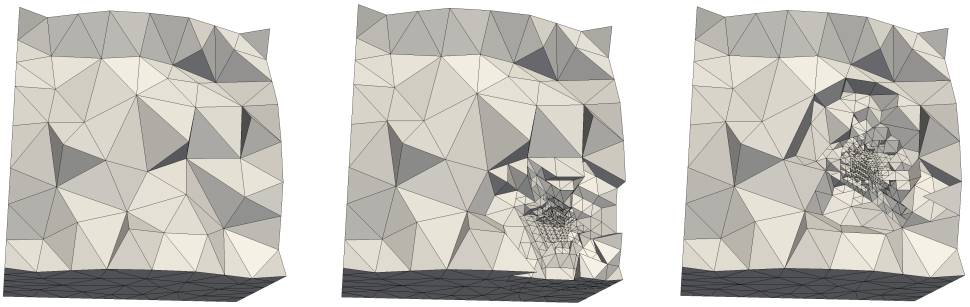


Figure 10: AMR for 3D tetrahedra mesh: number of active cells



(a) Position 1: Before Adaptation (b) Position 1: After Adaptation (c) Position 6: After Adaptation

Figure 11: Example AMR in 3D: cross-sections of tetrahedral mesh.

stages. Similar to the 2D results, the number of elements remained approximately constant during the adaptation process, see Figure 10.

6. Conclusion and Discussion

In this paper, we presented a simple but general array-based half-facet mesh data structure, called Hierarchical AHF, for hierarchical meshes under adaptive mesh refinement. We described the algorithms and a prototype implementation in MATLAB for both refinement and derefinement for 2D triangular and 3D tetrahedral meshes. We demonstrate that Hierarchical AHF is efficient in terms of both storage and computational costs. Hierarchical AHF is especially competitive if using 32-bit integers on 64-bit architecture when compared to a pointer-based implementation. Our framework could be easily integrated with finite element codes that support nonconformal meshes. The numerical results indicate the effectiveness of the adaptive procedures. In addition, our data structure is easily extended to support red-green refinement, so that it can also be used with finite element codes that require conformal meshes.

Hierarchical AHF stores all the information using arrays instead of pointers. Due to its array-based nature, it is well suited for parallel computations and is relatively easier to port onto GPUs. In addition, it facilitates easier interoperability with application codes. This tree hierarchy could be further utilized by multigrid or multilevel methods, which are often used as solvers of the arising linear systems for large scale simulations. We plan to explore these aspects in our future research.

Acknowledgements

This work was funded in part by the SciDAC program funded by the Office of Science, Advanced Scientific Computing Research of the U.S. Department of Energy, under Contract DE-AC02-06CH11357, and by a subcontract

to Stony Brook University from Argonne National Laboratory under the same. The last author was also supported by Army Research Office under Grant W911NF1310249.

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] R. B. Andrew, A. H. Sherman, and A. Weiser. Some refinement algorithms and data structures for regular local mesh refinement, 1983.
[3] D. N. Arnold, A. Mukherjee, and L. Pouly. Locally adapted tetrahedral meshes using bisection. *SIAM Journal on Scientific Computing*, 22(2):431–448, 2000.
[4] I. Babuska and W. Rheinboldt. Error estimates in adaptive finite element computations. *SIAM J. Numer. Anal.*, 15:736–754, 1978.
[5] I. Babuska and W. Rheinboldt. A posteriori error estimates for the finite element method. *Internat. J. Numer. Methods Engrg.*, 12:1597–1615, 1978.
[6] I. Babuska and W. C. Rheinboldt. A posteriori error analysis of finite element solutions for one-dimensional problems. *SIAM Journal on Numerical Analysis*, 18(3):565–589, 1981.
[7] W. Bangerth, R. Hartmann, and G. Kanschat. deal.II - a general-purpose object-oriented finite element library. *ACM Transactions on Mathematical Software (TOMS)*, 33(4):24, 2007.
[8] W. Bangerth and O. Kayser-Herold. Data structures and requirements for hp finite element software. *ACM Transactions on Mathematical Software (TOMS)*, 36(1):4, 2009.
[9] R. E. Bank. *PLTMG, a Software Package for Solving Elliptic Partial Differential Equations: Users' Guide 8.0*, volume 5. Siam, 1998.
[10] E. Bänsch. Local mesh refinement in 2 and 3 dimensions. *IMPACT of Computing in Science and Engineering*, 3(3):181–191, 1991.
[11] F. Bornemann, B. Erdmann, and R. Kornhuber. Adaptive multivlevel methods in three space dimensions. *International Journal for Numerical Methods in Engineering*, 36(18):3187–3203, 1993.
[12] 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.
[13] M. T. Jones and P. E. Plassmann. Adaptive refinement of unstructured finite-element meshes. *Finite Elements in Analysis and Design*, 25(1):41–60, 1997.
[14] D. Kelly, D. S. Gago, O. Zienkiewicz, I. Babuska, et al. A posteriori error analysis and adaptive processes in the finite element method: Part I-error analysis. *International Journal for Numerical Methods in Engineering*, 19(11):1593–1619, 1983.
[15] 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(3-4):237–254, 2006.
[16] I. Kossaczkỳ. A recursive approach to local mesh refinement in two and three dimensions. *Journal of Computational and Applied Mathematics*, 55(3):275–288, 1994.
[17] 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.
[18] P. Kus, P. Solin, and D. Andrs. Arbitrary-level hanging nodes for adaptive hp-fem approximations in 3D. *Journal of Computational and Applied Mathematics*, 270:121–133, 2014.
[19] A. Liu and B. Joe. Quality local refinement of tetrahedral meshes based on bisection. *SIAM Journal on Scientific Computing*, 16(6):1269–1291, 1995.
[20] W. Mitchell. Adaptive mesh refinement benchmark problems. http://math.nist.gov/amr-benchmark/index.html, 2013. Accessed:2015-04-24.
[21] W. F. Mitchell. A comparison of adaptive refinement techniques for elliptic problems. *ACM Transactions on Mathematical Software (TOMS)*, 15(4):326–347, 1989.
[22] R. H. Nochetto, K. G. Siebert, and A. Veeser. Theory of adaptive finite element methods: an introduction. In *Multiscale, nonlinear and adaptive approximation*, pages 409–542. Springer, 2009.
[23] D. Poirier, S. R. Allmaras, D. R. McCarthy, M. F. Smith, and F. Y. Enomoto. The CGNS system, 1998. AIAA Paper 98-3007.
[24] M.-C. Rivara. Mesh refinement processes based on the generalized bisection of simplices. *SIAM Journal on Numerical Analysis*, 21(3):604–613, 1984.
[25] M.-C. Rivara and C. Levin. A 3-D refinement algorithm suitable for adaptive and multi-grid techniques. *Communications in Applied Numerical Methods*, 8(5):281–290, 1992.
[26] P. Šolín, J. Červenỳ, and I. Doležel. Arbitrary-level hanging nodes and automatic adaptivity in the *hp*-fem. *Mathematics and Computers in Simulation*, 77(1):117–132, 2008.
[27] P. Solin, L. Korous, and P. Kus. Hermes2D, a C++ library for rapid development of adaptive hp-FEM and hp-DG solvers. *Journal of Computational and Applied Mathematics*, 270:152–165, 2014.
[28] T. Tautges, R. Meyers, and K. Merkley. MOAB: A mesh-oriented database. Technical report, Sandia National Laboratories, 2004.
[29] The CGNS Steering Sub-committee. *The CFD General Notation System Standard Interface Data Structures*. AIAA, 2002.