# Modifiable Array Data Structures for Mesh Topology

Dan Ibanez Mark S Shephard

February 27, 2016

#### Abstract

Topological data structures are useful in many areas, including the various mesh data structures used in finite element applications. Based on the graph-theoretic foundation for these data structures, we begin with a generic modifiable graph data structure and apply successive optimizations leading to a family of mesh data structures. The results are compact array-based mesh structures that can be modified in constant time. Specific implementations for finite elements and graphics are studied in detail and compared to the current state of the art.

# 1 Introduction

An unstructured mesh simulation code relies heavily on multiple core capabilities to deal with the mesh itself, and the range of features available at this level constrain the capabilities of the simulation as a whole. As such, the long-term goal towards which this paper contributes is the development of a mesh data structure with the following capabilities:

- 1. The flexibility to deal with evolving meshes
- 2. A highly scalable implementation for distributed memory computers
- 3. The ability to represent any of the conforming meshes typically used by Finite Element (FE) and Finite Volume (FV) methods
- 4. Minimize memory use
- 5. Maximize locality of storage
- 6. The ability to parallelize work inside supercomputer nodes with hybrid architecture such as accelerators

This paper focuses on the first five goals; the sixth will be the subject of a future publication. In particular, we present a derivation for a family of structures with these properties:

- 1. The representation centers around graph theoretic interpretations of topological adjacency
- 2. The mesh can remain topologically similar to and associated with a geometric model
- 3. The common element types of FE/FV methods can coexist in one structure
- 4. Additional data can be associated with entities to implement high order basis functions, including for geometric approximation
- 5. A mesh can be modified by adding and removing single entities in constant time
- 6. The entire mesh is stored in a few contiguous dynamic arrays

One major contribution of this paper is to show that the latter two properties, array storage and rapid single-entity modification, are not mutually exclusive and can be combined in a viable way.

Sections 2 and 3 begins with several concepts that form an abstraction of meshes from a topological perspective, and define the class of meshes we are concerned with. Section 4 then covers a derivation starting from a graph data structure, through successive optimization for graphs of meshes, to arrive at a data structure for representing adaptive, conformal meshes. Section 5 compares this structure in terms of quantitative performance measures and features with other similar implementations, and Section 6 presents several example uses of our implementation in practical simulations.

# 2 Nomenclature and Definitions

This section defines concepts and terms of graph and topology theory that are used throughout the remainder of this paper.

### 2.1 Nomenclature

Term	Meaning
r-edge	A graph edge representing a relation between two entities
r-vertex	A graph vertex representing an entity in a given relationship
topological complex	A breakdown of a domain in Cartesian space into topological entities
mesh	A topological complex whose entities have simple shape
m-entity	A topological entity of a mesh
m-vertex	A 0-dimensional m-entity
m-edge	A 1-dimensional m-entity
m-face	A 2-dimensional m-entity
m-region	A 3-dimensional m-entity
m-element	An m-entity not bounding another m-entity

#### 2.2 Graph

A directed graph G = (V, E) consists of a nonempty set of graph vertices V and a set of directed graph edges E. Each directed edge e is an ordered pair of vertices (u, v), and goes from u to v [17].

In order to distinguish between "vertices" and "edges" of relation graphs and mesh vertices and edges, we will refer to the directed graph edges of a relation as r-edges, and its vertices as r-vertices.

An r-vertex v is said to be reachable from u if there exists a path  $(v_0, ..., v_k)$ where  $u = v_0$ ,  $v = v_k$ , and  $(v_i, v_{i+1}) \in E$  for each consecutive pair of vertices in the path.

#### 2.3 Topological Complex

A point set is a subset of the points in some Cartesian space  $\mathbb{R}^D$ .

A topological complex T is a set of point sets containing points in  $\mathbb{R}^D$ . Each point set  $T_i^d$  in T is an open subset of some d-dimensional manifold embedded in  $\mathbb{R}^D$ , where  $0 \le d \le D$ . We say that d is the dimension of point set  $T_i^d$ . We can denote all point sets of dimension d in the complex by  $T^d$ .

All point sets in T are disjoint from one another, and their union  $\Omega = \bigcup T$ is a subset of some D-dimensional manifold, i.e. a D-dimensional manifold with boundary. We denote the boundary of this complex as  $\Gamma = \partial \Omega$ . Each point set is an open subset of a d-manifold, and the closed equivalent on said manifold, denoted  $\overline{T}_i^d = T_i^d \cup \partial T_i^d$ , is the set plus its boundary. Since the sets are disjoint, only their boundaries may intersect. In fact, all intersections of equal-dimension point sets must exist as unions of other, lower-dimensional point sets in T:

$$\forall T_i^d, T_{j \neq i}^d \in T : \exists S \subseteq \{T_k^q \in T | q < d\} : \bigcup S = \partial T_i^d \cap \partial T_j^d$$

Finally, to keep the surface properly divided, we require that the intersection of any point set boundary with the overall boundary also exist as a union of point sets:

$$\forall T_i^d \in T : \exists S \subseteq \{T_k^q \in T \big| q < d\} : \bigcup S = \partial T_i^d \cap \Gamma$$

Boundary-representation (BRep) CAD models are examples of topological complexes, as are meshes. Point sets of dimension 0 are called vertices, those of dimension 1 are called edges, faces have dimension 2 and regions have dimension 3. When discussing a topological complex, we refer to point sets simply as entities.

### 3 Conforming Unstructured Meshes

This section defines a conforming unstructured mesh, which the rest of this paper is concerned with representing in computer memory.

#### 3.1 Adjacency Relation

Given a topological complex T, we can describe the relations between point sets in terms of adjacency. If a point set b bounds a point set  $a, b \subseteq \partial a$ , then we say there is a downward adjacency (a, b). Note that downward adjacency is a transitive relation:

$$c \subseteq \partial b, b \subseteq \partial a \to c \subseteq \partial a$$

For every downward adjacency (a, b), there exists an upward adjacency (b, a). The union of upward and downward adjacencies are called first-order adjacencies.

The first-order adjacency relation defines a graph, which we call the topology graph. The majority of our work is concerned with finding efficient computer representations for topology graphs.

The topology subgraph between a pair of dimensions  $T^p$ ,  $T^q$  is a bipartite graph. We have a notation for queries of this bipartite graph:  $T_i^p\{T^q\}$  is the set of entities (point sets) in  $T^q$  adjacent to  $T_i^p$ . In general, one can query all entities adjacent to a set of entities:  $S\{T^q\} = \bigcup a\{T^q\}, a \in S$ . This makes it easier to define second-order adjacencies, which are found by two transitive queries, for example  $T_i^a\{T^b\}\{T^c\}$ .

Another useful concept will be the *entity use*. This is closely related to a downward adjacency: specifically, if entity b is in the boundary of entity a, then b is used by a, and that occurrence is an entity use.

#### 3.2 Mesh

A mesh M is herein defined as a special case of topological complex where the closure of each entity  $\overline{M}_i^d$  is topologically similar to a polytope of dimesion d. To distinguish meshes from other complexes, we use the prefix m resulting in the terms m-entity, m-vertex, m-edge, m-face, m-region. Mesh entities which do not bound other m-entities are the m-elements.

All m-entities have topological genus zero, i.e. they have no holes, so they do not need multiple loop or shell constructs to describe their boundary the way a BRep CAD model would.

#### 3.3 Finite Element Mesh

We further define *finite element mesh* as a special case of a mesh, with certain restrictions and requirements. For our current purposes, a finite element mesh is composed of m-entities whose closures are one of the following polytope types:

- 1. point (d = 0)
- 2. line (d = 1)
- 3. triangle (d = 2)

- 4. quadrilateral (d=2)
- 5. tetrahedron (d = 3)
- 6. hexahedron (d = 3)
- 7. (square-based) pyramid (d = 3)
- 8. triangular prism (d = 3)

This list can be easily extended to include additional polytope types of interest.

In addition, the finite element method uses fields which are defined as piecewise functions composed of basis functions, controlled by a finite set of degrees of freedom, where each degree of freedom is attached to one mesh entity. This requires a mechanism for associating degrees of freedom with mesh entities.

Finite element analysis procedures also require meshes where the number of elements around some boundary entity (such as a vertex or an edge) is limited to a reasonable upper bound, otherwise shape quality and numerical conditioning will degrade. Therefore, in such meshes, all upward adjacencies are bound by a constant. Any operation whose runtime is proportional to the number of upward adjacencies can be treated as a constant-time operation.

#### 3.4 Mesh Construction and Modification

This work is focused on unstructured meshes where the number of upward adjacencies between mesh entities is not fixed.

Such unstructured meshes have an advantage in representing complex geometry and in their ability to easily vary resolution throughout the geometry. There are two approaches to varying this resolution during a simulation. If an entirely new mesh is constructed, we say that the method is *remeshing*. If local changes are applied to the original mesh to transform it into the new mesh, we say the method *adapts*. Such local changes require adding and removing m-entities from the mesh within local portions of the domain. On the other hand, if the mesh is not changed during the simulation, we say that the mesh is *static*.

Finally, if there are multiple polytope types per dimension, such as having both triangles and quadrilaterals in 2D, then we say the mesh is *mixed*.

#### 3.5 Adaptation

Adaptation refers to a process of modifying the mesh by applying mesh entitylevel operations on mesh cavities. We can define a *mesh cavity* as the union of several mesh entities, in which the mesh modification changes the interior (open set) of mesh entities within the cavity, leaving its boundary unchanged.

Remeshing has a runtime cost at least proportional to the number of total elements, while the cost of adaptation is only proportional to the number of mesh entities modified. Moreover, the transfer of field values from the old mesh to the new mesh is a complex procedure when remeshing, requiring spatial search algorithms and tends to apply remapping operators that are diffusive and/or have to deal with conservation requirements at a global level.

Adaptation by local mesh modification supports local execution of solution transfer: refinement splits parent entities and is able to transfer solution exactly using shape function interpolation, and other operations are confined to a local cavity so that any searching is fast, and diffusive effects and conservation adjustments are local.

Local mesh adaptation requires unique properties of the mesh data structure that are otherwise unnecessary for static meshes. Adaptive procedures are composed of a series of mesh entity additions and removals. Therefore, at a minimum, we require that entity addition and removal be constant-time operations.

Furthermore, general local mesh modifications can introduce temporary topological inconsistencies. For example, the first modification made is either the addition of an entity which overlaps with existing entities or the removal of an entity. Adding an overlapping entity causes inconsistencies such as a face which has three adjacent regions in the temporary mesh. Removing an arbitrary entity can cause non-manifold configurations, such as that of a vertex adjacent to only two elements, which in turn only intersect at that vertex.

For several reasons, it is preferable to modify a cavity by first constructing all new entities that fill the cavity, which overlap with the old, and then destroying all old entities. First, this allows both versions to be considered by a solution transfer algorithm, which needs the mesh topology from both to operate properly in the general case. Second, we are able to evaluate quality and correctness metrics of the new entities and, if those are unacceptable, cancel the operation by destroying the new entities.

By using this approach to cavity modification, we typically start by adding entities and end by removing other, returning the mesh to topological validity. This means the intermediate, invalid state has overlapping entities and higherthan-usual upward adjacencies as opposed to having holes which make it nonmanifold.

Certain modifications, such as edge collapsing, can correctly preserve the boundary of the mesh if and only if there is a direct mapping from mesh entities to geometric (CAD) model entities. This mapping is referred to as classification [18]. For example, sharp edges can be preserved when it is known that a mesh entity is on such a CAD model edge. The inverse map of classification is called reverse classification, and it defines the groups of mesh entities to which boundary conditions are applied.

The implementations of edge collapsing which preserve topological similarity have so far required knowledge of the classification for all mesh entities, hence requiring a complete topological representation [19] to safely coarsen a mesh. Beyond that, it is convenient in any case to represent edges explicitly, given that many adaptive algorithms are based on edge lengths [2].



Figure 1: Two-triangle mesh with reduced and full topology graphs. On the left we see mesh edges and vertices. The middle and right figures depict mesh entities (triangles, edges, and vertices) as graph nodes, and their adjacency relations as arrows.

# 4 Flexible Graph Structures for Conforming Unstructured Meshes

Given the above discussion of requirements for finite element mesh information, we now discuss the common computer representations of unstructured finite element meshes.

Unstructured mesh applications represent of the portions of the mesh topology graph needed to support the operations carried out on the mesh. A representation which explicitly stores every m-entity is said to be a *full* representation. Any schemes which allow some m-entities to be represented implicitly (i.e. their presence does not consume memory) and be said to be *reduced* representations. We will cover full representations and show that our method can also produce reduced representations.

Once the set of explicit entities is chosen, one has options about which adjacencies to store. Recall from Section 3.1 that downward and upward adjacencies are transitive, so it is enough to store a subset of the r-edges such that reachability remains the same. One example of this is the *one-level* representation, in which we choose some ordered subset of the dimensions to explicitly represent and then store the upward and downward adjacencies between each consecutive pair of dimensions [1]. The full one-level representation, for example, stores region-face, face-edge, and edge-vertex adjacencies (both upward and downward). We will present both full and reduced one-level representations.

Figure 1 depicts two possible representations for a two-triangle mesh. The reduced representation stores triangle-to-vertex and triangle-to-triangle relations explicitly, while omitting mesh edges. Although the "dual" triangle-to-triangle graph is not an adjacency by our definition, it is used in some codes to avoid storing upward adjacencies. The full one-level representation stores triangles, edges, and vertices as well as one-level relations between them.

A comparison of representations based on the choice of dimensions and adjacencies between dimensions to represent was published by Garimella [12].

For any given representation, the computation of  $M_i^d \{M^q\}$  can either be done efficiently using stored information or using an exhaustive search if less information is available. If a structure can compute  $M_i^d \{M^q\}$ , where dimensions d and q are explicitly represented, in constant time, we say that it is *complete* [19]. Recall from Section 3.3 that upward adjacencies are bound by a constant, so they are computable in constant-time if enough information is stored.

For most applications, these structures must allow the association of user data with any m-entities.

For general adaptivity, a complete mesh representation is required to preserve mesh validity and similarity to the model during edge collapsing and swapping without imposing additional restrictions on how coarse the mesh can be.

#### 4.1 Graph Data Structure

A graph data structure which is powerful enough to accommodate any topological representation and has acceptable guarantees about the runtime cost of modifications is desired as a foundation for modifiable mesh structures.

Without restricting the types of entities and degrees of adjacency, the result would be a structure that utilizes a relatively large amount of memory. Section 4.3 will show how commonly satisfied assumptions about finite element mesh properties can yield a highly effective structure for use in mesh-based simulation workflows.

The graph structure to be used will store r-vertices and r-edges. Conceptually, r-vertices and r-edges are individual objects in memory which can be allocated and freed in constant time. The r-edges will be directed, but r-vertices will have knowedge of incoming as well as outgoing r-edges. Beyond this, the structure has the following runtime properties:

- 1. Given a pointer to the r-edge (A, B), pointers to r-vertices A and B are available in O(1) time.
- 2. Given an r-vertex pointer A, one can iterate over all outgoing r-edges (A, B) in time proportional to the number of outgoing edges.
- 3. Iteration over incoming r-edges (B, A) of the same A is also proportional to number of incoming edges.
- 4. removal of an r-edge is O(1).
- 5. removing an r-vertex A is proportional to the number of r-edges (A, B) or (B, A).

In order to satisfy Requirement 1, each r-edge object will store pointers to its two defining r-vertex objects. Satisfying Requirements 2 and 3 can be done by linking together all r-edges from the same r-vertex into a singly-linked list. The same is done for r-edges to the same r-vertex.

A singly-linked list is insufficient to satisfy Requirement 4 since one cannot remove a singly-linked list entry in constant time. Converting both lists to be doubly-linked, which involves adding "previous edge" pointers to all edges, fixes

```
Listing 1: Conceptual graph structure in C
struct rvertex {
   struct redge* first_edge_from;
   struct redge* first_edge_to;
};
struct redge {
   struct rvertex* from;
   struct redge* next_edge_from;
   struct redge* next_edge_from;
   struct redge* next_edge_to;
   struct redge* prev_edge_to;
   struct rvertex* to;
};
```



Figure 2: Example of graph structure layout

this. Finally, Requirement 5 is satisfied by virtue of assuming constant-time object deallocation and satisfying Requirement 4.

The result is that r-edges contain two endpoint pointers and two doublylinked nodes for a total of 6 pointers, while r-vertices contain just two list head pointers.

Listing 1 shows what the r-vertex and r-edge objects might look like if declared as structures in the C language. Correspondingly, Figure 2 shows an example layout of the data for a graph composed of four r-vertices and three r-edges between them.

#### 4.2 Structure Of Arrays

A structure composed of many small objects containing pointers to one another can be recast into a set of large arrays as opposed to individually allocated objects [20]. Pointers to objects are transformed into array indices. As an example, consider this transformation for a singly-linked list as shown in Figure 3.

All the list nodes are packed into an array, and a pointer to a list node



Figure 3: A singly-linked list recast into an array

becomes an index into this array, starting with zero. Since zero is a valid index, we can use -1 to denote a null pointer. Notice also that if we have knowledge about the maximum length of the array, we can choose an integer type that uses less bytes than a pointer, since pointers must be able to index the entire virtual address space.

Since indices can have the same meaning for several arrays, we can also separate the variables in an object into different arrays. For example, a particle simulation may have particle objects with variables for position, velocity, mass, and electrical charge. These can be separated into four large arrays, an approach that is often seen in scientific codes.

The new consideration in our case is how to allow constant-time addition and removal of objects from these large arrays. Our problem is made easier by allowing a new object to be placed wherever in the array is most convenient, much like the behavior of low-level memory allocators.

The problem of adding objects in constant time has a well-known solution called geometric growth. At any time we have some amount of array storage capacity c(t) which is filled entry at a time. When that storage is full, we reallocate it to a new capacity c(t + 1), such that the integer function c(t) approximates the real function  $f(t) = \alpha^t$ , where  $1 < \alpha \leq 2$ . Although reallocation has a runtime cost of O(c(t)), the geometric growth amortizes this cost such that adding objects is constant-time on average [8]. The tradeoff is that a bounded fraction of the storage is unused extra space. In our case, we use a growth formula of c(t + 1) = (3(c(t) + 1))/2 as a heuristic compromise between memory and runtime.

Removing objects is usually the more troublesome operation for array-based structures. First, we have no control over which object is being removed. This means a "hole" will be created at some arbitrary index. While some implementations opt to fill this hole immediately with an existing object, we will avoid this because it requires changing the index of a live object, which causes great confusion to users and leads to programming errors. We prefer that object in-



Figure 4: Extra space and hole tracking create modifiable arrays

dices are like object pointers: constant throughout the lifetime of the object. This way a handle/pointer/index may be maintained by the user and it will have clear meaning even during mesh modification.

If we cannot change indices then the holes must remain, and we will track them using a free list, or list of available space [14]. In our implementation, this means creating a new variable array along with those of the objects. This array which we call the free list contains pointers from each hole to the next hole, and a single head pointer outside this list points to the first hole. We can use a singly-linked list efficiently by adding and removing holes only to and from the front of this list, both of which are constant-time operations.

In summary, to add an object, we first check whether there are holes in the free list. If so, the first hole is used as space for the object, and it is removed from the free list. Otherwise the object is added at the end, which may trigger a geometric growth of all arrays simultaneously. To remove an object, we simply add the resulting hole as the first hole in the free list. See Figure 4 for a helpful layout diagram of modifiable object arrays.

One issue with this structure is that memory use does not decrease immediately when removing objects, and in theory we can only shrink the arrays if the last object is removed. This is connected to our decision to preserve identifiers, and can be fixed by temporarily relaxing that constraint. In a single collective step, all objects can be reordered, given new identifiers, and all links between them updated accordingly. If the new identifiers are contiguous, the arrays can shrink to minimal size. Beyond that, we can choose a reordering which makes subsequent queries cache-friendly. This is a supported operation in our implementation, combining hole removal with an adjacency-based reordering that improves locality.

An important benefit of this array-based structure is the ease with which additional variables can be added or removed at runtime. To add a new variable to objects of the same type, we just create a new array and ensure that subsequent resizing operations apply to that array along with the others.

#### 4.3 Mesh Structure

We can now take the graph structure developed in Section 4.1 and use it to represent a mesh topology graph as described in the beginning of Section 4.

We will take that structure and use its r-vertices to represent m-entities and r-edges to represent adjacencies. For the remainder of this section, an entity is an m-entity and a use is an m-entity use by another m-entity, i.e. an adjacency relation.

We begin by separating entities into groups based on topological type, as a prerequisite to our first optimization. Entities in the same group are all similar to some polytope, for example a triangle or a pyramid.

We first focus on representing the adjacency relations between two groups of different dimension. Once that representation is understood, it is easy to add or remove representations of adjacency between any pair of groups, and therefore between any pair of dimensions.

Our first optimization will use the fact from Section 3.3 that the number of downward adjacencies from each entity in the higher-dimension group is a known constant. This allows us to combine entity use objects and higher-dimensional entity objects together, removing any and all pointers between these objects. Such pointers account for half of the data in the graph structure: the doublylinked list from higher entities to entity uses and the pointers from uses back to higher entities.

Our second optimization is based on meshes of interest having bounded (but variable) upward adjacencies. This lets us exchange the doubly-linked list connecting uses to lower-dimensional entities with a singly-linked list. This will make the runtime cost of removing a use from such a list proportional to the number of upward adjacencies, which is bounded and thus O(1).

To recap, for each set of uses between pairs of polytope groups, we have omitted all pointers between the higher-dimensional entities and the uses. By using a singly-linked list we are left with three variables: the singly-linked list "next" pointers grouping uses of the same entity, the pointers from uses to lower-dimensional entities, and pointer from each lower-dimensional entity to its first use (a list head pointer).

As an example, consider the one-level adjacencies between triangles and their edges in a full representation. For a given triangle, one obtains the example graph from Figure 2. Figure 5 shows what the optimized structure's layout would look like for a triangle with three edges when stored in memory with contiguous objects.

Finally, our third optimization will be to re-cast the entire structure into arrays as described in Section 4.2. Our m-entities are separated into groups, or types, by similarity to a polytope. Each type is represented by its own set of contiguous arrays.

Figure 6 shows an example of the arrays involved for triangles and m-edges. The m-edge uses are represented by two arrays which have three entries for every triangle. One of them contains the indices of the edges used by each triangle, reminiscent of the simplest finite element connectivity arrays. Another stores links that connect uses of the same m-edge. Links to the first use of each m-edge are stored in an array accessed by m-edge index. A similar array exists for triangles in case there are m-regions involved that use triangles.

There is one more detail that needs resolving in the case of mixed meshes.



Figure 5: Optimization of topology graph structure for mesh



Figure 6: Optimized mesh structure recast to arrays

The problem is that different polytope groups may contain uses of the same mentity. For example, an m-edge may be used by a triangle and a quadrilateral. They must be linked together, but their arrays are separated by polytope group. In order to be able to jump between those arrays, we must enhance the indices being used. Our solution is to encode information about the polytope group into these indices, i.e. e = (t, i) where t is an integer uniquely identifying a polytope group, i is the index into the arrays of that group, and e is the extended index. In particular, the encoding we choose is e = iT + t, where T is the total number of polytope groups and  $0 \le t < T$ . This allows the extended index to remain an integer and be decoded using simple modulo and division instructions.

With the mesh entities organized by polytope group, we can add a mechanism for associating data with each entity of a polytope group by creating new arrays as described in Section 4.2. This is how vertex coordinates and entity-level fields are stored.

Recall also from Section 4.2 that reordering is sometimes needed to avoid fragmentation and improve locality. Beall and Shephard describe a reordering for mesh entities which improves the locality of subsequent adjacency queries and, more importantly, the sparsity pattern of matrices assembled from finite element meshes [1]. Adjacency-based reordering was expanded upon and tested by Zhou et al. [23]. Our implementation uses this algorithm to order each polytope group.

# 5 Analysis of Structures Implemented

We can now derive key characteristics of our structure and compare them to several structures used in production today for various applications.

#### 5.1 Representation

Although we focus on one-level representations, the system described here can individually control adjacency storage between any pair of dimensions.

Comparable flexibility can be seen in other array-based implementations such as STK and MOAB. FMDB is an object-oriented structure that stores full one-level representations [19]. It is capable of constant-time local mesh modifications and supporting adaptation code. Very similar to FMDB is the mesh databased used by Compère and Remacle in the MAdLib adaptation package [7]. STK is an array-based mesh structure being developed at Sandia National Laboratory [11]. MOAB is another array-based structure developed primarily at Argonne National Laboratory. STK allows users to specify each adjacency between a pair of entities explicitly; the results are then compressed into arrays. MOAB stores adjacency information from explicitly created non-vertex entities to vertices, and generates the inverse of this (vertices to others) on demand as required to efficiently answer adjacency queries. Adjacency between two non-vertex dimensions must be created more manually in MOAB [21]. By contrast, other implementations choose to represent elements and vertices with three adjacency relations: elements point to adjacent elements and vertices, while vertices point to one adjacent element. Celes, Paulino, and Espinha implement this scheme with additional systems for representing other entities implicitly [5]. A slightly different structure which was implemented using MOAB arrays is the Adjacent Half-Facet (AHF) structure. Their concept begins with half-facets, which we call side uses. Half-facets are identified with respect to the element using them, and arrays store mappings from each half-facet to its other half, from elements to vertices, and from vertices to one adjacent half-facet [10]. This can be viewed as the adjacencies described above enriched with orientation information between elements and from vertices to elements.

Another example of a modifiable structure is the GRUMMP system developed by Ollivier-Gooch and available at http://tetra.mech.ubc.ca/GRUMMP. This structure explicitly represents vertices, sides and elements. The adjacencies from elements to sides, sides to vertices, and vertices to sides are always stored.

#### 5.2 Memory Use

In analyzing the memory use of this structure, it is best to start by considering it outside of arrays as a pointer-based structure and count pointers per mesh entity. All m-entity memory is composed of pointers.

In a static one-level mesh, each of our mesh elements uses (2d) pointers where d is the number of entities in the next level down which are adjacent to an element. A lower-dimensional entity a uses (1+2d) pointers, where d is again the number of downward adjacencies from a, since a must keep a pointer to its first upward use. For changing meshes, each entity gets an additional free list pointer.

Consider a 2D triangle mesh with a reduced one-level representation of triangles and vertices only. Triangles, in this case, have 6 pointers and vertices have only one. Although the meaning of the pointers is quite different, this uses the exact same amount of memory as the more common triangle structure variant where each triangle stores three pointers to adjacent triangles and three pointers to vertices. Such a structure is used by the CGAL library, for example [4]. In our case, instead of pointers to adjacent triangles, we store list pointers that group triangles around one vertex. The benefits of doing this are discussed in 6.1.

To better compare to similar structures, we introduce an estimate of typical ratios between vertices, edges, and triangles in a surface mesh. The following are derived from an infinite square grid where all squares are split into two triangles:

$$n_v = n_e/3 = n_{tri}/2$$

In a mesh with these ratios, we use (2(6) + 1) = 13 total pointers per vertex with a reduced representation.

We can compare this to a linked structure developed at Carnegie Mellon University, which for each m-vertex builds a linked list of the m-vertices which are adjacent to it via an m-edge [3]. This results in 4 pointers per m-edge overall, and one pointer per m-vertex still. By the ratios above, this uses (3(4)+1) = 13 total pointers per vertex, which is again the same value. CMU's structure may use less memory in practice due to compression schemes they employ.

Finally, consider the commonly used half-edge or winged-edge data structure. We will choose a variant similar to what was presented by Weiler [22]. In this structure, there are triangles, vertices, and half-edges. Triangles and vertices each just store one pointer to an incident half-edge. A half-edge stores a pointer to at least one of its vertices, a pointer its face (half-edges are like edge uses, each is used by one face), a pointer to its other half, and a pointer to then next half-edge around its triangle. By our ratios, this structure consumes (2(1)+3(2(4))+1(1)) = 27 pointers per vertex. We can actually compare this to our *full* one-level representation, which includes edge structures. Our full representation uses (2(6) + 3(1 + 4) + 1(1)) = 28 pointers per vertex, which is very close to what the half-edge structure uses.

Moving beyond triangle meshes, we can consider tetrahedral (3D simplex) meshes. As before, we simplify matters using typical ratios between the entity types. The following ratios are derived from converting an infinite uniform cube grid using the method described by Dompierre et al. [9]. They are also consistent with ratios measured in completely unstructured tetrahedral meshes:

$$n_v = n_e/7 = n_{tri}/12 = n_{tet}/6$$

Using these ratios, a full one-level representation using our structure consumes (6(2(4)) + 12(1 + 2(3)) + 7(1 + 2(2)) + 1(1)) = 168 pointers per vertex. This closely resembles Remacle and Compère's estimate of 147 pointers per vertex, which may not account for auxiliary pointers which maintain data storage. When removing entities, the free lists increase this number 194 total pointers per vertex. Often these measurements are preferred with respect to element (tetrahedron) counts, which in our case is slightly less than 33 pointer per tetrahedra, free lists included. Our implementation uses 32-bit integers to encode pointers, which results in a memory consumption of about 130 bytes per tetrahedron.

In addition to analysis, we compare the actual memory use of this implementation against others.

Figure 7 shows that comparison; all structures have loaded the same mesh containing 100K tets. MDS is our full array representation and "reduced array" is the element-to-vertex representation, both using 32-bit indices. In addition, both MDS and FMDB are storing vertex coordinates, geometric model classification, and geometric model coordinates at mesh vertices. The MOAB structure was constructed to include element-to-vertex downward and upward adjacency. Only elements and vertices were defined in STK.

During the same test, we delete all non-topological information from the MDS array structure, e.g. coordinates and classification. The result was a memory use of 133 bytes per tetrahedron by our array-based structure, which



Figure 7: Memory usage comparison of several mesh data structures

agrees very closely with our earlier stated prediction of 130 bytes per tetrahedron.

### 5.3 Storage System

Our system for handling modifiable arrays is quite close to the work of Celes et al., whose arrays are also modifiable and use similar free list mechanisms [5].

STK and MOAB also group entities into arrays which are divided into polytope groups, and in their case further subdivided into sets. Separating the arrays into sets has some clear benefits in terms of memory use and organization. Neither STK nor MOAB are designed for individual entity modification in their compressed array form. STK has a modifiable form which is less compact. However, it should be possible to combine the majority of their design with our modifiable arrays, which is possible future work.

GRUMMP takes a hybrid approach between arrays and individual objects. Their entities are objects which refer to each other through true absolute-address pointers. These objects are stored in an array-based structure that never moves its entries. It is composed of a series of separate sub-arrays, each twice as large as the previous one. Resizing is done by allocating or deallocating a subarray at the end of the series. This method retains constant-time insertion guarantees and some of the memory-saving benefits of arrays. In exchange, it is not completely contiguous and requires some iteration to locate an item based on its index. Pointers to the adjacent sides of a single vertex are stored in a singly-linked list of fixed arrays, i.e. an array separated into individually allocated fixed-size blocks. Other adjacency pointers are stored in the element and side objects since their total size is fixed.

Storage using individual objects is a straightforward way for FMDB to allow constant-time modification, and our modifiable arrays are intended to surpass this with equal guarantees, lower memory use, and helpful locality.

Users typically interact with object-based structures via pointers to the objects, and most array-based structures for finite elements seem to be converg-

ing towards the same method of giving pointer-like identifiers to users. Our structure, the structure of Celes et al., and those of MOAB and STK all have identifiers which are small concrete types encoding at least two integers: and index into the arrays and some identifier of which arrays to index into.

# 6 Use in Adaptive Simulations

#### 6.1 Adaptivity

A key driver for modifiable mesh structures is to support the effective execution of adaptive mesh simulations, including execution on massively parallel computers. Although there are a number of alternative methods to implement mesh adaptation, the need to deal with a full range of operations to refine, coarsen and improve the mesh locally, while fully accounting for curved domains [15] favors the application of cavity based mesh modification. Thus in the examples show here, the MDS library, which implements the mesh data structure for the PUMI tools, employs the one-level adjacency representation to fully support cavity modifications as discussed in section 3.5. This allows us to drive an essentially unaltered version of the previously developed MeshAdapt adaptive procedures, which were originally based on the object oriented FMDB [16] mesh data structure.

We can compare and contrast implementations as well on the basis of adaptability. For example, while STK and MOAB can store full and complete representations, they are not modifiable in compressed form. On the other hand, while AHF and the structure of Celes et al. have modifiable arrays, they store element-to-element connectivity, which could cause difficulties in modification. As described in Section 3.5, modifications introduce a temporarily invalid state of overlapping entities in order to properly transfer solution. Such a state cannot be represented by element-to-element arrays which have a fixed number of entries per element, but they can be represented by our upward adjacency arrays, which contain linked lists that can expand to arbitrary size. For example, a triangle may have three adjacent tetrahedra. Celes et al. use modifications whose invalid state has missing entities rather than overlapping ones, and introduce extra vertex to element information to cope [6].

In addition, element-to-element connectivity means that changing a cavity affects data stored outside its boundary, namely elements near the cavity need to update their adjacent element pointers. With a one-level representation, changes only affect the entities in the boundary of the cavity. This makes parallelization of adaptivity easier.

Although we leave parallel considerations outside the scope of this paper, we have added partition information to this array structure. All the implementation details discussed above remain unchanged, and the resulting structure scalably represents meshes above 1 billion elements partitioned to above 32 thousand parts. The architecture of our partitioned representation is discussed in [13].



Figure 8: Floating object fluid-structure-interaction adaptivity

#### 6.2 Applications

In collaboration with the U.S. Army Corp of Engineers' Coastal Hydraulics Laboratory and their Proteus CFD code, we are using our flexible array structure to enable coupled mesh adaptation with mesh motion in the face of fluid-structure interaction with moving objects. Figure 8 shows a mesh of a buoyant object splashing down inside a tank of water. Proteus carries out an initial nodal repositioning and smoothing to track this motion, while adaptivity guided by error estimates ensure the discretization error and element quality remain controlled. The adaptation metric may also be anisotropic, as shown in Figure 9.

In collaboration with IBM and Sandia National Laboratories, we are using this array structure to manage very large meshes in a compact and scalable way. Figure 10 shows how the initial meshes use for this project represent multiple CAD model regions with graded resolution. The initial mesh is further refined and partitioned, and runs have exceeded one billion elements, utilizing up to 4 racks of an IBM BlueGene/Q computer. The memory use of our structure is quite small compared to the storage used for the stiffness matrix and Krylov vectors in this problem.

# 7 Conclusion

We present a modifiable and array-based structure for storing mesh topology and data associated with mesh entities. In addition, the structure is derived from an abstract topological adjacency graph given our particular requirements, making it easier for readers with different requirements to understand how to obtain a structure better suited for them. We show that we can combine a flexibility of representation (selection of explicitly represented dimensions and adjacencies between dimensions), constant-time single-entity modification, the locality of a few contiguous arrays, and the ability to handle transient invalidity.

Our implementation is available as part of the latest PUMI tools at this



Figure 9: Anisotropic mesh near fluid boundary



Figure 10: Graded multi-material mesh to initiate large scale runs

location: https://github.com/SCOREC/core. Documentation for these tools is available at this corresponding page: https://github.com/SCOREC/core/wiki.

# 8 Acknowledgements

This research was supported U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under award DE-SC00066117 (FASTMath SciDAC Institute). We gratefully acknowledge the use of the resources of the Rensselaer Center for Computational Innovations and the Leadership Computing Facility at Argonne National Laboratory.

### References

- Mark W Beall and Mark S Shephard. A general topology-based mesh data structure. International Journal for Numerical Methods in Engineering, 40(9):1573–1596, 1997.
- [2] Rupak Biswas and Roger C Strawn. Tetrahedral and hexahedral mesh adaptation for cfd problems. Applied Numerical Mathematics, 26(1):135– 151, 1998.
- [3] Daniel K Blandford, Guy E Blelloch, David E Cardoze, and Clemens Kadow. Compact representations of simplicial meshes in two and three dimensions. International journal of computational geometry & applications, 15(01):3–24, 2005.
- [4] Jean-Daniel Boissonnat, Olivier Devillers, Monique Teillaud, and Mariette Yvinec. Triangulations in cgal. In Proceedings of the sixteenth annual symposium on Computational geometry, pages 11–18. ACM, 2000.
- [5] Waldemar Celes, Glaucio H Paulino, and Rodrigo Espinha. A compact adjacency-based topological data structure for finite element mesh representation. *International Journal for Numerical Methods in Engineering*, 64(11):1529–1556, 2005.
- [6] Waldemar Celes, Glaucio H Paulino, and Rodrigo Espinha. Efficient handling of implicit entities in reduced mesh representations. *Journal of Computing and Information Science in Engineering*, 5(4):348–359, 2005.
- [7] Gaetan Compere, Jean-François Remacle, Johan Jansson, and Johan Hoffman. A mesh adaptation framework for dealing with large deforming meshes. *International journal for numerical methods in engineering*, 82(7):843–867, 2010.
- [8] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 17.4. MIT press Cambridge, 2001.

- [9] Julien Dompierre, Paul Labbé, Marie-Gabrielle Vallet, and Ricardo Camarero. How to subdivide pyramids, prisms, and hexahedra into tetrahedra. In *IMR*, pages 195–204, 1999.
- [10] Vladimir Dyedov, Navamita Ray, Daniel Einstein, Xiangmin Jiao, and Timothy J Tautges. Ahf: Array-based half-facet data structure for mixeddimensional and non-manifold meshes. In *Proceedings of the 22nd International Meshing Roundtable*, pages 445–464. Springer, 2014.
- [11] H Carter Edwards, Alan B Williams, Gregory D Sjaardema, David G Baur, and William K Cochran. Sierra toolkit computational mesh conceptual model. Sandia National Laboratories SAND Series, SAND2010-1192, 2010.
- [12] Rao V Garimella. Mesh data structure selection for mesh generation and fea applications. International journal for numerical methods in engineering, 55(4):451–478, 2002.
- [13] Daniel A Ibanez, E Seegyoung Seol, Cameron W Smith, and Mark S Shephard. Pumi: parallel unstructured mesh infrastructure. ACM Transactions on Mathematical Software, 2015.
- [14] Donald E Knuth. The Art of Computer Programming, volume 1, chapter 2.2.3. Addison-Wesley Professional, 3 edition, 1997.
- [15] Xiangrong Li, Mark S Shephard, and Mark W Beall. Accounting for curved domains in mesh adaptation. *International Journal for Numerical Methods* in Engineering, 58(2):247–276, 2003.
- [16] Xiangrong Li, Mark S Shephard, and Mark W Beall. 3d anisotropic mesh adaptation by mesh modification. *Computer methods in applied mechanics* and engineering, 194(48):4915–4950, 2005.
- [17] Kenneth Rosen. Discrete Mathematics and Its Applications 7th edition. McGraw-Hill Science, 2011.
- [18] Will J Schroeder and Mark S Shephard. A combined octree/delaunay method for fully automatic 3-d mesh generation. *International Journal* for Numerical Methods in Engineering, 29(1):37–55, 1990.
- [19] E Seegyoung Seol and Mark S Shephard. Efficient distributed mesh data structure for parallel automated adaptive analysis. *Engineering with Computers*, 22(3-4):197–213, 2006.
- [20] I-Jui Sung, Geng Daniel Liu, and Wen-Mei W Hwu. Dl: A data layout transformation system for heterogeneous computing. In *Innovative Parallel Computing (InPar), 2012*, pages 1–11. IEEE, 2012.
- [21] T. J. Tautges, R. Meyers, K. Merkley, C. Stimpson, and C. Ernst. MOAB: a mesh-oriented database. SAND2004-1592, Sandia National Laboratories, April 2004. Report.

- [22] Kevin Weiler. Edge-based data structures for solid modeling in curvedsurface environments. Computer Graphics and Applications, IEEE, 5(1):21–40, 1985.
- [23] Min Zhou, Onkar Sahni, Mark S Shephard, Christopher D Carothers, and Kenneth E Jansen. Adjacency-based data reordering algorithm for acceleration of finite element computations. *Scientific Programming*, 18(2):107–123, 2010.