Compact Array-Based Mesh Data Structures

Tyler J. Alumbaugh and Xiangmin Jiao

Center for Simulation of Advanced Rockets
Computational Science and Engineering
University of Illinois at Urbana-Champaign
{talumbau,jiao}@uiuc.edu

Summary. In this paper, we present simple and efficient array-based mesh data structures, including a compact representation of the *half-edge data structure* for surface meshes, and its generalization—a *half-face data structure*—for volume meshes. These array-based structures provide comprehensive and efficient support for querying incidence, adjacency, and boundary classification, but require substantially less memory than pointer-based mesh representations. In addition, they are easy to implement in traditional programming languages (such as in C or Fortran 90) and convenient to exchange across different software packages or different storage media. In a parallel setting, they also support partitioned meshes and hence are particularly appealing for large-scale scientific and engineering applications. We demonstrate the construction and usage of these data structures for various operations, and compare their space and time complexities with alternative structures.

Key words: mesh data structures, half-edge, half-face, parallel computing

1 Introduction

In scientific computing, mesh data structures play an important role in both numerical computations, including finite element and finite volume codes, and geometric algorithms, such as mesh adaptation and enhancement. Historically, the designs of data structures in the numerical and geometric communities have been based on quite different philosophies, due to the diverse requirements of different applications. Specifically, numerical solvers aim at space and time efficiency as well as ease of implementation in traditional programming languages such as Fortran 90, so they tend to use array-based data structures storing minimal information, as exemplified by the CFD General Notation System (CGNS), an AIAA Recommended Practice [21]. Geometric algorithms, on the other hand, require convenient traversal and modification of mesh entities, and hence tend to use comprehensive pointer-based data structures with substantial memory requirement, and frequently utilize advanced

programming features available only in modern programming languages (such as templates in C++), as exemplified by the CGAL library [9]. Increasingly, modern scientific applications require integrating geometric algorithms with numerical solvers, and this discrepancy in mesh data structures has led to difficulties in integrating different software packages and even more problems when implementing geometric algorithms within engineering codes. In a parallel setting, the necessity of accommodating partitioned meshes introduces additional complexities.

In this paper, we investigate mesh data structures that can serve both numerical and geometric computations on parallel computers. From an applications' point of view, it is desirable that such data structures meet the following requirements:

- Efficient in both time and space, so that mesh entities and their neighborhood information can be queried and modified without performing global search, while requiring a minimal amount of storage.
- Neutral of programming languages, so that it can be implemented conveniently in main-stream languages used in scientific computing, such as C, C++, Fortran 90, and even Matlab.
- Convenient for I/O, so that the data structure can be transferred between different storage (such as between files and main memory) and exchanged across different software modules.
- Easily extensible to support partitioned meshes, and easy to communicate across processors on parallel machines.

Meeting these requirements is decidedly nontrivial. Indeed, none of the preexisting data structures in the literature appeared to be satisfactory in all these aspects. In particular, the popular data structures used in numerical computations, such as the standard element-vertex connectivity for finite element codes [3], do not support efficient queries (such as whether a vertex is on the boundary) or traversals (such as from neighbor to neighbor) required by many geometric algorithms. The comprehensive pointer-based data structures used in geometric algorithms, such as edge-based data structures for surface meshes [16] and exhaustive incidence-based data structures for volume meshes [2], may require a substantial amount of memory, even after optimizing their storage to the bare minimum to store only the required pointers. These pointer-based representations are also difficult to implement in traditional programming languages, and special attention to memory management is needed (even in modern programming languages) to avoid memory fragmentation. In addition, they are inconvenient for I/O and interprocess communication.

In this work, we develop compact array-based data structures for both surface and volume meshes. Our data structures augment, and can be constructed efficiently from, the standard element-vertex connectivity. In the context of parallel computing, only the communication map of shared vertices along partition boundaries is needed as an additional requirement. Our data structures

require a minimal amount of storage, primarily composed of an encoding of the incidence relationship of d-dimensional entities along $(d-1)$-dimensional sub-entities, and a mapping from each vertex to one of its incident edges. In two dimensions, our data structure reduces to a compact representation of the well-known *half-edge data structure*. In three dimensions, it delivers a generalization of half-edges to volume meshes, which we refer to as the *half-face data structure*. With additional encoding of adjacency information of $(d-1)$-dimensional entities along partition boundaries, we then obtain a convenient representation of partitioned meshes for parallel computing.

The remainder of the paper is organized as follows. Sec. 2 presents some basic definitions, assumptions, and observations behind our data structures. Sec. 3 describes an array-based data structure for surface meshes, which resembles the well-known half-edge data structure. Sec. 4 introduces a half-face data structure and its array-based implementation. Sec. 5 extends the data structures for partitioned meshes in a parallel setting. Finally, Sec. 6 concludes the paper with a discussion of future work.

2 Preliminaries

Combinatorially, a d-dimensional *mesh* refers to a collection of topological entities of up to d dimensions, along with the *incidence relationship* of entities of different dimensions and *adjacency* of entities of the same dimension, where d is 2 for surface meshes and 3 for volume meshes. In this paper, we refer to the 0-dimensional entities *vertices,* the 1-dimensional entities *edges,* the 2-dimensional entities *faces* (or *facets*), and the 3-dimensional entities *cells*. We use *elements* as a synonym of the d-dimensional entities (i.e., faces in a surface mesh and cells in a volume mesh). We require a mesh to be *conformal*, in the sense that two adjacent cells intersect at a *shared* face, edge, or vertex, and two adjacent faces intersect at a *shared* edge or vertex.

In scientific and engineering applications, in general only a few types of elements are used in a mesh, including triangles and quadrilaterals for surface meshes and tetrahedra, prisms, pyramids, and hexahedra for volume meshes, either linear or quadratic. In this paper, we focus on linear elements. In general, the sub-entities within an element are ordered and assigned local IDs following a given convention, for consistent numerical and geometric computations (such as the calculation of the Jacobian and face normals) and for exchanging data across different software packages. In addition, the sub-entities within a sub-entity (such as the vertices within a face of a tetrahedron) are also ordered consistently. Targeted at these applications, we focus our attention on the meshes composed of the most commonly used element types, and adopt the widely used CGNS conventions [21] to number sub-entities. Fig. 1 depicts the conventions for the most common elements. In addition, we assume the vertices are assigned consecutive integer IDs ranging from one to the number of vertices, and similarly for elements. In a parallel setting, each partition is

assumed to have its own numbering systems for vertices and elements. Given an element, we assume one can determine its type from the element ID in negligible time, by comparing with the minimum and maximum IDs of each type of elements if the elements of the same type are numbered consecutively, or by performing a table lookup.

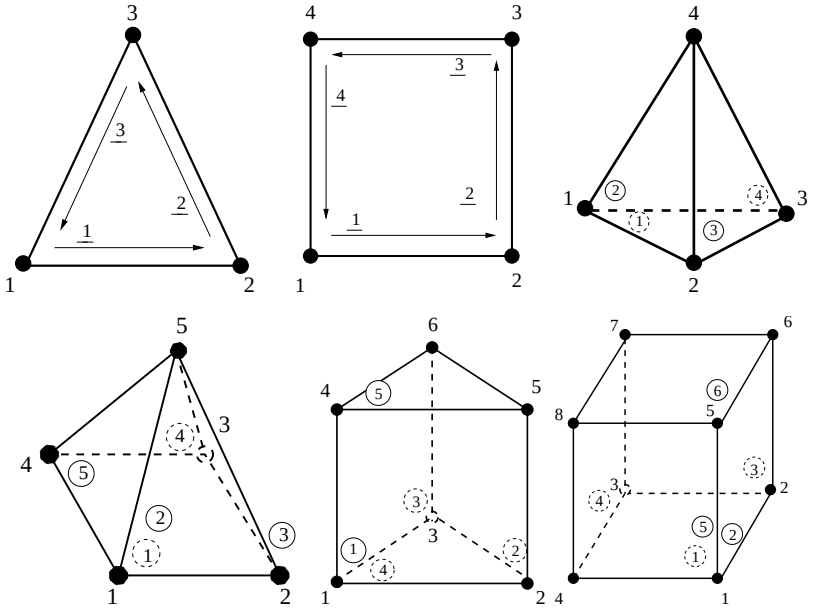


Fig. 1. Local numbering conventions for 2-D and 3-D elements. Underscored numbers correspond to local edge IDs, and circled ones correspond to local face IDs. The vertex next to an edge or face ID is the first vertex of the edge or face.

To simplify presentation, our discussions will mainly focus on *manifold* models with *boundary*. Extension to non-manifold models would involve generalizing the programming interface and tweaking the internal representation. In numerical and geometric computations, the boundary of a mesh frequently plays an important role to impose proper boundary treatments. We classify an entity to be a *border* entity if it is on the boundary, and otherwise to be a *non-border* or *interior* entity. Each border entity is said to incident on a *hole*. In our applications, a surface mesh is typically composed of the border entities of a volume mesh, and hence in general is *orientable* with consistent inward and outward surface normals. A manifold surface mesh with boundary has the following useful properties:

- Each *edge* is contained in either *one* or *two faces*.
- There is a *cyclic* sequence of the incident *edges* of each *non-border vertex*.

- There is a *linear* sequence of the incident *edges* of each *border vertex*.

Here, an ordered set of entities is said to be a *sequence* if each pair of consecutive entities are contained in a higher-dimensional entity. Analogously, a manifold volume mesh with boundary has the following properties:

- Each *face* is contained in either one or two *cells*.
- There is a *cyclic* sequence of the incident *border edges* of each *border vertex*.
- There is a *cyclic* sequence of the incident *faces* of each *non-border edge*.
- There is a *linear* sequence of the incident *faces* of each *border edge*.

To manipulate surface and volume meshes, scientific and engineering applications require efficient *mesh data structures* (abbreviated as *MDS* hereafter). An MDS allows iterating through the entities of a mesh, performing queries on incidence, adjacency, and classification (in particular, boundary classification) of entities, and modifying the mesh efficiently. We classify incidence relationships to be either *upward* or *downward*, which map an entity to other higher- or lower-dimensional entities, correspondingly. Furthermore, an entity in general is incident on one or more entities of a given dimension, so we further subdivide the incidence queries into *one-to-any* and *one-to-all*. We assume the *valence* of the mesh (i.e., the maximum number of edges incident on a vertex) is bounded by a small constant. We say an MDS is *comprehensive* if it can perform every incident, adjacent, or classification query in a time independent of mesh size. A data structure is *complete* (or *self-contained*) if it contains all the information necessary to construct a comprehensive MDS. In general, we require an MDS to be complete. Obviously, a comprehensive MDS is complete, but not vice versa. The goal of this paper is to develop complete and comprehensive data structures that require minimal storage.

3 Surface Mesh Data Structures

In this section, we investigate data structures for surface meshes in sequential applications. Issues related to parallelization will be discussed in Sec. 5.

3.1 Traditional Representations

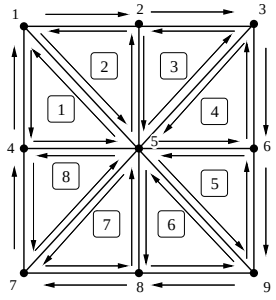
In the literature, two classes of representations of surface meshes have been commonly used: edge-based representations, and connectivity tables.

Edge-Based Representations

Edge-based representations have been studied and used extensively in computational geometry, for their generality and comprehensiveness. Three major

variants have been proposed: the winged-edge [1, 12], half-edge [22] (or the doubly-connected edge list or DCEL [8]), and quad-edge [13] data structures; see [16] for a comparison and discussion of implementation issues. These data structures allow efficient local traversal and modification of mesh entities, and are designed to handle arbitrary polygons. In practice, their implementations are typically pointer-based, and memory optimization has focused on omitting certain pointers to trade time for space, as in the Computational Geometry Algorithms Library (CGAL) [9, 16].

Among the edge-based structures, the half-edge data structure (abbreviated as HEDS hereafter) has been very popular for orientable manifold surfaces for its intuitiveness and ease-of-use. The HEDS is designed based on the observation that each face is bounded by a loop of directed edges in counterclockwise order, and each hole is bounded by a loop in clockwise order, as illustrated in Fig. 2. Therefore, every edge has two directed half-edges (with opposite directions), which are said to be the *twin* or the *opposite* of each other, one in each incident face (or hole). The programming interface of the HEDS allows a user to query the previous and next half-edges within a face (or hole), the opposite half-edge, an incident vertex or face of a half-edge, and vice versa. By flagging border half-edges or their incident holes, the HEDS also allows querying the boundary classification of any entity.



Element	V1	V2	V3
1	1	4	5
2	1	5	2
3	2	5	3
4	3	5	6
5	6	5	9
6	8	9	5
7	7	8	5
8	5	4	7

Fig. 2. Illustration of half-edges. **Fig. 3.** Connectivity of sample surface mesh.

In a pointer-based implementation of HEDS, an object (or record) is created for each vertex, half-edge, or face. A half-edge object stores pointers to its opposite, previous, and next half-edges, as well as to its origin (or destination) vertex and to an incident face. Each face or vertex stores a pointer to an incident half-edge. For applications involving computations on both faces and vertices, one can slightly reduce the storage by omitting either the previous or next half-edge. Therefore, such an implementation requires at least eight pointers per edge (four per half-edge), and one pointer per vertex or face. The winged-edge and quad-edge structures have comparable storage requirements (six pointers per edge) in this setting.

Element Connectivity

The element connectivity is a classic mesh representation for finite element analysis [3], and is also frequently used for file I/O and for exchanging meshes between different software modules. A connectivity table lists the vertices contained within each face in increasing order of their local IDs. Fig. 3 shows the connectivity table of the sample mesh of Fig. 2. This simple representation is self-contained but not comprehensive, as it does not support queries such as adjacent faces and boundary classification. A geometric software library typically constructs an internal edge-based structure from the connectivity table, and then discards this table.

3.2 Array-Based Half-Edges

Inspired by the HEDS and element connectivity, we design a hybrid data structure that combines the comprehensiveness of the former and the compactness of the latter. Our design is based on the following observations: The traditional HEDS must store many pointers in order to support arbitrary polygons. For surface meshes composed of only triangles and quadrilaterals, we can encode a half-edge based on its location within its incident face. The incident face and the previous or next half-edge can then be obtained with simple arithmetic operations without being stored explicitly. From a local edge ID and the element connectivity table, one can obtain the IDs of the incident vertices of an edge efficiently. Therefore, we only need to store the correspondence between twin half-edges and a mapping from each vertex to any of its incident half-edges.

More specifically, we assign each half-edge an ID composed of a pair of numbers $\langle f, i \rangle$, where f is the ID of its containing face, and i is the index of the edge (starting from 1, as shown in Fig. 1) within the face. To support boundary classification, we assign consecutive integer IDs to border edges (starting from 1), and encode the bth border half-edge as $\langle b, 0 \rangle$, where the zero value of the second part distinguishes a border edge from a non-border one. Since the number of edges per face is at most four, we can encode a half-edge ID in a single integer, using the last three bits to store the second part and the remaining bits the first part. We now define the arrays for the HEDS:

- V2e: Map each vertex to the ID of an incident half-edge originated from the vertex; map a border vertex to a border half-edge.
- E2e: Map each non-border half-edge to the ID of its twin half-edge.
- B2e: Map each border half-edge to the ID of its twin non-border half-edge.

Fig. 4 shows an example of this MDS for the sample mesh in Fig. 2. In terms of memory management, V2e and B2e are dense one-dimensional arrays, whose sizes are equal to the numbers of vertices and border edges, respectively. To allow efficient array indexing, E2e is a two-dimensional array with each

row corresponding to a face, and its number of columns is the maximum number of edges per face (3 for triangular meshes and 4 for quadrilateral or mixed meshes). Note that the full array-based HEDS is composed of these three arrays along with the element connectivity (EC), where EC is a two-dimensional array similar to E2e. Let n_i denote the number of i-dimensional entities in a mesh, where i is between 0 and 2. Assume that $n_1 \approx 3n_0$ and $n_2 \approx 2n_0$, then the array-based HEDS requires about $7n_0$ 32-bit integers. Compared to pointer-based HEDS, it reduces memory requirement by about four folds for 32-bit architecture and eight folds for 64-bit architecture.

V2e

Vertex	Inc. HE
1	$\langle 2,0\rangle$
2	$\langle 3,0\rangle$
3	$\langle 4,0\rangle$
4	$\langle 1,0\rangle$
5	$\langle 1,3\rangle$
6	$\langle 5,0\rangle$
7	$\langle 8,0\rangle$
8	$\langle 7,0\rangle$
9	$\langle 6,0\rangle$

E2e

Element	Half-edges		
1	$\langle 1,0\rangle$	$\langle 8,1\rangle$	$\langle 2,1\rangle$
2	$\langle 1,3\rangle$	$\langle 3,1\rangle$	$\langle 2,0\rangle$
3	$\langle 2,2\rangle$	$\langle 4,1\rangle$	$\langle 3,0\rangle$
4	$\langle 3,2\rangle$	$\langle 5,1\rangle$	$\langle 4,0\rangle$
5	$\langle 4,2\rangle$	$\langle 6,2\rangle$	$\langle 5,0\rangle$
6	$\langle 6,0\rangle$	$\langle 5,2\rangle$	$\langle 7,2\rangle$
7	$\langle 7,0\rangle$	$\langle 6,3\rangle$	$\langle 8,3\rangle$
8	$\langle 1,2\rangle$	$\langle 8,0\rangle$	$\langle 7,3\rangle$

B2e

Border	Opp. HE
1	$\langle 1,1\rangle$
2	$\langle 2,3\rangle$
3	$\langle 3,3\rangle$
4	$\langle 4,3\rangle$
5	$\langle 5,3\rangle$
6	$\langle 6,1\rangle$
7	$\langle 7,1\rangle$
8	$\langle 8,2\rangle$

Fig. 4. Array-based half-edge data structure of sample surface mesh.

Given the element connectivity, it is straightforward to construct the three arrays. In particular, we first construct E2e by inserting the half-edges into a hash-table (or map) with their incident vertices as keys and detecting collisions to match twin half-edges. A half-edge without a match is identified as a border edge and assigned a unique border ID, and its and its twin's half-edge IDs are then inserted into the corresponding entries in B2e and E2e, respectively. During the above procedure, we fill in the entry of V2e corresponding to the origin of each half-edge, allowing border half-edge to overwrite non-border ones but not vice versa.

3.3 Properties and Operations

The array-based HEDS has the following useful properties:

1. The full array-based HEDS delivers a *comprehensive* MDS.
2. E2e and V2e deliver a *complete* MDS.

To show the comprehensiveness of the full data structure, we summarize the basic queries as follows. Note that all array indices start from 1, and m denotes the number of edges of a given face.

- One-to-any downward incidence
  - ith edge (half-edge) of face f: return $\langle f, i\rangle$
  - ith vertex of face f: return $\mathrm{EC}(f, i)$
  - origin of non-border half-edge $\langle f, i\rangle$: return $\mathrm{EC}(f, i)$
- One-to-any upward incidence
  - the incident face of a non-border half-edge $\langle f, i\rangle$: return f
  - an incident half-edge of vth vertex: return $\mathrm{V2e}(v)$
- Adjacency
  - opposite of non-border half-edge $\langle f, i\rangle$: return $\mathrm{E2e}(f, i)$
  - opposite of border half-edge $\langle b, 0\rangle$: return $\mathrm{B2e}(b)$
  - previous of non-border half-edge $\langle f, i\rangle$: return $\langle f, \mathrm{mod}(l + m - 2, m) + 1\rangle$
  - next of non-border half-edge $\langle f, i\rangle$: return $\langle f, \mathrm{mod}(l, m) + 1\rangle$
- Boundary classification
  - half-edge $\langle f, i\rangle$: return $i = 0$
  - vertex v: return $\mathrm{V2e}(v)$.second$= 0$

Other types of queries are combinations of the above basic operations. In particular for downward incidence, the destination of a non-border half-edge is the origin of its previous half-edge; the origin and destination of a border half-edge are the destination and origin of its opposite half-edge, respectively; the one-to-all incidences from a face to edges and vertices involves enumerating i from 1 to m. For one-to-all upward incidence, the incident faces of an edge are those incident on its twin half-edges; the incident half-edges and faces at a vertex involve accessing all the half-edges incident on a vertex, enabled by the basic adjacency operations. Starting from a half-edge, we can rotate around its destination vertex in clockwise order following the links of opposite and then previous half-edges; the counterclockwise rotation around the origin vertex of a half-edge follow the links of opposite and then next half-edges. We also use these rotations to get the previous and next of a border half-edge: For the former, we loop around the origin of the border half-edge in counterclockwise order until reaching a border half-edge; for the latter, we loop around its destination clockwise. Since the valence of the mesh is a small constant, these rotations take constant time.

To show E2e and V2e are complete, we simply need to construct a procedure to compute the element connectivity (EC) from these arrays, because EC is a complete MDS itself. At a high level, the procedure goes as follows: First, allocate and fill in the element connectivity with zeros. Then, loop through all vertices, and for each vertex v, visit all the half-edges originated from v as shown above, starting from the half-edge $\mathrm{V2e}(v)$ and then rotating using the adjacency information in E2e. When visiting a half-edge $h = \langle f, i\rangle$ with $i > 0$, we assign $\mathrm{EC}(f, i)$ to v. After processing all the vertices, we then have the complete EC. We comment that E2e is independent of vertex numbering, but it can also be considered as complete if the vertices are allowed to be renumbered, since we can determine a vertex numbering from E2e using a

procedure similar to the above. These complete sub-MDS are useful to reduce the amount of data for I/O and inter-process communications.

3.4 Mesh Modification

A comprehensive mesh data structure allows not only querying but also modifying a mesh efficiently. In this subsection, we demonstrate how to modify the array-based HEDS, using *edge flipping* for triangular meshes as an example. This operation is used in many algorithms, such as Delaunay triangulation and mesh enhancement [10]. Fig. 5 illustrates a sample edge-flipping operation, which removes an edge composed of vertices $\{v_1, v_3\}$ and creates a new edge composed of $\{v_2, v_4\}$. As a consequence, the faces $\{v_1, v_2, v_3\}$ and $\{v_1, v_3, v_4\}$ (denoted by f and g) are replaced by $\{v_1, v_2, v_4\}$ and $\{v_2, v_3, v_4\}$, respectively.

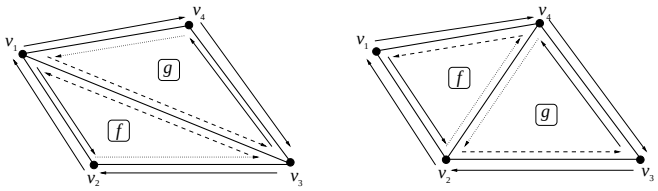


Fig. 5. Illustration of edge flipping.

Without loss of generality, suppose the fth row of EC is $\{v_3, v_1, v_2\}$ and gth row is $\{v_1, v_3, v_4\}$, so the first half-edges in f and g are the dashed lines in Fig. 5. When flipping the edge $\{v_1, v_3\}$, we need only update the entries associated with v_i, f, and g in E2e, B2e, V2e, and EC. Updating V2e and EC is relatively straightforward. For E2e and B2e, we need to map the half-edges opposite to those in faces f and g to the new half-edges, update mappings between the half-edges within f and g in E2e. In summary, an edge flip involves the following four steps:

1. **if** $\langle i, j\rangle \equiv$ E2e$(f, 1)$ is border **then** B2e(i) = E2e$(g, 3)$; **else** E2e(i, j) = E2e$(g, 3)$; perform the operation symmetrically by switching f and g;
2. E2e$(f, 1)$ = E2e$(g, 3)$; E2e$(g, 1)$ = E2e$(f, 3)$;
   E2e$(f, 3) = \langle g, 3\rangle$; E2e$(g, 3) = \langle f, 3\rangle$;
3. **if** V2e$(v_1) = \langle g, 1\rangle$ **then** V2e$(v_1) = \langle f, 2\rangle$;
   if V2e$(v_2) = \langle f, 3\rangle$ **then** V2e$(v_2) = \langle g, 1\rangle$;
   if V2e$(v_3) = \langle f, 1\rangle$ **then** V2e$(v_3) = \langle g, 2\rangle$;
   if V2e$(v_4) = \langle g, 3\rangle$ **then** V2e$(v_4) = \langle f, 1\rangle$;
4. EC$(f, 1) = v_4$; EC$(g, 1) = v_2$.

In the above, if v_3 was the ith (instead of the first) vertex of face f in EC, then we need to replace half-edge index $\langle f, j\rangle$ (and its corresponding array indices)

by $\langle f, \mathrm{mod}(j+i+2,3)+1\rangle$; similarly for g. Other modification operations, such as edge splitting and edge contraction, are slightly more complex but can be constructed similarly. For edge splitting, since the numbers of vertices and faces are increased by the operation, it is desirable to reserve additional memory for the arrays so that new vertices and faces can be appended to the end. For edge contraction, since the numbers of vertices and faces decrease, we need to swap the IDs of the to-be-removed vertex with the one with the largest ID (and similarly for faces), so that the vertex and element IDs will remain consecutive and the arrays can be shrunk after contraction.

4 Volume Mesh Data Structures

We now extend the array-based half-edge data structure to develop compact representations for volume meshes in serial applications.

4.1 Previous Work

As for surface meshes, the element connectivity is the classic representation for volume meshes in numerical computations, file I/O, and data exchange, but it is not a comprehensive representation. A few alternative mesh representations have been proposed to serve various special purposes, such as mesh generation [7, 17], mesh refinement [5, 20], and numerical computations [14]. In [4], a difference coding was proposed to compress a mesh representation, but vertices may need to be renumbered for its effectiveness. Recently, an index-based mesh representation was developed independently of this work [6], which shares some similarities in mesh-entity representations with our data structures. Another particularly noteworthy MDS is the algorithm oriented mesh database (AOMD) [19], which provides unified data structures for numerical and geometric computations.

AOMD is designed based on the observation that a typical application uses only a subset of incidences.[1] It stores the incidences used by an application and omits the unneeded ones. Its design aims at providing a unified programming interface for accessing the mesh database independently of the underlying storage. Unfortunately, the efficiency of AOMD critically depends on a given application. If an application requires nearly all types of incidences, then the efficiency advantage of AOMD diminishes. The implementation of AOMD is fairly complex, extensively utilizing template features of C++ to achieve customizability, and hence its design cannot be easily adopted in engineering applications written in traditional programming language (such as Fortran 90). As coupled applications become more and more commonplace, there are increasing demands for simple volume meshes data structures that are compact and comprehensive.

[1] In AOMD [19], the incidence relation are called adjacency.

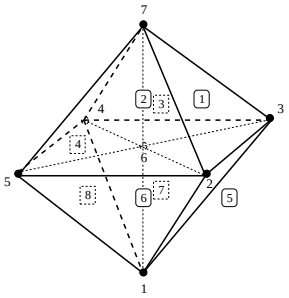


Fig. 6. Sample volume mesh.

Element	V1	V2	V3	V4
1	2	3	6	7
2	2	6	5	7
3	3	4	6	7
4	4	5	6	7
5	2	6	3	1
6	5	6	2	1
7	6	4	3	1
8	6	5	4	1

Fig. 7. Connectivity of sample volume mesh.

4.2 Array-Based Half-Faces

To achieve compactness and comprehensiveness for volume meshes, we propose a generalization of the array-based HEDS, called the *half-face data structure* (HFDS). Our generalization is based on the observation that faces in volume meshes play a similar role as edges in surface meshes: Each face is contained in two cells (or a cell and a hole), and the two copies of a face have opposite orientations when its vertices are ordered following the right-hand rule in each cell (i.e., in counterclockwise order with respect to the inward face normal of the cell). We refer to the two copies of a face as *half-faces*, which are said to be the *twin* or the *opposite* of each other. As for half-edges in surface meshes, we encode each half-face by a pair of numbers $\langle c, i\rangle$, where c is the element ID of its containing cell, and i is the face index (starting from 1) within the cell. Furthermore, we assign consecutive IDs to border faces (starting from 1), and encode a half-face with border ID b as $\langle b, 0\rangle$. Since there are at most six faces per cell, we can encode a half-face ID in one integer, using the last three bits for the second part and the remaining bits for the first part.

The above encoding scheme suggests a straightforward generalization of HEDS with three arrays: V2f, F2f, and B2f, which are the counterparts of V2e, E2e, and B2e, respectively, and in which the half-edge IDs are substituted by half-face IDs. Indeed, this generalization does provide a legitimate data structure. However, it is not an ideal generalization, because unlike E2e and V2e in HEDS, F2f and V2f no longer deliver a complete MDS. The incompleteness is due to the fact that the ordering of vertices in a half-face is cyclic without a designated starting point, so it is not always possible to infer the ordering of the vertices within a cell from this sub-MDS. When supplemented by the element connectivity, this simple generalization can suffer from inefficiency, as it requires comparing the vertex IDs to align the half-faces when performing one-to-all incidence queries.

To overcome these limitations, we define the *anchor* of a half-face as its designated first vertex, and each half-face then has m anchored copies, where m is the number of vertices (or edges) of the face. We encode an *anchored*

half-face (AHF) by a three-part ID $\langle c, i, j \rangle$, where the first two parts correspond to the half-face ID, and the third part corresponds to the *anchor index* (staring from 0), which is defined as follows: For a non-border half-face, if the anchor is the kth vertex in the face-vertex list of the face following the CGNS convention, then the anchor index is $k-1$; for a border half-face, the anchor index is $\text{mod}(m-t, m)$, where t is the anchor index of the vertex in its opposite half-face. Each AHF has one opposite (or twin) AHF, which is its opposite half-face with the same anchor. The last two parts of an AHF ID constitute the *local AHF ID*. The anchor index requires only two bits, and the local AHF ID requires only five bits, so the full AHF ID can be encoded in one integer. With 32-bit unsigned integers, this encoding is sufficient for meshes containing up to 2^{27} (more than 100 million) cells. In addition, we assign the AHF ID to the first edge of the AHF, and then obtain an ID for each edge within each face.

In the CGNS convention, each vertex has a local index within a cell. It is useful to store the mapping from the local AHF ID to the vertex's local index within the cell. We assign a unique ID (between 1 and the number of polyhedron types, 4 in general) to each type of element. To store the mapping for all element types, we introduce a three-dimensional array of size $4 \times 6 \times 4$, denoted by eA2v, whose dimensions correspond to the type IDs, local face IDs, and anchor indices, respectively. In addition, we define an array eAdj of the same size to store the mapping from each AHF to the local AHF ID of its adjacent AHF within a cell along its first edge. We now define the complete representation of the half-face data structure:

- V2f: Map each vertex to the AHF anchored at the vertex; map a border vertex to a border AHF;
- F2f: Map each non-border half-face with anchor index 0 to its twin AHF;
- B2f: Map each border half-face with anchor index 0 to its twin AHF.

Fig. 8 shows an example of this MDS for the sample mesh of Fig. 6. Similar to the array-based HEDS, V2f and B2f are dense one-dimensional arrays, whose sizes are equal to the numbers of vertices and border faces, respectively. F2f is a two-dimensional array with each row corresponding to a cell, and its number of columns is the maximum number of faces in a cell, ranging between four and six. The full HFDS is composed of these three arrays along with the element connectivity (EC). The construction of HFDS follows a procedure similar to that of HEDS, except for the additional operations needed to align the twin half-faces to determine the anchor indices.

4.3 Properties and Operations

Similar to HEDS, the array-based HFDS has the following useful properties:

1. The full array-based HFDS delivers a *comprehensive* MDS.
2. F2f and V2f deliver a *complete* MDS.

V2f

Vertex	Inc. HF
1	$\langle 7,0,1\rangle$
2	$\langle 2,0,2\rangle$
3	$\langle 3,0,0\rangle$
4	$\langle 4,0,0\rangle$
5	$\langle 6,0,2\rangle$
6	$\langle 1,1,1\rangle$
7	$\langle 1,0,1\rangle$

F2f

Element	Half-faces			
1	$\langle 5,1,0\rangle$	$\langle 1,0,0\rangle$	$\langle 3,4,1\rangle$	$\langle 2,2,1\rangle$
2	$\langle 6,1,1\rangle$	$\langle 1,4,1\rangle$	$\langle 4,3,1\rangle$	$\langle 2,0,0\rangle$
3	$\langle 7,1,1\rangle$	$\langle 3,0,0\rangle$	$\langle 4,4,1\rangle$	$\langle 1,3,1\rangle$
4	$\langle 8,1,1\rangle$	$\langle 4,0,0\rangle$	$\langle 2,3,1\rangle$	$\langle 3,3,1\rangle$
5	$\langle 1,1,1\rangle$	$\langle 6,3,1\rangle$	$\langle 7,4,1\rangle$	$\langle 5,0,0\rangle$
6	$\langle 2,1,1\rangle$	$\langle 8,2,1\rangle$	$\langle 5,2,1\rangle$	$\langle 6,0,0\rangle$
7	$\langle 3,1,1\rangle$	$\langle 8,4,1\rangle$	$\langle 7,0,0\rangle$	$\langle 5,3,1\rangle$
8	$\langle 4,1,1\rangle$	$\langle 6,2,1\rangle$	$\langle 8,0,0\rangle$	$\langle 7,2,1\rangle$

B2f

Border	Opp. HF
1	$\langle 1,2,0\rangle$
2	$\langle 2,4,0\rangle$
3	$\langle 3,2,0\rangle$
4	$\langle 4,2,0\rangle$
5	$\langle 5,4,0\rangle$
6	$\langle 6,4,0\rangle$
7	$\langle 7,3,0\rangle$
8	$\langle 8,3,0\rangle$

Fig. 8. Array-based half-face data structure for sample volume mesh.

To show the HFDS is comprehensive, we summarize the basic queries as follows. Again, m denotes the number of edges in a given AHF and is either 3 or 4, and all indices (except for anchor index) start from 1.

- One-to-any downward incidence
  - ith face of cell c anchored at jth vertex in the face: return $\langle c,i,j-1\rangle$
  - jth edge of AHF $\langle c,i\rangle$: return $\langle c,i,j-1\rangle$
  - local index of anchor of non-border AHF $\langle c,i,j\rangle$ within cell c: return eA2v$(e,i,j+1)$, where e is type ID of cell c
  - ith vertex of cell c: return EC(c,i)
- One-to-any upward incidence
  - incident cell of AHF (or edge) $\langle c,i,j\rangle$: return c
  - incident AHF (or edge) of vth vertex: return V2f(v)
- Adjacency
  - opposite of non-border AHF $\langle c,i,j\rangle$: return $\langle d,s,\mathrm{mod}(m-j+t,m)\rangle$, where $\langle d,s,t\rangle \equiv$ F2f(c,i)
  - opposite of border AHF $\langle b,0,j\rangle$: same as above, except that $\langle d,s,t\rangle \equiv$ B2f(b,i)
  - previous of AHF $\langle c,i,j\rangle$ within the face: return $\langle c,i,\mathrm{mod}(j+m-1,m)\rangle$
  - next of AHF $\langle c,i,j\rangle$ within the face: return $\langle c,i,\mathrm{mod}(j+1,m)\rangle$
  - in-cell adjacent AHF of non-border AHF $\langle c,i,j\rangle$ along edge: return $\langle c,\mathrm{eAdj}(e,i,j+1)\rangle$, where e is type ID of cell c
- Boundary classification
  - AHF (edge) $\langle c,i,j\rangle$: return $i=0$
  - vertex v: return V2f(v).second $=0$

Other types of queries are combinations of the above basic operations. Some queries are straightforward generalization of HEDS, including downward incidences (for border half-faces and one-to-all) and all incident cells of a face. Obtaining incident faces and cells along an edge involves traversing all the

half-faces incident on the edge, using the opposite and in-cell adjacent operators, so does determining the border half-face that is adjacent to a border half-face along an edge. The time complexity for traversing the boundary depends on the number of cells incident on a border edge. For applications that frequently traverse the boundary, a separate array B2b can be constructed to save the correspondence of border AHFs, similar to the E2e array in the HEDS, except that border AHF IDs will be stored instead of half-edge IDs.

A more complex query is to enumerate all incident AHFs anchored at a vertex, which is a useful building block for enumerating all incident cells and edges of a vertex. Using the in-cell adjacency and previous (or next) operators, we can iterate through the AHFs around an anchor within a cell. Together with the opposite operator, we can then visit the adjacent cells and their AHFs anchored at the vertex. This process essentially performs a breadth-first traversal over the AHFs around the vertex, and takes time proportional to the output size.

The argument for the completeness of F2f and V2f is similar to that of HEDS: the element connectivity (EC) can be constructed from F2f and V2f by looping through the AHFs around each vertex, starting from the AHF associated with the vertex in V2f. This complete sub-MDS can be used for efficient file I/O and interprocess communication, because EC and B2f can be constructed from them without requiring any additional storage.

4.4 Comparison

We now compare the storage requirements of the HFDS with some other MDS. In [2], three comprehensive MDS were reported: the "one-level adjacency representation", in which each i-dimensional entity stores points to its incident $(i+1)$- and $(i-1)$-dimensional entities when applicable, the "circular adjacency representation", in which $(i-1)$-dimensional incidence entities are stored for cells, faces, and edges, along with the incidence cells of each vertex, and the "reduced-interior representation", which omit the interior faces and edges in the representation. The storage requirements of these data structures are competitive with other existing mesh data structures [2, 18].

Let n_i denote the number of i-dimensional entities in a mesh, where i is between 0 and 3. Assume that $23n_0 \approx 4n_3$ for tetrahedral meshes and $n_0 \approx n_3$ for hexahedral meshes [2]. For a tetrahedral mesh, the HFDS data structure requires about $24n_0$ integers, of which $23n_0$ are for F2f, in addition to the $23n_0$ integers for EC. For a hexahedral mesh, the HFDS data structure requires about $7n_0$ integers, of which $6n_0$ are for F2f, in addition to the $8n_0$ integers for EC. Table 1 compares the memory requirements of the HFDS (excluding EC) against those reported in [2]. For meshes with fewer than 100 million cells, an integer in HFDS requires 4 bytes, whereas a pointer in other data structures requires 4- and 8-bytes on 32- and 64-bit architectures, respectively. Compared to the reduced-interior representation, the HFDS delivers roughly three to four folds of reduction in memory on 32-bit architectures, and six

Table 1. Memory requirements of mesh data structures. The units I and P stand for the numbers of integers and pointers, respectively.

Mesh type	One-level	Circular	Reduced interior	HFDS
Tetrahedral mesh	$201n_0P$	$153n_0P$	$76n_0P$	$24n_0I$
Hexahedral mesh	$71n_0P$	$55n_0P$	$31n_0P$	$7n_0I$

to eight folds on 64-bit architectures. Compared to the one-level and circular representations, the reduction is roughly an order of magnitude.

5 Parallelization

In a parallel environment (especially on distributed memory machines), a mesh must be partitioned so that it can be distributed onto multiple processors [11]. In this context, a border entity on the partition may or may not be on the physical boundary, and it is important for applications to distinguish the two types. Furthermore, when a process has multiple partitions, it is desirable to allow an algorithm to traverse across partition boundaries transparently. Our data structures can be extended conveniently to support such queries and traversals. We now describe the extension for meshes partitioned using an element-oriented scheme, which assigns each element to one partition along with its vertices.

Assume that each partition has a unique partition ID, and an efficient mapping exists for querying the owner process of a given partition. A partition typically has its own numbering systems for vertices and elements, from 1 to the numbers of vertices and elements, respectively. For each partition of a surface mesh, we construct an array-based HEDS. Note that a partition may not strictly be a manifold, as a vertex may be incident on more than two border edges. However, the HEDS is still applicable because an edge is always owned by one or two partitions. We define the *counterpart* of a border half-edge (on partition boundaries) to be the non-border half-edge in another partition. Given a mapping between the vertices shared across partitions, we then construct the following arrays to augment the HEDS:

- B2rp: Map each border edge to the partition ID of its counterpart, or map to -1 if the edge is on physical boundary.
- B2re: Map each border edge to the edge ID of its counterpart; undefined if on the physical boundary.

By checking the values in B2rp, we can identify whether entities are on the physical boundary. In addition, when determining the opposite of a half-edge within a partition, if its opposite is on the partition boundary, this extended data structure looks up the counterpart of its opposite border edge and returns the partition ID and the half-edge ID, so that multiple partitions can be

traversed seamlessly. From this data structure, one can also easily construct the communication pattern for shared edges across partitions. All the arrays in the extended HEDS are independent of the process mapping of the partitions, so a partition can be migrated easily across processes.

The generalization from surface meshes to volume meshes is straightforward. The *counterpart* of a border AHF is the non-border AHF with the same anchor. We introduce a similar set of arrays to map border faces to the partition and AHF IDs of their counterparts. The construction of these arrays also requires only the vertex mapping for vertices along partition boundaries.

6 Conclusion

In this paper, we introduced a compact array-based representation for the half-edge data structure for surface meshes, and a novel generalization to volume meshes. Our data structures augment the element connectivity by introducing three additional arrays. These data structures require minimal additional storage, provide comprehensive and efficient support for queries of adjacency, incidence, and boundary classification, and can be used to modify a mesh with operations such as edge flipping. Our data structures reduce memory requirement by a factor of between three and eight compared to other comprehensive data structures. A more compact subset of our data structures is also self-contained and can be used for efficient I/O and interprocess communication.

This work so far has mainly focused on two- and three-dimensional conformal manifold meshes, driven by the needs in the coupled parallel simulations at the Center for Simulation of Advanced Rockets [15]. These array-based data structures are readily extensible to higher dimensions, and it is also interesting to generalize them to non-manifold and/or non-conforming meshes. Another future direction is to compare the runtime performance of our data structures with other alternative structures, especially in the context of parallel mesh adaptivity.

Acknowledgements

This work was supported by the U.S. Department of Energy through the University of California under subcontract B523819, and in part by NSF and DARPA under CARGO grant #0310446. The first author would like to thank Phillip Alexander of CSAR for help with numerous software issues. We thank anonymous referees for their helpful comments.

References

1. B. G. Baumgart. A polyhedron representation for computer vision. In *National Computer Conference*, pages 589–596, 1975.
2. M. W. Beall and M. S. Shephard. A general topology-based mesh data structure. *Int. J. Numer. Meth. Engrg.*, 40:1573–1596, 1997.
3. E. B. Becker, G. F. Carey, and J. T. Oden. *Finite Elements: An Introduction*, volume 1. Prentice-Hall, 1981.
4. D. K. Blandford, G. E. Blelloch, D. E. Cardoze, and C. Kadow. Compact representations of simplicial meshes in two and three dimensions. In *Proceedings of 12th International Meshing Roundtable*, pages 135–146, 2003.
5. G. F. Carey, M. Sharma, and K. Wang. A class of data structures for 2-d and 3-d adaptive mesh refinement. *Int. J. Numer. Meth. Engrg.*, 26:2607–2622, 1988.
6. W. Celes, G. Paulino, and R. Espinha. A compact adjacency-based topological data structure for finite element mesh representation. *Int. J. Numer. Meth. Engrg.*, 2005. in press.
7. H. Dannelongue and P. Tanguy. Efficient data structure for adaptive remeshing with fem. *J. Comput. Phys.*, 91:94–109, 1990.
8. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 2nd edition, 2000.
9. A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, a computational geometry algorithms library. *Softw. – Pract. Exp.*, 30:1167–1202, 2000. Special Issue on Discrete Algorithm Engineering.
10. P. J. Frey and P.-L. George. *Mesh Generation: Application to Finite Elements*. Hermes, 2002.
11. J. R. Gilbert, G. L. Miller, and S.-H. Teng. Geometric mesh partitioning: Implementation and experiments. *SIAM J. Sci. Comp.*, 19:2091–2110, 1998.
12. A. S. Glassner. Maintaining winged-edge models. In J. Arvo, editor, *Graphics Gems II*, pages 191–201. Academic Press, 1991.
13. L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graphics*, 4:74–123, 1985.
14. D. Hawken, P. Townsend, and M. Webster. The use of dynamic data structures in finite element applications. *Int. J. Numer. Meth. Engrg.*, 33:1795–1811, 1992.
15. M. T. Heath and W. A. Dick. Virtual prototyping of solid propellant rockets. *Computing in Science & Engineering*, 2:21–32, 2000.
16. L. Kettner. Using generic programming for designing a data structure for polyhedral surfaces. *Comput. Geom. Theo. Appl.*, 13:65–90, 1999.
17. R. Löhner. Some useful data structures for the generation of unstructured grids. *Comm. Appl. Numer. Methods*, 4:123–135, 1988.
18. J.-F. Remacle, B. Karamete, and M. Shephard. Algorithm oriented mesh database. In *9th International Meshing Roundtable*, 2000.
19. J.-F. Remacle and M. S. Shephard. An algorithm oriented mesh database. *Int. J. Numer. Meth. Engrg.*, 58:349–374, 2003.
20. M. C. Rivara. Design and data structure of fully adaptive, multigrid, finite-element software. *ACM Trns. Math. Soft.*, 10:242–264, 1984.
21. The CGNS Steering Sub-committee. *The CFD General Notation System Standard Interface Data Structures*. AIAA, 2002.
22. K. Weiler. Edge-based data structures for solid modeling in curved-surface environments. *IEEE Computer Graphics and Applications*, 5:21–44, 1985.