# CONFORMAL MESH ADAPTATION ON HETEROGENEOUS SUPERCOMPUTERS

By

Daniel Alejandro Ibanez

A Dissertation Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: COMPUTER SCIENCE

Examining Committee:

_______________________________
Dr. Mark S. Shephard, Dissertation Adviser

_______________________________
Dr. Onkar Sahni, Member

_______________________________
Dr. Christopher D. Carothers, Member

_______________________________
Dr. Elliot Anshelevich, Member

Rensselaer Polytechnic Institute
Troy, New York

November 2016
(For Graduation December 2016)

# CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# ACKNOWLEDGMENT

This thesis owes its existence just as much to the support and collaboration of numerous individuals as it does to the efforts of the author. My family must be thanked first for providing so much support that I was able to give this thesis my full attention. I must thank Prof. Mark S. Shephard as my mentor for his unwavering support (and creation) of my career. My doctoral committee have provided me with many insights: Prof. Sahni on software development and the needs of advanced CFD simulations, Prof. Carothers on the ever-changing landscape of supercomputer architectures, Prof. Anshelevich on graph theory's elegant solutions to practical problems, and Prof. Shephard on how topology and adaptivity can greatly improve existing simulations.

I am honored to have worked alongside students including Cameron Smith, Brian Granzow, Daniel Zaide, Aleksandr Ovcharenko, and others whom I consider comrades in the difficult trade of developing good scientific software. Other past and present members of SCOREC including Fabien Delalondre, Seegyoung Seol, Max Bloomfield, and Assad Oberai have provided guidance in life as well as work.

I admire and thank those who pioneer the adoption of mesh adaptation in their important simulation workflows including Kenneth Jansen, Michel Rasquin, Glen Hansen, Chris Kees, and Mike Park.

This thesis is built upon the decades of work by SCOREC including that of Xiangrong Li, Rao Garimella, and Mark Beall. It also owes much to the research in mesh adaptivity by Frédéric Alauzet, Adrien Loseille, and Jean-François Remacle, and in message passing by Torsten Hoefler, Andrew Lumsdaine, and others.

# ABSTRACT

Mesh adaptation is a technique which dynamically modifies the mesh being used to approximately solve a Partial Differential Equation (PDE) in order to improve aspects of the approximate solution including the computer time and memory used to compute it as well as its level of accuracy. Even with the use of mesh adaptation, computing ever more accurate PDE solutions requires significant computer time and memory, motivating the use of supercomputers, which are constructed as networks of cooperating computational hardware. Trends in the computer hardware industry at large are introducing heterogeneous designs for current leadership-class supercomputers, which is both an opportunity and a challenge for programs aiming to make use of these machines.

This thesis presents implementations of mesh adaptation which are designed with memory efficient cache-friendly data structures and algorithms which can effectively leverage both distributed memory parallelism and shared memory parallelism (including GPUs). The data structures used in these implementations are widely applicable to other tasks involving meshes, and the programming paradigms introduced are general enough to be of use in most programs targeting leadership-class supercomputers. The implementations presented are being used by several simulation codes in production, and are available as open-source tools so they may continue providing value to the scientific community.

Several improvements to the design of mesh adaptation programs are presented, including solution transfer methods which preserve mass and momentum, methods for the maintenance of high-quality elements, scalable and deterministic methods for hybrid parallelization of mesh modification operations, and a combination of modification operators which reduce implementation complexity without sacrificing effectiveness.

# CHAPTER 1
# INTRODUCTION AND BACKGROUND

## 1.1 Introduction

A wide variety of aerospace, mechanical, and nuclear engineering problems require solving complex Partial Differential Equations (PDEs) in time and space. For efficiency and reliability, the solutions to these PDEs are found using computers.

Computers are equipped with a limited amount of memory to store information, and must use a mathematical representation of a PDE solution that can be described using a limited amount of information. For many engineering problems of interest, the exact solution as described by any known representation would require an infinite amount of information, therefore approximate solutions are sought.

A certain minimal amount of memory and processing power are required to obtain the lowest accuracy approximate solutions, and obtaining more accurate solutions requires more memory and/or processing power. For these reasons, computers with ever-increasing amounts of memory and processing power are designed and built to increase the accuracy of existing solutions and to solve previously unsolvable engineering problems. At any given point in history, the computers with the most memory and processing power are called supercomputers.

Supercomputers are expensive to acquire and even more expensive to operate in terms of electricity, cooling, and other maintenance, so their ability to produce accurate results at the lowest cost is of critical importance.

When solving PDEs over general domains, it is common to employ equation discretization methods that operate on a mesh, which is more precisely defined in Section 1.4.3. Chapter 2 presents the full design and implementation of two new computer representations of meshes, which are focused on minimizing the amount of computer memory and processing required to use an accurate mesh, and are also compatible with mesh adaptation.

Portions of this chapter submitted as: D. Ibanez and M. S. Shephard, “Modifiable array data structures for mesh topology,” *SIAM J. Scientific Comput.*, under review.

Mesh adaptation is a way in which the spatial discretization (mesh) can be altered over time to maximize the accuracy of the solution obtainable with a certain amount of computing power. There are several open areas of research in mesh adaptation, and Chapter 3 presents advancements made in this work to design and implement mesh adaptation that can run efficiently on present and near-future supercomputers.

Present supercomputers make extensive use of distributed memory parallelism by being constructed from tens of thousands of smaller computers (nodes) connected by a fast network. In an effort to reduce acquisition and maintenance costs for a given amount of computing power, present and future supercomputers will also make extensive use of shared memory parallelism, by having each node be constructed with hundreds to thousands of computing cores, all of which are capable of working in parallel. The combination of the two forms of parallelism is what makes a supercomputer *heterogeneous*.

Designing programs for heterogeneous supercomputers is a critical challenge, because such programs must be able to precisely coordinate a complex hierarchy of memory and interaction methods executing on millions of compute cores in order to solve a given problem at minimal cost. Chapter 4 presents contributions to the design and implementation of parallel programs, including both widely applicable tools and tools specifically designed to enable efficient parallel mesh adaptation.

An additional challenge to the design of parallel programs is the fact that supercomputers fall into different architectural design categories which currently have very different programming interfaces. Thus, in order to design a program which is *portably performant* over all architectures, one must try to abstract away their differences and, unfortunately, cater to the lowest common denominator of functionality. Throughout this thesis, we present two systems: the first only operates on some architectures and provides a wide array of adaptive functionality, while the second is portably performant across the major architectures and currently provides less adaptive functionality.

Finally, Chapter 5 presents several application programs which make use of the tools developed here in order to solve a variety of engineering problems.

## 1.2 Terminology

| | |
|---|---|
| Topological Complex | A breakdown of a Cartesian domain into topological entities |
| Mesh | A topological complex whose entities have simple shape |
| Entity | A topological entity of a mesh |
| Vertex | A 0-dimensional entity |
| Edge | A 1-dimensional entity |
| Face | A 2-dimensional entity |
| Region | A 3-dimensional entity |
| Element | An entity not bounding another entity |
| (Hardware) Node | A computer having a single hardware memory space, cooperating with others via a network |
| Core | A CPU core or a GPU hardware thread (the unit of hardware parallelism) |
| Process | An operating system process (has a software memory space, contains one or more threads) |
| Thread | An operating system thread or a CUDA thread (the unit of software parallelism) |
| (MPI) rank | One of several operating system processes cooperating using MPI to execute a parallel program |
| (mesh) partition | The subset of a mesh stored in a single data structure. |
| (mesh) part | Synonym for mesh partition |

## 1.3 Notation

| | |
|---|---|
| $\Omega$ | a subset of Cartesian space (a domain) |
| $T^d$ | the subset of $d$-dimensional entities in a topological complex $T$ |
| $T_i^d$ | the $i$-th entity in $T^d$ |
| $\partial e$ | the set of lower-dimensional entities adjacent to entity $e$, called its boundary |
| $\partial S$ | for a set of entities $S$, $\partial S = \bigcup_{e \in S} \partial e$ |
| $\bar{S}$ | for a set of entities $S$, $\bar{S} = S \cup \partial S$ |
| $a \subseteq \partial b$ | entity $a$ is a subset of entity $b$'s boundary, meaning $a$ is adjacent to $b$ |
| $a\{T^q\}$ | the set of entities in $T^q$ adjacent to entity $a$ |
| $S\{T^q\}$ | for a set of entities $S$, $S\{T^q\} = \bigcup_{e \in S} e\{T^q\}$ |
| $\mathcal{M}$ | the (symmetric positive-definite) mesh metric tensor |
| $h$ | the desired length of mesh edges (isotropic size) |
| $\text{diag}(a,b,c)$ | a diagonal matrix $A \in \mathbb{R}^{3\times 3}$ with $A_{11} = a, A_{22} = b, A_{33} = c$ |
| $\lceil x \rceil$ | the smallest integer $\geq x$ |
| $l_e$ | the length of edge $e$ in real space |
| $\tilde{l}_e$ | the length of edge $e$ in metric space |
| $V_K$ | the volume of tetrahedron $K$ in real space |
| $\tilde{V}_K$ | the volume of tetrahedron $K$ in metric space |
| $l_{K,\text{RMS}}$ | the root-mean-squared edge length of tetrahedron $K$ |
| $\eta_K$ | the mean ratio quality of simplex $K$ |
| $P$ | the number of processes in an MPI job |
| $T$ | the number of threads per process |
| $\rho$ | density |
| $v$ | velocity |
| $\mathbf{p}$ | momentum |

## 1.4 Mesh Definitions

We define here the basic concept of a *mesh* (Section 1.4.3), based on the general idea of a *topological complex* (Section 1.4.1) and focusing on its *adjacency relations* (Section 1.4.2). Next we go on to specify the type of meshes we are dealing with

(Section 1.4.4), and the unique way in which they are being modified, i.e. mesh *adaptation* (Section 1.4.5).

### 1.4.1 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 $0 \leq d \leq D$. 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 $\bar{T}_i^d = T_i^d \cup \partial T_i^d$, is the open set plus its boundary. Since the sets are disjoint, only their boundaries may intersect. For all pairs of equal-dimension point sets, the intersection of their boundaries must exist as the union of other, lower-dimensional point sets in $T$:

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

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 lower-dimensional 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 as *entities.*

Figure 1.1 illustrates two topological complexes, both in 2D. On the left is a cross-section of an airfoil in fluid, showing how the airfoil forms a “hole” in the fluid,

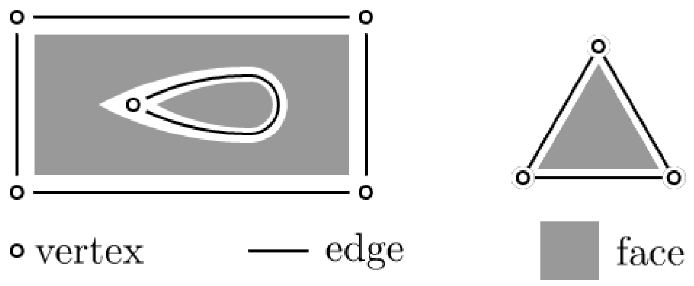

**Figure 1.1: Topological complexes: (left) an airfoil CAD model (right) a single triangle**

and a boundary may be represented by a single edge that is curved and connects to only one unique vertex. On the right is a simpler model of a triangular domain, which is representative of how triangles are viewed in a mesh.

## 1.4.2 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 $(b, a)$. Together, upward and downward adjacencies defined this way are called first-order adjacencies.

The first-order adjacency relations in a mesh define a graph, which we call a 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: $T^p_i\{T^q\}$ is the set of entities (point sets) in $T^q$ adjacent to $T^p_i$. 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

$T_i^a\{T^b\}\{T^c\}$.

Although these graphs have a natural direction for each edge (from higher dimension to lower), we are interested in being able to query both outgoing (downward) and incoming (upward) relations, so the storage will be bi-directional in many cases.

Another useful concept will be the *entity use*, which is essentially an edge of the topology graph. If entity $b$ is in the boundary of entity $a$, then $b$ is used by $a$, and that occurrence is an entity use. The term shows up when data is stored once for every adjacency relation.

### 1.4.3 Mesh

A *mesh* $M$ is herein defined as a special case of a topological complex where the closure of each entity $\bar{M}_i^d$ is topologically a polytope of dimesion $d$. Mesh entities which do not bound other entities are called *elements*.

Being polytopes topologically, mesh entities have no holes or internal empty spaces, so they do not need multiple loop or shell constructs to describe their boundary the way a BRep CAD model would.

### 1.4.4 Finite Element Mesh

We further define a *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 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 ($d = 3$)

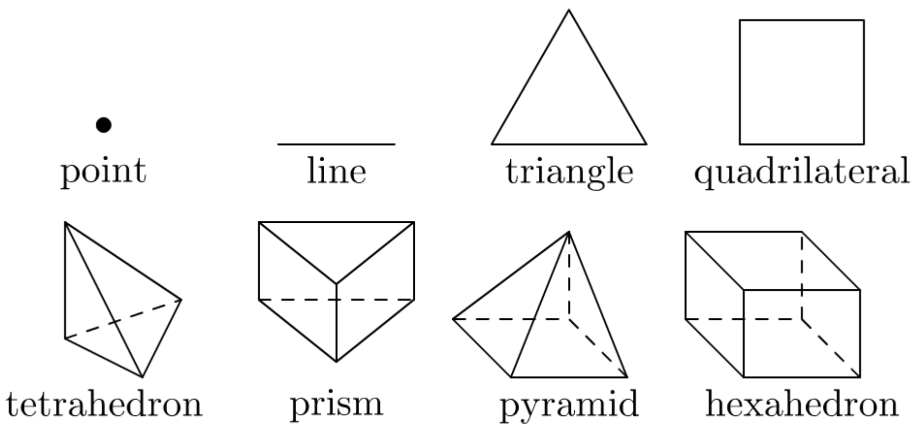

**Figure 1.2: Polytopes commonly used in FE and FV meshes**

7. (square-based) pyramid ($d = 3$)

8. triangular prism ($d = 3$)

This list of polytopes is also illustrated in Figure 1.2, and can be easily extended to include additional polytope types of interest.

This work is focused on *unstructured meshes*, meaning their topology must be explicitly stored because it is not originally defined by some simple pattern such as a grid. Such unstructured meshes have an advantage in representing complex geometry and in their ability to easily vary resolution throughout the geometry.

In addition, the Finite Element Method uses fields which are each defined as the weighted sum of finite number of basis functions, where the weights are referred to as *degrees of freedom* and are each attached to one mesh entity. This requires a data structure that can attach degrees of freedom to 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 (see Appendix B.1.1 for a proof of this bound).

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*.

#### 1.4.4.1 Topological Template

When dealing with stored mesh topology and attempting to describe the relationships between adjacent entities, one needs some frame of reference to begin with. This takes the form of a *topological template* [1,2] which describes, for a single polytope, a canonical numbering of its boundary entities. For example, the vertices of a tetrahedron are numbered such that the triple product $(x_1 - x_0) \times (x_2 - x_0) \cdot (x_3 - x_0)$ yields a positive value, where $x_i$ is the coordinate of vertex $i$. Furthermore, the triangles bounding a tetrahedron have a canonical ordering and orientation, for example the canonical second face of tetrahedron $(a, b, c, d)$ can be defined as the triangle $(a, b, d)$. Although these decisions are arbitrary, it is necessary to choose such a template for each topological type in order to have a frame of reference when programming operations that act on the boundary of an entity. It can also be useful to choose orderings to have certain special properties that ease the programming of certain algorithms, for example orienting all faces of an element such that the normal of the face points outwards from the element. See Section 2.5.3 for an example of how this information is used.

### 1.4.5 Adaptation

There are two approaches to modifying the topology throughout 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 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*.

Adaptation refers to a process of modifying the mesh by applying mesh entity-level 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 the mesh entities within the cavity, leaving its boundary unchanged.

Adaptation has a number of benefits compared to complete remeshing. 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. If adaptation is programmed as a series of entity additions and removals, then we require that *entity addition and removal be constant-time operations*. Section 4.8 presents two alternatives to scheduling cavity operations, the latter of which allows changes to be grouped into batches, which allows the use of simpler data structures.

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 and leaving the old entities in place.

As a consequence, the mesh structure must *tolerate temporary topological inconsistencies* introduced by adaptation. 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.

Certain modifications, such as edge collapsing, can only correctly preserve the boundary of the mesh the mesh structure *maintains a direct mapping from mesh entities to geometric (CAD) model entities.* This mapping is referred to as classification [3]. 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 require knowledge of the classification for all mesh entities, hence requiring a complete (though not necessarily full) topological representation [4] to safely coarsen a mesh. It would be possible to avoid storing some entities, so long as the classification of entities not stored could be inferred. Beyond that, it is convenient in any case to represent edges explicitly, given that many adaptive algorithms are based on edge lengths [5]. The adaptive applications in this work all use representations that store all entities.

### 1.4.6 Metric Field

The most common way in which one describes the desired effects on the mesh to an adaptation program is via a metric field [6–8]. From a continuous perspective, the metric field is a tensor $\mathcal{M}$ which varies over physical space. This tensor describes a metric because at any given point $\mathbf{x}$ in space, one can choose a direction expressed as a unit vector $\mathbf{u}$, then the product $\mathbf{u}^T\mathcal{M}(\mathbf{x})\mathbf{u}$ replaces the inner product $\mathbf{u}^T\mathbf{u}$ for measuring distance along the given direction. We require the metric tensor to be a real symmetric positive-definite $d \times d$ matrix for spatial dimension $d$, which means it can be diagonalized by an orthogonal matrix and has all positive eigenvalues, as expressed in Equation 1.1.

$$\mathcal{M} = R^T \Lambda R,\ R^T R = I,\ \Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_d),\ \forall i \in [1, d] : \lambda_i > 0 \qquad (1.1)$$

The matrix $R$ can be viewed as describing a rotation, while the diagonal matrix $\Lambda$ scales a vector by a positive amount along each coordinate axis. One can then consider the matrix $Q$ as defined by Equation 1.2 to be defining an affine transformation (rotation followed by scaling) that is applied to vectors before their inner product is taken, as illustrated in Equation 1.3.

$$Q = \Lambda^{\frac{1}{2}} R,\ \Lambda^{\frac{1}{2}} = \mathrm{diag}(\sqrt{\lambda_1}, \ldots, \sqrt{\lambda_d}) \tag{1.2}$$

$$\mathbf{u}^T \mathcal{M} \mathbf{v} = \mathbf{u}^T R^T \Lambda R \mathbf{v} = \mathbf{u}^T R^T \Lambda^{\frac{1}{2}} \Lambda^{\frac{1}{2}} R \mathbf{v} = \mathbf{u}^T Q^T Q \mathbf{v} = (Q\mathbf{u})^T (Q\mathbf{v}) = \tilde{\mathbf{u}}^T \tilde{\mathbf{v}} \tag{1.3}$$

Viewed in this way, the metric tensor is actually defining, at each point in space, an affine transformation that should be applied to vectors before computing local spatial qualities. For us, it is sufficient to consider how the metric tensor alters length along a direction **u** (Equation 1.4), as well as volume (Equation 1.5). We use a tilde to denote quantities "in metric space" (post-transformation).

$$\tilde{l} = l \cdot \sqrt{\mathbf{u}^T \mathcal{M} \mathbf{u}} \tag{1.4}$$

$$\tilde{V} = V \cdot \sqrt{\det(\mathcal{M})} \tag{1.5}$$

Note that for many applications, a metric which is isotropic may be sufficient, meaning that the entire tensor may be represented by a single scalar $\lambda$, or for convenience a single desired length value $h = 1/\sqrt{\lambda}$. It is useful for an adaptation code to consider this special case separately because storage and algorithmic complexity are greatly reduced (see Section 3.5).

### 1.4.7 Element Quality

Many researchers use a quality measure known as the mean ratio to evaluate tetrahedra and triangles [6–9]. In Equation 1.6, we define the mean ratio in an equivalent but slightly different form than presented by other authors. This is to clearly show that the mean ratio may be interpreted as comparing a tetrahedron's volume and the volume of an equilateral tetrahedron with the same root-mean-squared edge length.

$$\eta_K = \left(\frac{V_K}{\gamma_K \cdot l_{K,\mathrm{RMS}}^3}\right)^{\frac{2}{3}}, \; l_{K,\mathrm{RMS}} = \left(\frac{1}{6}\sum_{i=1}^{6}(l_{K,i})^2\right)^{\frac{1}{2}}, \; \gamma_K = \frac{1}{\sqrt{72}} \tag{1.6}$$

In Equation 1.6, $V_K$ is the tetrahedron volume, $l_{K,i}$ is the length of edge $i$ of the tetrahedron, and $\gamma_K$ is the volume of an equilateral tetrahedron with unit edge length. A similar definition exists for a triangle $T$ with area $A_T$ as shown in Equation 1.7.

$$\eta_T = \frac{A_T}{\gamma_T \cdot l_{T,\mathrm{RMS}}^2}, \; l_{T,\mathrm{RMS}} = \left(\frac{1}{3}\sum_{i=1}^{3}(l_{T,i})^2\right)^{\frac{1}{2}}, \; \gamma_T = \frac{\sqrt{3}}{4} \tag{1.7}$$

## 1.5 Reference Computer

Since the following sections will describe a variety of different computer hardware systems, it is useful to have a reference point of hardware to which they can be compared. We define a "reference computer" as follows:

1. It has a single CPU, which contains a set of registers (places to store single values), and can perform operations with the inputs and outputs being registers. Accessing a register is instant, but there are only a dozen or so of them.

2. It has memory which is used to temporarily store all the data required to solve the problem. The CPU can transfer data between registers and memory at a moderate cost.

3. It has a disk which is used to permanently store inputs and outputs of the overall program. The CPU can transfer data between memory and the disk, at a high cost.

4. It has a user interface, in our case we will simply say a human operator has some way to receive data from and provide data to the CPU in real time.

5. It has a network interface which transits data to or receives data from other computers.

6. Only one of the above actions may be performed at one time.

7. With the exception of a human or network choosing to provide data, all choice of actions is dictated by the CPU, which is in turn obeying the instructions of a single stored program.

We will also not pay much in-depth attention to the architecture of the cache system, because it is largely transparent to the program. In particular, the program cannot explicitly command the cache to behave in a certain way; it can only indirectly influence the performance of the cache in they way it transfers data to and from memory. The key performance principle regardless of cache design is to make sure the order in which values are stored in memory reflects the order in which they are accessed.

## 1.6 Heterogeneous Node Architecture

As mentioned in Section 1.1, a supercomputer is a networked collection of nodes, and the design of a single node is where much of their diversity has appeared in recent years. This Section describes the three key hardware component developments influencing this diversity. Any present-day supercomputer node is a combination components from each of these three categories.

### 1.6.1 Multi-core CPUs

Personal computers and similar mobile devices are limited to a single CPU due to the evolution of their hardware and software, so the computing power of a single CPU defines the quality of the products from which vendors make the majority of profits. Given the technology at any point in time, a single CPU core can only run so fast, and the way to add even more performance has been to include multiple cores per CPU, creating a miniature parallel system on a single chip [10]. At the time of this writing, most consumer devices have two or four cores, and the high-end CPUs from which the majority of clusters and previous generation supercomputers are constructed contain about sixteen cores.

A CPU core has its own set of registers, and more importantly it has the full capability to dictate transfers to memory, disks, the network, and the user interface. In comparison with the reference computer from Section 1.5, the key difference is that each core executes independently, and therefore two cores can execute *different* actions at the *same* time.

### 1.6.2 GPU Coprocessors

For the past two decades, many personal computers came equipped with pieces of specialized hardware called Graphics Processing Units (GPUs). These components were chips specifically hardwired to perform the highly parallelizable operations involved in displaying graphics, where each screen pixel could be handled by a separate parallel processor. As research in graphics progressed, there was a need to move from hardwired to programmable graphics processing to implement more realistic graphics algorithms. GPUs whose operations were somewhat programmable began appearing in 2002 such as NVidia's NV30 device. By 2004, Stanford University had developed a useful programming environment called BrookGPU [11], which enabled researchers to more easily use GPUs to solve new problems and was subsequently adopted by AMD. Usage spread from the graphics community to the closely related scientific computing community. GPUs were an affordable personal solution for researchers to access substantial computing power. The interest soon motivated NVidia to release a programming environment called the Compute Unified Device Architecture (CUDA), alongside its GeForce 8800 products in 2006. By 2009, not only was CUDA was being seriously considered as a programming environment to solve a wide variety of problems [12], but GPUs had made their way onto the world's top supercomputers. China's Tianhe-1A computer, revealed in 2010, contained over 7K NVidia GPUs, making it the world's most powerful supercomputer until 2011 [13]. American supercomputers soon followed suit, and in 2012 the Titan supercomputer at Oak Ridge National Laboratory was equipped with over 14K NVidia GPUs [14], making it the world's most powerful in the year 2012. By this point, GPUs had thousands of "cores" each.

In comparison to multi-core CPUs and the reference computer in Section 1.5, a GPU is in many ways limited in comparison to a multi-core CPU. The GPU and its cores cannot directly access CPU memory, disks, networks, or user interfaces. Rather, GPUs have their own memory hardware, separate from that of the CPU, and roughly equal in size. GPU cores can only execute a small program called a *kernel*, which is given to the GPU by the CPU. Since the GPU accepts a different instruction set than the CPU, kernel code must be compiled specifically for the GPU. Other actions of the GPU are dictated at a high level by the CPU, including transfer of data from CPU to GPU memory. Due to the lack of other access, CPU-GPU data transfer is required for network communication or disk file access. To make matters worse, such data transfer is slow (low bandwidth) compared to the rate at which the GPU can process data, meaning that without careful implementation, one can spend the majority of computer time simply waiting for data transfers [15].

Despite these limitations, the sheer number of cores in a GPU combined with the efficiencies gained in exchange for lack of generality means that GPUs can theoretically outperform CPUs by an order of magnitude, and should be substantially more energy efficient. This motivates researchers to re-design algorithms such that their practical performance can approach this theoretical promise.

### 1.6.3 The Intel Xeon Phi

During the years that GPUs were being introduced into leading supercomputers, Intel began developing a family of chips which were based on their multi-core CPUs but tended more and more towards GPU-like capabilities, by increasing the number of cores from tens to hundreds and while decreasing the amount of power per core. This family is called the Intel Xeon Phi line, and began with co-processors that behaved like GPUs, requiring a host CPU to send them specific compute requests. In 2013, China presented their TianHe-2 supercomputer, whose nodes had two Intel CPUs and three Intel Xeon Phi co-processors each. With 16K total nodes, TianHe-2 remained the world's most powerful supercomputer until 2016, the year of this writing. Intel has since evolved their Xeon Phi co-processor into a full-fledged processor with nearly one hundred CPU cores [16], which will be used to construct near-future supercomputers in the United States.

A Xeon Phi processor can for the most part be considered a CPU with many more cores than usual, with the additional benefit that they have performance properties close to those of a single GPU. Conversely, it may provide the hardware characteristic benefits of a GPU with the generality of a CPU, since all cores can access main memory, disks, and networks as described in Section 1.5.

## 1.7 Programming Environments

When writing programs for heterogeneous supercomputers, there are a few key programming environments which motivate choices about the implementation.

### 1.7.1 Operating System

Even when given hardware as simple the reference computer from Section 1.5, one can emulate the presence of more hardware by using an *operating system*, which is a program that directly executes on the given computer and in turn allows the execution of multiple other programs simultaneously on that same computer. The two operating system concepts most relevant to our work are the *process* and *thread*. An operating system process is an emulation of the reference computer: it has its own memory space, which emulates hardware memory, as well as access to disks, user interfaces, and networks, which the operating system manages such that the process appears to have unique access to these *resources*. A process also contains one or more threads, which are software emulations of CPU cores. They have their own emulated registers, and can execute different actions at the same time. A process has a single associated program whose instructions the threads are following, although each thread may be following a different subset of the instructions depending on its own local state (registers and stack). Operating system threads all have access to emulated process resources such as networks in the same way that CPU cores all have access to computer interfaces.

Despite having great complexity in their own right, operating systems are fundamentally just sharing mechanisms. Increasing the number of threads and processes that share a fixed amount of hardware resources will degrade the emulated performance of all the active threads as they will each be provided with a smaller fraction of the hardware resources.

If one instead intends to devote all the hardware of a machine to a single program, then it is natural to align the operating system constructs with their physical counterparts. This means binding an operating system process to a CPU, such that its memory space maps to the CPU's entire available memory, and binding operating system threads to CPU cores, such that the amount of parallelism and the amount of memory sharing perceived at the software level both match the amount supported at the hardware level.

### 1.7.2 MPI

Distributed memory computers have for the past two decades been programmed using the Message Passing Interface (MPI) [17,18]. This interface remains the most reliable and portable way to construct programs for all distributed memory supercomputers, and is thus central to any scalable program. Although its early versions (version 2.0 and earlier) had scalability challenges [19], MPI has evolved to effectively address these challenges and "modern" MPI as described in [20] scales well into the million-core range. We will present optimal usage of modern MPI features in Section 4.2 and show in Section 5.4 that one of the applications using tools developed in this thesis scales to 768K cores using only MPI [21].

MPI reflects in software the architecture of a distributed-memory machine: a set of compute nodes which execute in parallel and cooperate via a network. Each node has its own memory must explicitly send network messages to provide or query data stored in another node. An MPI rank is an operating system process, and MPI provides a consistent interface over the various operating system network interfaces. Whereas operating system networking uses complex addresses and protocols for generality, MPI maintains a concept of a group of $P$ ranks which are cooperating and may address one another by a consecutive integer from 0 to $(P - 1)$. MPI provides many complex algorithms for coordinating such groups of ranks, as well as a simplified method of transmitting data that allows programs to focus on the contents of the data as opposed to the details of what protocol may be used on the network. In software terminology, MPI is implemented as a library, which means it is simply a set of functions that can be called from a program.

**Listing 1.1: Serial for loop**

```
for (int i = 0; i < n; ++i) {
  b[i] = a[i] + 3.14159;
}
```

### 1.7.3 OpenMP

OpenMP is a programming environment mainly for multi-core CPUs. In the same way that MPI is a software reflection of distributed memory, OpenMP is a software reflection of shared memory. It is based on the concept of a single operating system process that contains $T$ operating system threads which are cooperating. Since memory is shared within an operating system process, OpenMP does not provide message-passing functionaliy the way MPI does. What OpenMP does provide is the concept of a parallel **for** loop, which is a key abstraction for shared memory programming. In structured serial programming, there is the concept of a **for** loop. Listing 1.1 illustrates a **for** loop in the C++ language, which adds a constant to the entries of an array `a` and stores the resulting values in the array `b`. This is done for a total of $n$ entries. As written, a single thread does this work one entry after another.

Unlike MPI, OpenMP is directive-based, meaning that it is integrated directly into a programming language compiler in order to introduce new language notation. As Listing 1.2 illustrates, this allows simple annotation of the loop from Listing 1.1 which turns it into a parallel **for** loop. What now occurs is that all of the cooperating threads in OpenMP will share the work of this loop. Given $T$ cooperating threads, each thread will do the work for approximately $(n/T)$ entries of the array, and OpenMP is responsible for the scheduling of which entries each thread works on. Assuming all threads are supported by proper CPU cores, the time to complete this work should be $O(n/T)$, whereas the serial loop would have completed in $O(n)$ time.

Like any form of parallelism, this places restrictions on what serial constructs may be parallelized. For example, even a simple sum operation as shown in Listing 1.3 could not be parallelized in the same simple manner, because the `sum` variable is being read from and written to by multiple threads without the necessary coordina-

**Listing 1.2: OpenMP parallel for loop**

```
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
  b[i] = a[i] + 3.14159;
}
```

**Listing 1.3: Dependent for loop**

```
for (int i = 0; i < n; ++i) {
  sum = sum + a[i];
}
```

tion. Such a situation is referred to as a *race condition*, because depending on the order in which different threads perform actions with respect to one another (which is indeterminate), incorrect results may be produced.

We refer to the execution of the body of the **for** loop for a single value of `i` as an *iteration.* In order to avoid race conditions, we introduce the following principles of array-based parallel **for** loop programming:

1. Avoid writing to and reading from the same array in the same loop.
2. No array entry should be written to by more than one iteration.
3. If an array entry must be read from and written to, it must only be accessed by a single iteration of the loop.
4. If the above rules must still be violated, it must be done via atomic operations.

Atomic operations exist to cover limited cases of multiple writes the same entry by different threads or loop iterations. For example, an atomic addition operation allows multiple iterations to add values to the same array entry, with the guaranteed outcome that after all iterations have executed, the correct sum appears at this entry.

### 1.7.4 CUDA

CUDA is NVidia's programming environment for their large family of programmable GPUs [22]. Like OpenMP's parallel **for** concept, CUDA's programming

**Listing 1.4: CUDA for loop**

```
__global__ add(double* a, double* b) {
  int i = blockIdx.x;
  b[i] = a[i] + 3.14159;
}
add<< n, 1 >>(a, b);
```

model is based on parallel execution of kernels, which are almost the same as parallel **for** loop iterations. Listing 1.4 shows how one might transform Listing 1.1 into the CUDA programming model. Like OpenMP, CUDA is integrated into the compiler in order to introduce its own non-standard language notation. The loop iteration becomes its own function which must be properly annotated to indicate it is a kernel.

The limitations of GPUs described in Section 1.6.2 make their way into the design of CUDA and software that uses it. Annotations required on CUDA functions because the code in them must be compiled specifically for the GPU. CUDA also provides the functions with which the CPU controls GPU memory allocations. In Listing 1.4, for example, both arrays `a` and `b` need to be explicitly allocated in GPU memory. As Section 1.6.2 mentions, CPU-GPU data transfer is slow and should be avoided. Also, for all practical purposes, kernel code should not allocate or deallocate memory. Although limited support for this exists, it should be avoided for best performance. This leads to the following principles of GPU programming:

1. Organize data in large allocations, such that the number of allocations is not proportional to the size of the data. This allows the CPU to manage allocations efficiently.

2. Allocate all data on the GPU, and only transfer it to the CPU for network and disk operations.

3. A kernel should do a small, constant amount of work using a small, constant amount of memory.

The first and third principles have a serious impact on data structure design, ruling out the many-small-objects approach. Any data structures used locally at the

**Listing 1.5: Kokkos for loop**

```
auto add = KOKKOS_LAMBDA(int i) {
  b[i] = a[i] + 3.14159;
}
Kokkos::parallel_for(n, add);
```

kernel level should be designed such that their size is known at compile time. The second principle is essentially the most aggressive approach to avoid CPU-GPU data transfer, and is motivated by the high cost of such data transfer. Although it can be argued this is too restrictive, we show in this thesis that very complex unstructured operations can be carried out effectively while adhering to this principle.

### 1.7.5 Kokkos

Kokkos is a C++ library developed at Sandia National Labs that provides a portable shared memory programming environment encompassing other environments such as OpenMP and CUDA [23]. Kokkos maintains the parallel **for** loop model of OpenMP while accounting for the separation of code and data that may occur if CUDA is used. Listing 1.5 illustrates what the loop from Listing 1.1 looks like when implemented using Kokkos.

The key here is that this code does not change when one moves from multi-core CPUs using OpenMP to GPUs using CUDA. Even for this simple example, the changes required would otherwise be significant (compare Listings 1.2 and 1.4).

Kokkos also provides a data structure describing an array allocation with features for controlling allocation to GPU memory if necessary and explicit control over any transfers to the CPU.

## 1.8 Overview of Software

Below we present the various pieces of software which contain the end products of the research and development done as part of this thesis.

### 1.8.1 PUMI

The Parallel Unstructured Mesh Infrastructure (PUMI) is a software package containing several libraries which together provide all the tooling necessary to store, query, and adapt partitioned meshes with simulation fields attached [24]. PUMI's design is based on locally serial software threads which cooperate with one another using message passing, assuming there is no sharing of memory. Each software thread is required to have access to all computer functions including memory allocation, message passing, file I/O, and the ability to call any available third party library. Because of these assumptions, PUMI's code could not execute on more restricted architectures such as GPUs without very substantial re-design and re-implementation. Key components of PUMI include:

#### 1.8.1.1 PCU

The Parallel Control Utility is a C library which implements the key parallel communication systems required to construct more complex parallel programs which remain scalable. Its algorithms and implementation will be covered in detail in Section 4.3. PCU uses MPI to transmit all its messages, making it portable across many distributed memory supercomputers. PCU also made an effort to implement hybrid threading in a way that was transparent to the rest of PUMI by providing MPI-like communication between threads.

#### 1.8.1.2 APF

Attached Parallel Fields is a C++ library whose original goal was to manage fields discretized over a mesh. In order to achieve that goal without being tied to a particular mesh implementation, APF included an abstract interface for interacting with any mesh implementation. Because algorithms based on this interface are immediately able to operate on multiple different mesh implementations, APF now contains many such algorithms dealing with topological and parallel operations.

#### 1.8.1.3 MDS

The Mesh Data Structure library is a C library implementing the main mesh data structure for APF and therefore PUMI. It is array-based, supports adding and removing entities in constant time, can represent multiple element types at once, and is augmented with parallel connectivity information for partitioned meshes. This structure will be described in detail in Section 2.4.

#### 1.8.1.4 MeshAdapt

MeshAdapt is a C++ library that leverages PCU, APF, MDS, and other libraries in PUMI to implement scalable parallel mesh adaptation. MeshAdapt developments which are part of this thesis will be covered in detail in Sections 3.3 as well as Section 4.8.1.

### 1.8.2 Omega_h

Omega_h is a single C++ library which was designed from the beginning with the restrictions and potential of GPU programming in mind [25]. It aims to provide as much of the core functionality of PUMI as is feasible in a portably performant way, which currently amounts to adapting triangular and tetrahedral meshes using anisotropic metrics. Omega_h represents the first such portably performant code capable of mesh adaptation, and is a major contribution of this thesis. It uses the Kokkos library to achieve portable on-node parallelism as described in Section 1.7.5, and techniques it uses to produce deterministic results are presented in Section 4.9. Its data structure will be described in Section 2.5. Its implementation of mesh adaption will be discussed mainly in Section 3.4, with key parallel aspects covered throughout Chapter 4.

## 1.9 Contributions

This thesis centers around contributions made to the design, implementation, and distribution of mesh adaptation libraries. In the following summary, any reference to the design of an algorithm or structure indicates that said design is a completely novel contribution of this thesis. Likewise, any reference to the implementation of a software library indicates that said library was developed in its entirety by the author in the course of this thesis work.

1. The design and implementation of a new mesh adaptation library, Omega_h,

built using the parallel **for** loop model to allow it to execute on GPUs, Intel Xeon Phis, etc. There are more novel algorithms encompassed in this contribution than can be included in this summary, see Sections 2.5, 3.4, 3.5, 3.6.1, 4.8.2, and 4.9.

2. The design of a new data structure and its implementation in MDS, which has an object-oriented interface and an array-based implementation (Section 2.4). This is the first such design that directly maintains the types of adjacency arrays needed for a full topological representation.

3. The design of scalable communication for unstructured mesh adaptation codes, exemplified by the implementation of the PCU communication library (Section 4.3) for inter-thread message passing and by certain components of the Omega_h library (Sections 4.5).

4. The implementation of the most recent version of MeshAdapt, which is based on the scalable communication of PCU and the novel structure of MDS. This library provides the algorithms developed by decades of research prior to this thesis (as described in Section 3.3) in a form that is scalable (Section 4.8.1), efficient, and has high quality source code.

5. The use of MeshAdapt and Omega_h to bring adaptive capabilities to real simulation codes used by universities, government researchers, and private companies to solve problems in structural mechanics, aerodynamics, hydrodynamics, and other areas. Chapter 5 describes these applications.

# CHAPTER 2
# ARRAY-BASED MESH REPRESENTATIONS

This Chapter presents two data structures, each used to represent the portion of the mesh (a.k.a. *mesh part*) stored in one shared memory space. Each of these structures are augmented with information connecting multiple mesh parts across distributed memory spaces as described in Chapter 4, the result of which is a pair of fully parallel mesh data structures.

## 2.1 Goals

An unstructured mesh simulation code relies heavily on multiple core capabilities to deal with the mesh, 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 thesis contributes is the development of a mesh data structure with the following capabilities:

1. The flexibility to adapt to evolving meshes
2. The ability to represent any of the conforming meshes typically used by Finite Element (FE) and Finite Volume (FV) methods
3. Low memory use
4. High locality of storage
5. Highly scalable implementation for distributed memory computers
6. The ability to parallelize work inside heterogeneous supercomputer nodes

The first goal is the most consequential. If adaptivity is implemented as a series of requests to add and remove mesh entities, then a much more complex structure is required (see Section 1.4.5 for further discussion). Such a structure is

Portions of this chapter submitted as: D. Ibanez and M. S. Shephard, “Modifiable array data structures for mesh topology,” *SIAM J. Scientific Comput.*, under review.

described in Section 2.4. However, this approach tends to conflict with our sixth goal regarding on-node parallelism. Therefore we also present an alternative approach in which the additions and removals required for adaptivity are accumulated into batches to be applied all at once (see Section 4.8.2), which allows us to use a less complex structure as presented in Section 2.5 while adhering to the principles of on-node programming established in Sections 1.7.3 and 1.7.4.

## 2.2 Related Work

There are several other implementations of mesh data structures which offer various subsets of the features described herein:

First, there are dynamic structures which support mesh adaptation. FMDB is an object-oriented structure that stores full or reduced representations [1,4]. It is capable of constant-time local mesh modifications and supporting adaptation code. Similar to FMDB is the mesh databased used by Compère and Remacle in the MAdLib adaptation package [7]. Celes, Paulino, and Espinha also implement a structure capable of adaptation [27]. Another example of an adaptive structure is the GRUMMP system developed by Ollivier-Gooch [28].

Second, there are array-based structures designed for efficient access of unchanging meshes. STK is an array-based mesh structure being developed at Sandia National Laboratory [29]. MOAB is another array-based structure developed primarily at Argonne National Laboratory [30], and working with MOAB an Adjacent Half-Facet structure was implemented which can perform some modifications [31].

## 2.3 Choices in Representation

There are key choices in terms of which entities and adjacencies to store which apply equally well to both of the structures presented in this Chapter.

### 2.3.1 Choosing Entities to Store

Unstructured mesh applications must represent the portions of the mesh topology graph needed to support the operations carried out on the mesh. A representation which explicitly stores every entity is said to be a *full* representation. Any schemes which allow some entities to be represented implicitly (i.e. their presence does not consume memory) are *reduced* representations [1, 32, 33].

As we began to discuss in Section 1.4.5, there are multiple factors in the decision of which entities to store:

1. For boundary condition application, it is necessary to be able to infer the classification of mesh faces.
2. For mesh adaptivity, it is necessary to be able to infer the classification of any mesh entity, whether stored or not.
3. In mesh adaptation, it is often useful to store values on the edges and vertices (for example, booleans indicating they are candidates for modification).
4. If the basis functions used for PDE discretization have degrees of freedom on edges or faces, those edges and faces must be stored.

There are also many consequences of storing only a subset of entities that must be considered:

1. If some entities of a given dimension are not stored, then downward adjacencies from a higher dimension to the partially-stored dimension would need to be more complex.
   (a) Given a single high-dimensional entity, the ordering of its adjacent bounding entities is crucial. If some of them are not stored, the remaining adjacency relationships would need to be annotated with ordering information such that the correct ordering can still be reconstructed.
   (b) Depending on the implementation, downward adjacencies would either not have constant degree or have "null" values where non-stored entities are adjacent. Both would complicate all code that queries and manages downward adjacencies.
2. How to query information (such as classification) about a non-stored entity is unclear. FMDB took the approach of temporarily creating and storing those

entities when queries were made [1]. Celes, Paulino, and Espinha use a similar solution in their reduced representation [27]. This workaround is incompatible with portable execution, as GPUs would not allow creating entities individually in this manner.

Beall and Shephard indicate other complexities involved with handling reduced representations [34].

Given that it is convenient to store edges for adaptation and boundary faces are needed for classification, the remaining consideration is whether to introduce complexity in exchange for not storing interior faces. The choice made in this work is to use a full representation, i.e. store all entities.

### 2.3.2 Choosing Adjacencies to Store

Once the set of explicit entities is chosen, one has options about which adjacencies to store. Recall from Section 1.4.2 that downward and upward adjacencies are transitive, so there are many subsets of the adjacencies from which the others can be reconstructed. 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 one can compute $M_i^d\{M^q\}$ for a single given entity $M_i^d$ in constant time, we say that the stored information is *complete* [4]. Recall from Section 1.4.4 that upward adjacencies are bound by a constant, so they are computable in constant-time if enough information is stored.

A comparison of representations based on the choice of dimensions and adjacencies between dimensions to represent was published by Garimella [32]. The choices we made in our structures are described separately for each data structure, in Sections 2.4.1 and 2.5.

## 2.4 MDS Data Structure

The MDS structure is built around the following key characteristics:

1. The representation centers around graph theoretic interpretations of topological adjacency.

2. The mesh can remain topologically consistent with and associated with geometric model entities.

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.

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 key contribution of this thesis 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.

### 2.4.1 Adjacencies stored in MDS

MDS is based on the full *one-level* representation, in which we store the upward and downward adjacencies between each consecutive pair of dimensions [34]. In particular, it stores region-face, face-edge, and edge-vertex adjacencies (both upward and downward). Despite this focus on the one-level adjacencies, the general MDS structure can be used to store any subset of adjacency relations.

Figure 2.1 illustrates geometrically the relationships stored in a full one-level representation. Each arrow representing an adjacency relation (a.k.a. entity use) is bi-directional to indicate that we store both the downward (high to low dimension) and upward (low to high) relations. Note that vertices are related only to edges, regions are related only to faces, etc.

MDS stores these one-level adjacencies directly and updates them as needed during entity addition and removals, so we can call them the "permanent" adjacencies. In order to fulfill requests for any adjacency information regarding a single mesh entity at runtime, MDS carries out derivations based on the permanent adjacency information as shown in Figure 2.2.

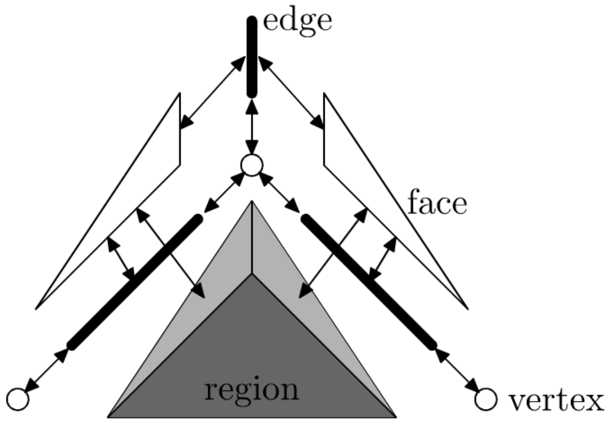

**Figure 2.1: A geometric representation of one-level relations among a subset of the entities that bound a tetrahedron**

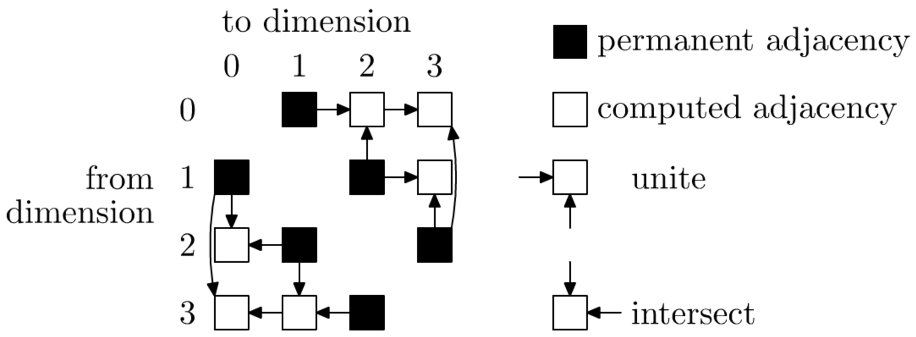

**Figure 2.2: Single-entity adjacency derivations in MDS**

Downward adjacencies not directly stored are derived locally by intersecting existing adjacencies. For example, the vertices of a triangle may be found by querying its edges and then the vertices of those edges, and ordering is preserved by finding a vertex that is adjacent to two particular edges. Such intersection logic moves down one dimension, and can be repeated if needed.

Conversely, upward adjacencies are found by successive unions of existing adjacencies. Given a vertex, one can retrieve adjacent edges, then for each edge retrieve adjacent faces, then take the union of these sets of faces, giving the set of unique faces adjacent to the edge.

These derivations are based on the algorithms described in detail in [1].

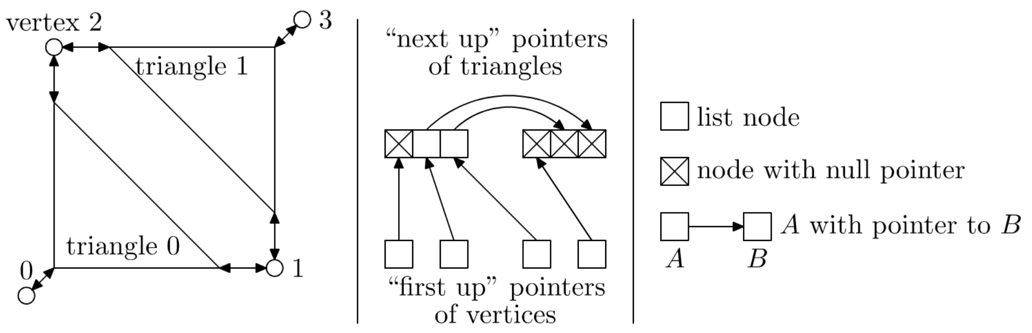

**Figure 2.3: (left) The example mesh: a reduced representation of two triangles sharing one edge and two vertices. (middle) The linked-list storage of upward adjacency information for vertices to triangles. (right) A key describing the meaning of symbols in the middle figure**

### 2.4.2 Object-Oriented Storage

We begin by describing how our structure would look if we stored it in an object-oriented manner. In object-oriented programming, data is organized into *objects* of a few *types*. All objects of the same type have the same *attributes*, and for each object and associated attribute a *value* must be stored. In the present case, the objects are mesh topological entities and the attributes are pointers encoding adjacency relations.

The goal is to organize all data into objects whose sizes are known at compile time based on their type. The first step to doing this is to create separate object types based on topological type. Objects of the same type are all topologically similar to some polytope, for example a triangle or a pyramid. This ensures that their downward adjacency degrees are all the same, which makes downward adjacency storage fairly straightforward. For example, a tetrahedron in a full representation would store four pointers to its bounding faces.

The second step is to use a linked-list technique [2] to distribute upward-adjacency storage amongst the entities of higher dimension, alleviating the need for variable-length storage in lower-dimensional objects. If an object stores $m$ downward pointers, then it also stores $m$ singly-linked list nodes. A singly-linked list node simply contains a pointer to the next node, which can also be null. Every low-dimensional object then stores a single "head" node which begins their upward linked list. If a high-dimensional object ($h$) points to a low-dimensional object ($l$) from its $j$th downward pointer, then its $j$th list node must be part of object $l$'s linked list. Figure 2.3 illustrates this portion of the data structure for a reduced mesh of two triangles. Notice how, for a shared vertex, one begins at the vertex's head node, reaches a node that is stored for triangle 0, then continues to a node stored for triangle 1, which contains a null pointer. This is fundamentally how upward adjacency is queried for a vertex. A singly-linked list can be used as opposed to doubly-linked one because per Section 1.4.3 the number of nodes in this list has a constant upper bound. This means we can afford the $O(m)$ runtime cost of removing a node from this list when a high-dimensional entity is removed ($m$ being the number of upward adjacent entities), in exchange for halving the memory use.

### 2.4.3 Structure of Arrays

Next we describe how we convert the object-oriented structure into an array-based one. Instead of storing all data for a single object contiguously, i.e. organizing by object first and then by attribute, we organize data first by attribute and then by object. We store all data for a single attribute in a contiguous array, hence this scheme can be called a *structure of arrays* [35]. In our case, examples of attribute arrays would be the "first up" nodes for all entities of dimension $d$ and likewise the "next up" nodes for all entities of dimension $(d+1)$. For a static mesh we would be done, but this structure needs to support efficient addition and removal of objects. We therefore need to manage the arrays which store attributes for all objects a single type, to account for addition and removal of objects of that type.

When users request the addition of a new entity, we are allowed to store it anywhere in the structure. In this case adding objects in constant time has a well-known solution called geometric growth [36]. 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 [36]. 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 indices 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 [37]. 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 2.4 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. As described in detail in Section 2.7, this is implemented in PUMI as an operation that

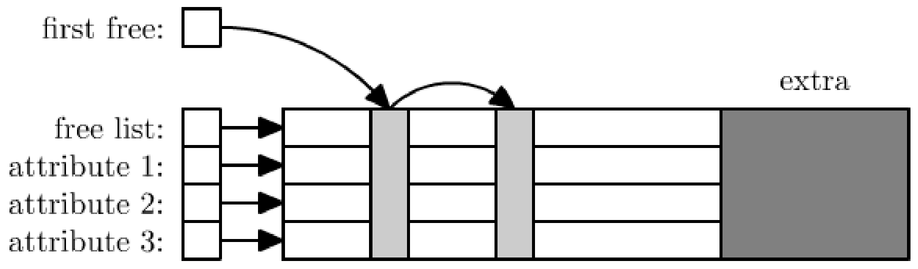

**Figure 2.4: Extra space and hole tracking create modifiable arrays**

combines hole removal with adjacency-based reordering to improve locality.

An important benefit of this array-based structure is the ease with which additional attributes can be added or removed at runtime. To add a new attribute 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. For example, different simulations may require different attributes such as position, velocity, mass, and electrical charge. These can each be allocated as a separate array, and each can be added and removed at runtime as needed with essentially no memory overhead.

### 2.4.4 Lists in Arrays

Since our initial structure composed of many small objects containing pointers to one another was recast into a set of large arrays as opposed to individually allocated objects, pointers to objects get transformed into array indices. As an example, consider the transformation for a singly-linked list as shown in Figure 2.5.

All the list nodes are packed into an array, and a pointer to a list node 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 separate the variables in an object into different arrays.

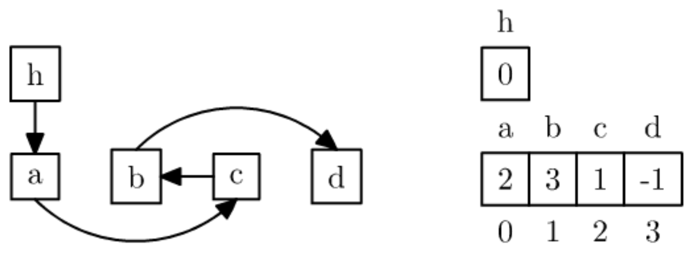

**Figure 2.5: (left) A linked list in object form (right) The same linked list packed into an array**

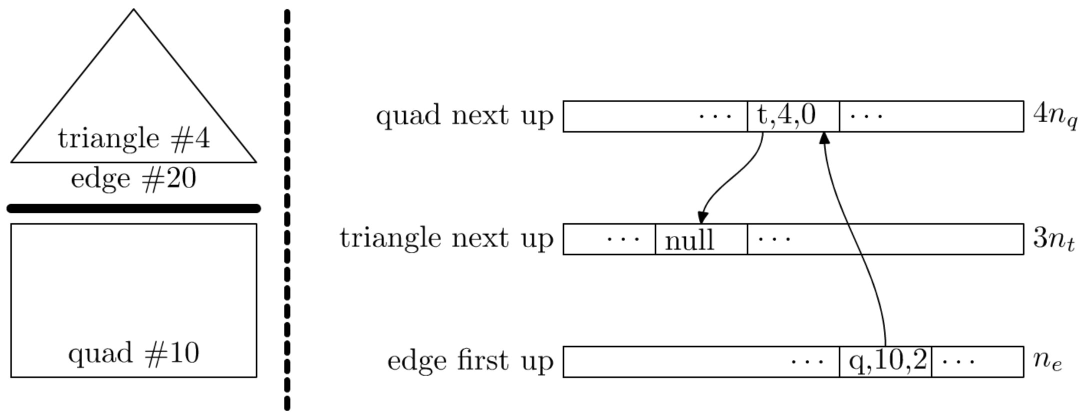

**Figure 2.6: (left) An example subset of a mixed mesh: an edge adjacent to a triangle and a quad (right) The corresponding upward adjacency linkage must traverse separate triangle and quad arrays**

### 2.4.5 Dynamically Modifiable Mesh Structure

There is one more detail that needs resolving in the case of mixed meshes. The problem is that different polytope groups may contain uses of the same entity. For example, an 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 $0 \leq t < T$. This allows the extended index to remain an integer and be decoded using simple modulo and division instructions. Figure 2.6 illustrates this representation. Starting with entry 20 in the "edge first up" array, we are directed to edge use 2 of quad 10. At entry $i = (10 \cdot 4 + 2)$ of the "quad next up" array, we are further directed to edge use 0 of triangle 4. At entry $i = (4 \cdot 3 + 0)$, we find a null pointer and stop.

To get clear picture of all the arrays and their groupings that are allocated for a real mesh, here is a listing including all polytope types that a complex CFD mesh might need when boundary layers are constructed with a mix of wedges and pyramids:

1. Vertices
    (a) Coordinates (size = 3×vertex capacity)
    (b) First edge (size = vertex capacity)
    (c) Free list (size = vertex capacity)
2. Edges
    (a) Vertices used (size = 2×edge capacity)
    (b) Next edge up (size = 2×edge capacity)
    (c) First face up (size = edge capacity)
    (d) Free list (size = edge capacity)
3. Triangles
    (a) Edges used (size = 3×triangle capacity)
    (b) Next face up (size = 3×triangle capacity)
    (c) First region up (size = triangle capacity)
    (d) Free list (size = triangle capacity)
4. Quadrilaterals
    (a) Edges used (size = 4×quadrilateral capacity)

   (b) Next face up (size = 4×quadrilateral capacity)
   (c) First region up (size = quadrilateral capacity)
   (d) Free list (size = quadrilateral capacity)

5. Tetrahedra
   (a) Faces used (size = 4×tetrahedron capacity)
   (b) Next region up (size = 4×tetrahedron capacity)
   (c) Free list (size = tetrahedron capacity)

6. Wedges
   (a) Faces used (size = 5×wedge capacity)
   (b) Next region up (size = 5×wedge capacity)
   (c) Free list (size = wedge capacity)

7. Pyramids
   (a) Faces used (size = 5×pyramid capacity)
   (b) Next region up (size = 5×pyramid capacity)
   (c) Free list (size = pyramid capacity)

Where the capacities refer to the geometric growth capacity described in Section 2.4.3 based on the current number of entities of that polytope group.

This structure provides a straightforward mechanism for associating data with each entity of a polytope group by creating new arrays as described in Section 2.4.3. This is how vertex coordinates and entity-level fields are stored.

## 2.5 Omega_h Data Structure

Omega_h is designed for portable performance across all the different hardware covered in Section 1.6. As such it makes a few key decisions differently from MDS, which affect much of its resulting structure and logic. Apart from the focus on GPU compatibility, when designing Omega_h we often prefer to use more memory as opposed to more compute time when such a trade-off is available. Finally, Omega_h is focused strictly on simplex mesh adaptivity, unlike PUMI which aims also to represent meshes containing prismatic boundary layers [38] or hexahedra [4].

### 2.5.1 Static Mesh

Recall from Section 1.7.4 that for effective performance on GPUs data must be consolidated into a few large arrays. Both MDS and Omega_h do this, and for the latter it is a matter of strict necessity.

However, in MDS, the usage of the structure is clearly in terms of requesting the creation or destruction of a single entity at a time. The clear benefit of this is ease of programming, but the drawback is the inability to parallelize access to a single MDS data structure (the free list, growth mechanism, and upward lists would become points of contention). For PUMI that is not a serious issue, because even with the use of threads there will be at least a few hundred elements per thread in typical PUMI use cases. That means it is acceptable to have one MDS data structure per thread. There are three reasons why this approach does not work on a GPU:

1. The GPU does not maintain persistent per-thread state from one loop to another
2. The GPU has more parallelism, closer to a dozen or even a single mesh element per compute core.

For these reasons, Omega_h takes a different approach: its mesh data structure will efficiently represent a static (unchanging) mesh topology, and the methods in Section 3.4 will construct a whole new mesh structure from the current one, incorporating many changes at once to amortize the cost of reconstruction. This is the main design difference in Omega_h: the mesh structure is a static read-only topological structure.

### 2.5.2 Adjacency Arrays

This static array structure is similar to those of the MOAB mesh database [30], and also of several solvers which implement their own structures. We assume for

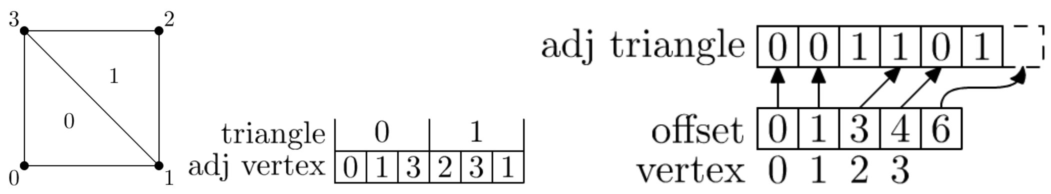

**Figure 2.7: (left) An example 4-vertex 2-triangle mesh (middle) the downward adjacency array (right) The upward adjacency array**

simplicity that either all elements are tetrahedra or they are all triangles (see Section 2.4.5 for the complexity associated with mixed meshes). This allows us to store downward adjacencies in a single array that can be thought of as two-dimensional (see Figure 2.7(middle)). Upward adjacencies (and all complex graphs in Omega_h, for that matter) are stored in the widely known compressed row format, used for example in the METIS graph partitioner interface [39]. This consists of two arrays, one containing the destinations of all graph edges, sorted by the source of the edge, and the other containing offsets indicating where in the first array the edges going out from a given graph node are stored.

### 2.5.3 Alignment Information

Alongside either of these adjacency representations, Omega_h also stores an array of useful alignment information. A high-dimensional entity will use a low-dimensional entity as part of its boundary. However, since that low-dimensional entity is shared with other high-dimensional entities, in general it will not be aligned according to the canonical orders described in the high-dimensional entity's topological template (see Section 1.4.4.1). For example, a face between two regions cannot have its normal point into both regions. Therefore it will be "flipped" with respect to the canonical boundary face of one of these regions.

An alignment array has one 8-bit entry for each entity use, and encodes data about how the entity being used lines up with the entity using it. Although 8 bits is the smallest value that current computers allow one to store, we only use 6 of these bits to store the following three things:

1. Downward ordering (3 bits): what is the canonical order of the low-dimensional

entity in the high-dimensional template.

2. Rotation (2 bits): how many times to rotate the low-dimensional entity before it lines up with the canonical boundary entity of the template.

3. Flip (1 bit): whether the low-dimensional entity also needs to be flipped before it can line up with the canonical boundary entity.

This alignment information, if stored, can speed up a wide variety of algorithms. It is also typically non-trivial to compute, but can be easier to construct at the same time as an adjacency is constructed. For example, downward ordering is easy to construct as one is constructing an upward adjacency based on a downward one. Therefore the adjacency construction algorithms are the optimal location to be computing and storing such information (more on this in Section 2.5.4). Because they occupy only 8 bits per entity use while the existing arrays occupy at least 32 bits per use, this additional information consumes at most 25% more memory, which is affordable.

### 2.5.4 Adjacency Cache

In MDS, if a user requests an adjacency other than those which are stored as per Figure 2.2, it is computed for the single entity the user specifies. As mentioned in Section 2.4.1, this computation may involve complex set intersections or unions, and the upward derivations in particular are typically implemented with heavy use of memory allocation. Per Section 1.7.4, we cannot allow such frequent and small allocations on GPUs. In addition, we should avoid doing a lot of work in a single GPU kernel, so it is preferable to have all the needed adjacency information stored in a convenient manner prior to entering a GPU kernel. Even in the MDS case, the user may request the same non-stored adjacency information multiple times in different contexts, and re-computing this can use significant amounts of time (see Section 2.6.1).

In profiling the execution of Omega_h mesh adaptation, we find the majority of time and memory is consumed by the derivation and storage of mesh adjacencies. To mitigate the runtime cost, our mesh structure acts as an adjacency cache,

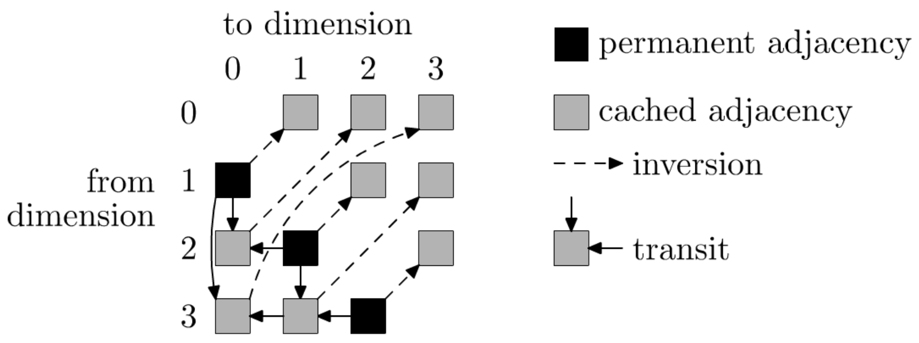

**Figure 2.8: Full-mesh adjacency derivations in Omega_h**

which derives new adjacencies when they are requested and stores them so they are available for later requests. Figure 2.8 shows the adjacencies that we start with and how the others are derived. The permanent adjacencies (those which are provided upon creation of the mesh and uniquely define topology) are the one-level downward adjacencies, i.e. the downward portion of Figure 2.2.

We use two forms of adjacency inference:

1. Inversion: an upward adjacency is derived from is corresponding downward adjacency. This reduces to a low-degree graph inversion problem, which we solve by using atomic operations to associate graph edges with their destination graph nodes. The key algorithm behind adjacency inversion is included in Appendix A.1.

2. Transiting: Similar to the intersection process described in Section 2.4.1 to derive downward adjacencies, we can combine the downward adjacencies from dimension $p$ to dimension $(q+1)$ and from $(q+1)$ to $q$ into a single adjacency from $p$ to $q$. By using the alignment information stored in each downward adjacency, we can avoid having to do any kind of set intersection, because we can map the $(q+1)$-dimensional entities onto the $p$-dimensional template and the template has all adjacency information stored *a priori*.

Note that the set of permanent adjacencies in Omega_h is not technically complete, because adjacencies cannot be efficiently derived for single entities at a time. However, because Omega_h does derivations on a full-mesh basis, it can use the

**Table 2.1: MDS adjacency timings (in milliseconds) for 100K tet mesh**

| dimension | entity count |
|---|---|
| 0 | 19468 |
| 1 | 129010 |
| 2 | 212846 |
| 3 | 103303 |

| | | to dimension | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 |
| from dimension | 0 | | 3.9 | 48.7 | 88.9 |
| | 1 | 2.4 | | 11.9 | 40.4 |
| | 2 | 16.2 | 4.2 | | 9.0 |
| | 3 | 21.1 | 13.1 | 2.2 | |

inversion algorithm to compute all upward relations based on all downward relations in $O(n)$ time, whereas there is no $O(1)$ equivalent operation for a single entity. Thus Omega_h obeys a relaxed definition of completeness that requires $O(n)$ full-mesh derivation operations as opposed to $O(1)$ single-entity derivation operations.

## 2.6 Data Structure Performance

### 2.6.1 Adjacency Query Performance

To substantiate our claims that MDS supports $O(1)$ adjacency queries and give an example of its typical query performance, we create two uniform meshes at different resolutions, having approximately 100K and 200K elements, respectively. We then time, for each pair of dimensions, how long it takes to traverse all entities of the first dimension and query their adjacent entities of the second dimension. The time required to do this should be approximately the number of entities of the first dimension times the constant time required for one adjacency query. Tables 2.1 and 2.2 list these timing results for both meshes, and show fairly clearly that all times grow by about 2X since the number of elements is about 2X larger in the second mesh. These timings were collected on a workstation with an Intel Xeon E5-2620v4 CPU.

Performing a corresponding study for Omega_h is not quite trivial. Because Omega_h uses an adjacency cache, the runtime cost of querying a particular adjacency depends on which adjacencies are already cached at the moment. In the absolute best case, that adjacency is already cached and need only be read directly from memory. This best case scenario is illustrated in Table 2.3, showing the time

Table 2.2: MDS adjacency timings (in milliseconds) for 200K tet mesh

| dimension | entity count |
|---|---|
| 0 | 39696 |
| 1 | 265267 |
| 2 | 439107 |
| 3 | 213535 |

| | | to dimension | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 |
| from dimension | 0 | | 8.7 | 104.8 | 191.5 |
| | 1 | 5.0 | | 26.3 | 84.7 |
| | 2 | 33.4 | 8.7 | | 20.3 |
| | 3 | 45.6 | 27.8 | 4.5 | |

Table 2.3: Omega_h traversal timings (in milliseconds) for 100K tet mesh

| | | to dimension | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 |
| from dimension | 0 | | 0.2 | 0.2 | 0.2 |
| | 1 | 1.7 | | 1.4 | 1.4 |
| | 2 | 2.6 | 2.7 | | 2.3 |
| | 3 | 1.2 | 1.2 | 1.3 | |

required to simply read and traverse cached adjacency information from memory, for all entities of a given dimension, using the same 100K mesh as Table 2.1. This data is presented as a lower bound for traversal cost, given that Omega_h has optimal structures for fast traversal and reading. Table 2.4 presents the more useful incremental construction times, i.e. the time required to construct an adjacency (for all entities of the relevant dimension) assuming any pre-requisite adjacencies used in the construction have already been cached. Values from Table 2.4 can be used in conjunction with the diagram in Figure 2.8 to determine Omega_h performance for a particular series of queries.

## 2.7 Reordering

Array-based structures have a natural order of traversal which is to traverse each array from start to finish. In Omega_h, all *d*-dimensional simplices are in one traversal order, and in MDS there is a traversal order for each polytope topology. Typically, mesh operations access multiple arrays but are based on the natural ordering of a single array. For example, a traversal based on elements may access

**Table 2.4: Omega_h incremental construction timings (in milliseconds) for 100K tet mesh**

| | | to dimension | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 |
| from | 0 | | 7.6 | 31.1 | 16.1 |
| dimension | 1 | | | 11.1 | 9.7 |
| | 2 | 5.5 | | | 4.7 |
| | 3 | 3.2 | 9.1 | | |

the vertices of each element. Recall from Section 1.5 that accessing data in the order it is stored is key to achieving performance on most cache architectures. Thus, we would like to establish orderings of each group (e.g. the elements) such that accesses to other groups (e.g. the vertices) are made in an approximately linear fashion. Since these accesses are based on adjacency, what we aim to do is order each group such that locality in the mesh (defined by graph distances) is well correlated to locality in the array (defined by indices). We further simplify the problem by focusing on ordering the vertices, and define orderings for other groups based on their adjacencies to vertices and the vertex ordering.

We interpret the mesh vertices as graph nodes and the mesh edges as graph edges, which enables us to use the linear arrangement measure from graph theory [40] as our metric of locality:

$$\text{la}(G = (V, E)) = \sum_{(i,j)\in E} |i - j| \tag{2.1}$$

Where $i$ and $j$ are the array indices of two graph nodes. There is additional rationale for using this metric in the case of finite element assembly for linear elements, which traverses mesh vertices and accesses adjacent vertices across mesh edges.

Finding an ordering of the vertices $V$ such that Equation 2.1 is minimized has been shown to be NP-complete for general graphs [40]. However, our mesh-based graphs have at least two useful properties: the degree of each vertex is bound by a small constant, and they are equipped with an embedding of vertices into $\mathbb{R}^3$ such that Euclidean distance closely reflects graph distance. As such, several heuristics for good linear arrangements have been developed for meshes. The Cuthill-McKee algorithm [41], developed to minimize the storage complexity of sparse matrices derived from meshes, is based on Breadth-First Search (BFS) of the graph and sorts each layer based on the degrees of the vertices. A similar algorithm by Zhou et al. [42] based on work by Beall and Shephard [34] also reduces to BFS for a linear mesh, and has special considerations for the starting point of BFS. Other (geometric) methods take advantage of the vertex coordinates, for example by arranging vertices along the Hilbert Space-Filling Curve (HSFC) [43]. There are also libraries implementing hierarchic methods of graph reordering, such as METIS [39]. The most effective methods at minimizing the linear arrangement metric are those which begin with one of the above methods and then apply a form of simulated annealing in which local modifications are made to the ordering that locally improve the metric [44]. However, for our purposes of cache locality it is not worth spending a large amount of computational power to obtain a slightly better ordering, so we limit ourselves to BFS-like algorithms and space filling curve algorithms, and we do not consider simulated annealing.

Table 2.5 shows the values of Equation 2.1 using several of these methods on three different meshes: a cube meshed with tetrahedra (1254 vertices), a fusion reactor cross-section meshed with triangles (3695 vertices), and a circuit solder geometry meshed with tetrahedra (2601 vertices). The “input” row represents the mesh’s linear arrangement value prior to reordering. Parentheses indicate the starting node used for a search procedure, for example BFS(0) uses the first input node as the starting point while BFS($n-1$) uses the last input node. For the cube, BFS(center) uses the node closest to the cube’s center. Our two main conclusions based on Table 2.5 are that BFS and HSFC give the best (smallest) linear arrangement values, and that the BFS result is heavily dependent on the starting point.

PUMI implements a BFS method with heuristics that use geometric classification for selecting a starting node [34, 42]. One of the difficulties with the BFS method is that it is difficult to parallelize, since layers need to be constructed sequentially. Early versions of Omega_h implemented a BFS method that was not

**Table 2.5: Linear arrangement values for different meshes and heuristics**

| | cube | fusion | solder |
|---|---|---|---|
| input | 3163141 | 1743275 | 8688982 |
| BFS(0) | 678166 | 910451 | 2085725 |
| BFS($n-1$) | 675903 | 471728 | 2098239 |
| BFS(center) | 1106864 | N/A | N/A |
| HSFC | 597009 | 627726 | 1877633 |
| Cuthill-McKee(0) | 822050 | 956364 | 2161626 |
| Cuthill-McKee($n-1$) | 784566 | 501883 | 2198386 |
| METIS | 1120368 | 1234170 | 2849735 |

**Table 2.6: Locality speedups including and excluding BFS reordering time**

| Platform | overall | excl. reorder |
|---|---|---|
| IBM BG/Q 1 thread | 3% | 7% |
| IBM BG/Q 16 threads | 10% | 15% |
| IBM BG/Q 64 threads | 5% | 11% |
| NVidia K40 | 2% | 8% |
| Intel Phi 1 thread | 26% | 30% |
| Intel Phi 120 threads | 24% | 29% |
| Intel Phi 240 threads | 9% | 19% |
| Intel i7 1 thread | 0% | 4% |
| Intel i7 4 threads | 11% | 16% |
| Intel i7 8 threads | 5% | 10% |

parallelized for this reason, but we did study its effects on subsequent adaptation operations on different hardware. Table 2.6 shows the speedup in Omega_h adaptation (see Section 3.4) when a reordering is carried out prior to each adaptation call. Note that even though this reordering was not shared-memory parallel, there was never a decrease in performance, and in the most extreme case reordering made a 26-30% difference. This result suggests that reordering is quite beneficial and should be incorporated into array-based adaptation workflows.

Despite non-parallel BFS reordering being adequate, we prefer a reordering method that takes advantage of shared-memory parallelism. The current Omega_h code implements a Hilbert curve method (HSFC). The Hilbert curve is a numbering

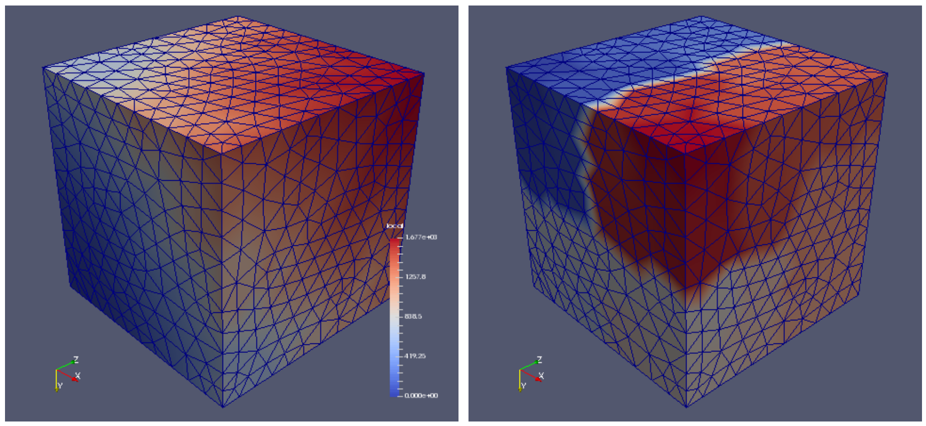

**Figure 2.9: Meshes colored by vertex ordering: (left) BFS, (right) Hilbert curve**

of the cubes of a regular cubic grid. Since our vertex coordinates are in a floating-point format with 52 binary digits of precision (see Section 4.9.2), we place our mesh in an implicit cubic grid that has $2^{52}$ cubes along each axis of the mesh's bounding box. This grid never has to be represented in its entirety; there is a constant-time algorithm for converting a coordinate in this grid to its integer position in the Hilbert curve [45]. We map each floating-point vertex coordinate to the cube it lies in, and assign that vertex the Hilbert curve position of that cube. We then sort vertices by their Hilbert curve positions. Although our resolution makes it very unlikely that two vertices fall in the same cube, the sorting procedure would arbitrarily order them if this did occur. More importantly, we can use efficient sorting algorithms for shared memory parallelism as described in Section 4.2.4, and the conversion of coordinates to Hilbert curve positions scales linearly (see Section 4.2.1), so this reordering method is more readily parallelizable than the BFS methods.

Figure 2.9 shows a cube mesh with its vertices colored by two different reordering methods: the BFS method implemented in PUMI, and the HSFC method implemented in Omega_h. Blue indicates a low vertex index while red indicates a high index. Note that while the HSFC method has a clear interface along which indices differ highly, it should have better locality among neighbors in the smooth areas because in the BFS method most vertices are adjacent to vertices in the previous or subsequent layer, and those differences are on average the size of one layer (the surface area of a cut through the cube, measured in exposed vertices).

# CHAPTER 3
# CAVITY-BASED CONFORMAL MESH ADAPTATION

## 3.1 In Context

The mesh adaptation methods in this work are conformal, general, and cavity-based. In the following Sections we define these terms in contrast to other techniques for modifying meshes.

### 3.1.1 Conformal and General

They are conformal in the sense that the boundaries of all elements (Section 1.4.1) are composed of the set of entities expected by that element's topological template (Section 1.4.4.1). In other words, we avoid the "hanging node" scenarios introduced by non-conformal mesh modification techniques. Typically, non-conformal mesh modification also restricts itself to subdividing input elements into more elements, or undoing such subdivisions which were done before. Figure 3.1 illustrates such a method, and clearly shows the hanging nodes introduced.

Non-conforming meshes require additional support from the PDE-solving code to deal with hanging nodes, and typically no more than one level of refinement

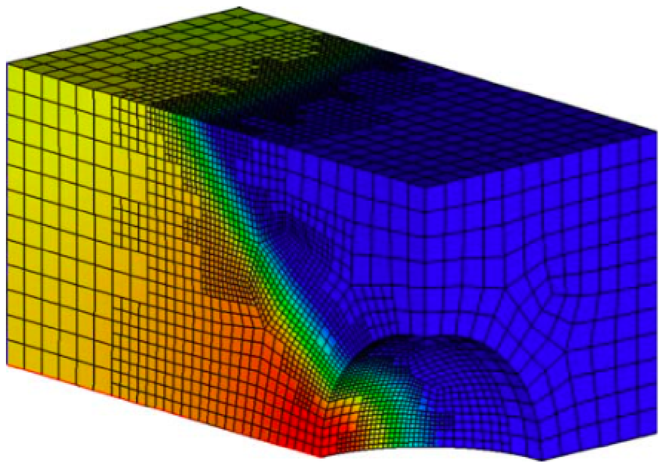

**Figure 3.1: Non-conforming parent-child adaptive mesh refinement [46]**

is allowed between adjacent elements. The more important limitation is due to non-conforming methods typically being parent-child methods, which fundamentally limits them to the topology (and geometry) of the coarse input mesh. If this input mesh is more fine than necessary in some areas, it cannot be coarsened. If moving objects or object deformation cause input elements to become highly compressed or even inverted, parent-child refinement can never correct or prevent this. New node placement when refining along a curved boundaries poses a similar issue to that of physical deformation. For these reasons we take a general approach, employing local cavity-based mesh modification operations which are able to coarsen beyond the input mesh and correct low-quality elements in the input mesh.

### 3.1.2 Cavity-Based

We restrict ourselves to local cavity operations, meaning that the transformation from input to output meshes can be expressed as a series of cavity modifications, each of which can in turn be expressed as the removal of a small number of mesh entities followed by the addition of a small number of mesh entities.

In general, a cavity can be defined as a manifold sub-domain of the mesh defined by a set of elements from the original input mesh. A cavity undergoes a modification, which changes the discretization (set of mesh entities) of its interior but leaves its "boundary" unchanged. In order to allow changing the discretization of the overall mesh boundary (i.e. the geometric model boundary) we use a relaxed definition of a cavity boundary (shown in Figure 3.2) which only includes entities that are also adjacent to elements outside the cavity. This is the minimum requirement for the method to be conformal: it must preserve the topology of this relaxed boundary. For efficient parallel execution, we also require that attributes of those boundary entities also remain unchanged. This includes geometric classification, as discussed in Section 3.3.2 and illustrated in Figure 3.8. This allows two cavities to be modified simultaneously by two threads or processes, so long as the cavities do not overlap (i.e. they do not share elements). They may be adjacent (share a subset of their boundaries) and need not coordinate because nothing about the shared boundary will be changed by either one.

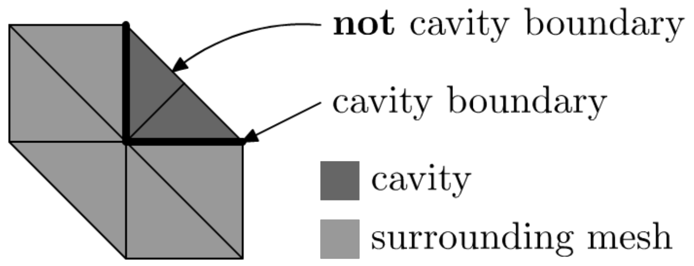

**Figure 3.2: Relaxed definition of cavity boundary excludes geometric boundary**

Some benefits of using local cavity operations are:

1. It allows more straightforward and reliable parallelization of mesh adaptation (see Section 4.8).

2. It allows much more careful control of the effects that mesh adaptation has on the simulation fields attached to the mesh.

On the other hand, the set of known cavity operations have been found by the trial and error of researchers, and there are many properties which they are not guaranteed to achieve. The most successful set of cavity operators are those which operate on simplex meshes, due ultimately to the fact that a simplex is the simplest polytope of a given dimension and that, conversely, more complex polytopes provide fewer valid configurations. In our work, we separate cavity operators into several categories:

1. Refinement: create a strictly more detailed discretization than the input. Guaranteed not to invert elements, but not to preserve any element quality. Guaranteed to exactly preserve the distribution of fields.

2. Coarsening: create a strictly less detailed discretization than the input. No guarantees it can be done without reducing or negating quality, so it must be checked. By definition, cannot exactly preserve the distribution of the fields.

3. Shape correction: typically maintains similar level of detail, modifies connectivity to improve minimum element quality. There is no known method

guaranteed to raise all elements to a quality that is useful for simulation and adaptation, but heuristic methods can achieve great results in practice [47].

4. Snapping: Many simulations are based on a geometric model, usually a CAD boundary representation in which the shape of an object is defined by parametric descriptions of curves and surfaces. Other boundary representations which are separate from the mesh may be used. In these cases, mesh adaptation should modify the boundary of the mesh to be as similar as possible to the geometric (CAD) boundary. We refer to this as "snapping", because for interpolating basis functions it involves snapping mesh nodes onto the geometric boundary.

## 3.2 Related Work

There have been several iterations of the MeshAdapt library developed at RPI. One of the earliest publications by De Cougny and Shephard [48] outlines the three basic steps and goes into some detail on a use of independent sets for coarsening purposes (an idea that we extend significantly in Section 4.8.2). Later, substantial extensions were introduced by Li on anisotropy using the metric tensor and the selection of operators for shape correction [8, 49]. Our implementations of tetrahedral edge swaps make use of guidance on fast implementation by Olivier-Gooch [50]. Researchers at INRIA have provided useful mathematical foundations for handling the anisotropic metric tensor field [6, 51, 52], and work at the Catholic University of Louvain explored the use of mesh adaptation to respond to moving objects [7], a path we continue with our Omega_h work.

## 3.3 MeshAdapt Methods

### 3.3.1 Template Refinement

The MeshAdapt library uses edge-based refinement templates for its refinement step. The way these work is that all edges whose metric length exceeds some threshold $l_{\text{up}} > 1$ are marked for refinement. Then each element takes into account the subset of its edges which are marked for refinement, and chooses one of many

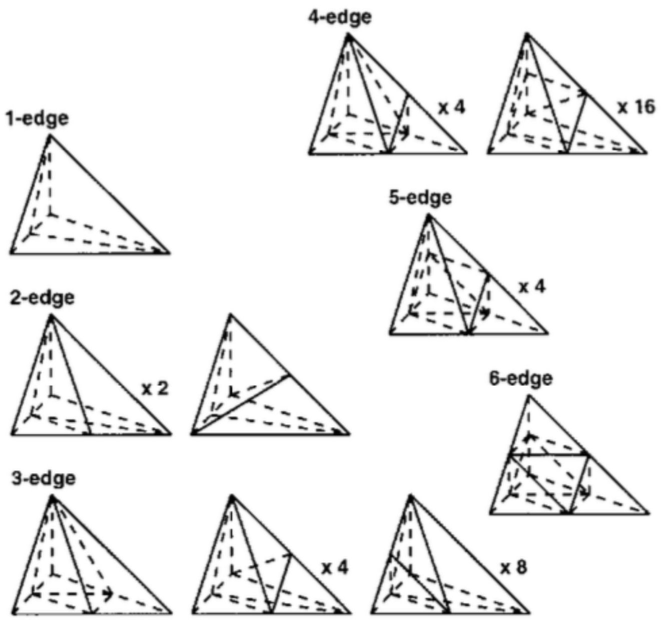

**Figure 3.3: Tetrahedral refinement templates**

possible subdivision patterns (refinement templates) based on this subset of edges. Figure 3.3 illustrates these templates in the case of tetrahedra. In fact, the center template shown for three marked edges has two variants which are symmetric by reflection but not by rotation. In total, this means there are 12 rotationally unique tetrahedron refinement templates.

One benefit of the use of refinement templates is that adjacent elements can be refined simultaneously, so all edges, faces, and regions of the mesh can be modified in a nearly embarrassingly parallel fashion once the set of marked edges is identified. Another benefit is that the gradation of the mesh is more explicitly controlled compared to methods which split edges independently. However, refinement templates have some drawbacks as well:

1. In some cases, a subset of the template is a polyhedron that cannot be subdivided into tetrahedra without introducing an extra vertex within the parent tetrahedron. In particular, Schönhardt's polyhedron can appear (see Figure 3.4). This reduces the predictability of refinement and makes it more difficult to transfer solution.

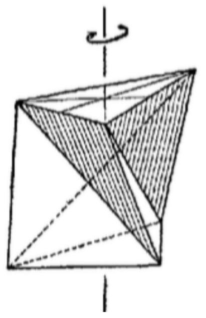

**Figure 3.4: Schönhardt's irreducible polyhedron [53]**

2. Other cases introduce a geometric decision, such as the case when all edges of a tetrahedron are refined, or even when two edges of a triangle are refined. This also reduces predictability.

3. It takes substantial code to implement all rotationally unique combinations for all the relevant element polytopes. This increases the likelihood of errors and decreases the productivity of modifying any aspect of refinement.

4. Due to the simultaneous nature of the operation and the difficulty of predicting the outcome, it is prohibitively difficult to reject a local portion of the refinement based on criteria such as new elements having too low quality.

### 3.3.2 Coarsening

Like other adaptation libraries, MeshAdapt implements coarsening via edge collapses. Figure 3.5 shows a typical edge collapse in a tetrahedral mesh for reference. One vertex in the mesh is "moved" onto another vertex which is adjacent via a mesh edge, collapsing this edge and all its adjacent faces and regions. For programming purposes, when dealing with sets of entities we can call those being removed the "collapsing" set and those being conceptually elongated to fill the cavity as the set to "keep". In practice all old entities are removed and the set of entities to keep is rebuilt with modified connectivity (where they were adjacent to the collapsed vertex, now they are adjacent to the kept vertex).

Edge collapsing is an operation which, if not properly controlled, can invalidate the mesh topologically or undo its topological similarity to a CAD model [3].

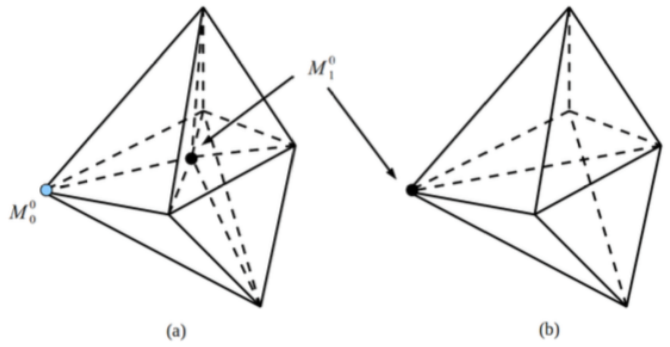

**Figure 3.5: Edge collapse in tetrahedral mesh [54]**

MeshAdapt uses a set of checks which is somewhat expensive to compute but is topologically robust:

1. The vertex being collapsed must have the same classification as the edge being collapsed. This preserves similarity to the CAD model by preventing collapses from the boundary into the interior.

2. If there exists a ring of three edges including the collapsing edge (Figure 3.6), then those three edges must bound a single triangle in the mesh. This prevents collapsing an empty hole in the mesh to zero volume as in Figure 3.7. While some [34] consider Figure 3.7 a valid operation, current mesh data structures assume an entity is uniquely defined by its set of bounding entities, which precludes the two overlapping mesh edges that Figure 3.7 produces. In addition, the non-collapsing edge adjacent to the collapsing vertex must have the same classification as the triangle. This prevents collapsing a cavity on a curved boundary down to zero volume (see Figure 3.8), which would require re-classifying entities on the cavity boundary (which breaks parallelism guarantees), and creates no elements in the new cavity (which is problematic for solution transfer).

3. Analogous to the edge ring check, in 3D we check for two triangles which share a non-collapsing edge and whose remaining two vertices are the endpoints of the collapsing edge (a "face ring"). The two triangles and the collapsing edge

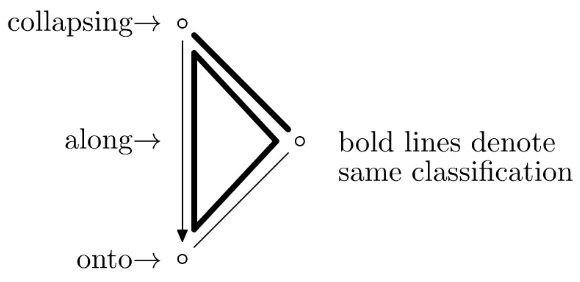

**Figure 3.6: Edge ring condition check during edge collapse**

must bound a single tetrahedron, and the triangle adjacent to the collapsing vertex must have the same classification as this tetrahedron.

4. All the resulting tetrahedra must have positive volume. In the 2D planar case, triangle normals should all be positive in the $Z$ axis. For straight-sided simplices in an equal-dimensional space (i.e. triangles in 2D and tetrahedra in 3D), this check on its own can prevent some topological invalidities, except those in Figure 3.7 and Figure 3.8.

The edge and face ring conditions are more complete versions of those described by Garimella [55]. Most of the expense of checking these conditions is in the search procedure to identify edge rings and face rings, as the naive approach costs $O(n^2)$ comparisons where $n$ is the number of edges (or faces) adjacent to one vertex. For typical tetrahedral meshes there may be 36 or more faces adjacent to a vertex, meaning close to 1000 comparisons would be needed. We can reduce the cost by using a set structure to store the $n$ vertices (or edges) that may complete a ring from one endpoint, and use its $O(\log(n))$ membership check capability to reduce the comparison cost to $O(n \log(n))$.

Finally, when applying a series of edge collapses to a mesh, one must additionally filter the set of edges targeted for collapse until it forms an independent set, in order to avoid a pathological sequence of edge collapses reducing large portions of the mesh down to a single edge. We resolve this by visiting vertices which are marked for collapse and unmarking them so long as each edge that needs collapsing still keeps one adjacent marked vertex. Similar methods of establishing an independent set were in use in early versions of MeshAdapt [48] and continue to be used by

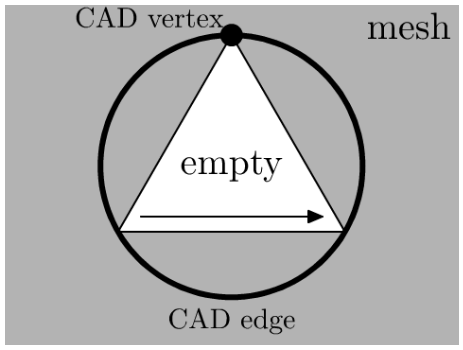

**Figure 3.7: Illegal collapse of a CAD hole represented by a periodic boundary**

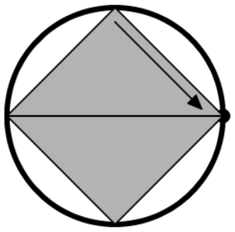

**Figure 3.8: Illegal collapse with no new elements and re-classification**

other adaptation codes [56].

### 3.3.3 Shape Correction

Shape correction, otherwise known in the literature as “sliver removal”, is one of the most important open problems in tetrahedral meshing. When meshing a planar domain with triangles, techniques such as Delaunay refinement can guarantee triangles of a certain quality, where quality can be measured by angles at corners or the mean ratio (see Appendix B.1.1 for the relationship between these measures). However, when meshing a volumetric domain with tetrahedra, there are no known methods to provably guarantee element quality (dihedral angles or the mean ratio). What is typically done is to carry out a method which satisfies edge length criteria and at a minimum produces positive-volume tetrahedra. Following this, heuristic methods (sliver removal) are carried out to attempt to remove tetrahedra with quality below a constant user-defined threshold.

Despite these methods being heuristic, their combination can often give very good results in practice. The most aggressive approaches [47, 57] can usually bring tetrahedral dihedral angles into the range $[30°, 130°]$, which is sufficient for most simulation purposes. However, these aggressive approaches may be either too expensive for use in mesh adaptation (which is executed within a performance-critical simulation), or they may modify the mesh too much (a large movement of all the nodes in the mesh would change the physical distribution of the fields defined by values at the nodes).

Mesh adaptation has the advantage of being provided with an input mesh, and therefore one can in theory reject any operation which would decrease the quality of the mesh lower than it was on input. In this sense, mesh adaptation can guarantee quality at least as good as the input. We will see this approach taken to some degree in the Omega_h methods described in Section 3.4. The danger of rejecting operations based on quality is that if too many operations are rejected then edge lengths will not conform well enough to the metric field. However, we are having increasing success with careful rejection of operations and some researchers are able to avoid using sliver removal techniques altogether if their quality requirements are low enough [56, 58].

Typically, mesh adaptation libraries implement a balance between prevention of quality degradation and repairing quality that has been degraded. The repair process uses a subset of the known sliver removal techniques, with a focus on being able to repair the average sliver using minimal computational resources, while resorting to more expensive techniques when the cheaper ones fail.

Li describes a fairly comprehensive set of sliver removal heuristics implemented in an earlier version of MeshAdapt [49]. The current version uses a similar approach, based on a taxonomy of sliver tetrahedra. The taxonomy can be described in terms of some boundary entity being too close to another boundary entity of the tetrahedron:

1. In the first case, we have two vertices being too close, which is really just an edge being too short. If quality is low in metric space, this is a case of an edge which should have collapsed but didn't. We attempt to more aggressively collapse it by trying to collapse each edge adjacent to either endpoint vertex.

2. In the second case, we have a vertex being too close to its opposing triangle

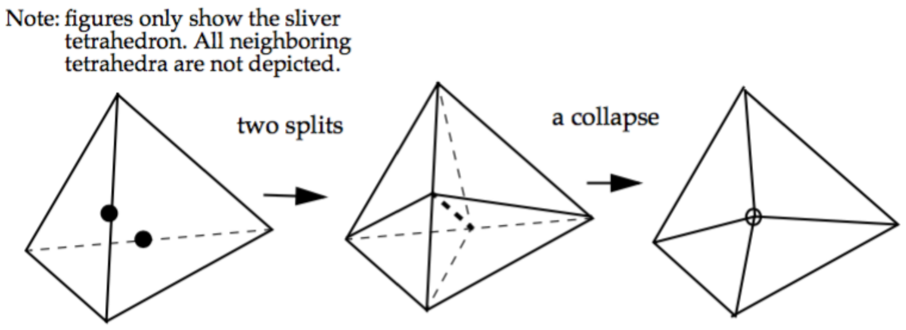
Note: figures only show the sliver tetrahedron. All neighboring tetrahedra are not depicted.
two splits
a collapse

**Figure 3.9: Double-split + collapse compound operator [49]**

face. Here we try to execute an edge swap (see Figure 3.10) on each of the three edges bounding the opposing triangle face. The work of Li suggests that a face split and collapse operation should also be implemented as future work [49].

3. In the third case, we have the classic sliver tetrahedron which has two opposing edges being too close to one another. We first attempt to perform an edge swap on each of the two opposing edges. If that fails, we attempt a double split collapse operation as shown in Figure 3.9.

Each of the above attempted operations is judged in terms of quality. If the minimum quality of any element in the cavity after a modification exceeds minimum quality of any element in the cavity before the modification, then we say the quality has locally improved. In most cases, one of the above attempts succeeds. However, it is possible in practice for none to succeed, in which case MeshAdapt will leave the sliver as-is.

#### 3.3.3.1 Edge Swap

The edge swap operation deserves a detailed consideration due to its complexity. Its goal is to remove the elements adjacent to an edge and replace them with a new set of elements such that the edge is not recreated. In the tetrahedral case, this essentially reduces to the problem of forming an output triangular mesh based on the "ring" vertices (those vertices which are opposite to the central edge across an input triangle), as Figure 3.10 illustrates. Furthermore, since this operation is used

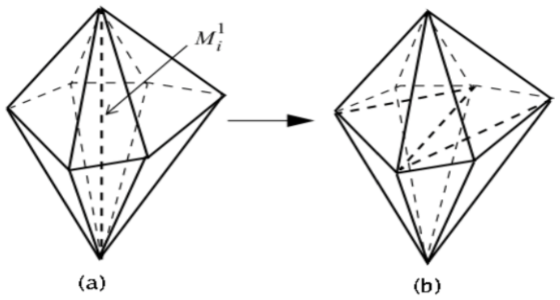

**Figure 3.10: Edge swap in tetrahedral mesh [54]**

almost exclusively for the purpose of repairing quality degradation (sliver removal), what we are interested in is finding the triangular mesh of the ring such that the resulting tetrahedral mesh of the cavity contains elements above a certain quality. The quality to beat is usually the lowest quality input tetrahedron, i.e. we want to at least improve the minimum quality.

A naive search of the space of possible triangular meshes can be quite expensive, so we use an optimized algorithm described by Freitag and Ollivier-Gooch [50]. The algorithm consists of limiting oneself to rings of a certain number of vertices (in our case, not attempting rings that have more than 7 vertices), and storing integer tables describing all possible trianglulations of such rings. The triangulations are described in terms of unique triangles (two triangulations may share the same triangle). Because the qualities of the two adjacent tetrahedra adjacent to such a triangle are uniquely defined, then if either quality is below the target threshold, all triangulations that include that triangle can be ignored. Otherwise, the lower of the two qualities is stored in association with the unique triangle, to avoid later re-computation.

### 3.3.4 Overall Steps

Listing 3.1 shows an abbreviated version of the C++ code for MeshAdapt's main function. The central loop executes a fixed number of iterations. At each iteration, we first coarsen as described in Section 3.3.2, followed by refinement as described in Section 3.3.1. We perform coarsening before refinement because this should reduce the peak memory usage between the two operations. Shape correction

**Listing 3.1: MeshAdapt main function**

```
void adapt(Input* in) {
  Adapt* a = new Adapt(in);
  preBalance(a);
  for (int i = 0; i < in->maximumIterations; ++i) {
    coarsen(a);
    midBalance(a);
    refine(a);
  }
  fixElementShapes(a);
  postBalance(a);
  delete a;
  delete in;
}
```

is performed after the main loop has completed, using the algorithm in Section 3.3.3. The functions whose names end in `Balance` execute parallel load balancing. We treat mesh elements as the units being balanced, and assign weights to them based on the metric space volume defined by Equation 1.5. Either graph partitioners [59] or diffusive partitioning methods [60] may be used at each step.

## 3.4 Omega_h Methods

The Omega_h code aims to provide MeshAdapt-like functionality on a wide variety of computing hardware. Due to the restrictions introduced by shared memory parallelism, Omega_h initially chose a simpler set of algorithms to reduce the cost of redesigning each algorithm for portability.

There are several key characteristics of MeshAdapt's design which need to be changed in order to execute efficiently with high degrees of shared memory parallelism (e.g. on GPUs):

1. The assumption of a persistent thread which can maintain substantial amounts of local data (e.g. a data structure representing a mesh partition) and act on that data in serial is invalid for GPUs, because the increased degree of parallelism requires lowering the per-thread overhead in memory use and runtime. Shared memory parallelism is better expressed at the loop level (see Section

1.7.3), and per-thread data should be avoided. This means data storage concerns (data structures) move up to the process level (in terms of software) and up to the node level (in terms of hardware).

2. Modification of the mesh cannot be based on a local *series* of single entity additions and removals that are immediately applied to the mesh, because this introduces increasing contention on the mesh data structure as the number of threads increase. Most other research in shared-memory parallelization of mesh modification has focused either on methods which assume the degree of parallelism is somewhat small [61] (as it is in CPUs) or methods which use entity-level non-deterministic mutual exclusion mechanisms [62]. We instead propose a deterministic independent set selection system in Section 4.8.2 that allows us to go from one simple static structure to another (the data structure is described in Section 2.5.1).

3. The operations that are performed in parallel should avoid making use of dynamic local memory allocation (see Section 1.7.4). For example, when threads need access to variable-length adjacency data such as upward adjacencies, it is better for them to cooperate and form a single data structure representing said information for all entities, instead of having each thread recompute its portion on demand during a higher-level operation.

Altogether, this results in the need to write the entire program in an array-based and loop-based style and requires algorithms that are intuitively expressed in that style.

We have designed such alternative algorithms for performing mesh adaptation, and so far they have demonstrated comparable and in some cases higher quality results for simplex mesh adaptation. Additional efforts are still underway to examine algorithmic details, and additional test cases are being developed in collaboration with other research groups to better qualify mesh adaptation methods, independent of parallelization.

The following sections describe the algorithms implemented in Omega_h in contrast to those of MeshAdapt.

### 3.4.1 Refinement

Instead of using template-based refinement, Omega_h uses single edge splits, meaning a single edge is bisected and adjacent elements are bisected at a single operation. Unlike the simultaneous execution that is possible with template-based refinement, only edge splits whose cavities do not overlap can be applied simultaneously (see Section 4.8.2 for how such splits are selected). Several researchers have similarly had success using only single edge splits rather than template-based refinement [7,56,58]. Like MeshAdapt, Omega_h marks all edges whose metric length exceeds some threshold (typically $\sqrt{2}$) as candidates for edge splits.

Omega_h uses a more accurate formula to measure edges when determining whether to split or collapse them. This formula, shown as Equation 3.1 is suggested by Loseille and Löhner [58] and is based on the assumption that the desired length varies geometrically along the edge ($h(t) = h_a^{1-t} h_b^t$ where $h_a$ and $h_b$ are the desired lengths at the endpoints). This assumption is consistent with the Log-Euclidean interpolation method we use for the metric tensor (see Section 3.5.1).

$$\tilde{l} = \frac{\tilde{l}_a - \tilde{l}_b}{\log\left(\tilde{l}_a / \tilde{l}_b\right)} \quad \tilde{l}_a = \frac{l}{h_a}, \quad \tilde{l}_b = \frac{l}{h_b} \tag{3.1}$$

Omega_h then unmarks any edges whose splitting would produce elements of low mean ratio quality (typically less than 20%). There is a trade-off here between two desired outcomes. On the one hand, when mesh adaptation is used to control discretization error, it is preferable to produce meshes of higher than desired resolution as opposed to lower than desired (in many cases the latter is unacceptable). Combined with the fact that refinement cannot technically produce inverted elements or invalid meshes, this is a strong motivation to immediately and unconditionally refine all edges that exceed a given threshold. On the other hand, immediate refinement can produce elements of arbitrarily bad quality, making it increasingly difficult to apply future mesh modifications and driving numerical conditioning of the physical simulation to unwanted extremes. Since computers use finite precision arithmetic, there are cases in practice where the refined element volumes are indistinguishable from zero to the computer. In Omega_h, we choose to avoid immediate refinement that would violate quality criteria under the assumption that, in most cases, subsequent operations (including other edge splits) will improve the situation sufficiently that the deferred refinement will eventually be accepted (see the loop in Listing 3.2). This is especially important when using single edge splits instead of applying refinement templates, because the order in which edges are split strongly affects the resulting quality. We have confirmed that at the end of adaptation we do meet the desired upper bound on metric edge length in the cases considered to date.

### 3.4.2 Coarsening

Like MeshAdapt, Omega_h uses single edge collapses for mesh coarsening. Since all operations in Omega_h use the quality-driven independent set system described in Section 4.8.2, that explicitly takes care of the need for an independent set of collapsing vertices. Edge collapses have some confusion in what their “key” entity is, because in reality it is the combination of an edge and one of its endpoint vertices that defines a collapse. We first mark all edges whose metric length is beneath some threshold (typically $1/\sqrt{2}$) as candidates to collapse via either endpoint. In addition, Omega_h will mark all edges adjacent to either endpoint of a short edge as additional candidates, since collapsing an endpoint along one of these additional edges may still elongate it. Then, for all candidate edges, collapses in the allowed directions are evaluated using almost all of the checks described in Section 3.3.2.

Omega_h saves time by skipping the full edge ring and face ring checks, replacing them with a requirement that each $d$-dimensional simplex adjacent to the collapsing edge must be classified on the same geometry as its $(d-1)$-dimensional side adjacent to the collapsing vertex. In other words, Omega_h checks for the problematic case shown in Figure 3.8, but not for the case in Figure 3.7. As a result, Omega_h requires that the input mesh, model or metric be constructed such that the case in Figure 3.7 doesn’t arise. One simple way to ensure this is to mesh “holes” in the geometry. For Omega_h’s first target applications, this makes sense because the interaction of both the fluid and the structure is of interest. Barring that solution, one can topologically break up periodic geometry, for example a periodic CAD edge becomes three or more edges, which is quite reasonable in the vast majority of cases.

Omega_h also applies the same quality restriction to coarsening that it applies to refinement, i.e. changes are not accepted if they produce elements below a user-defined threshold.

Finally, Omega_h implements what we call "overshoot protection" as suggested by Michal and Krakos [56]. This means that edge collapses are rejected if they would create new edges that are longer than the edge refinement threshold from Section 3.4.1. This restriction allows us to alternate refinement and coarsening without entering an infinite loop in which a collapse operation is exactly undone by a refinement operation, which is then undone by a subsequent collapse, etc.

We mark candidate collapses using two boolean flags per edge, one for each endpoint. Once all collapses that violate the checks mentioned above are unmarked, we move the focus of the problem from edges to vertices. To do this, every vertex chooses a single adjacent candidate edge collapse which is the one it will represent for the remainder of the coarsening operation. The selection is made first preferring the shortest edge (as suggested in [56]), preferring higher element qualities if lengths are equal, and ties in both length and quality are broken using global entity numbers. All others collapses in which the vertex could have participated for which it is an endpoint are discarded. Now the independent set selection in Section 4.8.2 is used with vertices as the keys and their selected collapse qualities as the values. The selected independent set of vertices is then collapsed, along each vertex's chosen edge.

### 3.4.3 Shape Correction

Omega_h uses simpler and yet possibly more effective shape correction logic. We have found that it is beneficial to consider a wider portion of the mesh surrounding a sliver when attempting to remove said sliver, rather than being overly concerned with the particular shape of said sliver. Omega_h has a user-defined parameter for how many layers of elements around a sliver will be considered. In this case, layers are defined by elements adjacent to one another across faces, simply because this graph is much cheaper to compute than that of elements adjacent via vertices. For every sliver, elements that are at most the specified distance via face adjacency are marked as being involved in sliver removal. Then we do one of two things:

1. Mark all edges of all considered elements as candidates for edge swaps. For each such edge we compute the best possible edge swap operation using the optimizations described in Section 3.3.3.1, and give those quality values to the method in Section 4.8.2 to select a set of edge swaps to perform. Only edge swaps which locally improve the quality in their cavity are accepted.

2. Mark all vertices of all considered elements as candidates for collapsing. For each of these vertices, we consider collapsing it along any adjacent edge. This is just a marking process, after which the procedure in Section 3.4.2 is carried out, with the only difference being that for a collapse to be accepted, the quality in the cavity must locally improve, instead of being above a constant threshold.

Note that although these algorithms describe using layers around a sliver, they remain cheap because the layers need only be marked, after which only small cavities around the marked elements are considered. The full set of elements marked around a sliver is never collected or examined as a whole in any way. The marking itself can be done in a local iterative fashion, each step adding marks to all face-adjacent elements of currently marked elements.

Unlike MeshAdapt, Omega_h requires that all elements be above a certain mean ratio quality, and will continue to attempt shape correction until this goal is satisfied or all attempts stall (see Section 3.4.4). So far, we have been able to achieve minimum mean ratios between 20% and 30% for all mesh elements in a variety of cases.

However, in general there are known examples where minimum quality cannot be improved above an arbitrary bound, for example if the CAD model being approximated has an arbitrarily small angle, elements adjacent to that angle will be limited in quality. Such forced angles have already begun to arise dynamically in deforming simulations, for example in Figure 5.3. This is a well-researched problem in mesh generation, and the literature still suggests that we identify such "forced" bad quality elements and treat them differently from elements which have a chance of being improved [63]. Implementing such identification and isolation is considered future work.

### 3.4.4 Overall Steps

Listing 3.2 shows an abbreviated version of the C++ code for Omega_h's main function. There are two loops involved: the first attempts to satisfy edge lengths and the second attempts to repair element quality. The edge length loop consists of one refinement pass as described in Section 3.4.1 (on line 5), followed by one coarsening pass as per Section 3.4.2 (on line 6). This first loop continues until neither of the passes could modify the mesh, which is detected by the `did_anything` boolean variable remaining false. The termination of this loop is dependent on the overshoot protection described in Section 3.4.2.

The second loop attempts passes of shape correction operations, in decreasing order of preference. First a pass of edge swaps is attempted as described in Section 3.4.3. Only if no swaps could be performed are edge collapses attempted on the edges around slivers. This loop continues until the worst element is above the desired threshold (checked on line 8) or no valid correction operations could be found (the break at line 11) is reached.

Listing 3.3 shows the options structure. It specifies the following user-adjustable parameters:

Line 2: The minimum desired length of edges in metric space (used to mark edges for coarsening).

Line 3: The maximum desired length of edges in metric space (used to mark edges for refinement).

Line 4: The minimum allowed quality of any edge-length focused operation. During the first loop of Listing 3.2, any operations that would create elements below this quality are rejected.

**Listing 3.2: Omega_h main function**

```
bool adapt(Mesh* mesh, AdaptOpts const& opts) {
  bool did_anything;
  do {
    did_anything = false;
    if (refine_by_size(mesh, opts)) did_anything = true;
    if (coarsen_by_size(mesh, opts)) did_anything = true;
  } while (did_anything);
  while (mesh->min_quality() < opts.min_quality_desired) {
    if (swap_edges(mesh, opts)) continue;
    if (coarsen_slivers(mesh, opts)) continue;
    break;
  }
}
```

**Listing 3.3: Omega_h parameters**

```
1 struct AdaptOpts {
2   Real min_length_desired;
3   Real max_length_desired;
4   Real min_quality_allowed;
5   Real min_quality_desired;
6   Int nsliver_layers;
7   Verbosity verbosity;
8 };
```

Line 5: The minimum desired quality of elements after adaptation. The second loop in Listing 3.2 will continue until all elements are above this quality or no improvements can be made.

Line 6: The number of face-adjacent element layers around a sliver to consider for shape correction.

Line 7: The amount of information to print to the user regarding the operations being carried out.

### 3.4.5 Approaching Displacements and Metrics

A common problem that arises when doing Lagrangian simulations (those in which the mesh follows the simulated fluid and objects) as described in Sections 5.2, 5.3, and 5.5, is that vertex motions may be requested such that after deformation the mesh has inverted or arbitrarily low-quality elements. Similarly, when aiming to satisfy an anisotropic metric tensor field, it may be that the elements of an initial mesh have arbitrary low (though not negative) quality when measured in the desired metric space.

Omega_h implements two methods (which are similar to one another) in order to maintain good element quality throughout adaptation in each of these respective cases. The idea is to establish an initial field which is known to be compatible with the initial mesh in terms of element quality, and then to use an interpolation method to gradually travel from the initial field to the given target field. Each step along this interpolation will degrade element quality, and the subsequent adaptation will repair it sufficiently that another step may be taken. Because it is difficult to predict a priori how far one may interpolate towards the target before quality limits are met, we use a predictor-corrector method which begins by stepping all the way to the target field and iteratively halves the step size until quality criteria are met. The quality criteria is usually the same as the minimum allowed quality during adaptivity, see Section 3.4.4. After each step, mesh adaptation is used to improve element qualities which enables a subsequent step.

In the case of large displacement problems, the initial displacement field is zero for all the vertices. Linear interpolation between this and the solved-for large displacement equates to linear motion in space of the vertices. The mesh is deformed until qualities reach a limit, after which point adaptation is used to correct low quality elements.

In the case of metric fields, the initial metric field is the implied metric field described in Section 3.5.2. This metric field is closely aligned with the initial mesh and its elements typically have good quality when measured in this metric space. We use the Log-Euclidean metric tensor interpolation described in Section 3.5.1 to step from the implied metric field to the desired metric field. Each change in the metric field will tend to decrease element quality in metric space, and subsequent adaptation will improve quality by bringing the mesh into agreement with the new metric field.

## 3.5 Size Field Algorithms

### 3.5.1 Metric Interpolation and Storage

The metric tensor field introduced in Section 1.4.6 is typically provided as a set of values defined at mesh vertices. This leads to an important question about how to compute metric tensor values at other points in the mesh, i.e. how to interpolate the metric tensor. We can illustrate the issues involved with the simplest interpolation case, that of deriving a metric tensor for the center of a mesh edge given the two metric tensors at its endpoints. Unfortunately, linear interpolation of the components of the tensor $\mathcal{M}$ itself has the undesirable property that interpolating between two highly anisotropic metrics of even slightly different orientations will tend to produce isotropic results with very small desired lengths, i.e. the longer desired length will be lost. For this reason, alternative interpolation methods have been developed focusing on getting desired results in the case of high anisotropy. Some of the most relevant are as follows:

1. MeshAdapt works with an internal representation consisting of the orthogonal matrix $R$ and the vector of desired lengths $\mathbf{h} = (h_1, h_2, h_3)^T$ which form the metric tensor:

$$\mathcal{M} = R^T \begin{bmatrix} 1/h_1^2 & 0 & 0 \\ 0 & 1/h_2^2 & 0 \\ 0 & 0 & 1/h_3^2 \end{bmatrix} R \tag{3.2}$$

   Each of these is then interpolated separately. The vector $\mathbf{h}$ is linearly interpolated, while the matrix $R$ is first linearly interpolated, and then re-orthogonalized using the Gram-Schmidt process [64]. The benefit of this is that anisotropy is fully preserved due to the separate interpolation of lengths. However, there are three main drawbacks to this method:

   (a) The interpolation of $R$ is fragile, because if the linearly interpolated value is rank-deficient then re-orthogonalization will fail. This can happen, for example, if the two input $R$ matrices represent rotations that are 180° away from each other. Even though these represent exactly the same metric tensor, interpolation would fail.

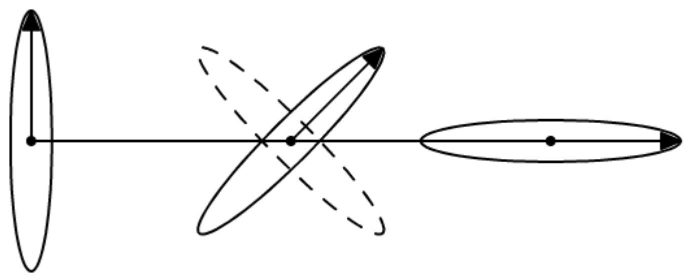

**Figure 3.11: MeshAdapt interpolation depends on eigenvector signs**

(b) The overall method gives results which depend too much on the details of how the tensor was decomposed. If we have two metrics which are 90° away from each other as shown in Figure 3.11, their interpolated result depends heavily on the signs of the eigenvectors chosen for $R$.

(c) This method stores 12 components per vertex on a 3D mesh, whereas the alternatives below store only 6.

2. One can instead compute the inverse of the metric tensors, then interpolate those linearly and invert the result. This is a method proposed by Alauzet and Frey as a compromise between the anisotropic fidelity and high runtime costs of the following two methods [65].

3. The highest fidelity interpolation suggested by Alauzet and Frey is the Power method which linearly interpolate $\mathcal{M}^{-1/2}$, which equals $\mathcal{M}$ replacing each eigenvalue $\lambda$ with $(1/\sqrt{\lambda})$. This requires the additional expense of computing an eigendecomposition.

4. An even better fidelity interpolation suggested by Loseille and Löhner [58] and later confirmed by Michal and Krakos [56] is the Log-Euclidean method which linearly interpolates $\log(\mathcal{M})$, which equals $\mathcal{M}$ replacing the eigenvalues $\lambda$ with $\log(\lambda)$. Figure 3.12 shows how this method better preserves very high levels of anisotropy. This is the method now used by Omega_h.

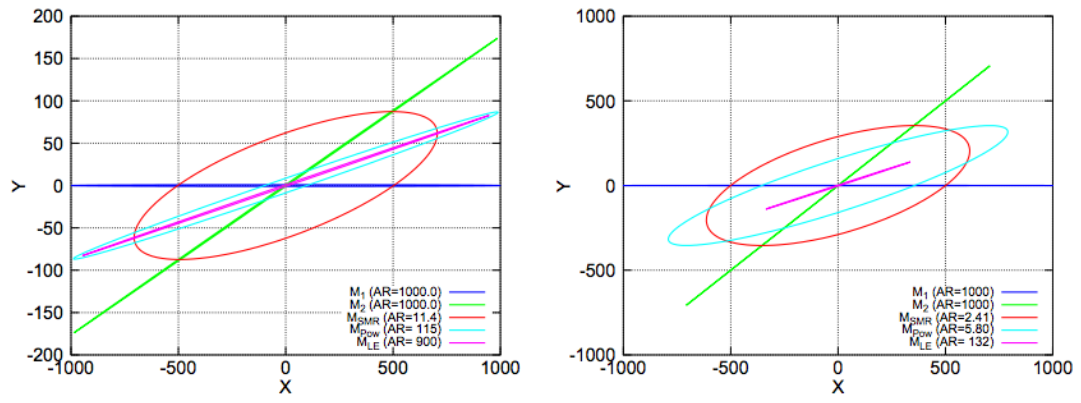

**Figure 3.12: Log-Euclidean versus Power interpolation at 1:1000 anisotropy [56]**

### 3.5.2 Implied Metric Field

It is often useful to develop an "implied" metric field [56]. Given a mesh, the implied metric field is the one which the mesh currently satisfies. Since there often does not exist a metric field which is exactly satisfied, the computed field is an approximation of the field which is most closely satisfied by the existing mesh. An implied metric field can be used to maintain a mesh's initial gradation while responding only to mesh motion. More importantly, it can be combined with the user's desired metric field to produce one or more intermediate metric fields which are more suitable for mesh adaptation (for example, gradually interpolating from one to the other).

For any simplex element, there exists a single metric tensor such that all its edges are unit length in metric space, which implies it is equilateral and has a perfect mean ratio quality in metric space. To find this metric, consider that each edge $i$ of the simplex forms a single scalar constraint:

$$1 = \sqrt{\mathbf{v}_i^T \mathcal{M} \mathbf{v}_i} \tag{3.3}$$

Where $\mathbf{v}_i$ is the real space vector along edge $i$. Also note that the metric tensor $\mathcal{M}$ has exactly as many independent scalars as the simplex has edges (3 in 2D, 6 in 3D). The constraint can be converted into a scalar equation for the independent variables:

$$1^2 = \left(\sqrt{\mathbf{v}_i^T \mathcal{M} \mathbf{v}_i}\right)^2$$

$$1 = \mathbf{v}_i^T \mathcal{M} \mathbf{v}_i$$

$$1 = \begin{bmatrix} x_i & y_i & z_i \end{bmatrix} \begin{bmatrix} a & d & f \\ d & b & e \\ f & e & c \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix}$$

$$1 = ax_i^2 + by_i^2 + cz_i^2 + 2dx_iy_i + 2ey_iz_i + 2fx_iz_i$$

In our 3D example, this leads to a $6 \times 6$ linear system:

$$\begin{bmatrix} x_1^2 & y_1^2 & z_1^2 & 2x_1y_1 & 2y_1z_1 & 2x_1z_1 \\ x_2^2 & y_2^2 & z_2^2 & 2x_2y_2 & 2y_2z_2 & 2x_2z_2 \\ x_3^2 & y_3^2 & z_3^2 & 2x_3y_3 & 2y_3z_3 & 2x_3z_3 \\ x_4^2 & y_4^2 & z_4^2 & 2x_4y_4 & 2y_4z_4 & 2x_4z_4 \\ x_5^2 & y_5^2 & z_5^2 & 2x_5y_5 & 2y_5z_5 & 2x_5z_5 \\ x_6^2 & y_6^2 & z_6^2 & 2x_6y_6 & 2y_6z_6 & 2x_6z_6 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \tag{3.4}$$

Where $y_3$ is the $y$ component of the length of edge 3, and so on. The matrix and right hand side are formed based on the element and the linear system is solved via QR decomposition [64], resulting in the implied metric tensor for an element. This derivation can be repeated for triangles in 2D, giving a 3 system for the symmetric tensor components.

In order to compute metric tensors at vertices, what is used is conceptually an average the implied values obtained at adjacent elements. First, we transform the element metric tensors into their "linear" form, which is one of the forms described in Section 3.5.1 used for metric interpolation (for example, Omega_h uses the matrix logarithm). Then we take at each vertex the average of these linear tensors at adjacent elements. Applying the inverse "de-linearizing" transformation to the average linear tensor gives the metric tensor at the vertex.

### 3.5.3 Implied Isotropic Size

If we are doing isotropic adaptation, we can follow a similar procedure to that in Section 3.5.2 for metric tensors, by identifying a desired edge length for each element and averaging this value to vertices. This is different from the typical approach of averaging the lengths of edges adjacent to a vertex, because only considering edges adjacent to the vertex can produce inaccurate results at corners of even simple meshes, where the axis-aligned edges are considered but diagonal edges are ignored. Unlike metric tensors, there rarely exists a single desired length which is satisfied by all edges of an element. The approximation we use is the root-mean-square of the element's current edge lengths ($l_{\text{RMS}}$) defined in Equations 1.6 and 1.7. This is done for consistency with our method of predicting output element counts, described in Section 3.5.4.

### 3.5.4 Targeting an Element Count

For users developing a desired metric field, it can be useful to have the ability to scale their desired metric field such that the resulting adapted mesh will have a certain desired number of elements. This is due to the limited amount of memory on computer systems; simulations making use of supercomputers will typically approach the limit of memory when storing their mesh. Therefore, it is preferable to limit resolution and execute correctly rather than exceed the memory limit in an attempt to reach high resolution. Together with re-partitioning onto a larger amount of hardware memory, metric field scaling should provide better control of parallel adaptive workflows.

Pain, Umpleby, de Oliveira, and Goddard describe one such method for predicting the number of elements produced and accordingly adjusting the metric field [66]. They begin with the assumption that after mesh adaptation all elements will have metric volume $\gamma_K$, where $\gamma_K$ is the volume of a perfectly equilateral tetrahedron with unit edge lengths. This leads to their volumetric prediction of the new element count $E_{\text{new}}$ based on the old elements which are here numbered from 1 to $E_{\text{old}}$:

$$E_{\text{new}} = \frac{\sum_{e=1}^{E_{\text{old}}} \tilde{V}_e}{\gamma_K} \tag{3.5}$$

Where $\tilde{V}_e$ is the volume of element $e$ in metric space, as we defined in Equation 1.5. They then seek a scalar $\beta$ such that the scaled size field $\beta\mathcal{M}$ will produce the desired number of elements $E_{\text{des}}$. They arrive at the following:

$$\beta = \left(\frac{\gamma_K \theta E_{\text{des}}}{\sum_{e}^{E_{\text{old}}} \sqrt{\det(\mathcal{M}_e)} V_e}\right)^{2/3} \tag{3.6}$$

Where $\theta$ is a correction factor which they state should be set to approximately 0.85. We claim this correction factor is compensating for the invalidity of the equilateral-volume assumption. It is impossible to mesh Euclidean space with equilateral tetrahedra, and should likewise be difficult in metric space. We propose using the mean ratios of input elements (in metric space) as an indication of how far typical element volumes would deviate from the equilateral volume (recall the interpretation of the mean ratio from Section 1.4.7). Instead of summing metric space volumes $\tilde{V}_e$, we sum the volumes that current elements would have if they were made equilateral while their root-mean-squared edge length was preserved. This makes them more comparable to the idealized equilateral elements being assumed on output:

$$E_{\text{new}} = \frac{\sum_{e=1}^{E_{\text{old}}} \gamma_K \tilde{l}^3_{e,\text{RMS}}}{\gamma_K} = \sum_{e=1}^{E_{\text{old}}} \tilde{l}^3_{e,\text{RMS}} \tag{3.7}$$

This leads to a simpler conclusion: the number of elements that input element $e$ will become after adaptation is approximately $\tilde{l}^d_{e,\text{RMS}}$, i.e. its root-mean-squared metric edge length raised to the power that represents area in 2D or volume in 3D. See Equations 1.6 and 1.7 for a definition or root-mean-squared metric length. This can be used to compute a new metric scaling factor:

$$\beta = \left(\frac{E_{\text{des}}}{\sum_{e}^{E_{\text{old}}} \tilde{l}^d_{e,\text{RMS}}}\right)^{2/d} \tag{3.8}$$

Equation 3.8 can also be used to compute element weights to estimate future memory usage for load balancing purposes.

## 3.6 Solution Transfer in a Cavity

When using mesh adaptation within the context of a simulation, one needs to maintain the simulation fields defined on the input mesh, and return the adapted mesh with the fields properly defined on it. Compared to full re-meshing methods, cavity-based adaptation has the advantage that field transfer can be localized to cavities, meaning that fields are only altered where the mesh topology was altered and that transfer algorithms need only consider this local problem as opposed to dealing with general mesh-to-mesh searches and global, typically expensive and/or unsatisfactory field fitting operations.

We are usually dealing with an input sub-mesh which has a field defined over it and an output sub-mesh over which we need to define a new field. While other researchers may have sub-meshes equal to the entire mesh, in our case these sub-meshes are always those associated with a local cavity before and after a mesh modification operation.

### 3.6.1 Conserving Integral Quantities

Integral conservation of specific fields may be required of solution transfer algorithms. There are typically infinitely many solutions which preserve the integral of some quantity over some portion of the mesh, and transfer algorithms differ in how well they preserve the variation of the solution field over space. In the context of conservative algorithms we use the terms “donor” and “target” to denote the input and output sub-meshes, respectively. Jiao and Heath develop a conservative method which minimizes the Sobolev error between the input and transferred fields by taking intersections of donor and target elements and solving a linear system based on a weighted residual formulation over these intersected elements which form a “common refinement” mesh [67]. Farrell and Maddison apply a similar method (in which they refer to the element intersections as the “supermesh”) with a focus on transferring quantities to support mesh adaptation [68]. Theirs can still be considered a full mesh-to-mesh method, which they apply to the meshes before and after adaptation.

Alauzet develops methods very similar to what we present in this section in their handling of $C^{-1}$ piecewise constants and $C^0$ piecewise linear functions as well as starting to account for different domains [69]. Surprisingly, Alauzet still applies this as a full mesh-to-mesh transfer despite building on a cavity-based mesh adaptation code.

We need conservative transfer in our work to develop the adaptive Lagrangian hydrodynamics application Alexa using Omega_h (see Section 5.3). In particular, our two goals to support Alexa are:

1. Conserve the mass of each physical object in the simulation. An object may be a fluid body or a solid mass. Mass is a piecewise constant scalar field defined over the element volumes. The algorithm we developed is described in Section 3.6.1.1.

2. Conserve the total momentum of the simulation system. This is done by using a special algorithm to transfer velocities (which are defined as vectors at mesh vertices) such that combined with the above mass transfer result in momentum conservation. This algorithm is described in Section 3.6.1.2.

In both these cases, the additional goal beyond conservation is the preservation of the variation or distribution of these fields, as much as possible within the bounds of conservation.

#### 3.6.1.1 Element-Centered Mass-Like Quantities

The integral conservation requirement for mass can be expressed in terms of a density field $\rho$, which has donor ($\rho_D$) and target ($\rho_T$) forms. Our goal is to find a target density $\rho_T$ such that Equation 3.9 is satisfied, where $\Omega_D$ is the domain of the donor cavity and $\Omega_T$ is the domain of the target cavity. Figure 3.13 illustrates such a scenario, and also illustrates the important case $\Omega_D \neq \Omega_T$, which we eventually attempt to address with Equation 3.14.

$$\int_{\Omega_T} \rho_T \, \mathrm{d}V = \int_{\Omega_D} \rho_D \, \mathrm{d}V \tag{3.9}$$

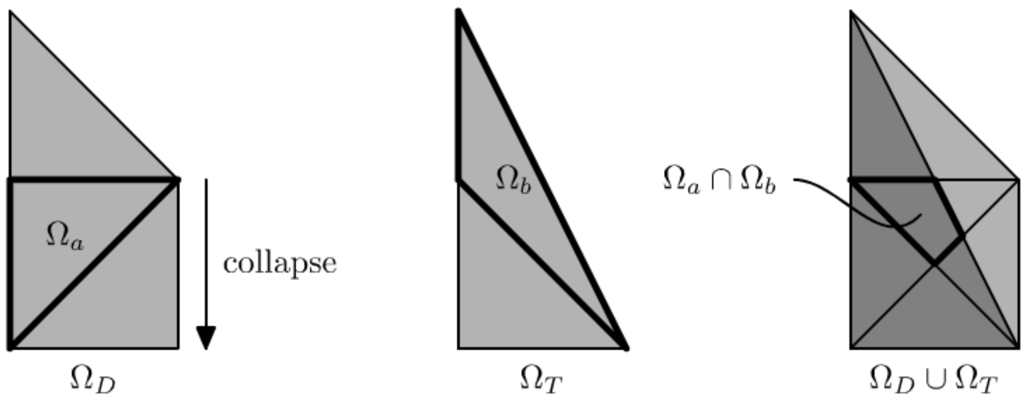

**Figure 3.13: Mass conservation case with changing cavity domains**

Our discretization of density is piecewise constant in that there is a single value over the domain of each element, and values are discontinuous between elements. The value of density in an element multiplied by the element's volume is the element mass. We define two sets of elements: the donor elements $E_D$ and the target elements $E_T$. Then discrete conservation is simply an equality of sums, and the goal is to find target element masses $m_a$.

$$\sum_{a \in E_T} m_a = \sum_{b \in E_D} m_b \tag{3.10}$$

To choose target masses that preserve the density distribution well, we begin with related work [67,68] which suggests that computing the geometric intersections of input and output elements allows the formulation of good conservative transfer algorithms. Their method is based on an assumption that donor and target domains are equal ($\Omega_T = \Omega_D$), and formulating the problem as a weighted residual of Equation 3.9:

$$\forall w \in H : \int_{\Omega_T} w \rho_T \, \mathrm{d}V = \int_{\Omega_D} w \rho_D \, \mathrm{d}V \tag{3.11}$$

Where $H$ is the space of weighting functions, and following [67] we choose $H$ to be the basis function space of the target mesh, meaning the space of piecewise constant functions on the target mesh. This just means $H$ can be used to single out each target element, giving the relationship:

$$\forall a \in E_T : \int_{\Omega_a} \rho_T \, \mathrm{d}V = \int_{\Omega_D \cap \Omega_a} \rho_D \, \mathrm{d}V = \sum_{b \in E_D} \int_{\Omega_b \cap \Omega_a} \rho_D \, \mathrm{d}V \tag{3.12}$$

Where $\Omega_a$ is the domain of element $a$, and the intersection of the domains of two elements $a$ and $b$ is written $\Omega_a \cap \Omega_b$. Knowing that $\rho_D$ is constant in $\Omega_b$, we get a simple sum of intersection volumes:

$$\forall a \in E_T : \int_{\Omega_a} \rho_T \, \mathrm{d}V = m_a = \sum_{b \in E_D} V(\Omega_b \cap \Omega_a) m_b \tag{3.13}$$

Where $V(\Omega_b \cap \Omega_a)$ is the volume of the intersection of elements $a$ and $b$. In order to compute this volume, we use the algorithms proposed by Powell and Abel [70] to first compute the polyhedron which is the intersection of the two element simplices, then compute its volume.

However, Equation 3.13 is based on the assumption that the domains of donor and target cavities are the same, and that there are not multiple objects requiring separate mass conservation. In the single object case, adaptation at the boundary of the object may produce different target and donor domains. We augment Equation 3.13 with a correction term to account for the fact that not all of $\Omega_D$ intersects with $\Omega_T$:

$$\forall a \in E_T : m_a = \sum_{b \in E_D} \frac{V(\Omega_b \cap \Omega_a)}{\sum_{c \in E_T} V(\Omega_b \cap \Omega_c)} m_b \tag{3.14}$$

We can confirm Equation 3.14 conserves mass by summing all target masses:

$$\sum_{a\in E_T} m_a = \sum_{a\in E_T}\sum_{b\in E_D} \frac{V(\Omega_b \cap \Omega_a)}{\sum_{c\in E_T} V(\Omega_b \cap \Omega_c)} m_b$$
$$\sum_{a\in E_T} m_a = \sum_{b\in E_D}\sum_{a\in E_T} \frac{V(\Omega_b \cap \Omega_a)}{\sum_{c\in E_T} V(\Omega_b \cap \Omega_c)} m_b$$
$$\sum_{a\in E_T} m_a = \sum_{b\in E_D} \frac{\sum_{a\in E_T} V(\Omega_b \cap \Omega_a)}{\sum_{c\in E_T} V(\Omega_b \cap \Omega_c)} m_b$$
$$\sum_{a\in E_T} m_a = \sum_{b\in E_D} m_b$$

Equation 3.14 can thus conserve mass between two cavities which may not have the same volume, so long as every donor element has a non-null intersection with the target cavity (the sum in the denominator). This requirement is satisfied for edge collapses (see Figure 3.5) and edge swaps (see Figure 3.10), because in both cases it is true that all donor elements are adjacent to at least one face of the relaxed cavity boundary (see Figure 3.2), and each face of the relaxed boundary is also adjacent to a donor element. If all donor and target elements have non-zero volume, then each donor element will overlap with at least one target element via this boundary face relationship.

Note that in the case of an edge split (with no snapping), every donor element becomes two target elements of equal volume, so we simply assign half the donor element mass to each.

#### 3.6.1.2 Momentum-Conserving Nodal Velocities

In the element formulation used in Alexa, velocity is a piecewise linear $C^0$ continuous function (its value is continuous between elements, but not its derivative). Such a field is defined in terms of common values at vertices, and the value at any point inside the element is a weighted sum of that element's vertex values, where the weights are the element basis functions, which in the linear case equal the barycentric coordinates of the point in the element.

The most important transfer done for piecewise linear fields is that during

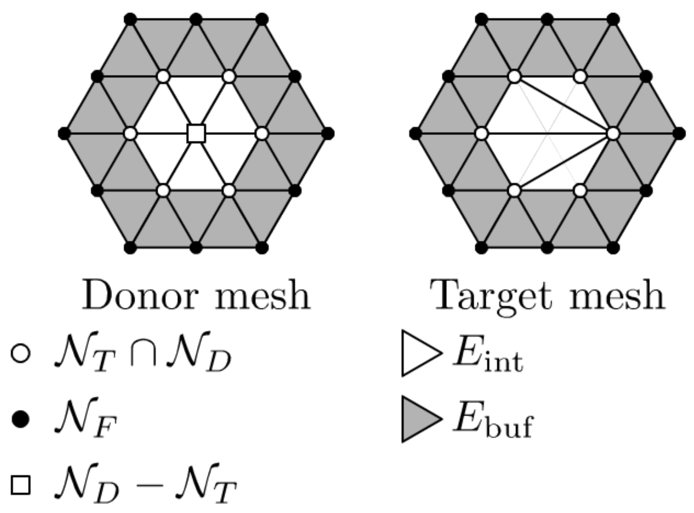

**Figure 3.14: Cavity with buffer layer and associated notation**

refinement, new vertices will receive values via interpolation. For the refinement operation, this transfer is exact because it preserves the value of the field at all points in space. For any other modifications (those that do not create new vertices) we would typically leave the values at vertices unchanged, accepting the small perturbations introduced by changes to the element connectivity.

However, in the case of momentum conservation, we find a need to modify the vertex velocity values in order to prevent changing element connectivity from destroying or creating momentum. One important difficulty here is that the way cavities for edge collapses and swaps are defined, *there are no vertices on the interior*. Recall from Section 3.1.2 that we require modifications leave their boundaries unchanged, making it impossible to alter vertex values without changing the cavity definition. This leads us to expand our cavity definition for collapses and swaps, as shown in Figure 3.14. By pushing the boundary outwards by one layer of elements, we bring the original boundary vertices into the interior which allows us to modify their values. In exchange for this, we pay a significant price in the complexity of parallelizing this method, because the selection of independent sets must now prevent cavities within distance three of one another from modifying simultaneously (see Section 4.8.2). This in turn requires that we increase the amount of ghosting to three layers during the selection procedure (see Section 4.8.2.2).

Because piecewise linear functions are continuous, we cannot use discontinuity to isolate the cavity as we did in Section 3.6.1.1. Instead, we must explicitly require that the new boundary vertices $\mathcal{N}_F$ remain fixed. This means the weighted residual technique from Equation 3.11 is no longer applicable.

The momentum field is the product of the velocity field and the density field, which in our case is stored as masses at elements. The integral of momentum over the domain of an element $e$ (in the case of mass and velocity being piecewise constant and linear, respectively) can be written as a discrete sum:

$$\mathbf{p}_e = m_e \frac{1}{d+1} \sum_{a \in \mathcal{N}(e)} v_a \tag{3.15}$$

Where $\mathcal{N}(e)$ is the set of vertices of $e$ and $d$ is the dimensionality of $e$ (assumed to be a simplex), with vertex velocity values $v_a$. We begin by leaving the velocity values unchanged between donor and target, and compute the momentum integrals over both donor and target cavities.

$$\mathbf{p}_T = \sum_{e \in E_T} m_e \frac{1}{d+1} \sum_{a \in \mathcal{N}(e)} v_a \tag{3.16}$$

$$\mathbf{p}_D = \sum_{e \in E_D} m_e \frac{1}{d+1} \sum_{a \in \mathcal{N}(e)} v_a \tag{3.17}$$

Note that the sum in Equation 3.16 can be rearranged to sum over vertices first:

$$\mathbf{p}_T = \sum_{a \in \mathcal{N}_T} v_a \frac{1}{d+1} \sum_{e \in E(a)} m_e \tag{3.18}$$

Where $\mathcal{N}_T$ is the set of vertices in the target cavity and $E(a)$ is the set of elements adjacent to vertex $a$. Our momentum conservation method will begin by computing the momentum loss $(\mathbf{p}_D - \mathbf{p}_T)$, then we use Equation 3.18 as the basis for altering vertex velocities such that each vertex contributes an equal amount of the momentum difference to the cavity, using these contributions to correct for momentum loss. In particular, for a target vertex $a$ the adjusted velocity $\tilde{v}_a$ is given by Equation 3.19.

$$\forall a \in \mathcal{N}_T : \tilde{v}_a = v_a + \frac{(d+1)(\mathbf{p}_D - \mathbf{p}_T)}{|\mathcal{N}_T| \sum_{e \in E(a)} m_e} \tag{3.19}$$

In the studies of this method to date we find that the momentum loss in a cavity is typically a small percentage, i.e. ($|\mathbf{p}_D - \mathbf{p}_T|/\mathbf{p}_D \leq 1\%$). For this reason it is less of a concern that we do not take into account the variation of velocity over space when distributing these corrections.

The final complication involved in altering these velocities is that Alexa (and simulation codes in general) apply boundary conditions involving velocity, meaning they require that vertices on a subset of the overall mesh boundary have velocity values which are equal to some user-defined function $v(x, y, z, t)$ over space and time. This means we should not apply velocity corrections to vertices with that restriction. Alexa will identify which boundary surfaces have boundary conditions applied to them and communicate them Omega_h. Omega_h will then reduce the set of modified target nodes $\mathcal{N}_T$ in Equation 3.19 accordingly. If no vertices remain in this set, we disallow that mesh modification from taking place. So far this has not been a serious limitation, but care needs to be taken in areas heavily affected by boundary conditions, such as the corner of a cube where all three surfaces are restricted.

Since refinement is exact, it automatically conserves momentum and does not need to employ the above cavity expansion or velocity correction. As discussed in Section 3.4.1, this is important because we prefer not to disallow refinement.

## 3.7 Serial Adaptation Performance

### 3.7.1 Analytic Anisotropy Test

In order to compare against other anisotropic adaptation software, we run Omega_h on the first test case presented by Park, Loseille, Krakos and Michal [71]. This case consists of a unit cube and an analytic metric defined in Equation 3.20. Figure 3.15 shows the mesh before and after Omega_h adapts it to conform to this analytic metric. We use the metric approaching method described in Section 3.4.5,

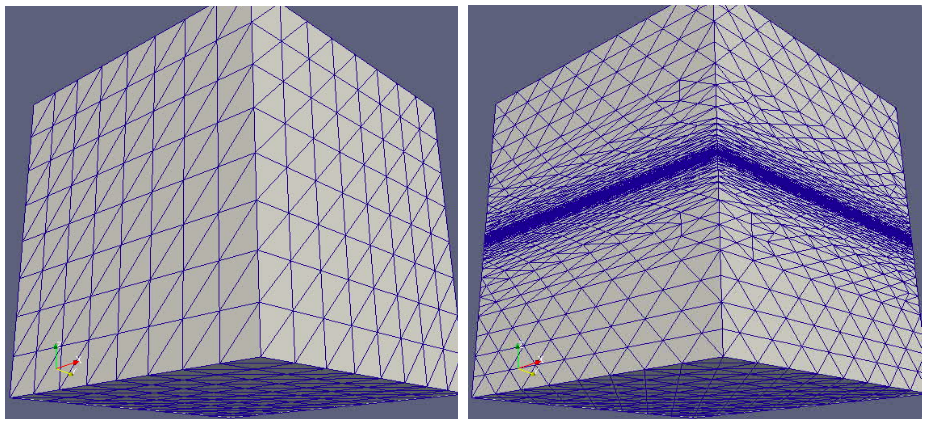

**Figure 3.15: Mesh before and after adapting to analytic metric**

which takes 10 steps to reach the final metric while restricted to keep all elements above 20% mean ratio quality. These 10 steps include 49 total refinement passes, 65 coarsening passes, and 22 swapping passes, and took 8.1 seconds to execute in serial on an Intel Xeon E5-2620v4 CPU. Table 3.1 shows the element quality and edge length statistics in metric space for the final adapted mesh. Note that shape correction was able to bring all elements above 30% quality while keeping edges in the length range shown in the histogram (the minimum edge length is 0.16).

$$\mathcal{M} = \begin{pmatrix} h_x^{-2} & 0 & 0 \\ 0 & h_y^{-2} & 0 \\ 0 & 0 & h_z^{-2} \end{pmatrix} \quad \text{where} \begin{cases} h_x = 0.1 \\ h_y = 0.1 \\ h_z = 0.001 + 0.198|z - 0.5| \end{cases} \tag{3.20}$$

### 3.7.2 Size Field Scaling Test

In order to demonstrate the implied size field formula described in Section 3.5.3 and the size field scaling to target a given element count described in Section 3.5.4, we use a series of mesh adaptations to increase the element count of a mesh by a factor of 2 at each adaptation. Note that uniform template refinement, which involves marking all edges for refinement and using the corresponding template described in Section 3.3.1, increases element count by a factor of 8 in 3D, so for a factor 2 increase we need general non-uniform refinement.

**Table 3.1: Metric quality and length histograms for analytic metric test**

| quality | #elements |
|---|---|
| 0.00-0.10 | 0 |
| 0.10-0.20 | 0 |
| 0.20-0.30 | 0 |
| 0.30-0.40 | 295 |
| 0.40-0.50 | 1268 |
| 0.50-0.60 | 4117 |
| 0.60-0.70 | 9241 |
| 0.70-0.80 | 13221 |
| 0.80-0.90 | 19660 |
| 0.90-1.00 | 5224 |

| length | #edges |
|---|---|
| 0.00-0.20 | 1 |
| 0.20-0.40 | 120 |
| 0.40-0.60 | 782 |
| 0.60-0.80 | 13644 |
| 0.80-1.00 | 18290 |
| 1.00-1.20 | 15847 |
| 1.20-1.40 | 16215 |
| 1.40-1.60 | 654 |
| 1.60-1.80 | 23 |
| 1.80-2.00 | 7 |

**Table 3.2: Element counts and adapt times for size field scaling**

| desired elements | actual elements | seconds to adapt |
|---|---|---|
| 7000 | 6912 | 0.19 |
| 14000 | 13079 | 0.50 |
| 28000 | 26255 | 0.96 |
| 56000 | 50441 | 1.62 |
| 112000 | 99320 | 3.07 |

We begin with graded mesh as shown in Figure 3.16(left), taken from our circuit simulation work described in Section 5.5. We compute its implied isotropic size, scale this size field such that a mesh conforming to the scaled size field would have approximately twice the elements, and adapt to the scaled size field. Notice that despite the curved boundary, no snapping takes place. We repeat this procedure 4 times, going from 7K elements to 112K elements, with the final mesh shown in Figure 3.16(right). Table 3.2 shows the desired and actual element counts at each step when this procedure is carried out using Omega_h. We do expect the actual element count to be less than desired due to the thresholds used to decide when to refine edges. Also as expected in serial, adaptation time is directly proportional to element count (given that this workflow does approximately equal work per element).

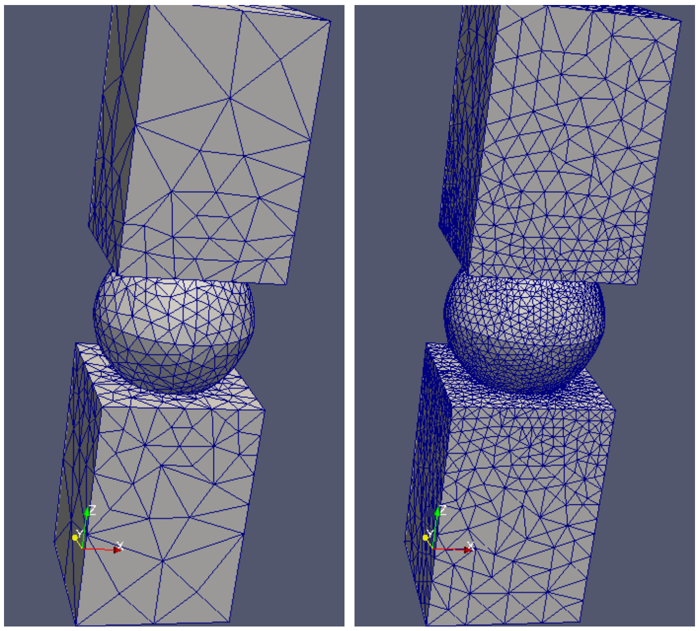

**Figure 3.16: Meshes before and after size field scaling**

# CHAPTER 4
# SCALABLE PARALLEL MESH ADAPTATION

## 4.1 Defining Scalability

It is important to first define scalability in the context of parallel computers. First, we define a single scalar $N$ describing the problem size. For mesh-based simulations, the number of mesh elements is a good approximate measure of problem size. Then we consider any operation which would take $O(N)$ time to execute on a single processor such as the reference computer in Section 1.5, and analyze the runtime bound of this operation when it is parallelized using $P$ cooperating processes. When the parallel runtime is $O(N/P)$, we say the operation has perfect linear scaling. Linear scaling typically only occurs when the processes do not have to coordinate, and coordination usually adds a factor $\log(P)$ to the runtime, meaning that the operation takes $O((N/P)\log(P))$ time in parallel.

Notice that we are considering a single operation which takes $O(N)$ time in serial. For example, a Recursive Inertial Bisection partitioning method is a divide-and-conquer algorithm which uses $O(\log(P))$ steps, each taking $O(N)$ time, to assign $N$ items to $P$ partitions for a total serial runtime which is $O(N\log(P))$ (analogous to an incomplete sorting algorithm) [75]. If the items are already well partitioned amongst $P$ processes and we run a parallel version of this procedure to adjust the partitioning, each $O(N)$ step becomes $O((N/P)\log(P))$ and the whole operation can be expected to take $O((N/P)\log^2(P))$.

Next, we define parallel runtime bounds which are considered non-scalable. The simplest example is one where an $O(N)$ serial operation becomes an $O(N)$ par-

Portions of this chapter previously appeared as: D. A. Ibanez, E. S. Seol, C. W. Smith, and M. S. Shephard, "PUMI: Parallel unstructured mesh infrastructure," *ACM Trans. Math. Softw.*, vol. 42, no. 3, pp. 17:1–17:28, May 2016.

Portions of this chapter previously appeared as: D. Ibanez, I. Dunn, and M. S. Shephard, "Hybrid mpi-thread parallelization of adaptive mesh operations," *Parallel Comput.*, vol. 52, pp. 133 – 143, Jan. 2016.

Portions of this chapter previously appeared as: D. Ibanez and M. Shephard, "Mesh adaptation for moving objects on shared memory hardware," in *Proc. 25th Int. Meshing Roundtable.* Washington, DC: Elsevier, Sep. 16–29, 2016.

allel operation, i.e. there is asymptotically no scaling. Although trivial, an alarming number of parallel programs in production use exhibit this degree of scaling for key parallel operations (for example, partitioning is done is serial prior to a parallel simulation). Because non-scalable operations may not consume a large percentage of runtime at low degrees of parallelism, these systems can show good parallel speedups up to a certain degrees of parallelism, but ultimately run into difficulties for large $P$ ($10^6$). The second example is an operation which has $O(P)$ runtime in parallel. Once again, may systems in production use today suffer from such trivial non-scalability. For example, it is natural when organizing communications for each process to create an array of size $P$ with information about what is being sent to each other process (or worse, to actually send data to every other cooperating process). The continued use of $O(P)$ algorithms is a significant obstacle to increasing the degree of shared memory parallelism in leadership-class supercomputers. The issue is made worse by the fact that until fairly recently, MPI (see Section 1.7.2) itself would either use or encourage the use of such algorithms [19]. Section 4.2 describes how to use modern MPI to avoid non-scalable operations during communication.

All the above considerations have analogues for the case of shared-memory programming, in which case one can exchange the number of parallel processes $P$ for a number of parallel threads $T$. In the case of shared memory only, there are fewer communication concerns because all threads may access all data directly. Although there are Non-Uniform Memory Access (NUMA) costs incurred if threads do not organize their data to minimize sharing, these costs are much smaller than the cost of network communication.

## 4.2 Parallel Operations

In this section we introduce a set of key collective parallel operations used to write parallel programs. All of these are scalable operations as described in Section 4.1. As such, a good approach to designing scalable programs is to use only these operations for parallelism.

### 4.2.1 Map

This is the most basic parallel operation, and is included to place our definitions in the context of the map-reduce programming model which has gained popularity in processing Internet data sets [76]. The simplest definition of a map operation takes in an array $\mathbf{a} = \{a_0, a_1, a_2, ...a_N\}$ and a unary operator $f$ and returns an array $\mathbf{b} = \{f(a_0), f(a_1), f(a_2), ..., f(a_N)\}$.

We generalize it to the idea of running $N$ independent operations in parallel with no need for coordination. The shared memory implementation of a map operation is the `parallel_for` concept introduced in Section 1.7.3. The need for these operations to be independent often forces the result to exist as one or more arrays of $N$ result values. The distributed memory implementation is anything inside a normal MPI program. Since MPI processes always execute in parallel, anytime all processes perform the same operation without coordination (message passing), it can be interpreted as an implicit map operation. This illustrates an important distinction between MPI programming and shared memory programming: MPI is parallel by default, while shared memory programming environments like OpenMP and CUDA are serial by default and parallelism must be explicitly requested per operation.

The map operation is the prime example of an operation which scales perfectly. Assuming the operator $f$ requires $O(1)$ time, a serial execution requires $O(N)$ time and a parallel operation over $P$ processes and $T$ threads takes $O(N/(PT))$ time.

### 4.2.2 Reduce

A reduction operation takes as input an array $\mathbf{a} = \{a_0, a_1, a_2, ..., a_N\}$ of scalar values and a binary operator $\otimes$, and returns a single scalar result. The result is related to the array entries via a binary expression tree, and in order to allow flexibility of implementation, any valid binary tree may be used. Figure 4.1 shows two valid binaries trees an implementation may use to reduce an array of five entries. The value of a leaf node in this tree is the value of a unique array entry, and the value of the non-leaf nodes is the result of applying the binary operator to the values of its two child nodes. The value of the root of the tree is the overall reduction result.

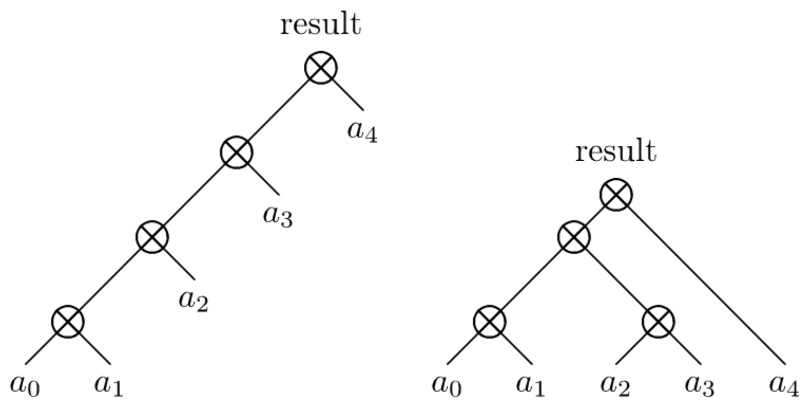

**Figure 4.1: Two possible binary trees used for reduction**

The left tree in Figure 4.1 illustrates the typical serial reduction process of maintaining an accumulator value and adding each consecutive array entry to it. The right tree in Figure 4.1 illustrates the most theoretically scalable implementation, i.e. a balanced tree. Assuming we have as many parallel processes as we have array entries, each of the $\lceil \log(N) \rceil$ layers of the balanced tree can be evaluated simultaneously and the runtime to evaluate the reduction is $O(\log(N))$ for arrays of size $N$. However, the balanced tree requires $O(N)$ memory for the intermediate result at one layer, while the serial tree can be evaluated with a single accumulator ($O(1)$ memory). The memory requirement is one reason why the serial tree is chosen in cases when no parallelism is available.

If we have fewer parallel processes than array entries, we can use a hybrid mix of the two trees in Figure 4.1, where each processor uses a serial tree to reduce $O(N/P)$ values in $O(N/P)$ time, and then each process' local results are reduced using a balanced tree in $O(\log(P))$ time resulting in an $O((N/P)+\log(P))$ runtime.

The binary operators used most often in Omega_h reductions are the sum, minimum, and maximum of the two inputs. Notice that the ability of implementations to choose any tree is a problem when the operator $\otimes$ is not commutative (the result depends on the tree structure), as is the case with typical computer implementations of floating-point summation. Section 4.9.2 will describe a technique for dealing with this issue.

The shared memory implementation of reductions is in our case provided by the

Kokkos library's `Kokkos::parallel_reduce` function [23]. The distributed memory implementation (which accepts a single value per process) is provided by MPI's `MPI_Reduce` and `MPI_Allreduce` functions (the later makes the result available to all processes, which is often desired). Combining these gives a reduction that can be expected to complete in $O((N/(PT)) + \log(PT))$ time.

### 4.2.3 Scan

Scan operations are key to scalable parallel programming, especially when using arrays for storage. Given an input array $\mathbf{a} = \{a_0, a_1, a_2, ..., a_N\}$ and a binary operator $\otimes$, an inclusive scan returns an array $\mathbf{b} = \{b_0, b_1, b_2, ..., b_N\}$ where each $b_i = \text{reduce}(\{a_0, a_1, a_2, ..., a_i\}, \otimes)$, i.e. the $i$-th output value is the result of reducing the first $i$ input values. An exclusive scan differs in that the $i$-th output value is the reduction of the first $(i - 1)$ values only (meaning the first output value must be defined separately). A scan with $(\otimes) = (+)$ is often referred to as a prefix sum.

Intuitively, a scan operation may be implemented with help from the intermediate values of a reduction tree. For example, the intermediate values in the serial tree (Figure 4.1(left)) are exactly the output values for scanning, meaning a scan takes $O(N)$ time in serial. Parallel scanning is a bit more complex and requires essentially traversing a balanced tree twice (Figure 4.1(right)), which still gives it a time complexity of $O(\log(N))$ assuming as many processors as input values.

We use the `Kokkos::parallel_scan` function to provide the shared memory implementation of scans, and the `MPI_Exscan` function to provide the distributed memory implementation, which accepts one value per process.

The utility of exclusive scans is in generating offsets for sub-groups of items in a global collection based on the size of each subgroup. For example, given the number of mesh elements per part, the `MPI_Exscan` takes $O(\log(P))$ time to returns offsets which can be used to establish a global numbering of mesh elements without further communication. Likewise, given the number of edges adjacent to each mesh vertex, a `Kokkos::parallel_scan` takes $O((N/T)\log(T))$ to produce offsets for all vertices that allow the construction of compressed-row storage for the vertex-to-edge adjacency (see Section 2.5.2).

### 4.2.4 Sort

Although we do not yet require a full distributed memory sorting algorithm in our work, we do have uses for sorting at the shared memory level (see Section 2.7), so we include this as a key parallel operation. In Omega_h, we use a slightly unusual definition of sorting: given an array $\mathbf{k} = \{k_0, k_1, k_2, ..., k_N\}$ of keys and a strict weak ordering $(\prec)$ [77], return a permutation array $\mathbf{p} = \{i_0, i_1, i_2, ..., i_N\}$ which indicates a sorted order for $\mathbf{k}$, meaning that $(k_{i_j} \prec k_{i_{j+1}})$. We use this definition so that the permutation array can be re-used to sort multiple other arrays that are associated with the array of keys. We also require that the algorithm be stable (keys which are "equal" in the ordering $(\prec)$ retain the same relative order) for determinism reasons. This is especially important when other arrays are being sorted using the permutation array.

Implementing a parallel sorting algorithm is considerably more challenging than implementing scans and reductions. At the time of this writing, Kokkos does not offer a highly general sorting capability. Instead, Omega_h relies on the Thrust library to provide sorting functionality on GPUs [78]. There several algorithms for sorting on GPUs [79], ranging in complexity up to $O((N/T)\log^2(N))$, which we still consider scalable. On manycore CPUs, we use a parallel merge sort implementation provided by researchers at Intel [80].

### 4.2.5 Exchange

We use the term exchange to refer to the distributed memory operation in which each process has a set of messages to send, each message having a unique destination process, and these messages are exchanged such that each process receives all the messages addressed to it during the operation. The difficult part of implementing an exchange is determining, for a given destination process, the full set of messages addressed to it. Hoefler, Siebert, and Lumsdaine [81] describe a communication-optimal algorithm for conducting a "sparse dynamic exchange". They define a sparse exchange as one in which the number of messages sent from or received by any single process is $O(\log(P))$. This is important, because if a process tries to send $O(P)$ messages then it is impossible to execute the exchange in a scalable way. They define a dynamic exchange as one in which the set of messages a process must receive is unknown to that process at first, hence the difficulty described above.

Many codes in use today solve this problem by using a variant of the "personalized consensus" algorithm in [81], reproduced here as Algorithm 1. By relying on a table (array) of size $P$, this algorithm is non-scalable. A scalable solution is the "non-blocking consensus" algorithm in [81], seen here as Algorithm 2. This algorithm uses synchronous sends, meaning that when a message is received at its destination, a confirmation is returned to the sender. Once a process receives confirmation that all its outgoing messages reached their destinations, it enters the non-blocking barrier. This barrier can execute concurrently with other message traffic, and once all processes enter it, it will signal its completion to all processes. Upon detecting completion, a process can be sure that all messages in flight have been received, thus it is safe to stop waiting for incoming messages. The barrier must be non-blocking because in order to make progress, processes must be able to continue receiving incoming messages until the time of overall completion. Knowing the communication is sparse, this algorithm runs in $O(\log(P))$ time [81].

**Algorithm 1:** Personalized Consensus [81]

```
  input  : List $S$ of destinations and data
  output: List $R$ of received data and sources
1 allocate local table with $P$ entries, initialize all entries to '0';
2 foreach $i \in S$ do
3     set row target($i$) in local table to '1';
4 $g$ = global sum of my table row;
5 foreach $i \in S$ do
6     start nonblocking sends to dest($i$);
7 for round from 1 to $g$ do
8     msg = blocking probe for incoming message;
9     allocate buffer, receive message, add buffer to $R$;
```

Section 4.3.3 describes how PUMI's communication library, PCU, implements Algorithm 2. This implementation goes beyond MPI because it is designed to allow communication between operating system threads instead of just processes.

As of the MPI 3.0 standard, however, it is no longer necessary to program

**Algorithm 2:** Non-Blocking Consensus [81]

```
input  : List $S$ of destinations and data
output: List $R$ of received data and sources
1  done=false;
2  barrier_active=false;
3  foreach $i \in S$ do
4      start nonblocking synchronous send to process dest($i$);
5  while not (done) do
6      msg = nonblocking probe for incoming message;
7      if msg exists then
8          allocate buffer, receive message, add buffer to $R$;
9      if barrier_active then
10         comp = test barrier for completion;
11         if comp then done=true;
12     else
13         if all sends are finished then
14             start nonblocking barrier;
15             barrier_active=true;
```

one's own exchange algorithm, as it can be accomplished by combining two key MPI systems: scalable process topology (graph communicators) [82], and neighborhood collectives [83]. This is the approach taken for distributed memory exchanges in Omega_h.

MPI's neighborhood collective functions perform non-dynamic exchanges, meaning that they must be informed of the incoming as well as outgoing messages. This is semantically the same as performing a series of single-message sends and receives, but comes with a few benefits. First, MPI's understanding that the messages are all to be sent at once enables it to perform certain optimizations. Second, the interface specifies that the data for all outgoing messages is to be packed in a single array (with data going to the same destination being contiguous). This is good for integrating MPI with shared-memory code, because the shared-memory code may efficiently prepare the outgoing data array and may efficiently process the incoming data array, thus minimizing the amount of work done by MPI (which is not guaranteed to use shared-memory parallelism to accelerate its work). Unfortunately, the neighborhood collective exchange functions have the name prefix `MPI_Neighbor_alltoall`, where `alltoall` is too reminiscent of the non-scalable practice of transmitting $O(P)$

messages from one processor.

The remaining problem of determining messages to receive can be delegated to MPI's scalable process topology description via the function `MPI_Dist_graph_create`. First, note that the above process is a graph theoretical problem of determining incoming graph edges to a graph node when each graph node knows only its outgoing edges. `MPI_Dist_graph_create` allows each process to specify any subset of the edges in the communication graph, and assumes responsibility for informing each process of all its incoming and outgoing edges. This is a more general and difficult problem than the one described above, and we restrict our use of this function to having each process specify all its outgoing edges. Given this restriction, we assume it executes in $O(\log(P))$ time.

Note that the exchange operation does not have a clear shared-memory equivalent in the context of Omega_h, because accessing shared memory directly is preferred over sending explicit messages amongst threads.

## 4.3 PCU: Scalable Inter-Thread Communication

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

The foundation of PCU is its point-to-point message passing routines where non-blocking synchronous message passing primitives are defined. There are two versions, one of which is a direct interface to MPI, and the second supports message passing between threads [73]. The two versions are interchangeable, and PCU can change which set of them is being used at runtime without affecting the rest of software components.

Building on the point-to-point primitives, PCU has an extensible framework for collective operations such as reduction, broadcast, scan, and barrier. Any collective whose communication pattern can be encoded as some kind of tree is supported, and the most common ones come built-in to PCU. These collectives are directly available to users.

Using both collectives and point-to-point communication, PCU provides a flexible user interfaces similar to MPI 3.0's `MPI_Neighbor_alltoallw`. The first phase allows tasks to construct messages to multiple neighbors at once and then send them. The second phase ensures that neighbors receive all the messages they have been sent. This "phased" communication algorithm is conceptually the same as Algorithm 2.

Finally, PCU has a system for creating a pool of threads within each process and assigning them ranks the way MPI does to processes. Users can call this API to enter a hybrid MPI+threads mode in which all the communication APIs (point-to-point, collective, and phased) work between threads. These capabilities support PUMI's overall hybrid MPI+thread operation.

### 4.3.1 Messaging Primitives in PCU

The main unusual design choice of PCU compared to other hybrid programming systems is its focus on inter-thread message passing. Since we must rebuild some of the high-level message passing capabilities, we identify a set of primitive operations as described by the MPI standard [18] which are sufficient for our applications:

1. Non-blocking synchronous send
2. Non-blocking send request completion test
3. Non-blocking probe
4. Blocking receive

These conceptual message-passing primitives are independent of their particular implementation. Note that because we require the send operation to be synchronous, it will complete if and only if the message is completely received at its destination.

All the remaining algorithms rely only on these guaranteed properties of the message passing primitives. To develop inter-thread message passing, we implement inter-thread message passing passing primitives. These are currently based on calling thread-safe MPI versions of the same primitives, but an area of future work involves implementing more efficient primitives.

In order to use MPI itself to pass messages between threads, we require that the implementation correctly handle self-sends. Then, we need to encode the source and destination thread IDs into the message metadata such that messages can be multiplexed out of a single process and demultiplexed at their destination process. The encoding of thread IDs makes use of the standard `MPI_TAG` metadata integer, which is typically a 32-bit signed integer. We use 10 bits of this integer to encode each of the local IDs for the source and destination threads. This encoding of source and destination means that threads must inspect messages with more sophisticated checking of the tag than `MPI_Recv` offers, since messages arriving at the same process may be destined for different threads within that process. We use `MPI_Iprobe` to inspect the tag before using `MPI_Recv` to commit to being the receiver. This combined probe and conditional receive procedure is specified in Algorithm 3.

**Algorithm 3:** Non-blocking pattern-match receive

```
  input  : pattern $P$
  output: received message metadata $M$ and data $b$, or null
1 let message $M \leftarrow$ non-blocking probe;
2 if $M$ is null (there is no message) then
3     return null;
4 if metadata of $M$ does not match $P$ then
5     return null;
6 allocate buffer $b$ per metadata of $M$;
7 blocking receive $M$ into $b$;
8 return $(M, b)$;
```

### 4.3.2 Simple Collectives in PCU

Collective operations are a necessary staple of distributed-memory high-performance computing. Operations such as parallel reduction, broadcast, and other collectives are key to coordinating threads [84]. More details on these collective algorithms and tradeoffs in their implementation can be found in [85]. Non-blocking collectives are more advanced implementations which are typically used to overlap communication and computation. By using them, PCU benefits from this overlap. Furthermore, in Section 4.3.3, we use Algorithm 2 for which a non-blocking implementation is necessary.

Since collectives are used by nearly all applications, we place a focus on developing built-in multithreaded collective operations based on the inter-thread message passing primitives.

We consider three fundamental collective operations: broadcast, reduce, and scan. Other operations such as exclusive scan and all-reduce can be built from the first three. These operations were selected as the minimal subset of collectives needed for our unstructured mesh operations.

These three collectives share many common features: they use $O(\log n)$ steps for $n$ threads, and at each step each thread is either idle, sending one message, or receiving one message. These shared characteristics make it easier to implement all the collectives in a general framework which abstracts away their differences, starting with the specific communication pattern used. A thread only needs to know which of the three actions to perform at each step and with which thread it is communicating, if any. This combined information is referred to as the communication pattern.

The second abstraction we can make is that of a merge operator, which is essentially the MPI reduction operator (e.g. `MPI_SUM` or `MPI_MAX`). The merge operator modifies the local data based on incoming data. For example, an reduction sum adds incoming values to local values. We do not refer to it as the reduction operator because it is used in all cases, including broadcast. As an interesting corner case, the merge operator for broadcast simply assigns the incoming value as the local value.

Following good software design, the communication pattern, merge operator, and data are each specified separately and are orthogonal from one another. This follows the example of interfaces such as `MPI_Reduce`.

With these abstract components specified, we can execute a collective operation using the non-blocking point-to-point message passing primitives developed in Section 4.3.1. Although the simplicity of collectives would allow blocking primitives to be used, using non-blocking primitives gives us a great benefit: we obtain a non-blocking collective operation. Such operations were implemented by Hoefler and Lumsdaine outside MPI [86] and their subsequent proposal to make them part of the MPI standard [87] was accepted. Our work implements hybrid threaded versions of such collectives.

Users of this system initiate a collective operation, and can interleave computation with communication progress queries. Communication progress consists of checking for incoming messages in the current step and proceeding to the next step when they are received.

Non-blocking collectives are useful from the perspective of of hiding latency, but they prove to be indispensable to Algorithm 2.

As a detailed example of how our non-blocking collectives work, Algorithms 4 and 5 show the functions that would begin and advance a non-blocking reduction, respectively. In these algorithms we use $\wedge$ to denote a bitwise and operation and $\oplus$ to denote a bitwise exclusive or operation. The operator $\otimes$ continues to represent the binary reduction, e.g. summation. Several helper functions were inlined for brevity, hence there is some repetition. For example, the termination condition on line 4 of Algorithm 4 and lines 1 and 14 of Algorithm 5. This condition is one of the aspects which varies depending on which type of collective is being executed (broadcasts and scans have different logic here). We implement a balanced binary tree (similar to Figure 4.1(right)) based on the binary digits of the thread ID (rank).

**Algorithm 4:** Starting a non-blocking reduction

```
    input  : local data $d_l$
 1  peers ← total thread count;
 2  rank ← local thread rank ∈ [0,peers−1];
 3  bit ← 1;
 4  if ((rank= 0) and (bit≥peers)) or
 5  ((rank≠ 0) and (((bit/2)∧rank)≠ 0)) then
 6      return;
 7  if rank⊕bit<peers then
 8      if bit∧rank≠ 0 then
 9          request ← send $d_l$ to (rank⊕bit);
10  return state as (bit,request)
```

**Algorithm 5:** Progress a non-blocking reduction

```
   input  : access to local data $d_l$
   input  : access to state tuple (bit,request)
1  if ((rank= 0) and (bit≥peers)) or
2  ((rank≠ 0) and (((bit/2)∧rank)≠ 0)) then
3      return false;
4  step_done ← false;
5  if rank⊕bit≥peers then step_done ← true;
6  else
7      if bit∧rank≠ 0 then
8          if request is done then step_done ← true;
9      else
10         if received message from (rank⊕bit) with data $d_r$ then
11             $d_l \leftarrow d_l \otimes d_r$;
12             step_done ← true;
13 bit ← 2·bit;
14 if ((rank= 0) and (bit≥peers)) or
15 ((rank≠ 0) and (((bit/2)∧rank)≠ 0)) then
16     return false;
17 if rank⊕bit<peers then
18     if bit∧rank≠ 0 then
19         request ← send $d_l$ to (rank⊕bit);
20 return true
```

### 4.3.3 Non-blocking Consensus in PCU

A common problem that arises when dealing with parallel graphs, and similar structures, such as the adjacency relations of unstructured meshes, has to do with transporting graph, or mesh entities, from one thread to another due to changes in the graph or to other operations which affect load balance. Such transportation is specified in a one-sided, push-driven manner, which means that each thread knows which entities it should send to which other threads, but does not know what it will be receiving.

Without *a priori* knowledge of the extent of information to be received, it is difficult to determine when to stop receiving information. A thread can perform a continuous loop which receives messages, but we must determine when to terminate that loop.

This problem has been solved previously in a slightly less efficient manner [88], and is an important special case of the general termination detection problem. PCU implements an optimal solution based on Algorithm 2 described in Section 4.2.5. Note that Algorithm 2 overlaps all incoming message processing with all outgoing message wait operations, so the potential for latency hiding is maximized.

When implementing phased message passing, we also optimize performance by buffering small messages. Users see an interface which allows sending small amounts of data to any destination. The user interface of phased communication allows us to pack all data traveling between the same pair of threads into a single message. This is done prior to executing Algorithm 2, which expects that the number of messages sent $|S|$ is equal to the number of unique destinations and the number of messages received $|R|$ is equal to the number of unique sources.

This relation of message counts to communication neighbors (sources and destinations) is quite useful in the process of determining runtime bounds. This is because, due to mesh partitioning, each thread should have a small and constant number of neighbors, with 40 being a maximum value among several meshes studied [89].

We have shown previously that buffering small messages can greatly improve performance due to the latency cost $\alpha$ and MPI's own management overhead per message, especially for applications with a tendency to send very small messages between the same pair of threads [88]. In the case of multiple threads per process, buffering also reduces the number of calls to `MPI_Send` and `MPI_Recv`, which reduces contention for the MPI library between the threads in that process. When these calls are protected by a lock, avoiding contention is important [90].

## 4.4 Remote Copies and Owners

For distributed memory parallelism, we are required to separate our data amongst multiple memory spaces. In order to find a piece of data across an entire distributed memory machine, one needs to pieces of information: which of the memory spaces it is in, and where in that memory space it is. The first value is provided by MPI and is called a "rank", which is a 32-bit integer identifying an MPI process. In object-oriented codes, the second value is a pointer to an object, which is a memory address (typically 64 bits). If data is stored mainly in arrays as we do in our work, the second value can be an integer, and can be 32 bits if that is the size chosen for array indices.

The way we partition a mesh onto a distributed memory machine is to have each MPI process create a data structure (one of those described in Chapter 2) and store in it a subset of the entities in the overall mesh. Partitioning begins with a subset of the elements that will reside in a given partition, and then requires that this partition also contain all entities on the boundary of those elements. The choice of elements is typically such that each element resides in exactly one partition (we call this an "element-based" partitioning), or an element may reside in multiple partitions (the most common example of this is a special case we call "ghosting" [72, 91]). A partitioning may be represented as a map from unique entities to a subset of partitions (MPI ranks). Each single mapping from a unique entity to a single partition causes a "copy" of that entity to be included in the data structure on that partition.

In order to maintain an understanding of the mesh as a whole, we need to maintain some information about which copies represent the same unique mesh entity. When working with a partitioned mesh, it is also very useful to designate one of the copies that represent the same unique entity as the owner copy, making it responsible for changes involving that unique entity [1, 4, 72]. Figure 4.2 shows the two ways in which we maintain this information. On the left, we have the PUMI implementation in which every copy stores a set of "remote copies", which are links to all other copies that represent the same unique mesh entity. This forms a complete graph of connectivity between the copies of one unique entity. What Omega_h maintains most of the time is a more sparse representation in which each entity stores a link to its owner (which may be itself, note the self-edge in Figure 4.2(right)). In both cases (PUMI and Omega_h), a remote link is a pair containing an MPI rank and an entity pointer, where the entity pointer decomposes into one or more array indices (see Section 2.4.5).

The main disadvantage of a complete set of links at every entity is the additional storage requirement and maintenance of the links during migration (see

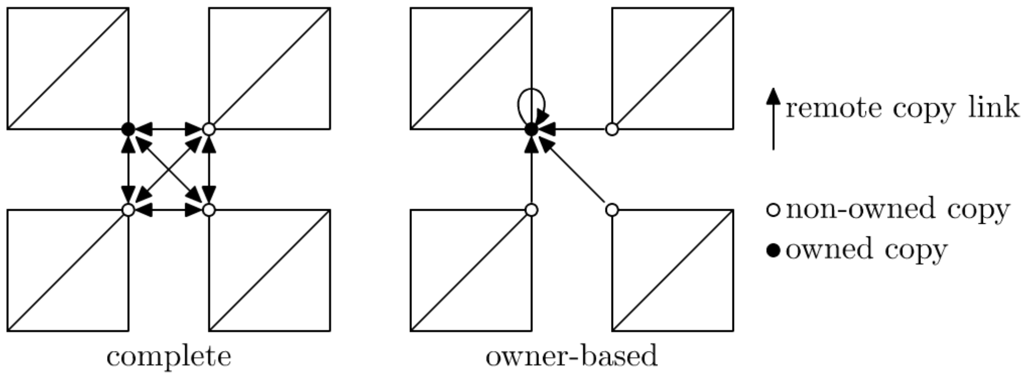

**Figure 4.2: Complete and owner-based remote links for a 4-partition mesh**

Section 4.6.2). The storage format is also more complicated because there is a variable number of links per entity, which requires some kind of indirection. In the case of PUMI, links for a single entity are allocated in their own small array and a single large array maps each entity to its associated small array of links. The advantage, as is often the case, is that this additional memory can save time in certain algorithms, for example ownership rules may be changed without communication.

## 4.5 Entity-Level Communication

A common operation in both PUMI and Omega_h is to send data in a way that can be interpreted as mesh entities communicating with one another. A good example of this would be mesh nodes communicating field values to one another to add up contributions from different mesh partitions. In a more general sense, we would like to have communications in which data is sent to and from many small objects, where there are more objects than threads.

In PCU, there is no explicit support for this, but it is straightforward to achieve by sending small messages which begin with the entity pointer of a copy (obtained from remote copies as described in Section 4.4), followed by the data aimed at that copy. When receiving data using PCU, a thread will loop over these small messages (which are packed into one large message for the entire thread) and bring up the relevant entity for each message.

In Omega_h, we have a more explicit implementation of this communication pattern in an object called the `Dist`, which is short for "distributor". The `Dist` model is that of a directed communication graph from one set of copies (the "sources") to another set (the "destinations"). The two sets may be the same, for example the copies of nodes of a mesh communicating with one another, or they may be different, for example the copies of vertices of the input mesh communicating with the copies of vertices of the adapted mesh. Each source may communicate with multiple destinations, although we assume the number of destinations per source (and the number of sources per destination) is bound by a small constant. It should be noted that both the sources and destinations are sets of copies which are distributed throughout the whole machine.

Along each edge of this communication graph, we will transmit a small amount of data. These transmissions will happen all at once. First a type of data (integer, floating-point, etc.) and a width (a very small constant, for example 3 values) are selected. Then the input values for each source are provided, and the `Dist` is responsible for relaying these to the destinations, such that each destination can readily access the data sent to it. Note that this is really the exchange algorithm from Section 4.2.5, it is simply implemented at a finer-grained level.

The `Dist` system goes through several stages in order to complete its task, as illustrated in Figure 4.3. Data begins as an array within the memory space of each source rank, with one entry per source. It is then expanded such that there is one entry per communication graph edge whose source is in this rank. The array is then permuted such that data whose destinations are on the same rank is stored contiguously. At this point, we use the MPI 3.0 exchange mechanism described in Section 4.2.5 to send out the contents of the given array. This MPI exchange then receives all the data whose destinations are on this rank into a new array whose contents are sorted by the rank of their sources. This received array is permuted such that data is sorted by destination object index. Finally (and optionally), the data may be reduced such that multiple values with the same destination become a single value (recall the common reductions in Section 4.2.2).

Figure 4.3 illustrates well the symmetry of this process, and the ease with which it can be executed in reverse. This symmetry is reflected in the program-

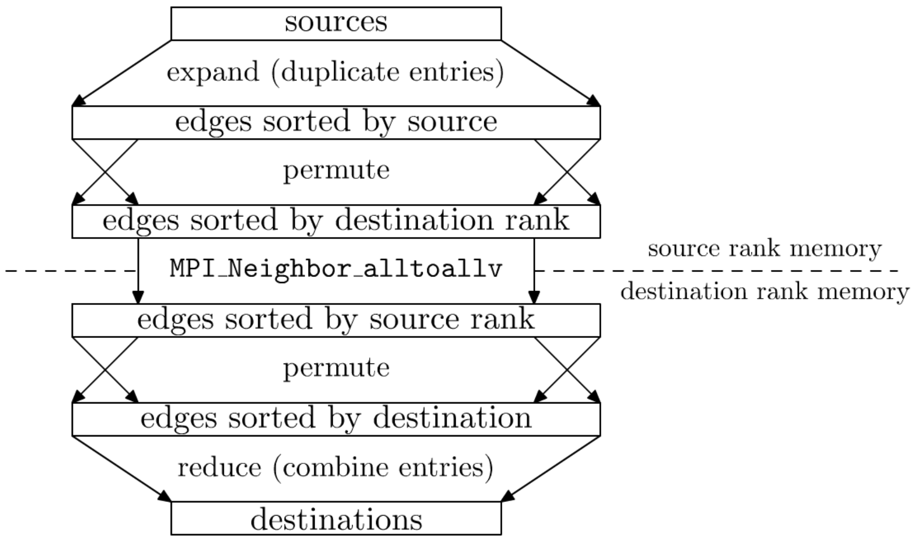

**Figure 4.3: The stages of a Distributor exchange algorithm**

ming interface. Each of the five transformations requires storing some data. The expansion and reduction steps can each be described by an array of offsets (see the compressed row format from Section 2.5.2). The permutation steps each store a permutation array. Finally, the message-passing step also stores arrays describing which portions of the input and output arrays are associated with which MPI ranks (these are also symmetric).

The generality of this interface covers a wide variety of communication needs. Most of the time, we use it such that each source has a single destination (e.g. entities to their owners), or symmetrically each destination may have a single source (owners back to entities). In order to create a `Dist` system, users need only specify the destination of each source on a given rank, where destinations are specified as pairs of rank and index (see Section 4.4). We use a shared-memory parallel sort (see Section 4.2.4) to obtain the first permutation, by sorting these destinations by their rank. We then construct the MPI information from the sorted ranks, and transmit the destination indices on their own. At the destination, we apply the mapping inversion algorithm described in Appendix A.1 to obtain the second permutation and the offsets used for reduction (this is where the small degree assumption comes into play). If there are multiple destinations per source, one need only specify the offsets from sources to the provided destination links.

Notice that all of these data manipulation steps can be performed in a shared-memory parallel fashion. In the case of a machine which has a CPU and a GPU, the GPU performs all the array transformations, and data is only copied to the CPU in the form appropriate for MPI's neighborhood exchange. After the exchange it returns to the GPU for the second half of the transformations. Despite this effective use of shared memory, it is still expensive to set up a `Dist` system. The ones which are most likely to be reused are those that Omega_h builds for communicating from copies of one dimension to their owners, and so we cache these systems the same way adjacencies are cached in Section 2.5.4.

## 4.6 Migration

As described in Section 4.4, a partitioned mesh involves careful duplication of mesh entities into "copies" in different memory spaces, and establishing the needed links between copies of the same entity. Creation of a partitioned mesh is handled by a procedure called "migration" [1, 4, 72]. It accepts as input a partitioned mesh (the input partitioning may be such that the entire mesh is in a single partition), and outputs a new partitioned mesh based on an input description of where the user would like certain mesh elements. This process is expected to take advantage of as much parallelism as possible, within the constraints of the input and output partitionings (for example, if either one has the whole mesh in one partition, portions of the migration will be inherently serial).

Following their respective styles, PUMI will modify the given mesh by incrementally adding and removing entities and remote copy links, while Omega_h will construct a new mesh structure.

The following sub-sections cover the key sub-problems involved in mesh migration:

### 4.6.1 Derive Lower Dimensional Partitionings

The user specifies only the element partitioning, and the required partitioning for lower-dimensional entities needs to be derived from it. This is a set union problem: each partition will request lower-dimensional entities adjacent to the elements it has been assigned. These requests are sent to the owners of the lower-dimensional entities in the input mesh, who will resolve duplicate requests (from multiple adjacent elements on the same partition) and derive the set of unique copies to generate.

### 4.6.2 Create and Link New Copies

For a given dimension, the new copies need to be set up and related via remote copy links. As above, the owner copy of an entity in the input mesh is responsible for setting up these links. This is where PUMI's task is more difficult due to it maintaining a complete set of remote links (see Figure 4.2). PUMI uses several stages of communication to first construct new copies, send their identities back to the old owner, and then have the old owner transmit all copy identities to all copies (all-to-all amongst copies is required to form a complete graph).

Algorithm 6 combines the full-mesh rebuild approach from Omega_h with the readability of PUMI-style communications to present an understandable implementation of new entity and link creation. A single conditional controls whether complete or owner-based links are used. The actual differences between Algorithm 6 and the actual C++ code are minor, for example PUMI only creates new copies that do not exist in the old mesh (because it is being directly transformed into the new mesh) and Omega_h transmits almost no information in the messages $m_1$ because creating a copy simply means establishing a local numbering of new copies.

Omega_h can use a subset of the `Dist` setup procedure to establish and number all new copies as described in Section 4.6.3 and illustrated in Figure 4.4. The sources are old owners and the destinations are new copies defined at first by rank only (step 3 in Figure 4.4). Their identities are transmitted back to the old owners, who then selects a new owner amongst the new copies and transmits that single identity back to all new copies (step 5 of Figure 4.4). Note that all three of these communications are carried out by a single `Dist`.

### 4.6.3 Build New Topological Adjacencies

Permanent topological adjacencies need to be constructed between different dimensions (see Figures 2.2 and 2.8). This brings about an interesting consideration, which is that we usually create adjacency information in a bottom-up fashion

**Algorithm 6:** Establish new entity copies and links of one dimension

```
input  : Old entities $E_O$, where old owners know their new ranks
input  : New entities $E_N$ with good remote links
1  foreach old owner $o \in E_O$ do
2      foreach new rank $r$ that needs a copy of $o$ do
3          Record a message $m_1$ to $r$ with data to build a copy of $o$;
4  exchange messages $m_1$;
5  foreach message $m_1$ received do
6      $c \leftarrow$ build a copy of an old owner $o$ based on $m_1$;
7      add $c$ to $E_N$ (usually implicit);
8      send the identity of $c$ back to $o$ as a message $m_2$;
9  exchange messages $m_2$;
10 foreach message $m_2$ received do
11     let $C_o$ be the set of new copies of $o$;
12     add the new copy $c$ to $C_o$;
13 foreach old owner $o \in E_O$ do
14     if not maintaining complete links then
15         select new owner $q \in C_o$;
16     foreach $c \in C_o$ do
17         if maintaining complete links then
18             send $C_o$ to $c$ in message $m_3$;
19         else
20             send $q$ to $c$ in message $m_3$;
21 exchange messages $m_3$;
22 foreach message $m_3$ received do
23     let $L_c$ be the set of links maintained for new copy $c$;
24     $L_c \leftarrow$ the links in message $m_3$;
```

(vertices are created, then edges are defined from vertices, etc.), but partition information is specified first at elements, and then derived in a top-down fashion. PUMI resolves this in a V-cycle fashion, meaning that it will first derive the new partitionings for all lower dimensional entities in a top-down fashion and then input all these partitionings to a bottom-up algorithm for actually building new entities and setting up remote links.

Unlike PUMI (and APF in particular), whose programming interfaces are based on the concept of an entity being an object, Omega_h has no concept of entity objects, only adjacency structures and associated data. This allows it to loosen constraints and build adjacencies in a top-down fashion. Each dimension $d$

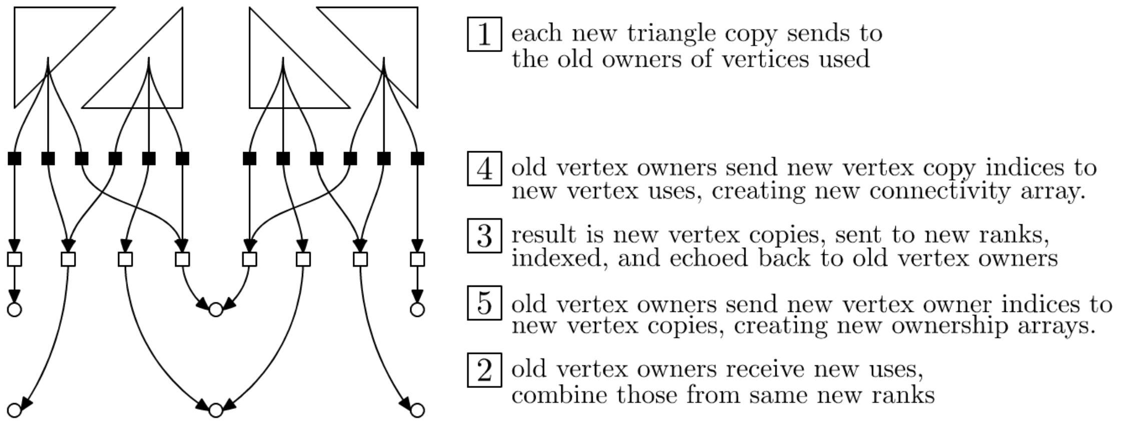
1 each new triangle copy sends to the old owners of vertices used
4 old vertex owners send new vertex copy indices to new vertex uses, creating new connectivity array.
3 result is new vertex copies, sent to new ranks, indexed, and echoed back to old vertex owners
5 old vertex owners send new vertex owner indices to new vertex copies, creating new ownership arrays.
2 old vertex owners receive new uses, combine those from same new ranks

**Figure 4.4: Omega_h migration steps for vertices based on triangles**

is handled as depicted for vertices in Figure 4.4. It first resolves the unique copies of $d$-dimensional entities as described in Section 4.6.1 (steps 1 and 2 in Figure 4.4). This results in an accurate prediction of the new identities (indices) for those entities, which can be fed back to their old owners (step 3), and forwarded again to form the new adjacency structure (step 4). It also sets up the partitioning for dimension $(d - 1)$ by sending new owners links to new copies (step 5). This can all be done with two `Dist` objects. One from new high-dimensional (e.g. triangle) copies to old low-dimensional (e.g. vertex) owners, formed in step 1 of Figure 4.4. The second from old low-dimensional owners to their new copies, formed in step 3 of Figure 4.4, and mentioned in Section 4.6.2. Repeating these steps in order of decreasing dimension, Omega_h can carry out a full migration.

## 4.7 Ghosting

As mentioned in Section 4.4, mesh partitioning can in some cases copy an element onto multiple partitions. The term ghosting typically refers to a partitioning algorithm that begins with an element-based partitioning (all elements exist on only one partition) and gradually copies layers of elements from neighboring partitions. Formally, a partition after ghosting one layer will have copies of all mesh elements adjacent to all $b$-dimensional entities it had copies of before ghosting, where $b$ is called the bridge dimension. In our work we only consider vertex bridges ($b = 0$).

The interface to PUMI's element-based migration procedure is “push-based”, meaning that each partition specifies, for each of its current elements, a single destination partition to send that element to. By definition, such a specification cannot express ghosting. Although PUMI has recently acquired a ghosting capability, it was not developed as part of this thesis and treats ghost elements in a specialized manner.

We focus here on Omega_h's implementation, which begins with a migration procedure capable of handling any amount of element duplication. The input to Omega_h migration specifies a list of element copies in the input mesh (which may be in other partitions) which define the element copies desired in the newly partitioned mesh. This way, one may request all the elements currently in the partition plus several elements in other partitions. Since the Omega_h migration procedure treats element duplication the same as duplication in lower dimensions, implementing ghosting only requires specifying the right set of elements. We form a graph from vertices to all adjacent elements (including those on other partitions) by having every vertex copy send its adjacent element identifiers to the vertex owner. The vertex owners send back the full list of adjacent elements, allowing partitions to request all elements adjacent to its current vertices.

The only care that needs to be taken in the migration procedure is to preserve the owner ranks of all entities when ghosting, because this is the only mechanism for distinguishing ghosted elements in case one needs to remove them and return to an element-based partitioning. This also gives a critical guarantee for parallelism: after ghosting, an owned entity will have all upward adjacent entities available as copies in shared memory. Any computations that need to consider entities adjacent to a central entity can be programmed without any communication, assuming all information is present in the data structure. As this is only true for owned entities, one need only communicate the result of this computation from owned entities to their other copies. This is an alternative to the dynamic migration system presented in Section 4.8.1.

## 4.8 Parallel Cavity Operations

When doing operations that account for higher-dimensional entities around a lower-dimensional entity, it is useful to have a mechanism for altering the mesh partition such that the cavity being operated on (defined by the elements adjacent to a lower-dimensional entity) is copied in its entirety onto at least one partition, so that this partition can perform the computation [48]. In this way, the code that operates within the cavity need not change when going from a serial (non-parallel) code to a scalable parallel code. Section 4.7 above hints at how ghosting plays this role in Omega_h. The equivalent system for PUMI is described in Section 4.8.1.

When the cavity operations alter mesh topology, one additionally needs mechanisms for scheduling the application of these modifications such that no two threads make conflicting modifications to the mesh, and threads are made aware of changes by other threads as needed. The following two Sections describe how this is done in PUMI and Omega_h.

### 4.8.1 Dynamic Migration

There is a system in APF called `CavityOp` which is responsible for carrying out dynamic mesh migration in order to apply a set of requested cavity operations. These operations may modify topology (e.g. MeshAdapt operators) or they may not (e.g. averaging element values to vertices). At all times, this system maintains an element-based partitioning, meaning that elements are copied onto only one partition. It begins by iterating over the mesh and performing any cavity operations for which the cavity is entirely within the local partition (we say the cavity is local). If the mesh is well partitioned, this should account for the majority of desired operations. Afterwards, it will mark any cavities which are distributed over multiple partitions and execute a single migration that aims to localize as many non-local cavities as possible. These two steps are repeated until all cavities have been operated on.

Cavities are required to be centered around a certain key entity, such as an edge or vertex. To use this system, users supply certain low-level functions to be executed. The first function accepts a central key entity and returns information about whether or not that entity represents a cavity that still needs to be operated on and whether the required cavity is local. If the cavity is not local, it indicates which low-entities need their upward adjacent entities localized. The second function is run after the first if the cavity is local, and applies the user-defined operation.

The dynamic migration system will try to localize a cavity onto the partition that owns the central key entity. Repeated migration is necessary because cavities may overlap, which creates a possibility of conflicts in which two overlapping cavities cannot be simultaneously localized by this heuristic, because their owners are different. We use a ranking of partitions to choose which competing partition has its request satisfied. This ensures at least that progress is guaranteed to be made at every iteration, because the highest ranked partition in a conflict will have its cavity localized. In practice, the number of iterations required is small and not dependent on the problem size, which makes this a scalable system.

There are a few drawbacks to the dynamic migration system, however. First, it alters the partitioning of the mesh, often significantly. Because it is focused on satisfying locality requests, it result in a poor partitioning (high surface areas) or even remove all elements from a partition. This has required us in practice to keep at least a few thousand elements per partition to avoid partitions being emptied by dynamic migration. Second, if the order of application of the cavity operations matters (as it does in the case of topological modifications), dynamic migration makes no attempt to order them intelligently; they will be applied in the order that key entities are encountered during mesh iteration and partition boundaries will by definition be acted upon later than partition interiors. This makes the results dependent on the partitioning of the mesh and the ordering of mesh entities within a partition.

Ghosting as described in 4.7 can be used instead of dynamic migration for operations that do not modify topology and where the order in which the cavities are processed does not matter. The original partition is a subset of the ghosted partition (so will never be empty), and can always be recovered afterwards. For operations which modify topology and therefore are order-dependent when cavities overlap, Sections 4.8.2 and 4.8.2.2 describe a system which depends on neither input partitioning nor traversal order of the mesh.

### 4.8.2 Independent Sets

As mentioned in Section 4.8.1, it may be beneficial to have some control over the ordering of cavity operations in the case when two or more cavities overlap. In the case of topology modifications, it is often the case that performing a modification will change topology such that overlapping modifications that used to be possible are no longer applicable. In this sense, the ordering also determines which operations will definitely be applied and which may be discarded in favor of others.

The following Sections 4.8.2.1 and 4.8.2.2 describe the algorithm used by Omega_h to select operations to apply. It aims to choose an independent set of operations (i.e. their cavities do not overlap), such that they may be applied in any order (including simultaneously) and produce the same result. This allows us to execute each modification using fine-grained shared-memory parallelism. Such an approach was suggested for GPU use by Pande et al. [92], although their implementation computed the independent set on the CPU. Recall also that independent sets are used in MeshAdapt [48] during coarsening to prevent a chain of overlapping edge collapses from removing too many mesh elements (see Section 3.3.2).

Similar to the dynamic migration system from Section 4.8.1, this algorithm may be repeated several times as each iteration discards some operations that can then be reconsidered. For example, when refining long edges as per Section 3.4.1, one iteration will split a set of long edges such that no two of them are adjacent to the same element. There usually remain long edges after these splits, so the algorithm is repeated after measuring the edges of the modified mesh. In the specific case of Omega_h, an independent set of edge collapses would first be tried, but the concept of iteration remains the same.

#### 4.8.2.1 Selection of a Set

We have to solve a graph independent set problem, with a graph whose graph nodes are possible modifications around certain key entities and the graph edges represent an overlap between their cavities, in our case meaning the key entities are adjacent to a common element. We have either vertices or edges as the key entities, and either triangles or tetrahedra for elements. Fast algorithms can construct the graph of keys that are adjacent to a common element, which we use as the basis for our conflict graph. Note that the expanded cavities used in Section 3.6.1.2 require more expensive algorithms to establish the conflict graph.

At the beginning of each pass, each key entity is annotated as either being a candidate or not based on the conditions described throughout Section 3.4. If it is a candidate, it is annotated with its output cavity quality (the minimum quality of any element that would be created by the modification). We would like to resolve conflicts in a way that prefers “better” mesh modifications, which in this case is defined by output quality.

In 1986, Luby presented a highly parallelizable algorithm for finding maximal independent sets of graphs [93]. A maxim*al* independent set is simply one that cannot be improved by adding more graph nodes to it, as opposed to a maxim*um* independent set which is NP-hard to find and has the most graph nodes of any possible independent set. We develop a variant of Luby’s algorithm that is still iterative, where at each iteration graph nodes which are local maxima of some function are added to the independent set. Each vertex can, in parallel, determine whether its function value is less than that of its neighbors, and alter its own state (whether or not it is in the set) with confidence that no neighbor will make an inconsistent decision. Luby’s original algorithm assigned random integers to each graph node at each iteration.

Instead of local maxima of random numbers, we use local maxima of output quality. Our modified Luby iteration is listed in full detail as Algorithm 7. Its parallel `for` loop will have its iterations scheduled by the current runtime (CUDA, OpenMP, etc.) onto the available hardware threads. In the extreme case, there may be enough threads for all iterations to execute simultaneously. Recall that the principles of shared memory programming described in Section 1.7.3 suggest careful control over array accesses. All arrays involved in Algorithm 7 are either read-only or write-only, and the latter (`new_state`) has each entry written by one thread only, by aligning its writes with the iterations of the parallel `for` loop.

The proof of the time complexity of Luby’s original algorithm relied on prob-

**Algorithm 7:** One iteration of independent set selection

```
input  : Conflict graph $G = (V, E)$ represented by $n$, xadj and adj
input  : Current vertex state in old_state (entries are either IN, NOT_IN,
         or UNKNOWN)
input  : Quality measure for each graph vertex in quality
input  : Unique graph node IDs in global
output: Updated vertex state in new_state
1  for $v \leftarrow 0$ to $n - 1$ do          // shared memory parallel for loop
2      if old_state $[v]$ $\neq$UNKNOWN then
3          return;
4      begin $\leftarrow$ xadj $[v]$;
5      end $\leftarrow$ xadj $[v + 1]$;
       // vertices adjacent to chosen ones are rejected
6      for $j$ $\leftarrow$begin to end $-1$ do
7          $u$ $\leftarrow$adj $[j]$;
8          if old_state $[u]$ =IN then
9              new_state $[v]$ $\leftarrow$NOT_IN;
10             return;
       // check if vertex is local maximum
11     v_qual $\leftarrow$quality $[v]$;
12     for $j$ $\leftarrow$begin to end $-1$ do
13         $u$ $\leftarrow$adj $[j]$;
           // neighbor was rejected, ignore its presence
14         if old_state $[u]$ =NOT_IN then  continue to next $j$ ;
15         u_qual $\leftarrow$quality $[u]$;
           // neighbor has higher quality
16         if u_qual >v_qual then  return;
           // neighbor has equal quality, tiebreaker by global ID
17         if (u_qual =v_qual ) and (global $[u]$ >global $[v]$) then  return;
       // only local maxima reach this line
18     new_state $[v]$ $\leftarrow$IN;
```

ability theory and changing the graph node numbers at each iteration [93]. In our algorithm, the number of iterations is bounded by the length of the longest path in the conflict graph whose nodes have monotonically increasing quality values. Although it may be theoretically possible to construct pathological meshes where this path length grows proportional to the number of elements, in practice such paths are bound by a constant. Figure 4.5 shows a histogram of the number of iterations required to find a maximal independent set during execution of a typical Omega_h

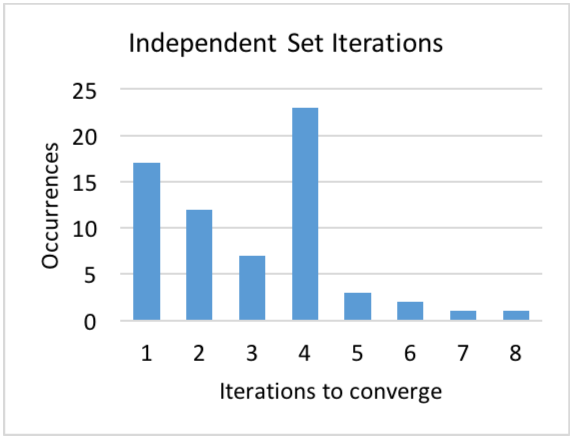

**Figure 4.5: Independent set convergence histogram**

mesh adaptation. The algorithm terminates in fewer than ten iterations in all cases, typically requiring about four iterations.

Finally, note that in line 20 of Algorithm 7 we compare graph node global IDs in the case of equal quality values. Ties could otherwise cause the algorithm to deadlock, and we prefer a deterministic tie-breaker. Thus in some cases the output is affected by the global ID values, however this does not mean it is ordering-dependent because we update global ID values in a way that is independent of the local ordering of entities.

#### 4.8.2.2 Ghosting for Set Selection

As mentioned in Section 4.7, a ghosted partition has the useful property that every owned entity (most importantly vertices and edges) has local copies of all its upward adjacent entities. All operations centered around a key entity which read information from upward adjacent entities and write information to the key entity can now be parallelized easily. Every MPI rank performs the local operation around the key entities that it owns, and then the information written to the key entities is communicated from owned copies to all other copies. For example, the worst element quality resulting from splitting an edge can be evaluated locally by the MPI rank owning that edge (because all surrounding element information is available) and then communicated to the MPI ranks that have copies of that edge with incomplete surrounding information.

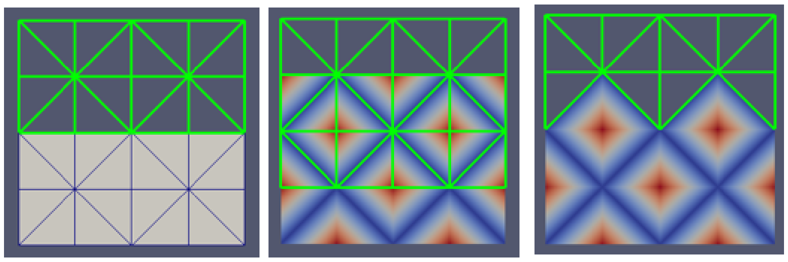

**Figure 4.6: Steps for distributed set selection: (left) non-ghosted partitions (mid) add ghost layers, compute independent set (right) trim away ghost elements not in owned independent cavities**

During each adaptation pass in Omega_h, we do what is possible without a ghost layer first. Typically, this means measuring edge lengths and determining whether any of them are too long or two short. Then a ghost layer is constructed and possible operations are evaluated as described in Section 3.4 and an independent set is chosen as described in Section 4.8.2. Once an independent set has been chosen, we return to an element-based partitioning, but one which is altered such that cavities in the independent set reside on one MPI rank. At that point, the code can proceed to apply the independent cavity modifications simultaneously and produce a new local mesh structure without further communication because entities on the partition boundary are not modified. Figure 4.6 illustrates this process at a simple partition boundary, in the case of selecting which mesh vertices ought to be collapsed.

We can create the global numberings of the new mesh based on the global numberings of the old mesh. Entities of the same dimension are numbered together. In an effort to preserve the spatial locality of the global numbering, we number newly created entities based on the numbers of their old cavity counterparts. Specifically, we designate an entity of dimension $d$ in the old cavity to represent all entities of dimension $d$ created in the new cavity. If the cavity created $r$ such entities, then we assign the value $r$ to the representative old entity. Other $d$-dimensional entities that were destroyed by the modification are assigned the value 0. Entities which stayed the same in the mesh are assigned the value 1. These values are put through a global exclusive scan operation (see Section 4.2.3), resulting in offsets for the old entities. These offsets can then be used to number the new entities. The exclusive scan is performed in the order that entities appear in the old numbering. Combined with the upward adjacency sorting mechanism described in Section 4.9.1, this results in a global renumbering mechanism that is independent of partitioning and local ordering.

This overall selection process based on ghosting allows Omega_h mesh adaptation method to be unaffected by partition boundaries, in the sense that the decision process of what modifications to apply is not influenced by the partition. The output is independent of local ordering so long as global IDs are independent of ordering as described above. Most other parallel adaptation implementations that we know of explicitly consider interior modifications first, followed by a repartitioning that allows consideration and modification of the near-boundary mesh [6, 48], making them at least partitioning-dependent.

## 4.9 Determinism

Section 4.8.2.2 described how Omega_h takes steps to select operations and number entities in a partitioning independent and ordering independent way. Omega_h is also deterministic, meaning that its output is the same if run twice with exactly the same inputs, which may not be the case if it were affected by the relative order in which distributed processes and shared threads reach a certain point in execution. That ordering is dependent on computer system state beyond the control of the program, such as other programs currently running, operating system state, or even the physical temperature of hardware cores altering their processing speed.

Each of these properties is important for software development purposes for two reasons. First, when symptoms of a bug are encountered, it is important to be able to re-run the program with additional instrumentation to determine the root cause. Non-determinism can cause symptoms to appear with arbitrarily low probability, requiring an arbitrarily large number of trials to reproduce the issue.

If a program is partitioning-independent, a highly parallel case that exhibits symptoms can be run with less parallelism for debugging purposes. As a practical example, MeshAdapt at one point had a bug which would exhibit symptoms with sufficient probability only using 4096 or more processes. Much of the time required to diagnose the issue was spent waiting for sufficient computing resources.

Conversely, if a code is partitioning-independent then one can construct regression tests which confirm that exactly the same results are produced with several different partitionings. This can catch a wide array of issues in parallel implementation. For example, if a bug causes 0.01% of the edge splits along a partition boundary to be incorrectly rejected, it may go unnoticed indefinitely for partition-dependent programs while it would be detected by a strict comparison of serial and parallel results.

Omega_h is developed and debugged using a small number of threads and processes, but has regression tests requiring the exact same results at varying degrees of parallelism. This tends to quickly catch parallel bugs, and it has been scaled to thousands of GPU CUDA cores and tens of thousands of MPI ranks without exhibiting any symptoms of bugs that depend on the degree of parallelism.

The following sections cover two more key operations that Omega_h uses to establish partitioning-independence and determinism.

### 4.9.1 Upward Adjacency Ordering

The mapping inversion algorithm in Section A.1 relies on atomic operations to determine the local ordering of upward adjacent entities, and so by itself is non-deterministic (it depends on the temporal order in which two threads attempt to access a single value). We run a post-processing step which locally sorts upward adjacent entities of a single entity by their global numbers. Recall from Section 4.8.2.2 that global numbers are independent of local ordering, so this operation likewise makes upward adjacency information independent of local traversal order and temporal thread access order. This is important because operations like floating-point summation are non-associative (see Section 4.9.2) and so higher-level operations like averaging values from elements to vertices produce slightly different values depending on the traversal order of upward adjacencies.

### 4.9.2 Order-Independent Sums

Computers typically implement floating-point numbers following the IEEE 754 standard [10]. This format represents real numbers as the nearest rational number which can be represented as $(m \cdot 2^p)$, where $m$ and $p$ are both integers within a predefined range. Varying the exponent $p$ can be interpreted as shifting the position of the decimal point, hence the term floating-point numbers. The exponent $p$ is adjusted such that the digits of $m$ capture the most significant digits of the real number being approximated. When adding a series of floating-point numbers whose exponents vary significantly, the resulting value can be different depending on the order of summation. If the values are added in descending order of magnitude, the intermediate values will all have exponents equal to or greater than the largest exponent $p_{\text{max}}$, so they will truncate all values less than $2^{p_{\text{max}}}$. When the sum follows the ascending order of magnitude, intermediate values start with the smallest exponent and gradually increase their exponent as small values are added. This more accurately accumulates digits of low significance, which may amount to a substantial difference in the final sum.

Mesh adaptation is highly sensitive to floating point values, because as described in Chapter 3 we choose to make modifications based on whether floating point values such as the length of an edge in metric space are above and below an exact threshold. Thus any perturbation could change the result of this comparison in certain cases, and produce a different mesh topologically. For our purposes, it is more important to have a consistent (ordering independent) set of operations rather than the most accurate method possible. Simple floating point summation can be performed very quickly relative to other mesh adaptation operations, so we prefer faster options so long as they are consistent. For this reason, we do not carry out a full global sort of the values into ascending order, as that would be too expensive. That said, we would prefer that the accuracy of our method be comparable to that of the non-deterministic algorithm.

We add global values using a fixed-point accumulator, meaning we select some $p_{\text{fixed}}$ and represent all intermediate values as $M \cdot 2^{p_{\text{fixed}}}$, where $M$ is an integer. We select $p_{\text{fixed}} = p_{\text{max}}$, which ensures that our accuracy is at least as good as the worst-case non-deterministic ordering, i.e. summation in descending order. The integer $M$ must have a large enough range to store the largest magnitude intermediate value (modulo $2^{p_{\text{fixed}}}$). The most common format for scientific computations is a 64-bit floating point number in which 52 bits are devoted to the integer $m$. We devote 128 bits to our intermediate $M$ values, meaning they can reliably add up to ($2^{76} > 10^{22}$) input values. By comparison, the largest-scale problems being solved at the time of this writing require summing at most $10^{12}$ floating-point values, meaning that 128 bits will remain sufficient until the maximum problem size of interest increases by an additional factor of $10^{10}$. To implement this algorithm, we need to perform two reductions: one to determine $p_{\max}$, and one to add up all the values in discrete units of $2^{p_{\max}}$. Since not all computers have built-in instructions for 128-bit integers, we sometimes have to implement intermediate values as a pair of 64-bit values emulating a 128-bit value.

The resulting method has a runtime that is small compared to the other operations of mesh adaptation and produces the exact same (bitwise consistent) values regardless of ordering. Combining this with other techniques such as upward adjacency sorting (see Section 4.9.1), we can carry out mesh adaptation such that all floating point values are partition-independent and ordering-independent, which is a prerequisite to having fully independent results for the overall adaptation algorithm.

## 4.10 Parallel Adaptation Performance

### 4.10.1 Generating Large Meshes

#### 4.10.1.1 MeshAdapt Uniform Refinement

In order to test the capability of the hybrid MPI-thread system, PCU, and the overall PUMI system supporting MeshAdapt, a 1.6 billion element mesh is created using up to 16K cores of an IBM Blue Gene/Q.

Mesh generation begins with a 4-part, 400K element tetrahedral mesh and proceeds so as to maintain a part density of 100K elements per part. Each up-scaling repartitioning uses uniform mesh refinement (see Section 3.3.1), which multiplies element counts and part counts by a factor of 8. At each step, we start with 2 processes per node and then create 8 threads per process, repartition the mesh among those threads with the help of migration (see Section 4.6), and then apply uniform refinement using those threads. The Blue Gene/Q has 16 cores per node, which is why we choose 2 processes per node so that the total threads per node equals the number of cores per node. At the end of each step, files are written out from each of the threads, which will be read in by the initial processes of the next step, where the next step allocates 8 times more nodes than the current step.

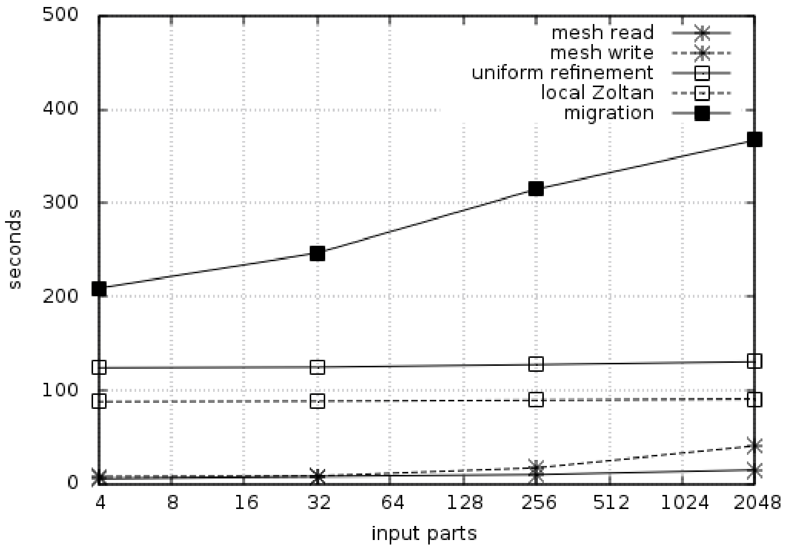

**Figure 4.7: Times for hybrid MeshAdapt uniform refinement**

Figure 4.7 shows the time consumed by each step of the PCU-supported MeshAdapt workflow as the number of parts is increased from 4 to 16K. The final step converts 2K parts into 16K parts. In this workflow, both migration and mesh file writing are being done using 8 threads per process. By comparison, file reading is done by the initial processes without multi-threading, and the times are very comparable even though each process is writing 8 times more data, so thread parallelism is achieved (the sizes of files are constant, so the times to read and write a single file should be similar). Migration shows an increase in runtime as part count is increased, which can be correlated with Figure 4.8, which shows how the number of neighbors of a mesh partition are increasing at the same rate as the migration runtime, both of which grow logarithmically with the number of elements and partitions. Recall that the exchange algorithm described in Section 4.2.5 and migration in general have runtimes proportional to the number of neighbors.

After generating this mesh, we study the overhead of threading with a simple workflow. Beginning with $P$ processes and $T$ threads per process such that $(P \cdot T)$ always equals 16K, we read the 16K-part, 1.6 billion element mesh. In terms of

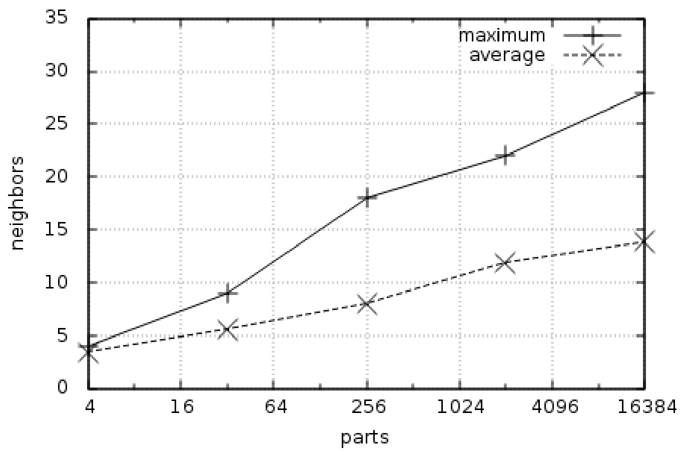

**Figure 4.8: Neighborhood increase during MeshAdapt uniform refinement**

hardware, we are using 16K cores of the Blue Gene/Q, which is 1024 nodes or one full rack. Then every thread migrates 10K elements to one of its neighbors. The resulting mesh is then written out. All operations are done in the hybrid mode, and the number of threads per process $T$ is varied. In all cases, the ideal outcome is that runtime remains the same.

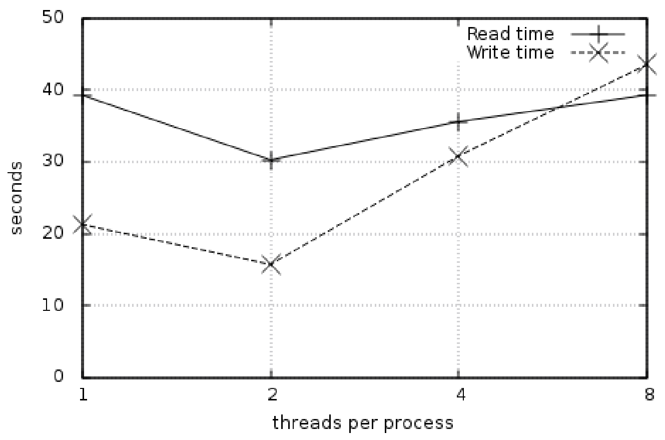

**Figure 4.9: Hybrid file IO performance**

Figure 4.9 shows the times for hybrid file IO. File IO is more prone to fluctuation because the filesystem and associated networks are shared by all users of the supercomputer. Despite this, we see file IO performance remains fairly constant as we move from process-only to hybrid operation.

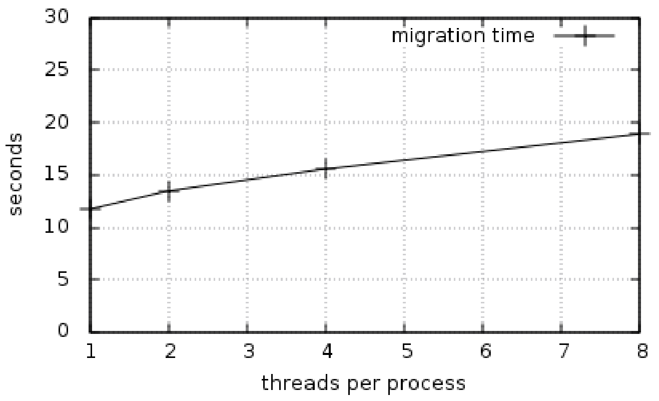

**Figure 4.10: Hybrid migration performance**

Figure 4.10 shows migration time over threads per process. Since the ideal result is constant, we see a logarithmic overhead in this part of the workflow.

#### 4.10.1.2 Omega_h Parallel Size Field Scaling

In order to carry out a study similar to the one in Section 4.10.1.1 using Omega_h instead of MeshAdapt, we choose to use general mesh adaptation with a scaled size field because Omega_h does not use refinement templates, including the uniform refinement template. We therefore parallelize the size field scaling workflow from Section 3.7.2. Our initial geometry, as illustrated by Figure 4.11, is a $4 \times 4$ array of the solder ball geometry used in Section 3.7.2. The initial mesh contains 80K elements. At each step $i$, two programs are run. The first program uses $P = 2^i$ processes to read a mesh of $2^{i-1}$ parts and use global Recursive Inertial Bisection to repartition it into $2^i$ parts [75]. The second program uses $P = 2^i$ processes to compute the implied isotropic size field, scale it with the goal of obtaining $(80 \cdot 10^3 \cdot 2^i)$ elements, and run mesh adaptation. At all times we use $\lceil 2^i/16 \rceil$ nodes such that there are at most 16 processes in a node. Note that because we use size field scaling, we can multiply by a factor of 2 at each step, whereas uniform refinement is limited to a factor of exactly 8, which can be too coarse a mechanism for achieving a desired element count.

Table 4.1 shows several performance metrics collected during this study. The

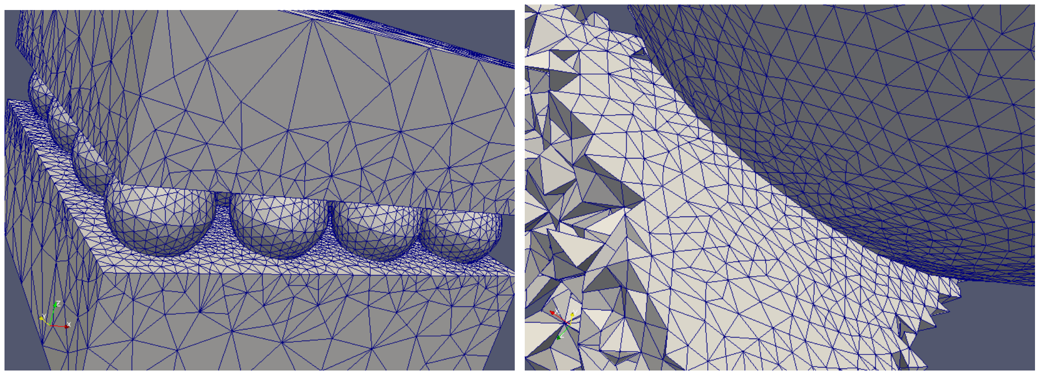

**Figure 4.11: (left) The $4 \times 4$ solder ball mesh with 160K elements, (right) One of 256 partitions of the mesh with 20M elements**

correlation between actual and target element counts is consistent with the behavior described in Section 3.7.2. Both repartitioning and adaptation times show good scalability, growing roughly logarithmically with the number of elements and parts, with the notable exception of the jump in repartitioning time to reach 8K parts. Note that here we are exercising all components of adaptation described in Section 3.4, not just refinement. Even collapses are exercised, as the edge length loop in Section 3.4.4 will carry out many collapses of short edges created during refinement (e.g. splitting the long edge of an obtuse triangle creates a very short edge). These components are all parallelized using the methods described in Sections 4.8.2, 4.8.2.1, and 4.8.2.2, and all of them scale well in this study. Memory usage per part shows a similar pattern of scalability. The speedup in the last column is the weak scaling speedup, computed as the number of processes times the parallel efficiency, where parallel efficiency is the slowdown in adaptation compared to ideal linear scaling.

### 4.10.2 Non-Uniform Size Field with Load Balancing

The test in Section 4.10.1.2 demonstrates the scalability of adapting to a uniform size field. However, many interesting load balancing and performance considerations come into play when adapting to a non-uniform size field, and this section studies that capability using Omega_h. First, recall from Section 3.5.4 that in the process of targeting a particular element count we derived a formula to predict how many output elements an input element would generate during adaptivity, based roughly on its size in metric space. We can use these values as weights in a load-

**Table 4.1: Performance metrics of Omega_h parallel scaling**

| target elements ($\times 10^3$) | actual elements ($\times 10^3$) | repart. time (sec.) | adapt time (sec.) | MB per part | parts | speedup |
|---|---|---|---|---|---|---|
| 160 | 160 | 4.52 | 398 | 149 | 2 | 2 |
| 320 | 283 | 6.46 | 362 | 125 | 4 | 4 |
| 640 | 566 | 6.26 | 409 | 133 | 8 | 8 |
| 1280 | 1146 | 6.18 | 370 | 119 | 16 | 17 |
| 2560 | 2340 | 8.13 | 385 | 117 | 32 | 33 |
| 5120 | 4615 | 8.78 | 482 | 131 | 64 | 53 |
| 10240 | 9320 | 10.5 | 422 | 110 | 128 | 121 |
| 20480 | 18784 | 10.8 | 525 | 129 | 256 | 194 |
| 40960 | 37273 | 16.8 | 544 | 138 | 512 | 374 |
| 81920 | 74847 | 22.1 | 607 | 148 | 1024 | 671 |
| 163840 | 150338 | 34.1 | 660 | 157 | 2048 | 1235 |
| 327680 | 299090 | 46.8 | 677 | 159 | 4096 | 2408 |
| 655360 | 599510 | 165 | 711 | 160 | 8192 | 4586 |
| 1310720 | 1201815 | 146 | 876 | 190 | 16384 | 7444 |

balancing problem, such that partitions are created not only on how many elements they have prior to adaptation, but also based on how many they will have at the end of adaptation. If a partition is sized based on input elements and it ends up refining much more than others, then during or after adaptation a load imbalance and possibly a memory overflow (of the memory space where this partition is) could occur. Conversely, if a partition is sized based only on output elements, and it is due for heavy coarsening, then before adaptation we again have severe imbalance and possibly a memory overflow. Thus, our load-balancing weight for an element is actually the average of 1.0 and the predicted output element count of the given element, where 1.0 represents how many elements the element is prior to adaptation. In this way we establish a partition which is halfway between the optimal partitions for input and output meshes.

The above load balancing algorithm is here referred to as predictive load balancing. After adaptation, we apply a post-processing load balance step based on the exact number of output elements. Figure 4.12 illustrates this process on a geometry that is purposely elongated to induce a 1D partitioning when our Recursive

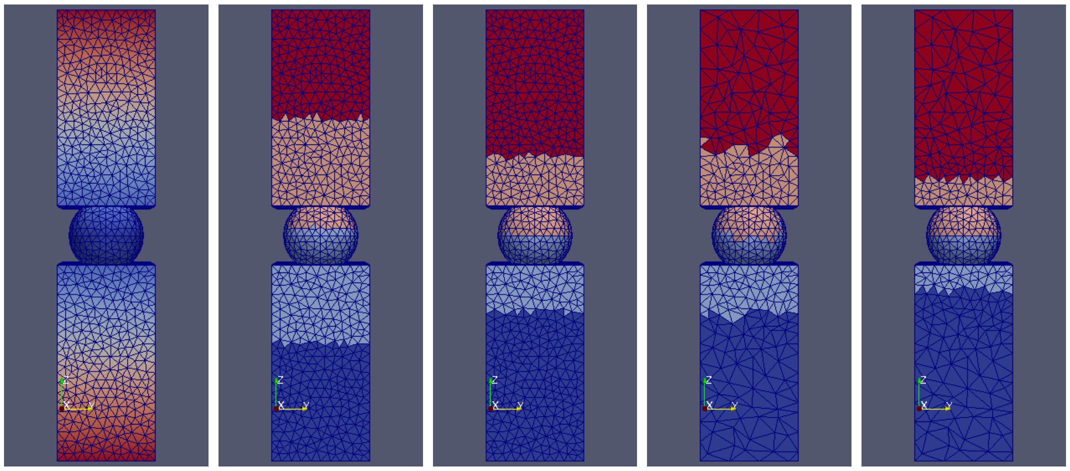

**Figure 4.12: Left to right: size field, input partitions, predictive partitions, adapted partitions, and post-balanced partitions for a simple 4-part problem**

Inertial Bisection method is used. The initial mesh is uniformly sized, and the first part of Figure 4.12 shows the desired size field, which is three times larger at the top and bottom than at the center. The following four parts of Figure 4.12 show how the mesh is initially partitioned, how the predictive load balancing increases the size of the top and bottom partitions to account for their coarsening (but only halfway), and how after adaptation the other half of the distance is covered by post-balancing to arrive at a well-balanced partitioning. In this example, the mesh is perfectly balanced at the start and end of the workflow (to within ±1 element), and the "halfway" partition used for adaptivity has an imbalance value of 1.32, where imbalance is defined as the maximum partition size divided by the average partition size.

We now take this same non-uniform size field in which the center of a solder ball geometry aims to be 3X finer than the top and bottom, and conduct a scaling study on an IBM Blue Gene/Q computer. We start with a uniformly-sized 90 million element mesh, which is partitioned into versions with 2048, 4096, and 8192 parts. A 195K element version of this mesh is shown in Figure 4.13. Then each version is adapted to the non-uniform size field using the predictive method described above. We use size field scaling to ensure the mesh has about 90 million elements after

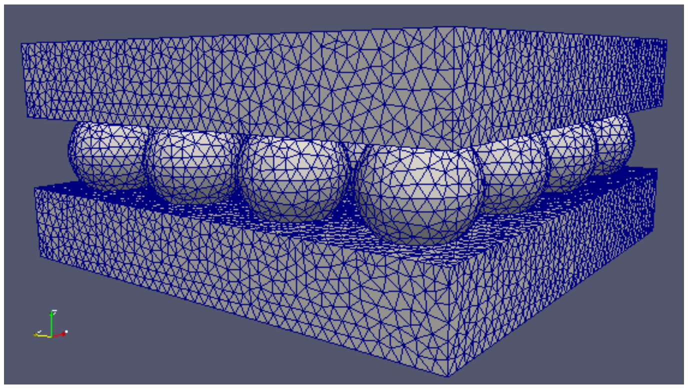

**Figure 4.13: A 195K element version of the 90M element mesh used for the non-uniform size field strong scaling study**

**Table 4.2: Strong scaling with non-uniform size field**

| #parts | predictive imbalance (max/avg) | adapted imbalance (max/avg) | adapt time (seconds) | speedup |
|---|---|---|---|---|
| 2048 | 1.658 | 1.797 | 1447 | 1.00 |
| 4096 | 1.722 | 1.874 | 828 | 1.74 |
| 8192 | 1.775 | 1.941 | 507 | 2.85 |

adaptation, and in all cases we do end up with 89.5 million elements (recall that Omega_h produces the same mesh regardless of partitioning). Since the size of the problem is constant (i.e. we are "strong scaling"), we expect runtime to decrease in proportion to the increase in parallelism. Table 4.2 shows the adaptation times, speedups relative to the 2048 part case, and imbalances after "halfway" predictive load balancing and after adaptation. Before predictive load balancing and after post-balancing, there is essentially no imbalance ($1.0 \pm 10^{-4}$). Since the goal of our predictive method is to find the middle ground between partitions optimal for the input and output meshes, then we expect that if we are successful the imbalance values after predictive balance and after adaptation will be similar, i.e. two values which vary inversely to one another are minimized by making them equal. We do see these two imbalances being close in Table 4.2.

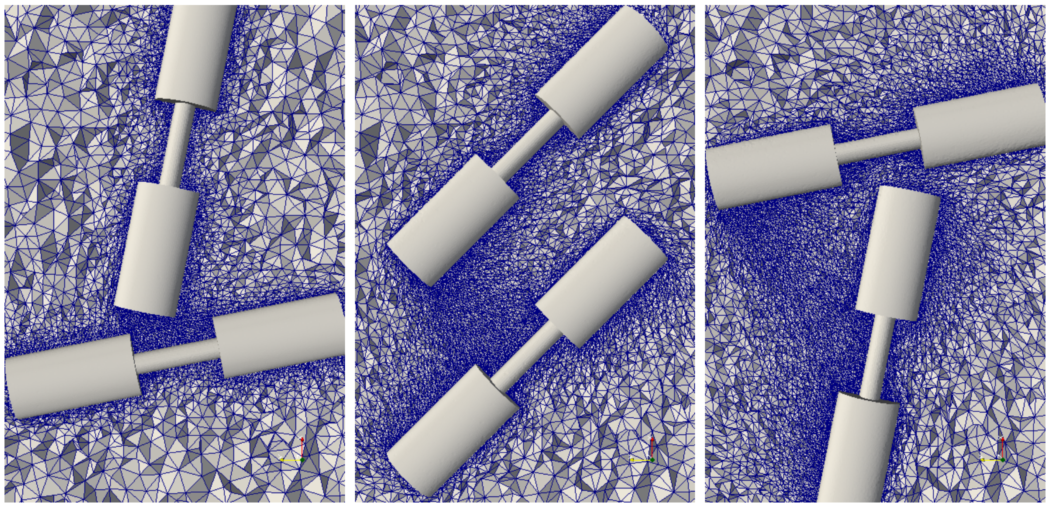

**Figure 4.14: Cutaway mesh views at steps 2, 8, and 14 of 16**

**Table 4.3: Runtime in minutes on different hardware**

| | number of OpenMP threads | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| hardware | GPU | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
| Intel Xeon 2620 v4 | | 203 | 115 | 76 | 55 | | | |
| Intel Knights Landing | | 1196 | 598 | 299 | 153 | 79 | 42 | 24 |
| NVidia K80 | 35 | | | | | | | |
| NVidia GTX 980 Ti | 15 | | | | | | | |

### 4.10.3 Moving Objects on Shared Memory Devices

In this test case, we focus on the potential for mesh adaptation to support simulations that have 3D solid bodies moving through a fluid. Mesh adaptation can be used to adjust connectivity between iterations of mesh motion, preventing tangling and inversion. Several other researchers have made good progress in applying general adaptation for these purposes [7, 94–96]. In this test case, we created a geometry consisting of two "rotors" whose ranges of motion overlap. This geometry is meant to be similar to simulations of interest such as helicopter rotors and manufacturing processes in which a fluid product is stirred.

The case is executed in a series of 16 time steps, where the following happens at each step:

1. A velocity field is prescribed for each object as vectors at mesh nodes.

2. This velocity field is spread onto the surrounding fluid mesh nodes by solv-

ing Laplace's equation using the objects and domain boundary as Dirichlet conditions.

3. The mesh is moved according to the velocity field and stops right before any element goes below 20% mean ratio quality. Using a lower threshold here causes quality repair to take much longer or fail.

4. Mesh adaptation is applied to the deformed mesh to recover edge lengths and ensure all elements are above 30% mean ratio quality. Our shape correction algorithms can have difficulties bringing all elements above a higher threshold than this. Mesh adaptation will rebuild the mesh about ten times when applied.

5. Steps 2 to 4 are repeated for any remaining motion that would previously have inverted elements. This repetition occurs ten to twenty times per time step.

We generated the initial mesh using Gmsh [97] with optimization by Netgen to remove sliver elements. All subsequent motion and adaptation is handled by our code. Figure 4.14 shows cutaway views of the mesh after time steps 2, 8, and 14 out of 16. The number of elements increases from one million to two million from start to finish.

Table 4.3 shows the runtime performance across different hardware. We first run on an Intel Xeon processor typical of current servers and clusters. Then, we run the same case on two pieces of hardware found on current supercomputers: the Intel Knights Landing CPU and the NVidia Tesla K80 GPU. Finally, we also run on a more consumer-market GPU, the NVidia GTX 980 Ti. Unlike CPUs, GPUs do not offer clear controls for using a subset of threads, so it is typical to simply show GPU speedup versus serial (in this case, 7X for the GTX 980 Ti) as opposed to some kind of scaling on the GPU. The Xeon 2620 has 8 cores and the Knights Landing CPU has 64 cores, and we see decent scaling until the number of OpenMP threads equals the number of cores, on both CPUs.

# CHAPTER 5
# APPLICATION TO ADAPTIVE SIMULATIONS

## 5.1 Workflow Integration

In order to realize its fullest potential to benefit simulations, mesh adaptation needs to be incorporated into a simulation workflow [98]. This can be implemented in a loosely coupled manner in which which a simulation runs several (or even all) steps with an unchanging mesh, then gives the solution and mesh files to an *a posteriori* error estimator, whose results are fed as files to a standalone mesh adaptation program, that outputs a new mesh file back to the original simulation program to run all over again. A loosely coupled implementation requires few modifications (sometimes none) to the simulation program, however the overhead of file I/O and switching programs introduces inefficiencies and complexities that may preclude some of the more interesting uses of mesh adaptation. By contrast, in a closely coupled system mesh adaptation and physics code are linked together into a single program, with minimal overhead in transferring data between the various components, and all components having comparable parallel efficiencies. This allows, for example, adapting the mesh after every simulation time step in a simulation that is massively parallel. In practice, all integrations of mesh adaptation into simulation programs fall somewhere along this spectrum.

A goal of this thesis has been to bring maximal value to simulation codes by integrating mesh adaptation into their workflows with the most appropriate degree of coupling. This chapter covers multiple instances of such integrations and discusses aspects unique to each of the simulation programs.

---

Portions of this chapter previously appeared as: D. A. Ibanez, E. S. Seol, C. W. Smith, and M. S. Shephard, "Pumi: Parallel unstructured mesh infrastructure," *ACM Trans. Math. Softw.*, vol. 42, no. 3, pp. 17:1–17:28, May 2016.

Portions of this chapter submitted as: D. Ibanez and M. S. Shephard, "Modifiable array data structures for mesh topology," *SIAM J. Scientific Comput.*, under review.

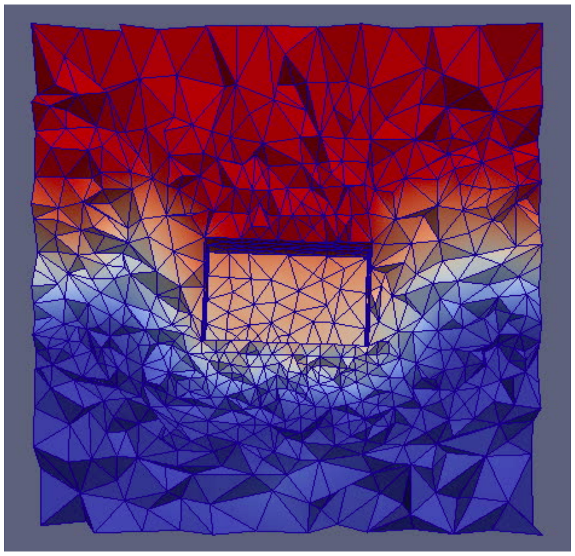

**Figure 5.1: Floating object fluid-structure-interaction adaptivity**

## 5.2 Proteus

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 5.1 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 5.2.

## 5.3 Alexa

In collaboration with Sandia National Laboratories, an experimental shock hydrodynamics application is being developed to explore the benefits of using a Lagrangian formulation with adaptivity and mesh motion. Figure 5.3 shows one of our early test cases, a triple point problem [99]. The figure compares the solution with and without adaptation included in the workflow. When adaptation is included, it is attempted after every explicit time step. Given the small amount of nodal motion per time step, relative to element sizes, each adaptivity call should be making very

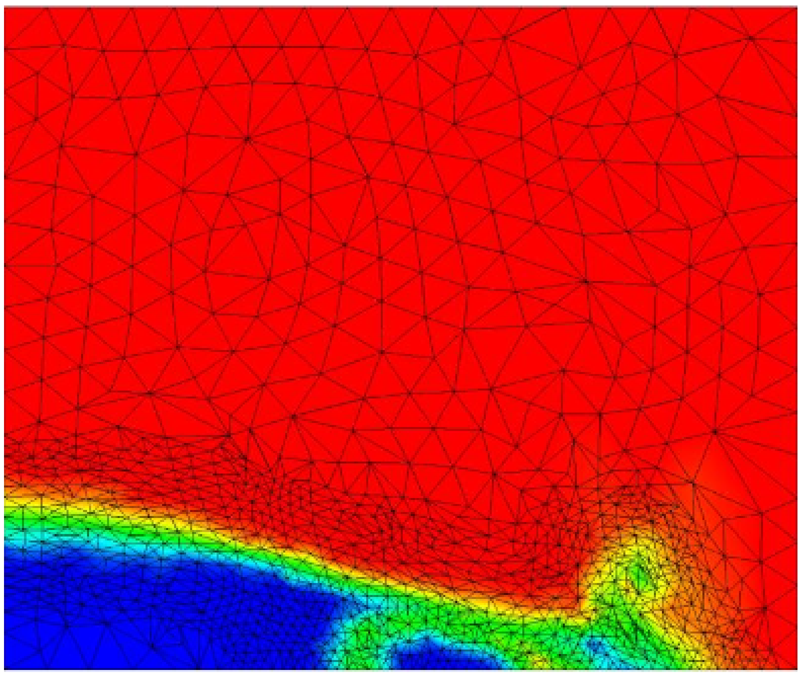

**Figure 5.2: Anisotropic mesh near fluid boundary**

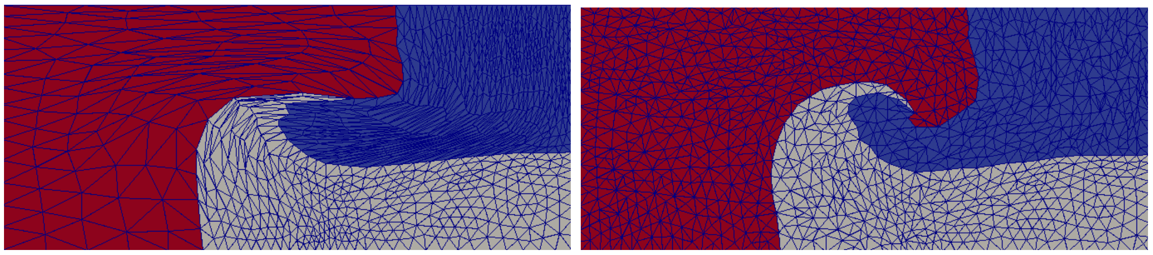

**Figure 5.3: Triple point problem: (left) purely Lagrangian (right) Lagrangian with adaptation**

minimal changes to the topology and solution, if any. The code can proceed quickly without rebuilding any structures if no mesh modification is deemed necessary after a given time step. For the triple point case, the adaptive workflow required 7722 time steps to reach the point ($t = 4$) in Figure 5.3. The non-adaptive workflow required 105,832 time steps (a 14X increase over the adaptive workflow) due to better element quality. We are just starting to develop the proper size field controls for this kind of application and its validation is ongoing work beyond the scope of this thesis. We have also developed solution transfer methods for cavity operators that satisfy certain properties, such as conservation of mass and momentum. Continued

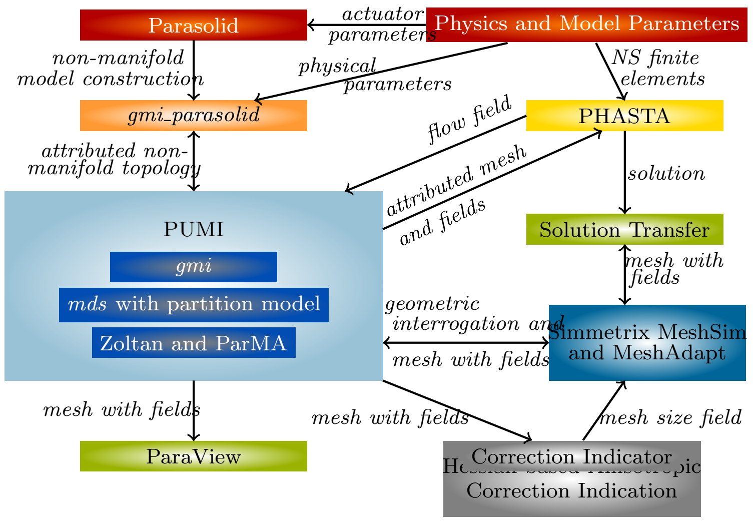

**Figure 5.4: Workflow of parallel PHASTA adaptive loop [72]**

development of transfer methods is important for bringing adaptivity to a wider range of applications.

## 5.4 PHASTA Active Flow Control

PHASTA [100, 101] is an effective implicit finite element based CFD code for bridging a broad range of time and length scales in various flows including turbulent ones (based on URANSS, DES, LES, DNS). It has been applied with anisotropic adaptive algorithms [38, 102–104] along with advanced numerical models of flow physics [105–107]. Modeling large-scale aerodynamic problems and active flow control's effects on large-scale flow changes (for instance, re-attachment of separated flow or virtual aerodynamic shaping of lifting surfaces) from micro-scale input [108–110] requires an efficient parallel adaptive workflow.

A workflow supporting parallel adaptive PHASTA flow simulations is illustrated in Figure 5.4. Figure 5.5 shows the initial and adapted mesh near the leading edge of TrapWing NASA test case [72, 111]. The Argonne Leadership Computing Facility's IBM BlueGene Q Mira system provides 4 hardware threads per core. Running this workflow in Mira system with 4 MPI processes per core, PHASTA achieved strong scaling in mesh-based computations on up to 786,432 cores using 3,145,728 MPI processes with a 92 billion element mesh [21].

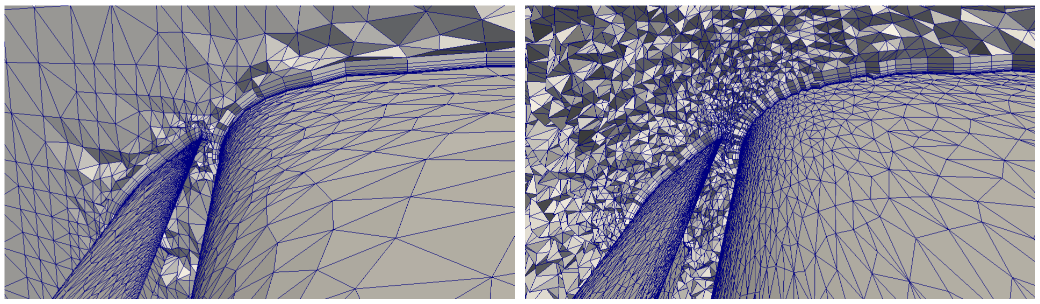

**Figure 5.5: Cut views of the initial (left) and adapted (right) anisotropic boundary layer meshes for NASA TrapWing [111]**

Key to this scaling and the efficiency of the workflow is controlling the load balance through PUMI interfaces to Zoltan and ParMA load balancing and partitioning tools. The workflow invokes load balancing after parallel mesh generation, during general unstructured mesh adaptation and before execution of PHASTA. Dynamic partitioning using a combination of ParMA and Zoltan is executed after parallel mesh generation to reach the partition sizes needed by mesh adaptation and PHASTA. During mesh adaptation, ParMA predictive load balancing procedures are used to ensure system memory is not exhausted and the resulting mesh is balanced. Lastly, before PHASTA execution, ParMA multi-criteria diffusive procedures are run to reduce both the mesh element and mesh vertex imbalance. During each of these stages the association of PHASTA solution data with mesh entities is maintained via field migration and local solution transfer procedures.

An in-memory coupling supporting a parallel adaptive PHASTA analysis loop [98] using the components depicted in Figure 5.4 is enabled through a functional interface to the FORTRAN 77/90 based flow solver. Through the use of FORTRAN 2003 *iso_c_bindings*, this interface supports interoperability with C/C++ components and supports the control of solver execution, and the interrogation and management of solver data structures.

## 5.5 Albany Adaptive Loop

Albany is a general-purpose finite element code built on the Trilinos framework [112, 113], both of which are developed primarily at Sandia National Laboratories.

This code is highly extensible, allowing the creation of new finite element numerical methods, which makes it an ideal platform for research in finite elements. The design of Albany is parallel from the start, and also includes an abstract interface for discretization storage, i.e. a mesh database, as well as various adaptivity codes.

As illustrated in Figure 5.8, PUMI was used to form a parallel adaptive loop using Albany and MeshAdapt [98,114]. This is entirely an in-memory coupling: the mesh database provides simple connectivity arrays and field data arrays to Albany for analysis, which Albany returns after a specified number of analysis steps on an unchanging mesh.

Once the data is back in PUMI structures, mesh adaptivity can be invoked on them to produce a new mesh, and solution transfer of key solution variables allows this new state to be sent back to Albany for further analysis, resulting in a self-contained, in-memory adaptive finite element code. The rich encoding of the PUMI mesh means that it is almost always a superset of the mesh information required for an analysis code. As such, we were able to convert not only to connectivity structures used internally by Albany, but also to another mesh data structure known as STK, part of the Trilinos framework. This makes PUMI more interoperable with any finite element codes involving the Trilinos framework. Figure 5.6 illustrates the initial and adapted mesh in large deformation analysis with Albany adaptive loop.

We are also using the MDS array structure to manage very large meshes in a compact and scalable way for a different Albany simulation studying the mechanical properties of computer circuitry. Figure 5.7 shows how the initial meshes used 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 (65536 cores) 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.

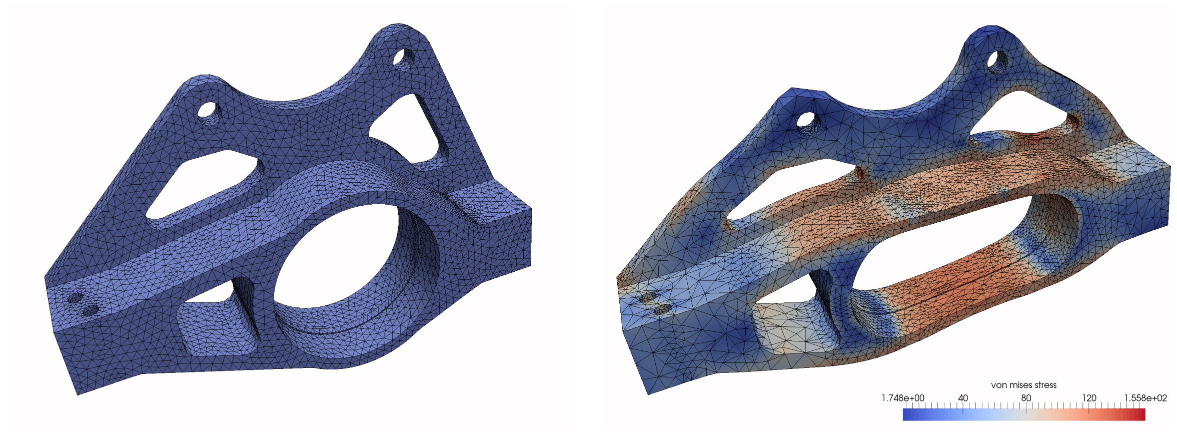

**Figure 5.6: Initial (left) and adapted mesh showing the Von Mises stress field that guides adaptivity (right) [72]**

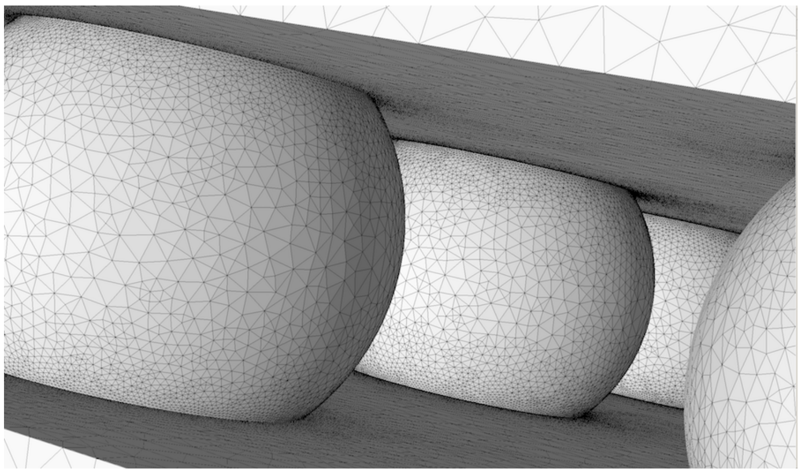

**Figure 5.7: Graded multi-material mesh to initiate large scale runs**

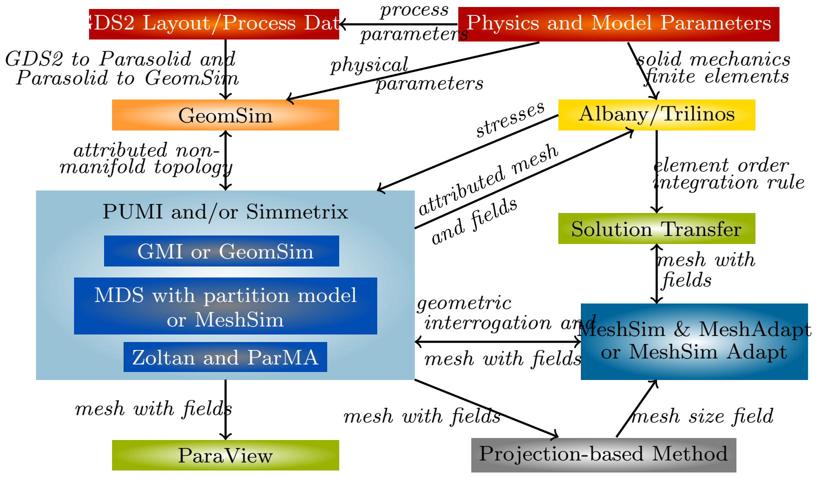
GDS2 Layout/Process Dat
process parameters
Physics and Model Parameters
GDS2 to Parasolid and Parasolid to GeomSim
physical parameters
solid mechanics finite elements
GeomSim
stresses
Albany/Trilinos
attributed non-manifold topology
attributed mesh and fields
element order integration rule
PUMI and/or Simmetrix
GMI or GeomSim
MDS with partition model or MeshSim
Zoltan and ParMA
Solution Transfer
mesh with fields
geometric interrogation and mesh with fields
MeshSim & MeshAdapt or MeshSim Adapt
mesh with fields
mesh with fields
mesh size field
ParaView
Projection-based Method

**Figure 5.8: Workflow of parallel Albany adaptive loop [72]**

# CHAPTER 6
# CONCLUSIONS AND FUTURE WORK

## 6.1 Conclusions

We presented two open-source implementations of general, conformal, cavity-based mesh adaptation with a focus on high-performance execution on heterogeneous supercomputers. Two techniques for supporting mesh adaptation using array-based data structures are presented, one of which uses rapidly modifiable arrays and the second uses batched independent sets of modifications to static arrays. Highly scalable techniques for the parallelization of mesh adaptation are developed, including the implementation of irregular sparse communications using MPI, as well as the efficient interaction of MPI with shared-memory array parallelism. One of the implementations of mesh adaptation presented is capable of carrying out general mesh adaptation in the context of the restricted programming model required to execute on GPUs, and effectively takes advantage of GPU parallelism. Some exploration of mesh adaptation operators and their scheduling is carried out, and the trade-offs of different approaches are demonstrated. Finally, the use of both implementations of mesh adaptation was demonstrated by integrating them into the workflows of several scientific simulation codes in use by the U.S. Army Corps of Engineers, Sandia National Laboratories, Boeing, and others.

## 6.2 Future Work

### 6.2.1 Convergence of PUMI and Omega_h

Section 2.5 presented the rationale for the development of Omega_h as a separate effort from PUMI, which opens the question of whether it is possible to combine the two codes in the future. A combination in this case would be defined as a strict union of their capabilities. In order to retain the ability to execute efficiently on GPUs and other shared memory hardware, a parallel `for` loop programming model such as that introduced in Section 1.7.3 would need to be used throughout. The MDS structure design which allows single additions and removals (see Section 2.4.3) would likely need to be replaced with a static design and independent set algorithms as is done in Omega_h. The resulting structure would need to be augmented to accept more than just simplices, as described in Section 2.4.5. Finally, several MeshAdapt algorithms not discussed in this thesis would need to be re-designed to use parallel `for` loops.

### 6.2.2 Snapping to Boundary Geometry

This thesis has not presented any new developments in the area of snapping (although MeshAdapt supports this in a limited capacity). This capability is required for many applications of interest, including aerodynamics simulations of rigid aircraft and mechanical simulations in which the shape of the solid objects does not significantly change, and thus the implementation of snapping in at least one of the codes presented here is considered important future work.

# REFERENCES

[1] E. S. Seol, “FMDB: flexible distributed mesh database for parallel automated adaptive analysis,” Ph.D. dissertation, Dept. Comput. Sci., Rensselaer Polytechnic Inst., Troy, NY, 2005.

[2] B. K. Karamete, R. Aubry, E. Mestreau, and S. Dey, “A novel double link structure (DLS) with applications to computational engineering and design,” in *54th AIAA Aerospace Sciences Meeting*, Jan. 4-8, 2016, pp. 1–17.

[3] W. J. Schroeder and M. S. Shephard, “A combined octree/delaunay method for fully automatic 3-d mesh generation,” *Int. J. Numerical Methods in Eng.*, vol. 29, no. 1, pp. 37–55, Jan. 1990.

[4] E. S. Seol and M. S. Shephard, “Efficient distributed mesh data structure for parallel automated adaptive analysis,” *Eng. with Comput.*, vol. 22, no. 3-4, pp. 197–213, Dec. 2006.

[5] R. Biswas and R. C. Strawn, “Tetrahedral and hexahedral mesh adaptation for CFD problems,” *Appl. Numerical Math.*, vol. 26, no. 1, pp. 135–151, Jan. 1998.

[6] A. Loseille, V. Menier, and F. Alauzet, “Parallel generation of large-size adapted meshes,” in *Proc. 24th Int. Meshing Roundtable*, Oct. 11-14, 2014, pp. 57–69.

[7] G. Compere, J.-F. Remacle, J. Jansson, and J. Hoffman, “A mesh adaptation framework for dealing with large deforming meshes,” *Int. J. Numerical Methods in Eng.*, vol. 82, no. 7, pp. 843–867, May 2010.

[8] X. Li, M. S. Shephard, and M. W. Beall, “3D anisotropic mesh adaptation by mesh modification,” *Comput. Methods in Appl. Mech. and Eng.*, vol. 194, no. 48-49, pp. 4915–4950, Nov. 2005.

[9] A. Liu and B. Joe, “Relationship between tetrahedron shape measures,” *BIT Numerical Math.*, vol. 34, no. 2, pp. 268–287, Jan. 1994.

[10] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach.* Waltham, MA: Elsevier, 2011.

[11] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, “Brook for GPUs: Stream computing on graphics hardware,” in *ACM SIGGRAPH 2004 Papers*, 2004, pp. 777–786.

[12] W.-M. Hwu, C. Rodrigues, S. Ryoo, and J. Stratton, “Compute unified device architecture application suitability,” *Comput. in Sci. & Eng.*, vol. 11, no. 3, pp. 16–26, May 2009.

[13] X.-J. Yang, X.-K. Liao, K. Lu, Q.-F. Hu, J.-Q. Song, and J.-S. Su, “The tianhe-1a supercomputer: Its hardware and software,” *J. Comput. Sci. and Technol.*, vol. 26, no. 3, pp. 344–351, May 2011.

[14] A. S. Bland, J. C. Wells, O. E. Messer, O. R. Hernandez, and J. H. Rogers, “Titan: Early experience with the Cray XK6 at Oak Ridge National Laboratory,” in *Proc. Cray User Group Conf.*, Apr. 29-30, 2012, pp. 1–21.

[15] C. Gregg and K. Hazelwood, “Where is the data? why you cannot debate CPU vs. GPU performance without the answer,” in *IEEE Int. Symp. Performance Anal. of Syst. and Software (ISPASS)*, 2011, pp. 134–144.

[16] J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor High-Performance Programming.* Waltham, MA: Elsevier, 2013.

[17] R. Hempel, “The MPI standard for message passing,” in *Proc. Int. Conf. High-Performance Comput. and Networking.*, May 18-20, 1994, pp. 247–252.

[18] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, “A high-performance, portable implementation of the MPI message passing interface standard,” *Parallel Comput.*, vol. 22, no. 6, pp. 789–828, Apr. 1996.

[19] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff, “MPI on a million processors,” in *Proc. 16th Eur. PVM/MPI Users' Group Meeting*, Sep. 7-10, 2009, pp. 20–30.

[20] W. Gropp, T. Hoefler, R. Thakur, and E. Lusk, *Using Advanced MPI: Modern Features of the Message-Passing Interface.* Cambridge, MA: MIT Press, 2014.

[21] M. Rasquin, C. Smith, K. Chitale, E. S. Seol, B. A. Matthews, J. L. Martin, O. Sahni, R. M. Loy, M. S. Shephard, and K. E. Jansen, “Scalable implicit flow solver for realistic wing simulations with flow control,” *Comput. in Sci. & Eng.*, vol. 16, no. 6, pp. 13–21, Dec. 2014.

[22] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable parallel programming with CUDA,” *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008.

[23] H. C. Edwards and C. R. Trott, “Kokkos: Enabling performance portability across manycore architectures,” in *2013 Extreme Scaling Workshop*, Aug. 15-16, 2013, pp. 18–24.

[24] (2014). “PUMI GitHub repository.” [Online]. Available: https://github.com/SCOREC/core (Date Last Accessed: Oct. 26, 2016)

[25] (2016). "Omega_h GitHub repository." [Online]. Available: https://github.com/ibaned/omega_h (Date Last Accessed: Oct. 26, 2016)

[26] D. Ibanez and M. S. Shephard, "Modifiable array data structures for mesh topology," *SIAM J. Scientific Comput.*, under review.

[27] W. Celes, G. H. Paulino, and R. Espinha, "A compact adjacency-based topological data structure for finite element mesh representation," *Int. J. Numerical Methods in Eng.*, vol. 64, no. 11, pp. 1529–1556, Sep. 2005.

[28] C. Ollivier-Gooch. (2016). "GRUMMP: Generation and refinement of unstructured, mixed-element meshes in parallel." [Online]. Available: http://tetra.mech.ubc.ca/GRUMMP (Date Last Accessed: Nov. 6, 2016)

[29] H. C. Edwards, A. B. Williams, G. D. Sjaardema, D. G. Baur, and W. K. Cochran, "SIERRA toolkit computational mesh conceptual model," Sandia Nat. Labs, Albuquerque, NM, Tech. Rep. SAND2010-1192, 2010.

[30] T. J. Tautges, R. Meyers, K. Merkley, C. Stimpson, and C. Ernst, "MOAB: A mesh-oriented database," Sandia Nat. Labs, Albuquerque, NM, Tech. Rep. SAND2004-1592, 2004.

[31] V. Dyedov, N. Ray, D. Einstein, X. Jiao, and T. J. Tautges, "AHF: array-based half-facet data structure for mixed-dimensional and non-manifold meshes," *Eng. with Comput.*, vol. 31, no. 3, pp. 389–404, Jul. 2015.

[32] R. V. Garimella, "Mesh data structure selection for mesh generation and FEA applications," *Int. J. Numerical Methods in Eng.*, vol. 55, no. 4, pp. 451–478, Jul. 2002.

[33] J.-F. Remacle and M. S. Shephard, "An algorithm oriented mesh database," *Int. J. Numerical Methods in Eng.*, vol. 58, no. 2, pp. 349–374, Jul. 2003.

[34] M. W. Beall and M. S. Shephard, "A general topology-based mesh data structure," *Int. J. Numerical Methods in Eng.*, vol. 40, no. 9, pp. 1573–1596, May 1997.

[35] I. J. Sung, G. D. Liu, and W. M. W. Hwu, "DL: A data layout transformation system for heterogeneous computing," in *Innovative Parallel Comput.*, May 13-14, 2012, pp. 1–11.

[36] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms.* Cambridge, MA: MIT Press, 2001, ch. 17.4.

[37] D. E. Knuth, *The Art of Computer Programming*, 3rd ed. Boston, MA: Addison-Wesley, 1997, vol. 1, ch. 2.2.3.

[38] A. Ovcharenko, K. C. Chitale, O. Sahni, K. E. Jansen, and M. S. Shephard, “Parallel adaptive boundary layer meshing for CFD analysis,” in *Proc. 21st Int. Meshing Roundtable*, 2013, pp. 437–455.

[39] G. Karypis and V. Kumar, “A fast and high quality multilevel scheme for partitioning irregular graphs,” *SIAM J. Scientific Comput.*, vol. 20, no. 1, pp. 359–392, Jul. 1998.

[40] M. Garey, D. Johnson, and L. Stockmeyer, “Some simplified NP-complete graph problems,” *Theoretical Comput. Sci.*, vol. 1, no. 3, pp. 237–267, Feb. 1976.

[41] E. Cuthill and J. McKee, “Reducing the bandwidth of sparse symmetric matrices,” in *Proc. 24th Nat. ACM Conf.*, 1969, pp. 157–172.

[42] M. Zhou, O. Sahni, M. S. Shephard, C. D. Carothers, and K. E. Jansen, “Adjacency-based data reordering algorithm for acceleration of finite element computations,” *Scientific Programming*, vol. 18, no. 2, pp. 107–123, May 2010.

[43] K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio, “New challenges in dynamic load balancing,” *Appl. Numerical Math.*, vol. 52, no. 2, pp. 133–152, Feb. 2005.

[44] J. Petit, “Experiments on the minimum linear arrangement problem,” *J. Exp. Algorithmics*, vol. 8, no. 1, pp. 1–29, Dec. 2003.

[45] J. Skilling, “Programming the hilbert curve,” in *23rd Int. Workshop on Bayesian Inference and Maximum Entropy Methods in Sci. and Eng.*, vol. 707, no. 1, Aug. 3-8, 2004, pp. 381–387.

[46] B. S. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey, “libMesh : a C++ library for parallel adaptive mesh refinement/coarsening simulations,” *Eng. with Comput.*, vol. 22, no. 3, pp. 237–254, Dec. 2006.

[47] B. M. Klingner and J. R. Shewchuk, “Aggressive tetrahedral mesh improvement,” in *Proc. 16th Int. Meshing Roundtable*, Oct. 14-17, 2007, pp. 3–23.

[48] H. L. De Cougny and M. S. Shephard, “Parallel refinement and coarsening of tetrahedral meshes,” *Int. J. Numerical Methods in Eng.*, vol. 46, no. 7, pp. 1101–1125, Sep. 1999.

[49] X. Li, “Mesh modification procedures for general 3D non-manifold domains,” Ph.D. dissertation, Dept. Mech. Eng., Rensselaer Polytechnic Inst., Troy, NY, 2003.

[50] L. A. Freitag and C. Ollivier-Gooch, “Tetrahedral mesh improvement using swapping and smoothing,” *Int. J. Numerical Methods in Eng.*, vol. 40, no. 21, pp. 3979–4002, Nov. 1997.

[51] P. Frey and F. Alauzet, “Anisotropic mesh adaptation for CFD computations,” *Comput. Methods in Appl. Mech. and Eng.*, vol. 194, no. 48-49, pp. 5068–5082, Nov. 2005.

[52] F. Alauzet, X. Li, E. S. Seol, and M. S. Shephard, “Parallel anisotropic 3D mesh adaptation by mesh modification,” *Eng. with Comput.*, vol. 21, no. 3, pp. 247–258, May 2006.

[53] E. Schönhardt, “Über die zerlegung von dreieckspolyedern in tetraeder,” *Mathematische Annalen*, vol. 98, no. 1, pp. 309–312, Mar. 1928.

[54] Q. Lu, “Developments of parallel curved meshing for high-order finite element simulations,” M.S. thesis, Dept. Mech. Eng., Rensselaer Polytechnic Inst., Troy, NY, 2011.

[55] R. V. Garimella, “Anisotropic tetrahedral mesh generation,” Ph.D. dissertation, Dept. Mech. Eng., Rensselaer Polytechnic Inst., Troy, NY, 1999.

[56] T. Michal and J. Krakos, “Anisotropic mesh adaptation through edge primitive operations,” in *50th AIAA Aerospace Sciences Meeting*, Jan. 9-12, 2012, pp. 1–16.

[57] F. Dassi, L. Kamenski, and H. Si, “Tetrahedral mesh improvement using moving mesh smoothing and lazy searching flips,” in *Proc. 25th Int. Meshing Roundtable*, Sep. 16-29, 2016, pp. 1–13.

[58] A. Loseille and R. Löhner, “On 3d anisotropic local remeshing for surface, volume and boundary layers,” in *Proc. 18th Int. Meshing Roundtable*, Oct. 25-28, 2009, pp. 611–630.

[59] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, “Zoltan data management services for parallel dynamic applications,” *Comput. in Sci. Eng.*, vol. 4, no. 2, pp. 90–96, Mar. 2002.

[60] C. W. Smith, M. Rasquin, D. Ibanez, K. E. Jansen, and M. S. Shephard, “Application specific partition improvement,” *SIAM J. Scientific Comput.*, under review.

[61] J.-F. Remacle, V. Bertrand, and C. Geuzaine, “A two-level multithreaded delaunay kernel,” in *Proc. 24th Int. Meshing Roundtable*, Oct. 11-14, 2014, pp. 6–17.

[62] C. Navarro, N. Hitschfeld-Kahler, and E. Scheihing, "A parallel gpu-based algorithm for delaunay edge-flips," in *The 27th Eur. Workshop on Computational Geometry, EuroCG*, Mar. 28-30, 2011, pp. 1–4.

[63] D. Engwirda, "Conforming restricted delaunay mesh generation for piecewise smooth complexes," in *Proc. 25th Int. Meshing Roundtable*, Sep. 16-29, 2016, pp. 1–13.

[64] L. N. Trefethen and D. Bau III, *Numerical Linear Algebra.* Philadelphia, PA: SIAM, 1997.

[65] F. Alauzet and P. Frey, "Estimateur d'erreur géométrique et métriques anisotropes pour l'adaptation de maillage. Partie I: aspects théoriques," INRIA, Palaiseau, France, Tech. Rep. RR-4759, 2003.

[66] C. Pain, A. Umpleby, C. de Oliveira, and A. Goddard, "Tetrahedral mesh optimisation and adaptivity for steady-state and transient finite element calculations," *Comput. Methods in Appl. Mech. and Eng.*, vol. 190, no. 29-30, pp. 3771–3796, Apr. 2001.

[67] X. Jiao and M. T. Heath, "Common-refinement-based data transfer between non-matching meshes in multiphysics simulations," *Int. J. Numerical Methods in Eng.*, vol. 61, no. 14, pp. 2402–2427, Oct. 2004.

[68] P. Farrell and J. Maddison, "Conservative interpolation between volume meshes by local galerkin projection," *Comput. Methods in Appl. Mech. and Eng.*, vol. 200, no. 1-4, pp. 89–100, Jan. 2011.

[69] F. Alauzet, "A parallel matrix-free conservative solution interpolation on unstructured tetrahedral meshes," *Comput. Methods in Appl. Mech. and Eng.*, vol. 299, no. 1, pp. 116–142, Feb. 2016.

[70] D. Powell and T. Abel, "An exact general remeshing scheme applied to physically conservative voxelization," *J. Computational Physics*, vol. 297, no. 1, pp. 340–356, Sep. 2015.

[71] M. A. Park, A. Loseille, J. A. Krakos, and T. Michal, "Comparing anisotropic output-based grid adaptation methods by decomposition," in *22nd AIAA Computational Fluid Dynamics Conf.*, Jun. 22-26, 2015, pp. 1–30.

[72] D. A. Ibanez, E. S. Seol, C. W. Smith, and M. S. Shephard, "PUMI: Parallel unstructured mesh infrastructure," *ACM Trans. Math. Softw.*, vol. 42, no. 3, pp. 17:1–17:28, May 2016.

[73] D. Ibanez, I. Dunn, and M. S. Shephard, "Hybrid MPI-thread parallelization of adaptive mesh operations," *Parallel Comput.*, vol. 52, pp. 133–143, Jan. 2016.

[74] D. Ibanez and M. Shephard, "Mesh adaptation for moving objects on shared memory hardware," in *Proc. 25th Int. Meshing Roundtable*, Sep. 16-29, 2016, pp. 1–5.

[75] H. Simon, "Partitioning of unstructured problems for parallel processing," *Comput. Syst. in Eng.*, vol. 2, no. 2, pp. 135–148, Feb. 1991.

[76] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[77] F. Roberts and B. Tesman, *Applied Combinatorics.* Boca Raton, FL: CRC Press, 2009, pp. 254–256.

[78] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for CUDA," in *GPU Computing Gems Jade Edition.* Waltham, MA: Elsevier, 2011, ch. 26, pp. 359–371.

[79] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *IEEE Int. Symp. Parallel & Distributed Process.*, May 25-29, 2009, pp. 1–10.

[80] A. D. Robinson. (2014). "A parallel stable sort using C++11 for TBB, Cilk Plus, and OpenMP." [Online]. Available: https://software.intel.com/en-us/articles/a-parallel-stable-sort-using-c11-for-tbb-cilk-plus-and-openmp (Date Last Accessed: Nov. 9, 2016)

[81] T. Hoefler, C. Siebert, and A. Lumsdaine, "Scalable communication protocols for dynamic sparse data exchange," *ACM Sigplan Notices*, vol. 45, no. 5, pp. 159–168, May 2010.

[82] T. Hoefler, R. Rabenseifner, H. Ritzdorf, B. R. de Supinski, R. Thakur, and J. L. Träff, "The scalable process topology interface of MPI 2.2," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 4, pp. 293–310, Aug. 2011.

[83] T. Hoefler and T. Schneider, "Optimization principles for collective neighborhood communications," in *Proc. Int. Conf. High Performance Comput., Networking, Storage and Anal. (SC).* IEEE, Nov. 10-16, 2012, pp. 1–10.

[84] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance analysis of MPI collective operations," *Cluster Comput.*, vol. 10, no. 2, pp. 127–143, Mar. 2007.

[85] R. Thakur and W. D. Gropp, "Improving the performance of collective operations in MPICH," in *Proc. 10th Eur. PVM/MPI Users' Group Meeting*, Sep. 29-30, 2003, pp. 257–267.

[86] T. Hoefler, A. Lumsdaine, and W. Rehm, “Implementation and performance analysis of non-blocking collective operations for MPI,” in *Proc. 2007 ACM/IEEE Conf. Supercomputing*, Nov. 10-16, 2007, pp. 1–10.

[87] T. Hoefler, P. Kambadur, R. L. Graham, G. Shipman, and A. Lumsdaine, “A case for standard non-blocking collective operations,” in *Proc. 14th Eur. PVM/MPI User’s Group Meeting*, Sep. 30, 2007, pp. 125–134.

[88] A. Ovcharenko, D. Ibanez, F. Delalondre, O. Sahni, K. E. Jansen, C. D. Carothers, and M. S. Shephard, “Neighborhood communication paradigm to increase scalability in large-scale dynamic scientific applications,” *Parallel Comput.*, vol. 38, no. 3, pp. 140–156, Mar. 2012.

[89] M. Zhou, “Petascale adaptive computational fluid dynamics,” Ph.D. dissertation, Dept. Mech. Eng., Rensselaer Polytechnic Inst., Troy, NY, 2009.

[90] D. J. Mavriplis, “Parallel performance investigations of an unstructured mesh navier-stokes solver,” *Int. J. High Performance Comput. Appl.*, vol. 16, no. 4, pp. 395–407, Nov. 2002.

[91] O. S. Lawlor, S. Chakravorty, T. L. Wilmarth, N. Choudhury, I. Dooley, G. Zheng, and L. V. Kalé, “ParFUM: a parallel framework for unstructured meshes for scalable dynamic physics applications,” *Eng. with Comput.*, vol. 22, no. 3, pp. 215–235, Sep. 2006.

[92] S. Pande, S. Biswas, and A. De, “GPU-based parallel algorithms for delaunay mesh refinement,” in *Proc. 24th Int. Meshing Roundtable*, Oct. 11-14, 2014, pp. 1–5.

[93] M. Luby, “A simple parallel algorithm for the maximal independent set problem,” *SIAM J. Comput.*, vol. 15, no. 4, pp. 1036–1053, Nov. 1986.

[94] M. Wicke, D. Ritchie, B. M. Klingner, S. Burke, J. R. Shewchuk, and J. F. O’Brien, “Dynamic local remeshing for elastoplastic simulation,” in *ACM SIGGRAPH*, Jul. 26-30, 2010, pp. 1–11.

[95] P. Clausen, M. Wicke, J. R. Shewchuk, and J. F. O’Brien, “Simulating liquids and solid-liquid interactions with lagrangian meshes,” *ACM Trans. Graph.*, vol. 32, no. 2, pp. 17:1–17:15, Apr. 2013.

[96] J. Chen, S. Li, J. Zheng, and Y. Zheng, “Parallel local remeshing for moving body applications,” in *Proc. 24th Int. Meshing Roundtable*, Oct. 11-14, 2014, pp. 1–6.

[97] C. Geuzaine and J.-F. Remacle, “Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities,” *Int. J. Numerical Methods in Eng.*, vol. 79, no. 11, pp. 1309–1331, May 2009.

[98] C. W. Smith, K. Chitale, D. A. Ibanez, B. Orecchio, E. S. Seol, O. Sahni, K. E. Jansen, and M. S. Shephard, “Building effective parallel unstructured adaptive simulations by in-memory integration of existing software components,” in *XSEDE16*, Jul. 17-21, 2016, pp. 1–6.

[99] S. D. Pino, “Metric-based mesh adaptation for 2D lagrangian compressible flows,” *J. Computational Physics*, vol. 230, no. 5, pp. 1793–1821, Mar. 2011.

[100] K. E. Jansen, C. H. Whiting, and G. M. Hulbert, “A generalized-$\alpha$ method for integrating the filtered Navier-Stokes equations with a stabilized finite element method,” *Comput. Methods in Appl. Mech. and Eng.*, vol. 190, no. 3-4, pp. 305–319, Oct. 2000.

[101] C. H. Whiting, K. E. Jansen, and S. Dey, “Hierarchical basis for stabilized finite element methods for compressible flows,” *Comput. Methods in Appl. Mech. and Eng.*, vol. 192, no. 47-48, pp. 5167–5185, Nov. 2003.

[102] O. Sahni, J. Müller, K. E. Jansen, M. S. Shephard, and C. A. Taylor, “Efficient anisotropic adaptive discretization of cardiovascular system,” *Comput. Methods in Appl. Mech. and Eng.*, vol. 195, no. 41-43, pp. 5634–5655, Aug. 2006.

[103] O. Sahni, K. E. Jansen, M. S. Shephard, C. A. Taylor, and M. W. Beall, “Adaptive boundary layer meshing for viscous flow simulations,” *Eng. with Comput.*, vol. 24, no. 3, pp. 267–285, Sep. 2008.

[104] O. Sahni, K. E. Jansen, C. A. Taylor, and M. S. Shephard, “Automated adaptive cardiovascular flow simulations,” *Eng. with Comput.*, vol. 25, no. 1, pp. 25–36, Jan. 2009.

[105] T. J. Hughes, L. Mazzei, and K. E. Jansen, “Large-eddy simulation and the variational multiscale method,” *Comput. and Visualization in Sci.*, vol. 3, no. 1-2, pp. 47–59, May 2000.

[106] A. E. Tejada-Martínez and K. E. Jansen, “On the interaction between dynamic model dissipation and numerical dissipation due to streamline upwind/Petrov-Galerkin stabilization,” *Comput. Methods in Appl. Mech. and Eng.*, vol. 194, no. 9-11, pp. 1225–1248, Mar. 2005.

[107] A. E. Tejada-Martínez and K. E. Jansen, “A parameter-free dynamic subgrid-scale model for large-eddy simulation,” *Comput. Methods in Appl. Mech. and Eng.*, vol. 195, no. 23-24, pp. 2919–2938, Apr. 2006.

[108] M. Amitay, B. L. Smith, and A. Glezer, “Aerodynamic flow control using synthetic jet technology,” in *36th AIAA Aerospace Sciences Meeting and Exhibit*, Jan. 12-15, 1998, pp. 1–19.

[109] A. Glezer and M. Amitay, “Synthetic jets,” *Annu. Rev. of Fluid Mech.*, vol. 34, pp. 503–529, Jan. 2002.

[110] O. Sahni, J. Wood, K. E. Jansen, and M. Amitay, “Three-dimensional interactions between a finite-span synthetic jet and a crossflow,” *J. Fluid Mech.*, vol. 671, no. 1, pp. 254–287, Apr. 2011.

[111] K. C. Chitale, O. Sahni, M. S. Shephard, S. Tendulkar, and K. E. Jansen, “Anisotropic adaptation for transonic flows with turbulent boundary layers,” *AIAA J.*, vol. 53, no. 2, pp. 367–378, Feb. 2014.

[112] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, “An overview of the Trilinos project,” *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 397–423, Sep. 2005.

[113] (2016). “The Trilinos Project: Sandia National Laboratories.” [Online]. Available: https://trilinos.org/ (Date Last Accessed: Oct. 26, 2016)

[114] (2016). “MeshAdapt: Parallel unstructured mesh adaptation library.” [Online]. Available: http://scorec.rpi.edu/meshadapt/ (Date Last Accessed: Oct. 26, 2016)

# APPENDIX A
# Key Algorithms

## A.1 Mapping Inversion

The shared-memory parallel map inversion algorithm used by Omega_h is not only a key building block but also an example of the limitations of typical shared-memory programming principles and how atomic operations may be unavoidable under certain constraints. In our case, a map means an array, `a2b`, describing a mapping from its accessible indices to another index space. This describes a bipartite graph (from set $A$ to set $B$) in which all graph nodes in set $A$ have degree 1, while nodes in set $B$ may have degrees greater than one.

Nodes in the second set are also assumed to have degrees which are bound by a small constant. These degrees correspond to upward adjacency degrees in mesh applications, the upward adjacency with the highest average degree is 36 triangles adjacent to a vertex in 3D, and the maximum degree in that graph may be over twice as much, so 100 is a reasonable upper bound. Although certain meshes for complex simulations have exhibited upward degrees of 300 or more, those cases also exhibit serious problems with adaptation and simulation accuracy near the high-degree entity. Although this algorithm can handle nodes in set $B$ having zero degree, this would only occur in mesh applications when there are "dangling" low-dimensional entities without upward adjacent higher-dimensional entities.

The goal for this algorithm is to construct the graph from the set $B$ to the set $A$, $f^{-1} : B \rightarrow \mathcal{P}(A)$, where $\mathcal{P}(A)$ is the power set of $A$.

Listing A.1 shows the actual C++ code used for map inversion. We are given the array `a2b` mapping each index in set $a \in A$ to an index in set $f(a) \in B$. We also given the size $|B|$ of set $B$ as `nb`. We first construct an array of degrees for each index in $B$, initialized to a degree of zero at every index (line 3). We then iterate over each ($a \in A$) and atomically increment the degree of $b = f(a)$ (lines 4 and 5). We then use an exclusive scan (see Section 4.2.3) to convert the array of degrees into an array of offsets. We call this offsets array `b2ba`, where `ba` refers to the edges

**Listing A.1: Invert map by atomics**

```
 1 Graph invert_map_by_atomics(LOs a2b, LO nb) {
 2   auto na = a2b.size();
 3   Write<LO> degrees(nb, 0);
 4   auto count = LAMBDA(LO a) {
 5     atomic_increment(&degrees[a2b[a]]);
 6   };
 7   parallel_for(na, count);
 8   auto b2ba = offset_scan(Read<LO>(degrees));
 9   auto nba = b2ba.get(nb);
10   Write<LO> write_ba2a(nba);
11   auto positions = Write<LO>(nb, 0);
12   auto fill = LAMBDA(LO a) {
13     auto b = a2b[a];
14     auto first = b2ba[b];
15     auto j = atomic_fetch_add<LO>(&positions[a2b[a]], 1);
16     write_ba2a[first + j] = a;
17   };
18   parallel_for(na, fill);
19   auto ba2a = LOs(write_ba2a);
20   return Graph(b2ba, ba2a);
21 }
```

$(a, b)$ of the bipartite graph sorted by their destination node $b$ (line 6). The total number graph edges `nba` is given by the last offset (line 7). We then construct an array with one entry for each edge (edges are sorted by destination), which will store the source node index of each edge (line 8). We again iterate over $a \in A$, this time trying to fill an entry in the edge to source array. Since the order of edges with the same destination is unspecified, we use atomic operations again to determine where edges are placed. An array of $|B|$ position indices is created to keep track of how many edges have been processed for each destination node (line 9). Each iteration of the final loop will atomically read the current position and increment it by one, thereby obtaining an allocated slot (line 13). It then writes the source vertex $a$ into this slot (line 14). The resulting graph from set $B$ to set $A$ is represented by the two arrays `b2ba` (destinations to edges) and `ba2a` (edges to sources), which are exactly the inverse of the input mapping `a2b` (sources to destinations).

Using $T$ threads, this algorithm can be expected to run in time $O(r(|A|/T) +$

$\log(|B|))$, where $r$ is the maximum degree of any node in $B$ and the $\log(|B|)$ term is introduced by the scan operation. As such, it is specifically designed for low-degree graphs, such as mesh adjacencies. There is an alternative, which is to explicitly sort the edges by destination node using a general sorting function. As described in Section 4.2.4, this could take up to $O((|A|/T)\log(|A|))$ time to run. Explicit sorting is preferable for higher degrees, but for Omega_h usage the two are comparable in runtime and we use the variant based on atomic operations.

# APPENDIX B
# Topological Ratios

## B.1 Maximum Upward Adjacencies

### B.1.1 From Vertices

We begin by proving an upper bound on tetrahedra sharing a vertex, which must be based on some assumed geometric restriction, because topology alone does not dictate any such bound. We will begin with the most straightforward restriction, that of solid angles, and correlate it to the mean ratio quality measure defined by Equation 1.6 from Section 1.4.7.

Each tetrahedron adjacent to a vertex forms a solid angle at the corner where the adjacency occurs. For a given vertex, the sum of these solid angles for all adjacent tetrahedra cannot exceed $4\pi$, the solid angle of a sphere. Conversely, if there are $n$ tetrahedra adjacent to one vertex, then one or more of the tetrahedra will satisfy Inequality B.1, where $\mathbf{\Omega}$ is the solid angle of the relevant corner.

We assume that the way to maximize the mean ratio of a tetrahedron that has one solid angle equal to $\mathbf{\Omega}$ is to have its other three corners form an equilateral triangle.

Since the mean ratio is scale-invariant, we can consider this without loss of generality for a tetrahedron $(o, a, b, c)$ where $o$ is the center of a unit sphere and $(a, b, c)$ are on the surface of that sphere, and form an equilateral triangle as shown in Figure B.1. Let $\theta$ be the angle $\angle oab = \angle obc = \angle oca$, Intuitively, as $\theta$ increases from zero to $\frac{2}{3}\pi$, the solid angle $\mathbf{\Omega}$ at $o$ monotonically increases from zero to $2\pi$. The exact relation between $\mathbf{\Omega}$ and $\theta$ is given by Equation B.2 using an intermediate $\phi$ (the dihedral angle between any pair of triangular faces meeting at $o$).

$$\mathbf{\Omega} \leq \frac{4\pi}{n} \tag{B.1}$$

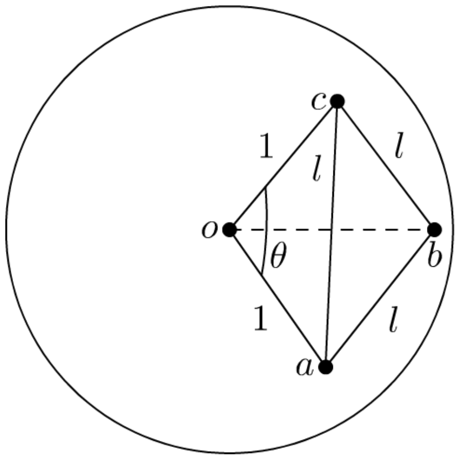

**Figure B.1: Maximizing quality versus solid angle**

$$\begin{aligned}\phi &= \arccos\left(\frac{\cos\theta - \cos^2\theta}{\sin^2\theta}\right)\\ \mathbf{\Omega} &= 3\phi - \pi = 3\arccos\left(\frac{\cos\theta - \cos^2\theta}{\sin^2\theta}\right) - \pi\end{aligned} \tag{B.2}$$

Let $l = 2\sin\left(\frac{\theta}{2}\right)$ be the length of any cord $(a, b)$, $(b, c)$, or $(c, a)$. We can use properties of equilateral triangles and isosceles tetrahedra to derive the mean ratio quality of this tetrahedron in terms of $l$ as shown in Equation B.3. The ranges for valid tetrahedra are $\theta \in [0, \frac{2}{3}\pi]$ and $l \in [0, \sqrt{3}/2]$, in which quality varies monotonically with solid angle.

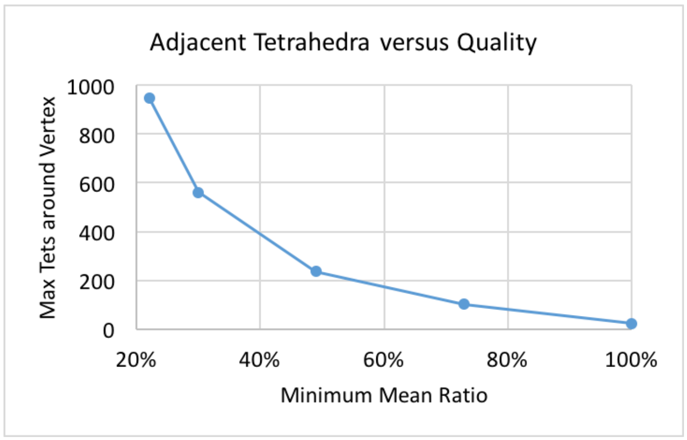

**Figure B.2: Maximum vertex-tetrahedron degree given a minimum tetrahedron mean ratio**

$$\begin{aligned}
h &= \tfrac{\sqrt{3}}{2} l \\
r &= \tfrac{1}{3} h = \tfrac{1}{2\sqrt{3}} l \\
R &= 2r = \tfrac{1}{\sqrt{3}} l \\
A &= \tfrac{\sqrt{3}}{4} l^2 \\
H &= \sqrt{1 - R^2} = \sqrt{1 - \tfrac{1}{3} l^2} \\
V &= \tfrac{1}{3} AH = \tfrac{1}{4\sqrt{3}} l^2 \sqrt{1 - \tfrac{1}{3} l^2} \\
l_{\mathrm{MS}} &= \tfrac{1}{6}(3l^2 + 3) = \tfrac{1}{2}(l^2 + 1) \\
\eta^3 &= \frac{V^2}{\gamma^2 l_{\mathrm{MS}}^3} = \frac{2^3}{4^2 \cdot 3} \frac{l^4 (1 - \frac{1}{3} l^2)}{\gamma^2 (l^2 + 1)^3}
\end{aligned} \tag{B.3}$$

In conclusion, if we ensure that all tetrahedra in a mesh have quality $\geq Q_{\min}$, then we also guarantee that no vertex in the mesh can have more than some $n_{\max}$ tetrahedra adjacent. We can use the extreme case in Figure B.1 to plot this relation, as shown in Figure B.2

A similar yet much simpler analysis, applied to triangles from the origin to the edge of a unit circle, results in the plot given by Figure B.3.

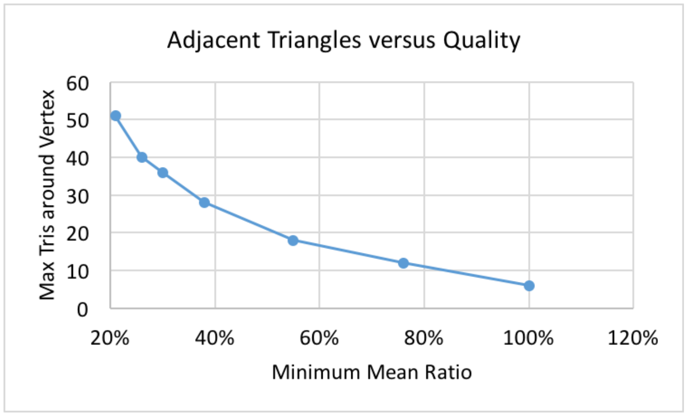

**Figure B.3: Maximum vertex-triangle degree given a minimum triangle mean ratio**