A GENERIC MESH DATA STRUCTURE WITH PARALLEL APPLICATIONS

BY

WILLIAM KENNETH COCHRAN JR.

B.S., Wake Forest University, 1999

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2009

Urbana, Illinois

Doctoral Committee:

Professor Michael Heath, Chair
Professor Robert Dodds
Associate Professor Jeff Erickson
Professor Laxmikant Kale
Abstract

High performance, massively-parallel multi-physics simulations are built on efficient mesh data structures. Most data structures are designed from the bottom up, focusing on the implementation of linear algebra routines. In this thesis, we explore a top-down approach to design, evaluating the various needs of many aspects of simulation, not just the implementation of a matrix-vector product. With this as motivation, we have developed a generic data structure that both provides efficient linear algebra subroutines by optimizing the computation at a fine-grained level and allows for rapid, reusable implementations of complex geometric algorithms. We demonstrate both through various experiments including directly measuring the efficiency of matrix-vector multiplication; implementation and analysis of a multi-frontal indefinite direct solver; approximation of the medial axis; and the development of a hybrid, two-phase mesh partitioner. The efficiency of matrix-vector multiplication is compared against a theoretical value derived from a simple model of computing hardware. The direct solver uses our data structure to remove a search step normally required for pivoting in indefinite solvers. We demonstrate that pairwise pivoting may have advantages over partial pivoting for ill-conditioned sparse matrices arising from meshes. We also present a novel, parallel algorithm that consistently approximates the medial axis of a domain of arbitrary dimension. By leveraging our data structure, a single implementation can be used for any type of mesh (e.g., 2-D, 3-D, space-time, or mixed element). Finally, we develop a hybrid approach to mesh partitioning in parallel. Using the medial axis of the mesh, large features are separated and partitioned independently using a geometric partitioner. In this way, complex domains are broken down into pieces that are better suited for geometric partitioning.
To my wife
Acknowledgments

After too many years, I am finally finished with school. And like all achievements, mine required a great deal of sacrifice, support, and encouragement from many friends, mentors, and loved ones. Those that helped me are too many to name and I cannot hope to repay their generosity in full. At the risk of leaving some out, I would like to thank a few people personally.

First, I would like to thank my committee. My discussions with Professor Jeff Erickson helped focus the final results of my medial axis algorithm. Thanks also go to Professor Sanjay Kale’s CHARM++ group. Their help and patience allowed me to learn about production parallel supercomputing code. I would like to extend a great deal of thanks to my advisor, Professor Mike Heath. He allowed me to tackle a large problem, let me direct the study of that problem, and gave me the time and freedom to complete my work. He showed me how much fun, and exasperating, science can be.

The Center for the Simulation of Advanced Rockets richly deserves my thanks. Not only did they offer me financial and computing support, but they exposed me to the problems facing large-scale engineering simulation. The staff, especially Damrong Guoy and Bob Fiedler, had infinite patience with me and valued my opinion from the start.

Professors Bob Plemmons and Todd Torgersen of Wake Forest University also merit credit for this thesis. Thanks to a wonderful experience working in scientific computing one summer a decade ago, I am not a lawyer today. I would also like to thank the staff of Information Services at the Wake Forest University Baptist Medical Center. Not only did they hire me in the first place, their flexibility allowed me to complete my undergraduate degree while remaining gainfully employed.

I would like to thank my father. He was the first to show me what a computer could do. I still remember spending weekend afternoons programming a Commodore VIC-20 in our living room almost thirty years ago. He pulled no punches in teaching me to program. He showed me sorting algorithms, databases, and, on one occasion, posed the knapsack problem to me—all before I turned ten. He also had the best toy I could ever imagine: a programmable calculator.

My mother also deserves my deepest appreciation. She understood my capabilities and never let me compromise my ability. Without her unwavering support, I would never have graduated college, let alone apply to graduate school. Thanks to her, I am not answering phones in technical support somewhere.
Finally, I would like to thank my wife. We did it! One day we will look back fondly on our time in graduate school. I always want to remember that I would never have started without you. The only reason I had the nerve to finish was you. Now that we have five decades of schooling between us, let’s see what the real world is like.

Champaign, Illinois
July 2009
Table of Contents

List of Tables ................................................................. ix
List of Figures ............................................................... x

1 Meshes ................................................................. 1
  1.1 Supercomputers ......................................................... 2
  1.2 Philosophy ............................................................ 3
  1.3 Jargon ................................................................. 4
  1.4 Goals ................................................................. 7
    1.4.1 Parallel Software Engineering ................................ 7
    1.4.2 Computational Geometry ..................................... 8
    1.4.3 Linear Algebra ................................................ 9
    1.4.4 Grand Scheme .................................................. 9

2 Generic Parallel Mesh Data Structures ............................ 11
  2.1 Generic Implementation ............................................. 13
    2.1.1 Design Criteria ............................................... 15
    2.1.2 High-Speed Memory Considerations ........................... 20
  2.2 Other Efforts in Mesh Data Structures ......................... 21
    2.2.1 Mesh Data Structures ........................................ 22
    2.2.2 Solver Data Structures ...................................... 24
    2.2.3 Generic Implementations .................................... 25
  2.3 Mesh Interface ...................................................... 26
    2.3.1 Containers and Iterators .................................... 26
    2.3.2 Recursive Data Structures .................................. 27
    2.3.3 Action-Method Interface ..................................... 29
    2.3.4 Future Work .................................................. 30
  2.4 Optimization ......................................................... 32
    2.4.1 Compiler-Only Optimization ................................ 32
    2.4.2 Future Work .................................................. 38
  2.5 Matrix-Free Computation .......................................... 38
    2.5.1 Experimental Determination of Sustained Matrix-Vector
          Product Performance .......................................... 39
    2.5.2 Results ....................................................... 40
    2.5.3 Future Work .................................................. 43
  2.6 Parallel Algorithms ................................................ 44
  2.7 Parallel Scalability ................................................ 46
    2.7.1 Parallel Efficiency .......................................... 46
    2.7.2 Surface Effect ............................................... 49
    2.7.3 Measuring Scalability ....................................... 50
  2.8 Reading Serial Meshes .............................................. 52
    2.8.1 Algorithm ..................................................... 52
    2.8.2 Scalability Study ............................................ 55
    2.8.3 Future Work .................................................. 55
## List of Tables

4.1 Importance of various indirect measures on computation rate (products per second). .......................... 114  
4.2 Importance of various indirect measures on computation rate (rows per second). .......................... 114  
4.3 Correlation coefficient matrix for cut dual edges and number of neighbors. .......................... 116
List of Figures

1.1 A quadrilateral has two topological dimensions since it can be divided in two by a curve which has one dimension. At left, the figure is embedded in a space with two geometric dimensions. At right, the figure is embedded in a space with three geometric dimensions. .................................................. 5
1.2 Mesh, in blue, and corresponding dual mesh, in red. ............... 7

2.1 Traditional computation cycle involving many different representations of meshes. ............................................................ 12
2.2 Algorithm for “skinning” a mesh. ........................................... 13
2.3 Generic surface mesh extraction routine in C++. ....................... 15
2.4 High level algorithm for large-scale engineering simulation. ...... 16
2.5 Non-naturally parallel data structure in action. Local data are processed in red ovals. Parallel data management occurs in green ovals [50]. ................................................................. 17
2.6 Sparse matrix-vector multiply with indirection. ......................... 21
2.7 Sparse matrix-vector multiply with “dense” core. ...................... 21
2.8 The recursive data type Recursive actually generates five separate data types. This recursion has the effect of unrolling the factorial so the compiler can perform the multiplication. ........... 28
2.9 An example action interface dual_partition implementation in the mesh data types. The last line is an example implementation of the dual_partition action using the ParMetis GeomKPart method. ......................................................... 30
2.10 Hypothetical complex solver constructed through the action-method interface. ................................................................. 31
2.11 Various meshes used in optimization experiments. .................. 34
2.12 Wall clock time reduction of parallel program using compiler optimization with input of 140,000 tetrahedra. ....................... 35
2.13 Wall clock time reduction of parallel program using compiler optimization with input of 450,000 tetrahedra. ....................... 36
2.14 Wall clock time reduction of parallel program using compiler optimization with input of 2,700,000 tetrahedra. ................. 37
2.15 The maximum rate of computation is relatively independent of problem size. ............................................................. 41
2.16 As currently implemented, our data structure is around 12% efficient compared to optimal CRS style sparse matrix-vector multiply. 42
2.17 Sparse matrix-vector multiply routine for Laplacian stencil. ....... 43
2.18 In the conjugate gradient method, only the test for convergence needs to be parallelized explicitly. .................................... 45
2.19 For mesh-based computation, as the number of compute nodes in a simulation increase, the time to completion reaches a minimum then starts to increase. .................................................. 47
2.20 Scalability results in mesh-based computations are often reported in number of cells per compute node. 49
2.21 A surprising number of mesh partitions have a reduced communication volume due to their location, even on a cube. 50
2.22 As problems get larger, these curves tend to move higher and to the right. 51
2.23 Algorithm to read arbitrarily large serial meshes in parallel environment. 54
2.24 Overall performance of reading serial meshes from disks never exceeds 6500 cells per second, no matter how many processors are used or how large the mesh is. 56
2.25 As problem size increases, algorithm exhibits some gains in efficiency. 57

3.1 Medial axis of a simple rectangle. 63
3.2 As $\epsilon \to 0$, $h$ is not altered, but for $\epsilon = 0$, $h$ disappears. 65
3.3 Wave propagation models illustrate difficulties in computing medial axis. 66
3.4 For any $y$ that generates a non-minimal $\beta$, there must be another surface point $w$ that generates a smaller $\beta$. 69
3.5 Inefficient algorithm to compute points on medial axis. 70
3.6 Sharp features do not need to be resolved by the mesh. 71
3.7 Interior distance and surface point approximation computed via marching method. 76
3.8 Centroid 4 is associated with the surface centroid $a$. It should be associated with $b$, indicated by the red dotted line. The dual graph (in blue) is derived from a mesh, partially illustrated in gray. 77
3.9 Dual edge detection algorithm. 79
3.10 Hairs introduced by discretization. 80
3.11 Hair, in green, governed by very small portion of surface, in red. 81
3.12 For dual edges close to set boundary and at low angle to medial hair, algorithm detects medial axis of discretization, not underlying surface. 81
3.13 Identification of stable dual edges. 83
3.14 Medial node averaging method for orienting facets taken from mesh. 84
3.15 Complete algorithm to compute medial axis. 84
3.16 Parallel interior distance and surface point approximation computed via marching method. 88
3.17 Complete algorithm to compute medial axis in parallel. 89
3.18 Medial axis computed on 64, 128, 192, and 256 compute nodes. 90
3.19 Detail of medial axis approximation (right) of 450,000 cell mesh (left). Bottom shows cutaway. 91
3.20 Effects on computed medial axis of varying derivative threshold $\tan \theta$ (left) with cutaway (right). 93
3.21 Detail of medial axis computation around bottom of fin. 94
3.22 Detail of medial axis computation around bottom of fin. 95
3.23 Detail of medial axis computation around bottom of fin. 96
3.24 Great variance in cell volume can lead to poorly resolved medial axis ($\tan \theta < 4$). 97
3.25 Wall clock time for computing dual mesh. 98
3.26 Wall clock time for computing dual mesh. 100
3.27 Wall clock time for computing interior distance and surface point fields. .................................................. 101
3.28 Wall clock time for computing interior distance and surface point fields. .................................................. 102
3.29 Wall clock time for identifying and smoothing medial facets. .............................................................. 103
3.30 Wall clock time for identifying and smoothing medial facets. .............................................................. 105
3.31 Wall clock time for entire medial axis computation. ................................................................. 106
3.32 Wall clock time for entire medial axis computation. ................................................................. 107

4.1 High level algorithm for large-scale engineering simulation given in Figure 2.4. ........................................ 110
4.2 Matrix-vector product algorithm with overlapped communication. .................................................... 112
4.3 Examples of a sliced cube. .............................................. 113
4.4 For most scalable applications, load balance appears correlated with matrix-vector multiply time. .......... 115
4.5 As number of neighbors increases, the rate of computation on each node decreases. ........................... 116
4.6 As the sample space becomes smaller, disregarding good meshes, negative correlation between maximum number of patches and computation rate increases. ........................................ 117
4.7 Differences between partitioners can be difficult to detect. Averages of columns given by solid line. ........ 118
4.8 Disjoint partition from Cartesian nested dissection. ................................................................. 120
4.9 Medial axis features identified by intersection of three or more facets. .................................................. 123
4.10 Algorithm to identify features. .............................................. 123
4.11 Mesh (top) and features of medial axis detected by algorithm (bottom). ............................................. 125
4.12 Cut-away view of mesh (top) and features of medial axis detected by algorithm (bottom). .................. 126
4.13 Detail of fin in star grain. .............................................. 127
4.14 Updated marching method to compute number of cells between facet and surface, with additions shown in red. .............................................. 128
4.15 Interior distance and surface point approximation computed via marching method. .......................... 129
4.16 Algorithm to complete rough partitioning of dual mesh. ......................................................... 130
4.17 Rough feature partitioning. .............................................. 131
4.18 Complete topological partitioning algorithm. ............................................................. 132
4.19 On three compute nodes, thirteen actual medial features are detected as nineteen. .......................... 133
4.20 Features detected independently on distinct compute nodes (top) are resolved into single features. Bottom left shows facets colored by partitioning. Bottom right shows facets colored by feature. .............................................. 134
4.21 Simple addition to ADDFACETTOFEATURE to track which nodes on remote compute nodes are in which local feature. .............................................. 134
4.22 Global feature identifier unification algorithm. ............................................................. 135
4.23 Parallel merge feature algorithm. .............................................. 135
4.24 Example of mesh partitioned with hybrid partitioner. Discolorations are artifacts of the rendering process. .............................................. 137

5.1 Pairwise pivoting allows for overlapping elimination of entries in a matrix. Elements with the same index can be eliminated simultaneously. .............................................. 144
5.2 Resulting matrix structure after nested dissection. $M_{1,1}$ and $M_{2,2}$ have this same structure. ........................................ 145
5.3 Multifrontal elimination stabilized by partial pivoting. ........ 146
5.4 Elimination implementation using constant time access of STL containers. ....................................................... 147
5.5 Parallel Gaussian elimination with partial pivoting. ........... 151
5.6 Algorithm to compute number of partial pivots performed by UMFPACK given permutation matrix. ......................... 152
5.7 As $\kappa^2$ approaches 6, more and more rows require pivoting. 154
5.8 For pairwise pivoting, number of pivots quickly reaches a maximum. 154
5.9 As number of pivots increases, so does number of non-zeroes in factors. .............................................................. 155
5.10 As number of pivots increases, so does number of non-zeroes in factors. .............................................................. 157
Scientific simulations often require the solution of very difficult equations on complex domains. Often the equations do not have a closed form solution and the mathematical descriptions of the domains are too cumbersome to work with. To solve these problems, the domain is discretized and the solution to the equations is approximated on the discretized domain. The discretized domain is referred to as a mesh.

For instance, consider the ordinary differential equation $y' = f(y(t), t)$ with $t$ a non-negative real number, $t \in [0, \infty)$, and with $y(0) = 1$. The derivative term can be approximated by the first order finite difference equation

$$y'(t) \approx \frac{y(t+h) - y(t)}{h},$$

where $h$ is some positive real number. Substituting Eq. 1.1 into the ordinary differential equation and solving for $y(t+h)$ yields Euler’s method: $\bar{y}(t+h) = \bar{y}(t) + hf(\bar{y}(t), t)$ with $\bar{y}(0) = y(0)$. Since we know $\bar{y}(0)$, we can compute $\bar{y}(h) \approx y(h)$. By iterating this equation, we obtain an approximate solution to the differential equation at $t = 2h, 3h, \ldots$ a mesh. This style of computation, approximating on a mesh, has been used with great success for many years.

Then came computers. While much work has been done to program computers to solve differential equations exactly, computers are far better at approximating solutions on meshes. A computer program to implement Euler’s method is short, executes quickly, and can fill hard drives with data in a matter of minutes or hours. This is also true for more sophisticated methods. These sophisticated methods lead to a system of simultaneous equations to solve instead of the one equation in the example. To that end, a great deal of work has been put into optimizing a computer’s ability to solve such systems. Nowadays, a programmer can take a mesh, derive the system of equations from an approximation method, and solve them with any of the widely available software...
libraries. We will argue that developing software in this fashion is at odds with the process of approximating on meshes.

A mesh is intricately linked to the method of approximation. Some methods, like Euler’s method, require points at regular intervals while others need various shapes to perform computations. Mesheres can also be categorized by many other characteristics such as conformal/non-conformal, two-dimensional/threedimensional, and structured/unstructured. The type of mesh used to describe the domain dictates what kind of approximation can be used. In the (very) trivial example above, divorcing the mesh from the method makes little sense because the mesh is intricately used in the computation.

State-of-the-art solvers rely on the mesh to accelerate computing a solution. As solvers become more sophisticated, they require more information about the mesh to produce fast, accurate answers. In this thesis, we seek to re-marry the mesh back to the computation. We develop a mesh solver.

1.1 Supercomputers

As pointed out above, computers can generate vast amounts of results from approximations on meshes. On the assumption that “more is better,” these approximations have grown larger and more complex. We now use computers for dynamic simulations, approximate solutions of difficult equations on complex domains with the hope of replicating or predicting the result of a physical experiment.

Accurate dynamic simulations require large meshes, large in the sense that there are many simultaneous equations to solve. And these equations must be solved many times. Consider the parameter \( h \) in Euler’s method given above. By definition, as \( h \to 0 \), the finite difference approximation approaches the derivative term. Under reasonable assumptions, smaller \( h \) leads to greater accuracy at each step of Euler’s method. Unfortunately, a smaller \( h \) means it takes more iterations, and more computer time, to reach a particular \( t \) in the solution. Simulation complexity can quickly exceed the abilities of even a powerful personal computer, especially if the simulation is in three dimensions.

Thus, parallel supercomputers were invented. Larger problems drive the need for faster supercomputers, which in turn require larger problems. Without the larger problems some of the vast array of processors will fall idle and the supercomputer is under-utilized. Modern supercomputers can be thought of as many thousands of powerful personal computers connected by very efficient networks. The goal of connecting so many computers is to solve problems thousands of times larger or thousands of times faster or some combination of both. As problems increase in size, it quickly becomes apparent that generating and managing large problems for supercomputers is a lot more complex than just making the problem larger.

Algorithms that work well on a workstation may not be the optimal choice
for a supercomputer. The primary reason for this is data locality. On a workstation, all data can be accessed in (roughly) the same amount of time. Each data point in a mesh can be accessed in the same amount of time as any other. On a supercomputer, this is not the case. Some data points in a mesh can be accessed quickly while others require more time. Orders of magnitude more time. Worse still, most of the memory of a supercomputer is not random access. The programmer must coordinate the memory access explicitly. This environment is referred to as a parallel supercomputer or distributed memory supercomputer.

Developers write one software application that is run simultaneously on each compute node of the parallel supercomputer. Since each compute node is running the same application but with different subsets of the problem data, this environment is often called single program multiple data or SPMD environments. In an SPMD environment, the programmer uses a message-passing library to coordinate memory access.

In summary, any discussion of mesh-based dynamic simulations must include a discussion of supercomputers. And any discussion of supercomputers must consider this data locality problem. Some algorithms lend themselves to computing in a parallel environment. Those algorithms that maximize the number of simultaneous floating point operations performed by a supercomputer while minimizing the total time the simulation runs are said to have good parallelism. Of these algorithms, some are capable of performing this feat as input size varies with supercomputer size. They are said to be scalable. If an algorithm is not scalable, using said algorithm on a supercomputer is not a good idea.

State-of-the-art techniques for managing mesh data, including solving systems of equations, are scalable. Divide-and-conquer algorithms such as those used in mesh partitioning are also scalable. Matrix-vector multiplication is wonderfully scalable. In this thesis, we seek to make these operations faster and demonstrate elusive unconditional stability for asymmetric multi-frontal solvers in parallel. We develop a set of scalable mesh solvers.

1.2 Philosophy

Computational science is a study in optimization. Given the choice between algorithm $A$ and algorithm $B$, choose the fastest. What is meant by fastest? That depends on the specifics of the problem. More often than not, it is impossible to determine which algorithm is best without bench testing dozens of algorithms with your simulation. Even for small simulations, it can take weeks or months to write the code necessary to perform each bench test. Anyone who has tried to choose a good preconditioner for an iterative solver is familiar with this process.

What about large simulations? The reason we run large simulations is because small simulations are insufficient. This insufficiency is hard to quantify, or we would just perform “error correction,” and our small simulation would
be sufficient. To complicate matters, large simulations are often the product of many developers and theorists. Each part of the simulation code is intimately connected to every other in ways that are not obvious, even to those who wrote the software. Swapping between algorithms can be tedious, fraught with error, and not in the budget. Therefore, the choice of an algorithm sometimes rests on the scientific equivalent of a slick sales brochure.

This is not to say that algorithms chosen in this way will necessarily lead the computation astray or extend computation times beyond some reasonable limit. Quite the contrary, excellent results can be obtained in this fashion. Most ineffective algorithms can be removed from consideration using simple thought experiments. But are the excellent results optimal? Answering this question will often lead to a re-write of large sections of code for one reason or another. Older programming languages are easy to misuse by a summer intern just learning how to program. Programming decisions made by seasoned developers just so the code will run may turn into long term design choices. Everyone who has ever written a computer program knows how to write that program better.\footnote{Except, maybe, Kazushige Goto}

The problem is there are three forces at work: algorithm design, software design, and software implementation. Unfortunately, the third, which is only tangentially related to the first two, can dominate the decision making process unless great care is taken. For instance, consider a simulation with two different meshes that need to use the same linear solver. In the first mesh, boundary conditions are given as displacements; in the second, as pressures. The very data structures used to store the boundary conditions have to be fundamentally different. This leads to significant differences in software implementation that must be translated between. More code means more places for bugs to hide and exacerbates the implementation problems hinted at above.

State-of-the-art programming languages have mechanisms to minimize this impact. By presenting programming interfaces rather than data structures, these languages allow for extremely broad application of algorithms to problems without the entanglement issues described above. Techniques such as parametric polymorphism allow programmers of varying capabilities to add or remove portions of code without affecting unrelated aspects of the overall simulation. In this thesis, we seek to harness these techniques to allow maximal code reuse and rapid prototyping for even large scale simulations. We develop a set of generic scalable mesh solvers.

### 1.3 Jargon

The ideas presented in this thesis extend work from various fields of computational science. As such, many of the terms are overloaded, used by different people in different fields to mean different things. Consider the sentence “Element 12, a non-linear element, has 2 nodes on node 2.” This should be parsed
Figure 1.1: A quadrilateral has two topological dimensions since it can be divided in two by a curve which has one dimension. At left, the figure is embedded in a space with two geometric dimensions. At right, the figure is embedded in a space with three geometric dimensions.

to mean “Element 12 (a mesh construct), a non-linear element (a discrete approximation of a portion of space), has 2 nodes (another mesh construct) on node 2 (a processing unit).”

This contrived example demonstrates how convoluted algorithm descriptions can become. To complicate matters further, we develop all of our algorithms in a mesh neutral fashion. That is, all algorithms described in this thesis work on meshes of arbitrary type—two-dimensional, three-dimensional, “mixed element,” etc. In order to describe the algorithms, we employ the mesh-neutral vocabulary described here.

First, there are two different dimensions that we speak of, geometric dimension and topological dimension. When we use geometric dimension, we refer to the minimum number of vectors that span the space in which an object is embedded. Topological dimension is defined recursively. By definition, a point in space has topological dimension 0. An object has topological dimension $n$ if it can be separated by an object of topological dimension $n - 1$. For instance, the quadrilateral depicted in Figure 1.1 has topological dimension two because it can be separated into two pieces by a curve, regardless of the geometric dimension of the space in which the quadrilateral is embedded. As the figure at right shows, the geometric dimension can be larger than the topological dimension. The opposite does not hold. When we use the term dimension without geometric or topological, we mean both—that is, the topological and geometric dimension are the same.

In the example given at the beginning of this chapter, the mesh was defined to be a series of points. We refer to the points as nodes. Whenever the term node is used without decoration, we always mean a point in space relevant to the mesh computation. The nodes themselves are related to other points in the mesh through computational dependencies. Returning to the example, point 2h
cannot be computed without point \( h \) which, in turn, relies on point 0. These relationships are captured by other types of mesh objects such as edges or faces.

Generally, the term *edge* is used to describe a mesh object that is bounded by nodes, the term *face* is used to describe a mesh object that is bounded by edges, etc. It is not difficult to see that if an object is bounded by another, then it can be separated by the same. Thus, edges have topological dimension 1, faces have topological dimension 2, etc. Unfortunately, much of this terminology is cumbersome in describing the mesh neutral algorithms given later.

Rather, we adopt a recursive approach to naming mesh object. Mesh objects of the highest topological dimension in a mesh, say \( n \), are called *cells*. Objects of the second highest degree, \( n - 1 \), are called *facets*. Finally, objects of the third highest degree, \( n - 2 \), are called *connectors*. We keep the term *edge* to mean a mesh object of topological dimension 1.

Here are two examples. A three-dimensional simplicial mesh is made up of tetrahedral cells. Each facet is a triangle. A connector is an edge. For a two-dimensional mesh composed of quadrilaterals and triangles, each cell is either a quadrilateral or triangle. Each facet is an edge. And so forth.

Individual processing units in a supercomputer are sometimes referred to as “nodes.” This usage may be losing favor as new supercomputers are using processing units with multiple cores and boards with multiple processing units, giving each “node” in the network topology the ability to run multiple computing processes simultaneously. Software packages such as Charm++ [72] further muddy the definition of “node” by allowing a developer to “over partition” a problem into many threads per processor, where each thread can migrate from one processor to another to improve load balance. In spite of this confusion and despite the possible loss of real world analog, we use the term *compute node* to refer to a process that is given a rank in MPI [107].

We also reserve the term *element* to mean a discrete approximation of a variational form of a differential operator, as in the finite element method. We used the term “mixed element” above to refer to a mesh, as is common use when discussing particular types of dynamic simulations. In this case, “mixed element” means that the domain is broken into elements of different types. The mesh itself is composed of cells of different shapes. Henceforth, we will not use the term “mixed element” to refer to meshes.

Throughout this thesis we also refer to a *dual mesh*. The dual mesh of a mesh is one where the topological dimensions of the mesh are reversed: a mesh cell is represented by a dual mesh node, a mesh facet is represented by a dual mesh edge, etc. For example, the mesh in Figure 1.2 is given in blue. The dual mesh is given in red.

Finally, we generalize the term *medial axis* for the sake of simplicity. Technically, a medial axis is a curve, an object having only one topological dimension. Since we are describing algorithms that work on domains of many dimensions, we extend the term medial axis to include its relatives of higher topological
1.4 Goals

This thesis comprises several aspects of computational science: parallel software engineering, computational geometry, and linear algebra. To read from beginning to end without a guide or map is like navigating an unfamiliar metropolitan area with only a compass. Without points of reference, the compass is almost useless.

To help navigate the pages of this thesis, we provide the following points of reference, divided into the aspects mentioned above. These points will help keep the complete scope of the thesis in perspective in the face of the mountain of detail presented.

1.4.1 Parallel Software Engineering

Merriam Webster defines engineering as

- **2 a**: the application of science and mathematics by which the properties of matter and the sources of energy in nature are made useful to people
- **b**: the design and manufacture of complex products

Civil, mechanical, electrical, aerospace, and other engineering fields clearly use the definition **2 a** of engineering. This anecdote demonstrates how far software engineering should evolve. Take another example. Some define engineering as “minimizing cost within constraints.” This definition gives a metric for how
good an engineer is. Contrast this with the tongue-in-cheek Ninety Ninety rule attributed to Tom Cargill:

The first 90% of the code accounts for the first 90% of the development time. The remaining 10% of the code accounts for the other 90% of the development time. [17]

Like most humorous juxtapositions, this comparison is unfair. Most engineering projects across all fields finish overdue and over budget. Even still, software engineering should be patterned after other engineering fields—two distinct phases, both governed by engineers with different qualifications. Civil engineering has design and build. Mechanical engineering has prototype and production. Software implementation has two natural phases: the first ninety percent and the last ten. We believe that Cargill’s assessment comes from the tendency of software engineers to misunderstand the importance of either phase of implementation. There is a painful veracity to this statement. For this reason, we will on occasion refer to the pernicious 10%.

What are the first ninety and last ten? Unfortunately, this thesis does not delve into a general solution for this. Rather, in the next chapter, we hope to illustrate a clear divide between these two phases of implementation in parallel applications. The use of programming techniques such as parametric polymorphism will help demarcate the two phases and shed some light on the difficulty of parallel programming.

1.4.2 Computational Geometry

In Chapters 3 and 4, we explore and use a curious oddity: the medial axis. The medial axis is a useful representation of the “shape” of a domain. For this reason, we employ the medial axis in computing a partitioning of a mesh. Computing the medial axis for general domains is an ill-posed problem, sensitive to noise. Luckily, we do not need a bulletproof general solution, as our domains generally do not display degenerate behavior.

Many algorithms compute adequate approximations to the medial axis. Generally, these algorithms start with a point cloud. The points are assumed to be sampled from the surface of a domain. The approximate medial axis computed from the points is that of the sampled surface. These algorithms are sophisticated but, for one reason or another, do not fit our application. We therefore explore computing the medial axis from a simpler yet more complex perspective.

The algorithm we develop is simpler in that we start with a better description of the problem domain, namely a mesh. Having a complete description of the surface of the domain greatly simplifies the process of computing a medial axis. Moreover, having an interior mesh aids in producing topologically correct results.

The algorithm we develop is more complex in that we compute the medial axis in parallel. And since the medial axis will be used in computing a mesh
partitioning, the computation must take place without an intelligent distribution of the mesh across the memory of a supercomputer in advance. We develop an “embarrassingly parallel” test for distance to the medial axis after a march through the mesh.

1.4.3 Linear Algebra

In Chapter 5, we develop an algorithm for parallel Gaussian elimination with partial pivoting. The fundamental problem with partial pivoting in parallel is scheduling communication of pivots and the subsequent matrix row swap. In order to coordinate the computation, we present an algorithm that is sub-optimal serially but has distinct advantages in parallel.

The algorithm is sub-optimal in the number of pivots: in the elimination of a column sub-diagonal, more than one row may be pivoted to ensure the diagonal entry has the largest absolute value of the elements remaining to be eliminated. While this seems like a “bad idea,” the mesh data structure described in the next chapter allows row pivoting by swapping memory pointers, limiting the computational impact.

In order to make this algorithm attractive in parallel, the sub-optimal algorithm provides a schedule for the necessary communication to perform the pivoting. The scheduling is designed to minimize wait times to coordinate pivots. In this case, a sub-optimal serial algorithm can be very useful in parallel environments.

We organize the elimination process as a multi-frontal calculation. Pivoting rows in a multi-frontal calculation requires the use of a very expressive mesh data structure to ensure that matrix elements not in the current scope of the recursion are handled appropriately.

1.4.4 Grand Scheme

In the end, this thesis demonstrates how science can be accomplished using an expressive parallel mesh data structure. We explicitly identify two benefits of our approach. First, some applications will benefit from the ease with which parallel applications can be developed. For instance, the mesh partitioner can be implemented with nearly any type of data structure. But, for example, if the boundary values are not handled by the partitioner, then they must be handled by the simulation developer. Using our data structure, the boundary values are handled by the partitioner, reducing the amount of supporting software necessary to use the partitioner.

The second benefit comes from relying on the data structure to provide extra information not immediately available as inputs to software libraries. For instance, the linear solver described in Chapter 5 requires an expressive mesh data structure. Normally, the nested dissection based multi-frontal method
requires square submatrices to function properly. Pivoting in Gaussian elimination requires entire rows (or columns) of a matrix to be exchanged, violating the recursive structure of the data processing. In order to move only the correct data during pivots, the solver relies on the expressive nature of the mesh data structure.

These two benefits aid development of large scale simulation codes. When developing these codes, designers choose among several options for each part of the process. The choice of partitioner, preconditioner, or solver often dictates how large sections of code must be written—specifically translating the output of one part into the input of another. Translating between data types is error prone, time consuming, and often masks other subtle problems. For instance, using a parallel mesh partitioner is more complicated than just invoking a function. A mesh may need to be loaded into memory, translated into a dual mesh (which better represents calculation dependencies), and placed into the data structure taken as input to the mesh partitioner. The output then must be translated back to the mesh data structure of the overall computation and the elements of the mesh moved around the memory of the supercomputer.

This thesis demonstrates that there is a better way. Beyond unification of programming interfaces, a unification of assignment of software library responsibilities is needed. For instance, mesh partitioning usually happens on the dual mesh, a better representation of the computation discussed at length in Chapter 4. Once the dual mesh is constructed, it is partitioned using a graph partitioning algorithm. The responsibility of translating to and from the dual mesh is generally the responsibility of the simulation programmer, since representations of a mesh are subtly different between simulations. Therefore, the translation software must be implemented many different times. This thesis demonstrates there is a better way to divide responsibilities such as construction of the dual mesh.

Simulation codes are generally divided into autonomous phases. The solver is implemented independently of mesh management routines. Unfortunately, the mesh management and solver routines rarely have mechanisms to handle boundary values. The programmer must spend a good deal of time writing software ensuring mesh boundary data are moved from the mesh manager to the solver appropriately. The responsibility of handling “edge cases” or “supporting data” is passed from the library writer to the simulation designer. We suspect that much of the “remaining 10%” Cargill described is handling this responsibility.

Much of the work necessary to handle this type of supporting code can be eliminated with the appropriate choice of development tools. We show that it is possible to describe and implement efficient algorithms that do not rely on a specific mesh data structure and let advances in compiler technology bring together the supporting code. We begin by describing our mesh data structure and interface.
2 Generic Parallel Mesh Data Structures

Peace, commerce, and honest friendship with all nations – entangling alliances with none.
— THOMAS JEFFERSON, Inaugural Address (1801)

The key to performance is elegance, not battalions of special cases.
— JON BENTLEY AND DOUG McILROY (1993)

Most high performance simulation codes that use computational meshes rely on simple, contiguous arrays to store data. For instance, a three-dimensional mesh with \(n\) nodes and \(c\) cells is stored in an \(n \times 3\) floating point array and various \(c \times x\) integer arrays, where \(x\) varies with the type of cells in the mesh. This approach lends itself well to construction of matrices to be solved by libraries. Unfortunately, these data structures are awkward for many of the supporting computations of engineering simulation, especially in parallel environments. For instance, identifying common nodes and cells between processors requires several supporting arrays used to reference the mesh data indirectly, as well as a voluminous amount of code to maintain and use these arrays. Other supporting code, such as preconditioner computation, node numbering, and mesh partitioning, often explicitly builds more expressive data structures, such as elimination trees and dual meshes, in order to achieve their goals.

The reason for this involves how algorithms interact with data structures. Operations such as matrix assembly require random access to node data and iterative access to cell data. For applications that spend the bulk of computation solving matrices, multiple-array storage makes sense. Due to the contiguous nature of the arrays, both random and iterative accesses are achieved through one addition instruction, often optimized by the compiler for a given computer architecture.

Unfortunately, other supporting computations need a more descriptive data structure. Most mesh partitioners, for example, require translation of the mesh from a multiple-array data structure into one optimized for handling sparse graphs. Such translation is necessary because these algorithms require graph traversal access to the mesh. Other supporting calculations require still other types of access to the mesh, or the data stored on the mesh require more translations. These translations can be time consuming.
Until recently, simulation designers have accepted the cost of translation since they were considered one-off costs or part of a pre-processing phase. For most simulations, a single mesh was used, and the supporting computations could be considered inconsequential relative to the total simulation time. However, computers have become larger, simulation techniques have advanced, and meshes often must evolve in order to improve various numerical properties [53]. As the meshes change, they must be smoothed, patched, or regenerated entirely. This requires re-partitioning, recalculation of any preconditioners, or other support computation once considered inconsequential. The bulk of what was considered pre-processing is now a standard part of the iterative simulation cycle depicted in terms of mesh maintenance in Figure 2.1.

One way to improve overall performance of these complex simulations is to redesign the mesh data structure. However, any change to these data structures may increase the amount of memory consumed by an application to store the mesh. When working on serial simulations, or even parallel simulations with a modest number of processors, this is a significant concern. However, modern parallel architectures routinely have a large amount of RAM available per computing node, typically one gigabyte or more. Large scale simulations can leverage this massive amount of memory to accommodate more flexible mesh data structures. Even with eight-byte integers and sixteen-byte doubles, each computing core can store roughly three million nodes and twenty-eight million tetrahedra. Scalable mesh simulations generally require less than one percent of this capacity to store the mesh data structures [2, 21, 48, 76]. Further, the total amount of memory used by an application may increase very little, or even decrease, because multiple copies of the mesh in various representations are no
longer needed to perform supporting computations.

A more expressive mesh data structure should not only increase efficiency at run-time by reducing the number of translations between alternative representations of the mesh, it may increase the efficiency of the programmers themselves—both in reducing the total amount of code to be written and in providing a more intuitive approach to application development. Consider maintaining shared data on a mesh in a parallel environment. Often there are arrays of arrays; the former are indirect references to other compute nodes and the latter are indirect references to mesh objects on the corresponding compute node. This indirection often leads to voluminous code that can be very difficult to debug, maintain, and improve.

Rather than rely on a multiple-array data structure that serves serial and small scale simulation best, we have designed an expressive data structure that guarantees that many of the data access patterns necessary for engineering simulations occur in $O(1)$ time. Furthermore, we extend the ideas presented by Vidwans [115], in which any matrix derived from the mesh is considered yet another translation of the mesh and this translation can be removed from the computation.

### 2.1 Generic Implementation

Consider the simple algorithm for computing the surface of a mesh given in Figure 2.2. This algorithm requires a way to iterate through entities in the mesh, a way to determine incidence of entities on others, and a way to add entities to a set. Since this algorithm works regardless of mesh dimension or types of elements involved, any data structure we construct should present the same interface to this algorithm independent of all other information about the mesh. This polymorphism allows algorithm designers to require only a prescribed interface without having to worry about how the data are stored. Because C++ has several tools for polymorphic methods, we chose it as the development platform.

C++ offers two kinds of polymorphism: ad hoc [116] and parametric [98]. Ad
hoc polymorphism relies on fully describing all objects that implement the interface before compile-time, whereas parametric polymorphism does not require any mention of a concrete type in the method specification. C++ implements the former through its inheritance model. For instance, in the algorithm given above, we could have a super class that allows for incidence checking, a super class that allows for iteration, a super class that allows for adding an object to a set, or any combination of these three. Then, through method virtualization, the application would choose which method to call at run-time based on a virtual function table. There are several drawbacks to this approach.

First, to implement a mesh object, a programmer must implement the entire set of virtual functions called for in the inheritance heirarchy, even if they are not used or do not make sense for a particular application. The programmer has the option of using empty methods or methods that trigger errors, but the code will not be as easy to maintain or debug.

Second, virtual methods are not as efficient as their non-virtual counterparts. For instance, consider the example above with a virtual method `GetNext()`. This method will be the most frequently invoked in the algorithm. Not only is there an extra indirection step for each invocation, but the compiler has limited ability to optimize the function in this context. This can have serious repercussions on the performance of the software [40, 66]. Consequently, there has been extensive research into optimizing virtual function calls [3, 12, 13, 69, 92, 122].

Parametric polymorphism, on the other hand, requires neither a virtual table nor run-time information to dispatch functions. Rather, the compiler infers which method should be called with the proper data structure. This eliminates the need for a virtual table and allows the compiler to perform better optimizations. Using parametric polymorphism is often called generic programming since algorithms can be coded completely independently of the data structures they use and manipulate. This approach has many advantages for large-scale simulation. For instance, a mesh partitioner can be fully implemented without a priori knowledge of the types of cells, the dimensionality of mesh (whether it is a three-dimensional mesh, a space-time mesh, a surface mesh, etc.), how the mesh is stored in memory, how the communication lists for parallel support are constructed and maintained, and other details that often affect algorithm design and clutter implementation.

Generic programming with C++ templates dramatically reduces the amount of code that must be written and simplifies the implementation of algorithms. Consider the algorithm given in Figure 2.2. While this algorithm is well understood, its implementation is cumbersome using the multiple-array data structure. Knowledge of element connectivity, communication lists, and construction of interim data structures are required for a complete, working implementation, ready for production. Using generic programming, on the other hand, the implementation matches the algorithm description almost line for line. Figure 2.3
template<typename MESH>
void ExtractSurface ( const MESH &mesh ,
  typename MESH::object_set &surface )
{
  typename MESH::
    template dimensional_types<MESH::topo_dim-1>::object_iter cur;

  for ( cur = mesh.template begin<MESH::topo_dim-1>();
    cur != mesh.template end<MESH::topo_dim-1>(); cur++ )
    if ( cur->second.template count<MESH::topo_dim>() == 1 )
      surface.insert ( cur.element->second );
}

Figure 2.3: Generic surface mesh extraction routine in C++.

gives the actual C++ listing of the method used to extract the surface of a mesh using the mesh data structure described below. This example software works for all meshes, regardless of mesh dimensionality, connectivity representations, types of elements, or even if the mesh is distributed across a number of processors in a parallel environment.

The C++ standard specifies the Standard Template Library (STL) [119]. Not only does this library provide common data structures (such as sets, vectors, and associative arrays), it also provides proven memory management through its implementation of containers. As pointed out previously, a computational mesh is a dynamic object that moves, grows, and shrinks. For the classic multiple-array data structure, the data must be copied and arrays re-allocated as the mesh changes. Using the STL for memory management not only frees the programmer from the task of tracking memory allocation but reduces the risk of memory leaks and other memory errors that are difficult to debug [110].

2.1.1 Design Criteria

Before the design of any software begins, a careful analysis of its goals should be carried out. Through such analysis, well-organized software can be designed so that it is easy to maintain and extend. Most mesh-based engineering simulations follow the algorithm specified in Figure 2.4. Each step of the algorithm accesses the mesh in different patterns. Any general purpose mesh data structure for simulation should provide constant-time access, where possible, for each of these steps. In other cases, such as memory allocation, the data structure should introduce no overwhelming inefficiency. We will address each line of the algorithm in turn. We start with the most obvious criterion for a parallel data structure.

Criterion 2.1. The data structure must be naturally parallel.
This criterion has the unfortunate characteristic of being vague. While this cannot be tested for, its absence can be observed. For instance, the Fortran code snippet in Figure 2.5 is an implementation of the conjugate gradient method. The portions circled in red are the conjugate gradient method. The portions in green are the parallel supporting code. The preponderance of code highlighted in green suggests that the underlying data structure is not naturally parallel.

Why is this important? The amount of time to develop code grows exponentially with the length of the code. The COnstructive COst MOdel (COCOMO) indicates that the effort expended in software programming can be estimated as

\[ E = aS^b, \]  

where \( E \) the amount of time in person-months, \( S \) the number of lines of code in the finished product, and \( a \) and \( b \) experimentally derived constants [24]. For most projects, \( b \) ranges from 1.05 to 1.2, depending on the complexity of the project and other factors. For parallel programming, we expect \( b \) to be high. According to this simple model, expanding code by 400\% results in more than a five-fold increase in development effort.

This model makes sense. As development proceeds and the code base becomes larger, each added feature increases the probability that the existing code requires modification. This probability increases as the code base increases since each added feature reveals subtle inefficiencies, bugs, and programming structure inadequacies. With minimizing the number of lines of code as the guiding principle of our design, we turn to the simulation algorithm.

The first step, Pre-process mesh, can be very simple, such as placing a serial mesh into the memory of a distributed memory supercomputer for partitioning. Or it can be very complex, such as generating a mesh in parallel given a CAD model. We shall focus on the former, since the other steps of the simulation should prescribe the necessary functions to complete the more complex oper-
Figure 2.5: Non-naturally parallel data structure in action. Local data are processed in red ovals. Parallel data management occurs in green ovals [50].
ations. Memory allocation and de-allocation are at the heart of each. Use of the stack for memory allocation is preferred because it is most efficient, just an addition operation. Meshes of sufficiently rich simulations evolve over time, however, so that the data structure for a mesh must be able to re-allocate space. Unfortunately, allocation from the heap is the only way to do this.

The time for removing elements in a multiple-array data structure is constant for cells (copy data from the end of an array to the now-empty position and update a counter) and more complex for nodes, up to \( O(n) \) in the number of cells depending on the type of removal. Unfortunately, the time for adding one element can be significant, with contiguous storage needing to be re-allocated, worst case \( O(n) \) in the number of elements. There are mechanisms to amortize this cost, but at the expense of unused memory. In the simplest example of pre-processing, reading a mesh from disk, the size of the data on each processor is known in advance and read into memory after just one allocation, giving a worst case of \( O(n) \) in the number of elements. This gives the second criterion for the data structure.

**Criterion 2.2.** Any mesh data structure should support bulk insertions/deletions in \( O(n) \) in the number of elements inserted/deleted and transactional insertions/deletions in \( O(n) \) in the number of elements inserted/deleted.

The second step of the algorithm, *Partition mesh*, requires iteration through mesh objects in graph traversal order. Topological partitioners by their very definition require graph adjacency information and often mandate that the programmer provide the mesh in specialized graph data structure [74, 75]. Even some geometric partitioners require adjacency information to compute local data [115]. The third criterion for a mesh data structure comes from this observation.

**Criterion 2.3.** Any object should enumerate its adjacencies in \( O(1) \).

Once the graph has been processed, the mesh data need to be moved to the appropriate compute node. Normally, after translating the arrays that describe the mesh into an adjacency representation and calling the partitioner, the programmer will then move the necessary data from the arrays to the appropriate compute node, often requiring a new allocation for the partitioned mesh. To simplify the task of the programmer and optimize the overall execution of the simulation, we give the next criterion.

**Criterion 2.4.** If a step in a simulation computes a transform of a mesh (e.g., change in connectivity, movement of elements, computation of values stored on elements), then that transform is computed directly on the mesh data structure.

For instance, when a partitioning of the mesh is computed, the mesh objects are automatically redistributed across the entire computer based on the partitioning. This criterion is the heart of the overall optimization described in
this chapter: *meshes are not translated from one form to another to compute transforms that are then translated back to be applied*. This assures that the various responsibilities of method coders abut the responsibilities of the simulation coders.

This criterion affects the next step in the simulation algorithm, *Generate and solve (non-)linear system*. Through various transformations, many engineering simulations generate a system of equations from the mesh and then solve them. As part of this process, a sparse linear system $Ax = b$ is generated and solved either directly or iteratively. This matrix is constructed in $O(n)$ in the number of cells or stencils. For instance, given a simplicial mesh and Galerkin finite element formulation, each cell is locally integrated (in constant time) to generate a local stiffness matrix, which is then aggregated with the local stiffness matrices of each cell. On the other hand, finite difference stencils on a structured mesh give rise to individual equations at each node so that each row of the matrix is computed in constant time. For clarity, we call the local stiffness matrix or stencil associated with a node a *stencil object* to be differentiated from a mesh object.

**Criterion 2.5.** *Each stencil object must be constructed in $O(1)$.*

Often, preconditioners and direct solvers require an explicit representation of the mesh to complete the computation efficiently. They will reconstruct the mesh in a way that exposes the information they need. For instance, parallel sparse direct solvers are often organized in a multi-frontal manner, with the node order generated by the mesh partitioner. This organization is stored in an elimination tree, an object that exposes which computations can be done independently of others and how these computations should be ordered for improved parallel efficiency. The final result is computed as a level-order traversal of the tree. Since the efficacy of direct solvers for large-scale simulation depends to a large extent on the shape of the domain being simulated, it is difficult to posit a more concrete criterion on the data structure *vis-a-vis* direct solvers.

**Criterion 2.6.** *The stencil objects derived from the mesh can be organized and evaluated in any order.*

Iterative methods, on the other hand, require two mesh-related criteria. First, methods such as GMRES, Bi-CG, and other iterative solvers rely on the efficiency of computing matrix-vector multiplication. For a sparse matrix $A$, the product $Ax$ can be computed in $O(n)$ in the number of rows. In solving the system, this matrix-vector product may be evaluated thousands of times. It is of utmost importance that any mesh data structure meet this specification.

**Criterion 2.7.** *Any iteration through all stencil objects to compute a matrix-vector product is completed in $O(n)$ in the number of stencil objects.*

The next criterion involves the construction and application of a suitable preconditioner. Briefly, for the identity matrix $I$, the system $Ix = b$ is trivial
A preconditioner is a matrix $M$ such that $Mx = b$ is easy to solve and, for some matrix $A$, $MA$ is “closer” to $I$ than $A$ is. The idea is that an iterative method can solve the linear system $MAx = Mb$ in much less time than $Ax = b$.

The more sophisticated preconditioners generally try to approximate $A^{-1}$ by some other iterative or direct method. For instance, a few iterations of algebraic multigrid (AMG) can be an effective preconditioner. Or, using a domain decomposition approach with an approximate Schur complement has shown great success for a variety of applications. There are also incomplete factorizations that limit time consuming (and communication intense) fill. Description (and implementation) of these preconditioners can be quite tedious. Consider a domain decomposition-based preconditioner that uses a constant number of preconditioned Bi-CG iterations (preconditioned with AMG) to approximate the $A^{-1}$ term in the Schur complement. Here we have a preconditioned method accelerated with a preconditioned method! Since each of these preconditioners requires multiple stencil objects for each mesh object, and each set of stencil objects is computed in a different order from the others, we have another design criterion.

**Criterion 2.8.** The application of preconditioners can be nested.

The final three lines of the high level simulation algorithm do not impose any additional requirements on the mesh data structure. Rather, they emphasize the need for a robust data structure with an expressive interface.

### 2.1.2 High-Speed Memory Considerations

Designing parallel data structures to make the most effective use of high-speed processor cache is notoriously difficult. Consider the eigenvalue problem $Ax = \lambda Bx$. This can be solved very well by several preconditioned eigensolvers. If multiple eigenvalues are recovered simultaneously, then the computation can be reordered to use cache better, with considerable speed up [10]. This experiment demonstrates that appropriate use of cache is a major concern for high performance computing.

The current problem lies in the use of the compressed row (or column) storage (CSR) format for matrices. The unpredictable indirection of this format requires large portions of the left-hand-side vector and right-hand-side vector be loaded into cache (as opposed to multi-frontal direct methods designed to use dense Basic Linear Algebra Subroutines (BLAS) [80], which use the cache in a predictable fashion).

Consider the matrix-vector multiply algorithm given in Figure 2.6. Unless $x$ fits entirely in cache, there is no guarantee that cache can be used effectively. When $x$ does not fit into cache, then the sparse matrix-vector operation slows down dramatically. Contrast this algorithm with the algorithm presented in
Figure 2.6: Sparse matrix-vector multiply with indirection.

\[
\text{SparseMatVec}(M, x)
\begin{align*}
1 & \text{ for } i \leftarrow 1 \text{ to } n \\
2 & \text{ do } \\
3 & \text{ for } j \leftarrow 1 \text{ to } \text{row_length}[i] \\
4 & \text{ do } \\
5 & \quad y[i] \leftarrow y[i] + M[\text{row_ind}[i] + j] \ast x[\text{col_ind}[\text{row_ind}[i] + j]] \\
6 & \text{ return } y
\end{align*}
\]

Figure 2.7: Sparse matrix-vector multiply with “dense” core.

\[
\text{DenseMatVec}(M, x)
\begin{align*}
1 & \text{ for each row in } M \\
2 & \text{ do } \\
3 & \text{ for each col in row } \\
4 & \text{ do } \\
5 & \quad y[\text{row}] \leftarrow y[\text{row}] + M[\text{row}, \text{col}] \ast x[\text{col}] \\
6 & \text{ return } y
\end{align*}
\]

Figure 2.7. Since off-diagonal entries are predicted by mesh connectivity, this algorithm promotes better use of cache. This demonstrates the final criterion for the data structure.

**Criterion 2.9.** \textit{Vector values are stored so that they are accessed by stencil object rather than by position in a contiguous array.}

Before we introduce our data structure, we evaluate several existing efforts to build robust data structures for meshes. As you will see, we are not the first to apply the various techniques we used to implement our data structure. Rather we combined the ideas in a way to meet most if not all of the needs of simulation designers.

### 2.2 Other Efforts in Mesh Data Structures

There are many diverse efforts in designing data structures and frameworks to make parallel programming more accessible to researchers. Generally, they can be divided into two categories: mesh-oriented and solver-oriented. The mesh-oriented data structures are used for partitioning, mesh maintenance, and mesh generation. The solver-oriented ones are designed to maximize the number of floating point operations per second during the \textit{Generate and solve (non-)linear system} step. We present six different mesh data structures, three in each category, and describe what we find to be the most useful in each.
2.2.1 Mesh Data Structures

Mesh data structures track nodes, edges, and other mesh entities. We start with the simplest graph representation, a matrix $A$ where nonzero value $a_{i,j}$ indicates the weight of an edge originating at $i$ and terminating at $j$. A mesh data structure, at a minimum, identifies cells in some way. Some data structures accomplish this with two graphs: the first graph is the nodes and edges, and the second is the dual mesh (cells and facets) [27, 28].

From this data structure, we can add complexity. For instance, we can provide mechanisms to add or subtract cells, split cells, or flip edges/faces. These data structures are formulated to make these operations easy. The following mesh-oriented data structures provide just these sorts of routines in a variety of ways.

**CGAL**

The Computational Geometry Algorithms Library (CGAL) provides a large set of algorithms and data structures for performing many, many tasks in computational geometry [29, 30]. It relies heavily on the use of C++ templates for the implementation. This allows programmers to code only what they need and relies on the compiler to bring together the algorithm. The breadth and scope of this library is impressive.

In fact, it is possible to develop a complete set of solver routines much like the ones described below developed entirely within the framework of CGAL. The solver routines may even work well enough to satisfy all of the criteria outlined above, save one. Unfortunately, the library does not lend itself to parallel computation without violating Criterion 2.1. In the end, the result would end up like that of the conjugate gradient example in Figure 2.5—the parallelism would appear to be an after-thought.

We patterned our style of algorithm development around the style of CGAL. Algorithms are classes that are constructed around mesh templates. For instance, the medial axis algorithm described in the next chapter is a class that takes a mesh as a template parameter. In this way, the algorithm is developed using generic iterators and topological relations. In the end, we hope to establish a substantial base of algorithms for various parts of numerical simulation in much the same way CGAL has a substantial base of algorithms concerning computational geometry.

**TSTT Mesh Interface**

The Terascale Simulation Tools and Technologies (TSTT) mesh interface is an attempt to bring descriptive mesh data structures into high performance computing [90, 91]. Unlike CGAL, the TSTT does not appear to leverage the full capabilities of generic programming. Rather, the interface is enforced though naming convention, as with *ad hoc* polymorphism.
The reason for this is primarily for backward compatibility. Without going too deeply into details, using generic C++ in Fortran is problematic for the Fortran programmer. Unlike conventional library calls, the methods are not instantiated until they are needed. When we write a generic class, the methods do not exist until a user of the class provides a specification for the template. Providing this specification in Fortran is tedious and limits the flexibility of some aspects of the code.

As mentioned earlier, *ad hoc* polymorphism is difficult to optimize and may lead to a loss of “free” efficiency provided by a compiler. The TSTT mesh interface does offer expert insight into what particular mesh operations are required for many mesh maintenance routines. Some of our design, such as the idea of sets of entities, we borrowed directly from their specification.

We do not, however, implement their efficient set container, rather we rely on standard containers provided by the STL. The reason for this is Criterion 2.6. Mesh simulation results are sensitive to orderings of computations. And, we do not want to segregate different dimensional mesh objects in storage just in case they need to be iterated over in an interleaved fashion, as in some electromagnetic simulations.

**Parallel AOMD**

The Parallel Algorithm Oriented Mesh Database (AOMD), like the TSTT mesh interface, relies on *ad hoc* polymorphism [96]. And, like the TSTT mesh interface, it is a wonderful resource for identifying important mesh access and manipulation routines. It would also be possible to write a set of solver routines using the data structure.

While we feel this data structure would have the greatest chance of success for such a solver, we feel that there are still some issues that can be handled better using generic polymorphism. First, the use of cells is extremely specific in the application programming interface. There are explicit classes for hexahedra, tetrahedra, prisms, and the like. Contrast this with the face class, which can be made up of a variable number of edges.

Rather, we approach mesh entities in a more generic fashion. This gives developers greater freedom in specifying meshes and allows for even higher dimensional elements for space-time meshes.

These three successful data structures were invaluable guides when designing our data structure. They indicated valuable access patterns and demonstrated that some types of access are less useful than we believed when we started our project. Once we had determined the total breadth and scope of the mesh data structure routines, we set about understanding the needs of high performance solvers.
2.2.2 Solver Data Structures

The solver-oriented data structures concentrate on two aspects of the computation: fast data access and distributed memory support. Generally, the fast data access is taken care of by contiguous memory access. The mesh is stored explicitly in contiguous arrays discussed earlier. These arrays are used to construct vectors and sparse matrices. The sparse matrices can be stored in contiguous memory by constructing indexing arrays, one that lists the offsets for each row and one that translates the current offset into a column.

The idea of doing this in C++ has been around for a long time [38], much longer than the parametric polymorphism and generic programming constructs in the language. By providing a uniform interface, the amount of reused code, in theory, should increase. In fact, large development libraries leverage this fact and are very successful in providing simulation designers great flexibility in the types of available solvers, preconditioners, etc.

The other part of the computation, managing the parallel aspect, is handled by indexing arrays. An array of arrays maintains maps of local mesh objects to mesh objects on other compute nodes. By keeping the objects ordered on each compute node, the volume of communication can be minimized. We consider three different high performance solvers and their data structures.

**ParFUM**

One of the most ambitious projects is ParFUM, a finite element mesh management package [79]. While this shares some of the interfaces of the data structures already mentioned, the mesh is stored in a manner similar to the data structures below. ParFUM is an excellent project in that it tackles a major challenge in large scale scientific computation: mesh evolution.

As mentioned earlier, meshes are dynamic. Some parts of the mesh require greater resolution, while others require less. The scope of ParFUM is far greater than this, since it is built on Charm++ [72]. Unfortunately, the framework is somewhat inflexible in that it is geared toward the solution of finite element analyses.

**Trilinos**

Epetra, the heart of the Trilinos suite of linear algebra solvers [63, 64], is a set of C++ classes that manage mesh data in parallel and provide efficient access for solver routines. Epetra provides several sparse matrix storage formats, a dual mesh representation, and a mapping class to manage the parallel indexing.

Switching between types of matrix representations is facilitated by a pure virtual base class. The indexing is handled by a map class that translates local offsets into offsets on other compute nodes. In this way, different kernels can be written with a common interface to the data and the communication layer.
Trilinos also offers two mesh representations: phdMesh and ABMesh. Both are similar to ParFUM in that they offer contiguous memory storage of the data. There is also a separate partitioning layer in Zoltan based on a hypergraph representation of the mesh [26]. It is plain that Trilinos violates Criterion 2.4, a common pitfall of very large development efforts.

The individual packages of Trilinos are superb. Combining two packages, however, such as partitioning and solving, still involves several unnecessary data translations.

**PETSc**

Finally, any discussion of high performance solvers must include PETSc [14, 15]. PETSc does not provide any direct mesh manipulation interface. Rather, it provides solvers, both linear and non-linear, and a parallel management library. The speed with which equations can be solved is nearly optimal [52] and the software is widely used through the scientific computing community. It contains dozens of solvers and hundreds of preconditioners. PETSc is a demonstration that code reuse techniques can be efficient on parallel supercomputers. It does not offer any sort of mesh management interface, violating several of our design criteria.

### 2.2.3 Generic Implementations

Applying generic programming techniques to scientific computing is not a new idea [19, 56, 73, 106]. Our data structure differs from these projects in scope: we intimately combine meshes and solvers through generic interfaces. Meshes and linear solvers, in the context of simulations, are not independent. Though they can be used in applications apart from each other, they should be specialized for the task when used together.

A robust mesh management library and a fast sparse linear solver can be plugged together to obtain excellent results. But, as noted at the beginning of Chapter 1, the sparse linear solver often needs to know something about the mesh, and vice-versa. This is obvious for domain-decomposition-based preconditioners, multilevel solvers, and adaptive mesh refinement solvers.

Ordering the computation based on the mesh structure, a subtle optimization, can enhance the performance of a sparse matrix-vector product three fold [51]. In the same paper, Gropp et al. demonstrate that ordering the memory requests to match the connections in the mesh will boost performance. Knowledge and use of the mesh at such a low level in the computation can yield a total of seven times performance increase over a naïve integration of libraries [52].
2.3 Mesh Interface

One of the benefits of generic programming is the ability to specify and use the mesh application programming interface (API) completely independently of developing the code necessary to make the API work. For instance, in the next chapter we describe an algorithm for computing the medial axis of a meshed domain. This algorithm relies entirely upon the concepts described in Section 1.3 and does not require a mesh description until compile time. Similarly, we implemented a matrix-free sparse matrix-vector multiplication routine based on the abstract concept of a stencil object.

The mesh data structure we developed provides these abstractions and others to help speed development of high performance computation supporting code and the actual simulation itself. Before we describe these abstractions and their implementation, we discuss our approach to data storage.

2.3.1 Containers and Iterators

The Standard Template Library provides containers to manage memory. Instead of relying on the programmer to check for memory leaks and debug complex memory errors, the STL manages its own memory. By using containers, the programmer is free to allocate space almost as if there were a garbage collector.

The STL containers come in two basic flavors: sequence containers and associative containers. A sequence container is analogous to a contiguous memory array (e.g., std::vector) or linked-list (e.g., std::list). These provide various guarantees on asymptotic speed of insertion, removal, and random access. To gain performance in one of these, a developer must be prepared to pay a penalty in another.

Associative containers, on the other hand, can be thought of as hash maps (e.g., std::map) or trees (e.g., std::set). They provide a middle ground for insertion, removal, and random access speed. Generally, these operations require $O(\log n)$ operations, for $n$ the number of elements in the container. The interesting point is that iteration, moving from one element in the container to the next, can be achieved in constant time.

A mesh data structure requires associative indexing. Mesh objects are associated with an index. In serial, the map can consist of integers enumerated from 0 or 1, depending on your programming language. In parallel, prevailing wisdom assumes that the local index should still be enumerated integers with a separate map for relating indices on one compute node to indices on another. This just reinforces the notion that mesh objects are stored in an associative index. Object indices on one compute node are associated with object indices on another node.

Why not, then, use an associative container to store mesh objects? Most will claim they are too slow. And they are right, if one is talking about random access to the objects. Solvers that use contiguous memory data structures require
random access to objects. A close look at the sparse matrix-vector routine in Figure 2.6 shows that the access pattern for the left- and right-hand-side vector is not iterative. However, if the data storage could be reorganized somehow so that it resembled the dense matrix-vector routine in Figure 2.7, then the apparent random access would be replaced with iterative access. In other words, satisfying Criterion 2.9 removes objections to using associative containers for mesh objects.

For this reason, we designed the mesh data structure to rely on std::map\(^1\) for managing the objects. Once we do this, we are freed from maintaining index maps in a parallel implementation, as all compute nodes with a particular mesh object agree on the associated index for the object. The std::map supports \(O(n)\) bulk insertions and \(O(\log n)\) transactional insertions, satisfying Criterion 2.2.

### 2.3.2 Recursive Data Structures

The mesh objects listed in Section 1.3 are defined recursively. For this reason, we implemented a recursive data structure for generating the mesh object types. A recursive data structure generates a set of data structures at compile time based upon instantiation requests. They are very useful when, during development, a programmer does not know how many different data types will be needed.

As a simple example, consider the recursive data structure to compute the factorial of an integer in Figure 2.8. The compiler instantiates \texttt{Recursive<5>} to call the static function fact(). This function requires \texttt{Recursive<4>::fact()} to complete, so the compiler instantiates the \texttt{Recursive<4>} class as well. This class requires \texttt{Recursive<3>::fact()} and so on until the compiler instantiates \texttt{Recursive<1>} and its static fact() method.

Once everything has been instantiated, the compiler sees a list of return values and through constant propagation and constant folding compiler optimizations generates code that is equivalent to std::cout << "5! = " << 120 << std::endl. In essence, recursive data structures can be used to program the compiler to generate efficient code.

Recursive data structures are also a way of generalizing related or similar types. We can have a common interface for mesh objects and have the compiler instantiate the mesh objects as needed. At compile time, the dimensionality of the mesh is known, so the compiler can construct them as necessary. A user instantiates a mesh by specifying the topological dimension of the mesh and a set of parameters specific mesh objects of each dimension.

The mesh instantiates a class that contains a recursive data structure that describes the containers used to hold the objects. The compiler uses the parameters to determine what the particular mesh object will inherit from. Through these inheritances, the mesh objects can store arbitrary data, have boundary

\(^1\)Or, if available, std::hash_map
```cpp
template <int i>
class Recursive
{
    
    public:
    
    static int fact ()
    {
        return i*Recursive<i-1>::fact();
    }

};

template <>
class Recursive<1>
{
    public:
    
    static int fact ()
    {
        return 1;
    }

};

int main ( int argc , char **argv )
{
    std::cout << "'5!' = '" << Recursive<5>::fact() << std::endl;
}
```

Figure 2.8: The recursive data type `Recursive` actually generates five separate data types. This recursion has the effect of unrolling the factorial so the compiler can perform the multiplication.

conditions, have positions in space, and have different index types.

One of the parent classes of a mesh object is a class that stores a list of pointers to itself. This class is used to keep incidences. For instance, if a node is incident on an edge, then a pointer to each resides in the other. No topological consistency checking occurs for this incidence list. If a particular three-dimensional mesh application does not require knowledge of edges, then it need not be constructed. Nodes can be incident on faces only. In this way, the mesh object data structure can enumerate its adjacencies in \(O(1)\) time, consistent with Criterion 2.3.

The inheritance mechanism allows the developer to choose between mesh objects storing local arrays or storing references to arrays in memory. For instance, the former can be implemented as a class that contains an array of double precision floating point numbers, whereas the latter stores a pointer to an array of doubles. As long as the double array is parceled up so that each mesh object refers to a contiguous block within the array, then the data access looks exactly the same.

In either case, this mechanism can be used to implement Criterion 2.9. This happens through the use of stencil objects. A stencil object is analogous to a row or set of rows in a matrix. The stencil collects pointers to the mesh objects that must be operated on during matrix-vector multiply, preconditioning, or direct
solution. Each stencil is associated with a mesh object. By enumerating the incidences of the mesh object in $O(1)$, each stencil can be constructed in constant time. Assuming the underlying stencil has local support, this is consistent with Criterion 2.5.

Since the mesh objects and stencils are stored in associative memory, the data structure is consistent with Criterion 2.6: the stencils can be organized in any order. And, since the stencils hold pointers to mesh objects relevant to the computation of each row in a matrix-vector operation, the matrix-vector operation can occur in $O(n)$ in the number of stencils. This satisfies Criterion 2.7.

Using recursive data types also allows mesh data types to infer other mesh data types. Given a particular mesh, the medial axis algorithm in the next chapter needs to construct a mesh of the same geometric dimension, but of one less topological dimension. Also, dual meshes can be inferred from the mesh type. We developed a generalized inferred mesh class that makes dual meshes.

The generalized inferred mesh class allows the developer to decide which mesh objects will be represented by nodes and which by edges. For instance, the dual mesh will have nodes for each cell and edges for each facet. The user may wish to represent so-called edge neighbors in the dual mesh. An edge neighbor is any mesh object that shares a common edge. This information is taken as a template parameter and the compiler generates the appropriate code to infer the dual mesh.

This seems to contradict Criterion 2.4. Why create an entirely different mesh representation? For one reason only: the dual mesh is smaller in memory footprint than the original mesh. For serial and purely local computations, this does not matter. However, to reduce the overall communication overhead, it may be necessary to replicate the mesh in another form. Knowing when to violate 2.4 is the key to good optimization. The criterion should always be enforced across mesh method invocations.

### 2.3.3 Action-Method Interface

The code relies on abstracting common patterns from most simulations and implementing them in an action-method interface. The action-method interface allows the user to call methods through one template function and the mesh takes care of the rest. For instance, it is common to partition a mesh by constructing the dual mesh, computing a graph partitioning of the dual mesh, and then redistributing the entire mesh based on the dual mesh partitioning.

In this case, the mesh data type has an action called `dual_partition`. The algorithm for the action is simple: derive a dual mesh, partition the dual mesh, and redistribute the mesh accordingly. The partitioner used is called the method. Figure 2.9 gives the source code and a sample instantiation of the code. To use a different mesh partitioner, say the exact Cartesian nested dissection partitioner [115], only one line of code needs to be changed:
template <int TOPO_DIM , template <int> class DEFS>
template <template <typename M> PARTFCN>
void mesh<TOPO_DIM,DEFS>::dual_partition
    ( typename PARTFCN <this_type>::options &o )
{
    typename PARTFCN<this_type>::dual_mesh dm;
    PARTFCN<this_type>::dm_factory::
        get_dual_mesh ( *this , dm );
    PARTFCN<template< typename M> PARTFCN<this_type>::dual_mesh>::
        partition ( dm , o , gComm );
    PARTFCN<this_type>::dm_factory::
        redistribute_mesh ( dm , *this );
}

my_mesh<parmetis<1>::partitioner>::dual_partition ( 1.05 );

Figure 2.9: An example action interface dual_partition implementation in the mesh data types. The last line is an example implementation of the dual_partition action using the ParMetis GeomKPart method.

    my_mesh.dual_partition<parmetis<1>::partitioner> ( 1.05 );

becomes

    my_mesh.dual_partition<exact_cnd> ( 0.05 );

for the different instantiation. The arguments of the function are specific to the partitioner and should be adjusted depending on the desired parameters of the partition.

Other common patterns are file input/output, solving a right-hand side, applying a preconditioner, sparse matrix-vector product, and other BLAS routines. By using the action-method interface, these patterns can be applied using different algorithms with minimal change in the source code by the software developer.

2.3.4 Future Work

We have not implemented a preconditioner to test Criterion 2.8. As such, the solver action is not fully tested. We have tested the concept by switching between Cholesky factorization and conjugate gradient method, achieving results as expected. If it does work, then extremely complex iterative solvers can be constructed in a single typedef statement.

Consider the domain decomposition-based preconditioner example from Section 2.1.1 that uses five preconditioned Bi-CG iterations (preconditioned with three “V” cycles of AMG) to approximate the \( A^{-1} \) term in the Schur complement used to accelerate GMRES. This could be implemented as in Figure 2.10. Our experience with generic programming indicates this is a completely viable paradigm for developing mesh-based simulation codes.
class AMG_options {
    public:
    enum { iterations = 3, cycle = 'V' }; 
};

class BiCG_options {
    public:
    enum { iterations = 5 }; 
};

template <typename MESH>
class my_solver:
    public GMRES <
        MESH ,
        domain_decomposition <
            MESH ,
            BiCG <
                MESH ,
                AMG<MESH, no_precond, AMG_options>::
                preconditioner ,
                Bi_CG_options
                >::preconditioner ,
                no_options
                >::preconditioner ,
                no_options
                >::preconditioner,
                no_options
            >::preconditioner,
            no_options
        >::preconditioner,
        no_options
    >
};

mesh<3, mesh_opts> my_mesh;
my_mesh.read_file <native> ("mesh.file");
my_mesh.dual_partition <parmetis<1>::partitioner> (1.05);
my_mesh.solve <my_solver, LHS, RHS> ( );
my_mesh.write_file <hdf> ("results_file");

Figure 2.10: Hypothetical complex solver constructed through the action- method interface.
2.4 Optimization

The main reason for choosing generic programming is to leverage the ability of the compiler to generate code and perform code optimization. Understanding how this works requires a little background on how compilers work and how templated C++ is developed. Modern compilers translate high level languages into an internal representation. This internal representation is optimized, then code is generated. In the linking step, the generated code is ordered and placed into an executable file.

This four step process allows for pre-compiled libraries to be assembled once for all applications on a particular system. Many applications that uses BLAS [80], for instance, need only a portion of the routines. Yet, including only the appropriate subroutines would be unnecessarily confusing and prone to error. These types of libraries are compiled in full, a single header file is generated, and combined with applications at link time.

Generic software implementation works differently. Since virtual tables are not used to dispatch functions at run-time, every template method and the data structures used for the templates must be explicitly declared at compile time, before the compiler generates the internal representation. Because libraries have been compiled already and are generated code, they cannot be used by the compiler for templates\(^2\). Because most compilers require the C++ (or internal representation) of the templates before code generation, templates are implemented completely in header files.

The internal representation has full access to the data structures in use in each function, not just a virtual base class. This allows the compiler to build a single internal representation of the entire executable. The work flow of compilers for C, Fortran, and non-template C++ cannot do this since object files and libraries each have code generated and their internal representations do not connect together in a useful way.

Since the compiler automatically decides which methods need to be implemented, often much of a library need not be compiled. Not only that, by using templates, the amount of code that needs to be written can be cut in half or more. Consider the BLAS subroutines. Each subroutine exists in a single precision and double precision implementation. With templates, only one needs to be coded and the compiler will do the rest.

2.4.1 Compiler-Only Optimization

We ran several experiments with various mesh sizes to demonstrate how well the compiler can optimize templated C++. The experimental software reads a file into memory, generates a dual mesh, computes the medial axis of the mesh (using the algorithm presented in the next chapter), computes a geometric

\(^2\)The `export` keyword can be used to a limited extent to generate a library-like file.
partitioning (using exact Cartesian nested dissection [115]), and writes the file out to disk. To fulfill some of these tasks, the mesh is partitioned in a rough sense, and redistributed to match the partitioning.

Reading a mesh from disk is a serial process and uses the algorithm presented in Section 2.8. Since it is bound by a large serial thread, it does not scale very well. Computing a dual mesh is almost “embarrassingly parallel,” meaning that the finest grain part of the computation can be completed independent of all others. The medial axis algorithm is complex, but is dominated by a marching algorithm. The partitioning uses a nested dissection algorithm, becoming more parallel as the computation proceeds down the recursion tree. The rough partitioner is nearly embarrassingly parallel, if the number of cells per compute node is equal across all compute nodes. Distributing the mesh is a communication intensive operation. Finally, writing to disk is an embarrassingly parallel implementation that is limited by the read/write head on the disk and mitigated by the buffers on the file server.

The size of the mesh was varied along with the number of processors used in the computation. Figure 2.11 shows the three meshes used in the experiments. For our largest mesh (in green), we used a 2.7 million element tetrahedral mesh of the solid fuel domain of a Titan IV rocket. The two smaller meshes were subsets of this mesh: a 450,000 element tetrahedral mesh of the star grain (in red), and a 140,000 element tetrahedral mesh of the top-most quarter of the star grain (in blue). Further, each experiment was executed on up to forty-eight compute nodes. We used an Apple 2GHz G5 cluster running Linux. We used g++ version 4.1.2 to compile the software with -O2 optimization on optimized runs.

Figure 2.12 shows the results for the smallest mesh. Generally, for more than 10,000 cells per compute node, we see a 2.5 to 4.5 times speed-up just by turning on optimization. Unfortunately, the timing experiment soon became dominated by the mesh reading method. This is discussed further in Section 2.8. Figures 2.13 and 2.14 show similar results. As the meshes increase in size, the relative efficiency of the optimizer lessens.

There is a noticeable outlier in the large mesh optimization results. This occurs for six processors, or roughly 225,000 cells per node. In the next chapter, we discuss a similar outlier that arises due to poor cache use. In this case, we suspect it is not a cache use issue. The actual time to perform the computation is roughly four times higher than it should be for all portions of the computation except creation of the dual mesh. Since this is the only part that contains no communication, we suspect there is a communication bottleneck.

When compared to the four-processor case, both the communication volume and the minimum number of communications to perform a global operation increases by roughly 50%. When eight processors are used, the application runs in roughly one-third the time. In this case, eight processors incur a similar number of messages and better use the interconnection network.
Figure 2.11: Various meshes used in optimization experiments.
Figure 2.12: Wall clock time reduction of parallel program using compiler optimization with input of 140,000 tetrahedra.
Figure 2.13: Wall clock time reduction of parallel program using compiler optimization with input of 450,000 tetrahedra.
Figure 2.14: Wall clock time reduction of parallel program using compiler optimization with input of 2,700,000 tetrahedra.
2.4.2 Future Work

This experiment needs to be extended in four ways: more diverse meshes, greater numbers of processors, across multiple platforms, and with different compilers. The importance of this experiment is to identify which platforms and compilers, if any, do not provide adequate optimization of templated C++. We expect to see better results from commercial high performance compilers than what we received from a two year old build of the GNU C++ compiler.

Further, none of these algorithms have been tuned for performance. According to Knuth\(^3\), “premature optimization is the root of all evil.” [77]. Like much of research code, the software is in a “proof-of-concept” phase and has not been tuned to use cache optimally or reordered to removed redundant computation. Also, the code is lacking several iterators to aide the compiler in optimization, such as a \texttt{const iterator} for enumeration of objects without changing them.

2.5 Matrix-Free Computation

The primary motivation for a robust data structure is to remove unnecessary mesh representations from a simulation, including matrices used in computations. For large-scale, three-dimensional simulations, researchers generally rely on iterative solvers to arrive at a solution. The heart of these iterative solvers is the matrix-vector multiply, or MATVEC.

The sparse matrix-vector multiply operation is a memory bound operation in serial in the sense that the computation speed is limited by the speed of the memory subsystem [52]. The processing unit is frequently starved for data and must wait for the matrix entries to be loaded into registers. This limits the overall performance of the computation to that of the bandwidth to get data into and out of the processing unit.

This bottleneck is easy to demonstrate. Let matrix \(M\) be a sparse matrix with each nonzero entry stored in eight bytes. Assume that \(M\) has \(r\) rows and each row of \(M\) has, on average, \(n\) nonzeros. The total memory consumed by the matrix is \(8nr\) bytes. To perform the matrix-vector multiply, the entire input and output vector must be stored. Assuming eight-byte entries, the vectors need \(16r\) bytes of memory. In total, the matrix-vector multiplication requires \(8r(n + 2)\) bytes of memory.

Assume that a particular three-dimensional simulation has one thousand nodes per processor with three degrees of freedom per node. This implies \(r = 3000\). Assume also this simulation is performing finite element analysis. Each row, at a minimum has twelve nonzeros (three degrees of freedom per node with four-node tetrahedra). The absolute minimum amount of memory required to perform a matrix-vector product is 336,000 bytes.

\(^3\)Knuth attributes this quote to Tony Hoare, though Hoare does not recollect making the statement
Even this paltry computation is six times larger than the data memory provided on most high performance processor chips, the L1 cache. Therefore, each matrix-vector multiply requires that the matrix and both vectors be copied to memory on the chip, manipulated, and the result copied back off of the chip. What’s worse, the memory usage given above is at least an order of magnitude smaller than actual sparse matrix-vector products in practice. The matrix-vector product generally does not fit in fast memory on the motherboard, the L2 cache. The limiting factor in computing matrix-vector products is the memory subsystem.

2.5.1 Experimental Determination of Sustained Matrix-Vector Product Performance

One way to determine the efficiency of a memory-bound computation is to compute the amount of data the particular computation needs and divide that by the memory bandwidth. This gives the amount of time the memory subsystem needs to move the data from RAM to registers. Dividing this by the number of floating point operations in the computation gives the theoretical maximum sustained computation rate in floating point operations per second.

Unfortunately, the internals of data access of STL containers are intentionally opaque. Computing the amount of data necessary to perform a sparse matrix-vector product, while technically possible, violates the spirit of the STL. Also, this information would be of limited value for our purposes. In order to compare this data structure to those of existing real-world applications, we chose to measure our matrix-vector implementation against the theoretical maximum speed of the current de facto standard matrix-vector implementation: compressed row storage (CRS).

It is trivial to compute the minimum amount of data that must be sent to the processing unit [51]. Let $M$ be a sparse matrix with $r$ rows and $n$ the total number nonzeroes in $M$. Assume the memory bandwidth, the number of bytes that can move to the processing unit per second, is $B$. Let $N$ be the number of vectors being multiplied simultaneously. And, let the computing architecture use four-byte integers and eight-byte double precision floating point numbers. Then, using the formula provided in [51], the theoretical maximum rate of computation for the sparse matrix-vector product is

$$R = \frac{2B}{(16 + \frac{4}{N}) \frac{r}{n} + \frac{12}{N}}.$$  \hspace{1cm} (2.2)

This simple model ignores several important aspects of the computation and tends to overstate the maximum achievable rate of computation.

All of the variables except $B$ are defined by the problem. Finding $B$ experimentally can be accomplished using the STREAM benchmarking software [86, 87]. This software measures the amount of time to perform several types of
computations on data that are much larger than the size of the cache.

2.5.2 Results

The matrix used in the experiment arises from Laplace’s equation,

\[ \nabla^2 u = 0, \tag{2.3} \]

on a square or cube, with Dirichlet boundary conditions specified on all boundary nodes. The discretization used for the equation was the second-order centered finite difference stencil

\[ \frac{\partial^2 u}{\partial x_i^2}(x) \approx \frac{u(x + he_i) - 2u(x) + u(x - he_i)}{h^2}, \tag{2.4} \]

where \( e_i \) is a unit vector in the \( i \)th direction. This gives rise to a linear system of the form

\[
\begin{bmatrix}
A & B \\
0 & I
\end{bmatrix}
\begin{bmatrix}
x_j \\
x_b
\end{bmatrix}
=
\begin{bmatrix}
0 \\
b
\end{bmatrix},
\tag{2.5}
\]

where \( x_j \) are degrees of freedom on the interior nodes of the mesh, \( x_b \) are the nodal values for which a Dirichlet boundary is set, and \( b \) is the appropriate right-hand sides. This equation is simplified to \( Ax_i = -Bx_b \). The matrix-vector product \( Ax_i \) is the one we tested.

To gauge the effect cache, we varied the mesh sizes, the number of dimensions of the mesh, and the number of “simultaneous” matrix-vector products. By simultaneous, we mean a number of independent matrix-vector products that can be computed at the same time. This situation can arise in solving for multiple right-hand sides, finding multiple eigenvectors, or other reasons.

The experiments were performed on a dual-processor Apple G5 with four gigabytes of memory running Linux. The experiment was single processor, single thread in order to isolate the matrix-vector computation. Execution of the STREAM benchmarking software indicated that the memory bandwidth on this computer is roughly 2150 MB/s. This gives the theoretical maxima of the rates of matrix-vector multiply in Figure 2.15.

The fact that these lines are level indicates the computation rates given are unrealistic. Due to the unpredictable access patterns of the vectors, we would expect to see some slow-down due to unavoidable cache misses as the matrix size grows.

The values in this plot are far smaller than the peak performance measured with flops.c [1], which performs numerical integrations of various transcendental functions. One of the integrations reports a capability of over two gigaflops. Most others report in a range from 250 to 900 megaflops. The theoretical peak performance derived from Eq. (2.2) is in line with these results under the assumption that the computation performance is limited by the memory subsystem.
We performed the timing experiments using the MPI_Wtime function. The results are given in Figure 2.16. For one matrix-vector right-hand side, the code executed at 15% to 20% of optimal performance for systems with 100,000 or more nonzeroes. With four right-hand sides, the code executed at 10%-15% of maximal performance for the same systems. In the end, the performance of the code peaked at 115 MFlops for four right-hand sides with a problem that fit almost entirely in memory of the chip.

Many factors make up this computation, the most important being use of L1 and L2 cache. By L1 cache we mean the memory on the processor chip. And by L2 cache we mean high speed DRAM placed on the motherboard. Generally, to approach this sort of optimization, code needs to be tuned in several ways to avoid abusing the cache [10, 52]. The code as run was tuned only by ordering nodes for good use of cache.

These results are promising in that with appropriate tuning for optimal cache use, this data structure may rival the high performance implementations of MATVEC in PETSc [14, 15, 16], Trilinos [63], or others. Perhaps one of the more interesting results is the less-than-expected gain in performance for performing multiple right-hand side computations. According to [52], this computation should take roughly 1.5 times the single right-hand side computation. In our experiments, this was taking about twice to three times as long.
Figure 2.16: As currently implemented, our data structure is around 12% efficient compared to optimal CRS style sparse matrix-vector multiply.
template <typename MESH>
template <int LHS, int RHS, bool include_boundary>
void laplacianStencil <MESH>::matrix_multiply_row ()
{
    double &result = (*this->diagonal)[RHS];
    result = get_diag_mult() * (*this->diagonal)[LHS];
    for (int i = 0; i != 2*MESH::GEOM_DIM; i++)
    {
        if (*this->off_diagonals[i] != 0)
        {
            if (include_boundary || (*this->off_diagonals[i]->is_boundary())
                result -= (*this->off_diagonals[i]))[LHS];
        }
        else
            break;
    }
}

template <typename STENCIL , typename MESH>
template <int LHS, int RHS, bool include_boundary>
void mesh_operations<STENCIL, MESH>::mat_vec (stencil_set &s)
{
    typename stencil_set::iterator cur = s.begin();
    while (cur != s.end())
    {
        cur->second.template
        matrix_multiply_row<LHS,RHS,include_boundary> ();
        cur++;
    }
}

Figure 2.17: Sparse matrix-vector multiply routine for Laplacian stencil.

2.5.3 Future Work

First, this work needs to be extended to different processing units and different problems. The Laplacian stencil is simple and the algorithm takes advantage of this simplicity. Notice in Figure 2.17, in the matrix_multiply_row function, the multiplication of the \(-1\) off-diagonal is optimized out by the programmer. This sort of optimization is not possible within CRS style matrix-vector product routines. Other optimizations may have different impact. For instance, optimizations that reduce the multiplication overhead in the assembly of global stiffness matrices in the finite element method may show similar improvements [120].

Also, rigorous cache tuning can be pursued. To obtain the results in [51] and [52], the researchers tuned the code to make the most efficient use of cache. In particular, they implemented three optimizations including node reordering. We made use of the node reordering in our experiments. Other cache-based optimizations can be implemented via the template interface so that they can be tuned for particular architectures.

The code snippet in Figure 2.17 shows the MATVEC implementation. The code does not do any data blocking or memory request reordering to improve
cache use. Rather, it is a straightforward implementation of the algorithm given in Figure 2.7.

Finally, we wish to implement the same timing experiments using PETSc on the same computer. According to [52], they achieve a 50%-70% optimal rate for the hardware they tested. While we do not expect there to be a radical difference in their results, it would help complete the comparison and accurately measure the performance of our data structure against a very successful solver code.

2.6 Parallel Algorithms

As we have demonstrated, our data structure performs computation with reasonable efficiency and has the potential to compete with mature high performance libraries. One thing we have not yet demonstrated is Criterion 2.1: exhibition of natural parallelism.

Generally, we speak of algorithms exhibiting natural parallelism. As we will show, it is straightforward to establish that an implementation of an algorithm exhibits parallelism through careful profiling and timing experiments. To think of a data structure as being parallel generally means the data structure works in parallel. We wish to alter that perception.

As mentioned in Chapter 1, we seek to isolate the pernicious 10% of code that takes so much time. One way to do this is to isolate unrelated portions of software from each other. For instance, the conjugate gradient implementation depicted in Figure 2.5 shows that parallelization of the algorithm had significant impact on the software. Why?

The conjugate gradient method can be phrased in terms of matrix-vector products, daxpy operations, and dot products (see Figure 2.18). Each of these components is made of parallel code; the developers of sophisticated parallel libraries have subroutines to perform each of these tasks. But technically, there should be no reason why a conjugate gradient method coded in serial and parallel should be radically different. In fact, the only step that requires explicit parallelization is the test for convergence.

The underlying matrix-vector routines, daxpy operations, and dot products do require explicit parallelization. Some portions of a vector need to be shared and some need to be ghosted. Shared and ghosted data are the data that reside on multiple compute nodes. For instance, if node \( n \) is on both compute node \( i \) and compute node \( j \), then it is shared or ghosted. A datum is shared between two compute nodes if there is a stencil on both nodes that contributes to the value of the datum. Otherwise, the datum is ghosted on the node that does not contain such a stencil.

During each of the linear algebra operations, the shared and ghosted data must be communicated with neighbors. Upon receipt, the compute node will need to update the data it contains. For ghosted data, the currently stored...
ConjugateGradient($A, b$)
1. $x_0 \leftarrow$ Starting vector
2. $r_0 \leftarrow b - Ax_0$
3. $p_0 \leftarrow r_0$
4. $k \leftarrow 0$
5. while not converged do
6. $\alpha_k \leftarrow ||r_k||^2 ||p_k||^2_A$
7. $x_{k+1} \leftarrow x_k + \alpha_k p_k$
8. $r_{k+1} \leftarrow r_k - \alpha_k Ap_k$
9. $\beta_k \leftarrow ||r_{k+1}||^2 ||r_k||^2$
10. $p_{k+1} \leftarrow r_{k+1} + \beta_k p_k$
11. $k \leftarrow k + 1$
12. 13. return $x_k$

Figure 2.18: In the conjugate gradient method, only the test for convergence needs to be parallelized explicitly.

values are always replaced with the incoming data. Shared data, if they need to be communicated, are summed with the currently stored values.

Depending on the type of matrix-vector computation, the communication of shared data may or may not be necessary. Consider a finite difference stencil as in Eq. (2.4). If nodes are ghosted to complete the stencils for all the shared nodes, then the shared nodes will be computed redundantly. However, for finite element methods, nodes do not need to be ghosted to complete stencils. In this case, the stiffness matrix $K$ is made up of the sum of all the local stiffness matrices $K_l$. These local stiffness matrices are implemented as stencils. Since there are no ghosts under this scheme, the shared data must be summed between compute nodes. For example,

$$Kx = \left( \sum_i K_i^t \right) x = \left( \sum_{j=0}^{N} \sum_{i \in p_j} K_i^t \right) x = \sum_{j=0}^{N} \left( \sum_{i \in p_j} K_i^t x \right), \quad (2.6)$$

for $N$ the number of compute nodes and $p_j$ an enumeration of the compute nodes.

Computing dot products is a little more complex. Obviously, a compute node need not consider ghost values when computing dot products. However, shared values should be used in the calculation only once. Given this concern and the requirements of the matrix-vector product, we designed our communication strategy.

First, along with the regular mesh data structure, the mesh maintains a ghost mesh data structure. As mentioned in Section 2.3.2, the mesh data structure

---

4Local in the sense of local support
instantiates a class that instantiates a recursive data structure of mesh object factories. A second instance of this class is used to store ghost mesh data. In this way, data ghosted on the compute node will not interfere with stencil creation. Further, storing the data in this manner allows for the ghost data to be integrated seamlessly into stencils since ghosts and local objects have exactly the same type.

The mesh data structure also tracks which of its objects are ghosted or shared. These are stored in `std::deque` containers stored in `std::map` containers keyed with the MPI rank of the appropriate neighbor. The `std::deque` containers are sorted in the mesh object index order. The actual MPI messages sent between compute nodes is exactly the same as other systems for maintaining communication lists: the data sent are in an expected order that can be unpacked on the receiving end.

### 2.7 Parallel Scalability

The efficiency of software on a single processing unit is easy to demonstrate. The core can compute only so many floating point computations per second, limited by the known speed of the gates in the integrated circuit. Simply timing a computation, counting the number of operations, and comparing against the known speed of the gates gives a very accurate measure of how efficiently software uses the processing unit.

Performing the same evaluation on parallel software is much more subtle. Simulation partitioning, problem domain, interconnection networks, cache effects, and other difficult to quantify variables radically affect the performance of code on a parallel supercomputer. For this reason, it is common to measure parallel speed-up, a ratio of wall clock times relative to the number of processors used in a given execution of the code. But, measuring two different problems to demonstrate scalability is perilous—the sheer number of variables to control for can render even the most careful experiment ineffective. In order to minimize the obfuscation due to these effects, we use four indicators of scalable parallel applications.

#### 2.7.1 Parallel Efficiency

Consider some mesh-based computation such as a matrix-vector product. In parallel, for a constant matrix, the amount of time necessary to perform the computation will decrease as the number of compute nodes used increases, down to some minimum, as depicted schematically in Figure 2.19. If the computation substantially decreases the wall clock run time for each added compute node, the application is *computation bound*. A computational bound simulation has enough work to perform that most of the communication can be performed simultaneously with computation.
Figure 2.19: For mesh-based computation, as the number of compute nodes in a simulation increase, the time to completion reaches a minimum then starts to increase.

Sometimes, to demonstrate parallel efficacy, experimenters will show linear or near linear speed-up. By linear speed-up, we mean that as the number of processors doubles, the time to execute the code halves. These studies concentrate on the computation bound portion of the curve in Figure 2.19. These tests are important since they are proof that the implementation is making use of most, if not all, of the available parallelism. This is the first indicator we use to demonstrate scalability.

As the number of processors increases for a fixed problem, Amdahl’s law [4] governs the curve. Amdahl’s law is based on the simple observation that any software implemented in parallel can be divided into two parts: the serial, $\sigma$ and the parallel, $\rho$. On a single processor, the time $T$ to complete execution is the sum of the time to perform the serial part $\sigma_t$ plus the time $\rho_t$ to perform the parallel part. Adding compute nodes to the computation reduces the amount of time needed to perform the parallel part. Unfortunately, the extra compute nodes cannot affect the rate at which the serial portion completes. Formally stated, for $p$ compute nodes, Amdahl’s law bounds the minimum time to execute software in parallel:

$$T_p \geq \sigma_t + \frac{\rho_t}{p}, \quad (2.7)$$

with a strict equality in the case of $p = 1$. Basically, the relative increase in performance diminishes as the amount of serial work dominates the calculation.

Eventually, the marginal increase in communication volume exceeds the marginal decrease in wall clock time due to adding another compute node to the computation. This can happen for several reasons. For instance, the parallel
portion of the code can be divided into parallel pieces only so many ways. A matrix-vector product contains a certain number of floating point operations and can be divided into so many pieces before the work can no longer be split up. In the end, mesh-based parallel codes will become communication bound. A communication bound simulation does not have enough work to hide a significant portion of the communication.

Amdahl’s law is a useful guide for understanding how changing the number of compute nodes will affect the run-time of a particular simulation, even if the logical conclusion reached ad infinitum is pessimistic: \( T \geq \sigma_t \). Further, it offers no predictive mechanism for demonstrating whether a particular algorithm or implementation will be effective for a particular problem on various supercomputers. Luckily, there is another measure of parallel scalability: isoefficiency [49].

The efficiency, \( E \), of a parallel simulation is defined to be the ratio of the serial execution cost to the total parallel execution cost. The serial execution cost is simply the time it takes one compute node to complete the simulation, \( T_s \). The total parallel execution cost is the number of compute nodes, \( p \), multiplied by the time it takes the parallel simulation to complete, \( T_p \). Mathematically,

\[
E = \frac{T_s}{pT_p} \tag{2.8}
\]

An algorithm for which \( E \) is bounded away from zero as \( p \to \infty \) is said to be scalable. A necessary condition for scalability is that the serial portion of a problem must be inversely related to some aspect of the problem. This can be seen from applying Amdahl’s law to Eq. 2.8:

\[
E \leq \frac{T_s}{T_s + (p - 1)\sigma_t}. \tag{2.9}
\]

Unless \( \sigma_t \to 0 \) as \( p \to \infty \), the algorithm is not scalable. If the problem can be altered in such a way that the serial portion can shrink as \( \sigma_t = \mathcal{O}(1/p) \), then the number of compute nodes will dictate the problem parameters to ensure that the upper bound on efficiency does not approach zero. For mesh-based simulations, increasing the number of cells in the mesh tends to reduce the serial portion of the computation. The isoefficiency function describes how fast a problem must grow in terms of the number of processors so that \( E \) remains constant as the number of compute nodes is increased.

As such, we can measure the wall clock time in terms of how many cells are on each processor, as in Figure 2.20. This plot is similar to Figure 2.19 with the exception that the communication bound portion and the computation bound portion are on opposite ends of the graph. Scalability studies of mesh-based applications are often described in terms of cells per compute node.

Isoefficiency analysis has one subtle caveat: if the problem size grows too fast, then the wall clock time necessary to complete the task will increase at
an unacceptable rate. Even though it may be technically possible to increase
the problem size to maintain a particular efficiency, the algorithm may still be
considered unscalable. For instance, if the size of the problem must grow as the
square of the number of compute nodes to maintain constant efficiency, then
the algorithm may not be fit for use on large supercomputers for two reasons:
a problem large enough for scalability may not exist, and the amount of time
necessary to solve the problem may be unacceptable.

2.7.2 Surface Effect

In practice, computing the isoefficiency function described above is not possi-
ble. Rather, the problem domain is idealized. For instance, the variable $\sigma_t$ is
extremely sensitive to the partitioning of the mesh. Various assumptions about
the partition are made to aid the calculation. The shape of the domain also
strongly influences $\sigma_t$. Partitions that lie on the surface of a mesh have a re-
duced communication volume compared to those that are strictly on the interior
of the mesh. Communication volume, in general, varies directly with $\sigma_t$. Some
domains have a better communication profile than others.

A cube demonstrates this effect. Let a domain be some cube meshed and
partitioned into smaller cubes of equal size and equal number of cells per com-
pute node. Assume only facet neighbors need to communicate. Then, interior
partitions communicate with six other partitions while surface partitions com-
municate with fewer. On such a cube, partitioned into 1000 pieces, the actual
communication volume is 90% of the idealized amount used in estimating effi-

Figure 2.20: Scalability results in mesh-based computations are often reported
in number of cells per compute node.
Figure 2.21: A surprising number of mesh partitions have a reduced communication volume due to their location, even on a cube.

Complex domains, in general, have a more favorable ratio of surface area to volume. The actual communication volume is often much less than the theoretical volume given by the cube. On the downside, the curve in Figure 2.21 is monotonically increasing. As the number of compute nodes increases, the communication volume will increase by more than the theoretical results indicate. For mesh-based simulations, increasing $p$ drives $\sigma_t$ in the wrong direction. This surface area effect can lead to scalability studies that do not indicate theoretical or desired performance.

One indication of this effect is the scaling width. In Figure 2.20, we indicate the area between computation bound and communication bound regions of the plot as the scaling width. This width will shrink as the actual communication volume approaches the theoretical communication volume. But for truly scalable applications, this width will never disappear. This is the second indicator we use to demonstrate scalability.

### 2.7.3 Measuring Scalability

When comparing execution times of several simulations of different mesh sizes, we expect to see similar curves for each mesh, with the curves shifted according to a derived isoefficiency function. If the mesh size must grow faster than the number of compute nodes to maintain efficiency, we expect to see an increase in overall run-time. This corresponds to a shift upward in both Figure 2.19 and 2.20. Also, since there will be more cells per compute node, this causes a shift
to the right in 2.20. Since the idea of isoefficiency is to increase the number of processors used while maintaining a similar efficiency, we expect the curves to be shifted to the right as the problem size grows as in Figure 2.22.

If we pick points on these curves where the efficiency is the same, we can plot a scaling curve. The scaling curve is related to the isoefficiency function described above. The isoefficiency function describes how the work load must increase for a parallel application to maintain its efficiency. Work done by the simulation, $W$, is related to walk-clock time on $p$ compute nodes as $T_p = W/p$. This implies that the scaling curve is the isoefficiency function divided by the number of compute nodes. The scaling curve is the third indicator of parallel scalability. If the scaling curve increases linearly or faster, the algorithm scales poorly. The size of the problem would need to grow too rapidly for increased numbers of compute nodes, thereby requiring unacceptably long simulation times.

Computing the scaling curve requires computing the efficiency of the implementation. For various reasons, obtaining the serial time $T_s$ for the efficiency computation may be difficult. Rather, we choose to compute an approximation we call $p$-efficiency. This efficiency is the ratio of the work done on distinct numbers of compute nodes. For instance, the 2-efficiency of a simulation using 20 compute nodes is

$$E_{2,20} = \frac{2T_2}{20T_{20}}.$$  \hfill (2.10)

When plotted on a curve, the efficiency of a computation decreases as the number of compute nodes increases. For scalable applications, the efficiency
curve for larger simulation problems should lie above the efficiency curve for smaller problems. This is our fourth indicator of scalability. Unfortunately, memory hierarchy effects can skew the efficiency plot. For instance, for a large problem on a small number of compute nodes, the data being operated on may not fit into cache. As the number of compute nodes increases, more and more of the data can reside in cache. This apparent increase in efficiency happens on fewer compute nodes for smaller problems, leading to an inversion in the $p$-efficiency plot.

### 2.8 Reading Serial Meshes

The first step of many mesh simulations is reading mesh data from disk. The data are generally stored in a file that has at least two parts: a list of node locations and a list of mesh objects. Usually, a single compute node reads the data and prepares it for computation. This process is obsolete for large meshes on 64-bit computers. On these architectures, a 100 million cell mesh requires more than eight gigabytes of RAM in contiguous memory. Even if the compute node possesses the ability to allocate the memory, there is no guarantee that the data will fit in RAM.

To avoid this problem, we have developed an algorithm to construct the mesh in parallel. The first step of the algorithm is to construct a skeleton of the mesh based on the data on disk. This skeleton is then partitioned and the remaining mesh objects are constructed in serial for reasons given in the next section.

We present this algorithm for several reasons. First, the data structure we developed is memory intensive. For moderate to large meshes, a single compute node cannot hold the entire mesh in memory. Second, this algorithm also demonstrates how difficult demonstrating parallel scalability really is. We show that this algorithm does exhibit limited scaling by some metrics but scales poorly by others. In the end, we conclude this algorithm is not very scalable. This does not mean it is not useful, however, since it needs to be executed only once over the entire life of a mesh.

#### 2.8.1 Algorithm

The algorithm is divided into three phases. The first phase is limited by the speed of the disk head and network file system. Each processor reads a contiguous block of data from the mesh object description portion of the serial mesh file. If the mesh objects described are cells and there are $c$ cells in the file, then each of $p$ compute nodes reads $\lfloor c/p \rfloor$ cells, with compute node $p-1$ picking up the balance.

Once the mesh object descriptions are read, each processor reads the appropriate nodes from the node location portion of the mesh file. The nodes are
directly associated with the cells without creating the edges, facets, or other mesh objects. At this point, the data structure contains all the information necessary to construct the mesh. A rough partitioning of the mesh is computed to reduce the amount of communication necessary to complete the construction of the data structure.

We employ a very simple geometric partitioning for this step. Consider a sliced loaf of bread. The cuts in the bread are orthogonal to the long dimension. The rough partitioning works very similarly. The longest Cartesian axis is computed using a collective reduction operation. Once this is computed, the mesh is over-partitioned geometrically into pieces along this line.

To accomplish this, a number of bins are set up along this long dimension. These bins are of fixed width, e.g., with one thousand bins on a domain that is 1 unit long, each bin will be .001 units across. Next, each cell is placed into the appropriate bin based on its location along the axis. In this case, we compute the centroid of the cell using the appropriate coordinate.

Once the mesh has been placed in these bins, the bins are placed on the appropriate compute nodes. The bins are assigned as one would expect, neighboring bins are grouped together until the number of cells in the group exceed \( \lfloor c/p \rfloor \). This assignment starts at one end of the domain and proceeds to the other. In this way, the mesh is roughly partitioned in a manner similar to sliced bread.

After partitioning, the remaining mesh objects are constructed. To ensure that the objects are assigned distinct identifiers, this construction is done in serial. The compute nodes obtain ghost cells from neighboring processors. The ghosts share a node with a cell in the partition. Then, in turn, each compute node constructs its mesh objects, including for the ghost cells. After construction, the compute node sends ghosted cells to its neighbors, and the next compute node can begin construction of its mesh objects. Once the last compute node has constructed its cells, all of the compute nodes delete their ghosts.

This algorithm, fully presented in Figure 2.23, is guaranteed to construct the mesh, no matter how large the mesh is. Anecdotally, our data structure begins to run into memory limitations when reading two million cell serial meshes on one 32-bit compute node. On 64-bit compute nodes, where pointers native types are twice as large, one million cell meshes and larger require multiple compute nodes. This algorithm guarantees that a mesh of any size can be read, given enough compute nodes. Also, this needs to happen only once for the lifetime of the mesh. Once the serial mesh is in memory, it can be written out in a binary format native to the data structure. This binary format can be read in embarrassingly parallel fashion.
**READ SERIAL MESH**($fname, M$)

1. $p \leftarrow$ current compute node numbered from 0
2. $s \leftarrow$ total number of compute nodes
3. $c \leftarrow$ number of mesh objects in $fname$
4. $l \leftarrow \lfloor pc/s \rfloor$
5. $h \leftarrow \lfloor (p + 1)c/s \rfloor$
6. **for each** mesh object $c$ in file $fname$
   **do**
   **if** $l \leq c.id < h$
   **then**
   - Create mesh object $c$ in $M$
   - Create nodes for mesh object $c$ in $M$
   - Add nodes from file to $c$

7. **for each** node $n$ in file $fname$
   **do**
   **if** $n \in M$
   **then**
   - Read node location

8. Compute rough partitioning
9. Ghost nodes
10. **for** $i \leftarrow 0 \ldots s - 1$
11. **do**
12. **if** $p = i$
13. **then**
   - Receive ghost information
   - Receive highest used identifiers
   - Construct all cells
   - Send ghost information to neighbors
   - Send highest used identifiers to $p + 1$

---

Figure 2.23: Algorithm to read arbitrarily large serial meshes in parallel environment.
2.8.2 Scalability Study

In one sense, this algorithm scales poorly. No matter how many compute nodes in the simulation or cells in the mesh, the total number of cells created per second never exceeds some threshold. The reason for this is simple: in three-dimensional meshes, there are many more faces and edges than cells and nodes. As an example, consider the results of loading three different size meshes on an Apple G5 2GHz supercomputing cluster with 2 GB of RAM for each compute node. The wall clock times are given in Figure 2.24.

In another sense, the algorithm shows some scalability. Figure 2.25 demonstrates that as the problem size gets larger, the $E_2$ efficiency increases for a given number of compute nodes. The reason for the efficiency gain is the growing portion of the computation that can be parallelized. The amount of data that can be processed in parallel is a sizable chunk of the overall data processed. Anecdotally, as much as 30% of the mesh can be constructed in parallel. Why, then, the poor performance?

Even though the algorithm demonstrates that the mesh objects and nodes in the file can be read in parallel, there are only $O(1)$ hard drive “heads” reading the data. The parallel portions of the code are implicitly serialized by the underlying storage media. This serial bottleneck explains why there is a cap on the overall performance of the algorithm, no matter how many processors are employed to perform the computation.

One important consideration when employing this algorithm is the overall wall clock time of the mesh processing. Once the algorithm begins to lose an unacceptable amount of efficiency, there are steps that can be taken. For instance, once the derivative of the lower curve depicted in Figure 2.24 becomes negative, then the wall clock time on the mesh load will begin to increase. To prevent this from seriously impeding performance, we recommend that the algorithm be implemented on a limited number of processors so that the distributed data structure can be constructed.

For instance, the number of cells on a compute node for scalable implementations of various simulations is generally less than one hundred thousand (and more often less than twenty thousand). When constructing the parallel data structure under these conditions, the serial processing may take hours. Rather, we suggest constructing the data structure on far fewer processors once, saving this result in the provided native parallel format, and using this file for subsequent simulations.

2.8.3 Future Work

The current implementation can be improved in two different ways. The first is to use a better mesh partitioner than the one currently employed. One way to do this would be to construct a node neighbor dual mesh and use one of the several mesh partitioners that are available with this data structure. For instance,
Figure 2.24: Overall performance of reading serial meshes from disks never exceeds 6500 cells per second, no matter how many processors are used or how large the mesh is.
rather than using the simple bread loaf strategy, a Cartesian nested dissection partitioner can be employed both to balance load and to reduce communication volume.

Perhaps the greatest limitation of this algorithm is the obvious serial thread that runs through the second half of the algorithm: constructing the supporting mesh objects. One way this serialization can be minimized is by moving the computation of non-ghosted data out of the loop and making it parallel. Identifiers can be assigned uniquely on each processor by using the modulus operator. Using the same notation as in Figure 2.23, each compute node can use an identifier \( i \) if \( i \mod s = p \). Then the serial step would be to unify identifiers of objects between processors. The lowest numbered compute node that contains an object would have the authoritative identifier for that object. All other neighbors that contain that object would update their copy to that of the lowest numbered compute node.

By reducing the overall communication volume and eliminating the bulk of the work in the serial step, we hope to make this algorithm somewhat scalable. Even though the cost of the algorithm can be amortized over the life of the mesh, we hope to reduce this cost and make loading an exceptionally large serial mesh into distributed memory less costly and remove the extra step of saving a serial mesh into a parallel file.

Figure 2.25: As problem size increases, algorithm exhibits some gains in efficiency.
2.9 Software Development

There are two kinds of software development with our data structure. The first kind we call *method development*. In method development, the algorithms are developed, tested, and debugged. For instance, if a developer wishes to code a new iterative solver, the new code would be method development. In terms of the actor interface, the developer writes code to the interface provided by a template.

The second kind of development we call *application development*. The application developer selects the actions necessary to accomplish a task. Then, the application developer will choose among the various methods that implement the interface of the action.

In this way, someone coding a file input/output routine, for example, need not be concerned with how boundary conditions are stored. Another programmer implementing a new preconditioner need only code the matrix-vector multiply or solving routine without worrying about which solver is to be used. In fact, this programmer could easily switch between solvers to test the efficacy of the new preconditioner.

This illustrates some of the difficulty of software development. The preconditioner developer is simultaneously developing both a method and an application. To test the preconditioner, the developer needs a solver, a linear system representative of the types of problems the preconditioner is trying to accelerate, a mesh, and other parts of a complete simulation. The action interface we developed for our data structure provides a clearly defined boundary between method and application. This allows the preconditioner developer to assemble the relevant simulation in relatively few lines of software.

The semantics used in C++ are different for each type of development. Generic method programming requires the programmer to flag templates and templated types so the compiler can correctly interpret the syntax of the code. Since the compiler does not know what the exact data structures are during development, explicit indication of templates is necessary. On the other hand, application development does not require the same syntax since the application developer is composing instructions for the compiler to generate code. For instance, an application developer may loop over nodes in a mesh using the following for loop:

```c++
for ( node_iterator n = m.begin<node>();
    n != m.end<node>(); n++ )
```

with the body of the for loop querying the node in some way. A method programmer would need to flag the begin and end functions as templates (e.g., `m.template begin<node>`).

We believe the unanticipated difficulty of the pernicious 10% occurs when method programming and application programming overlap. For instance, when external libraries and developed methods need to be combined, the “glue” used
is often dependent on subtle interactions between the libraries. The individual pieces of the simulation are simple to code. Even the most complex solvers are straightforward applications of available libraries, which themselves are simple to code. Programming a mesh partitioner is easy, as most are simple nested dissection based algorithms. Compiling matrices and preconditioners are not that difficult. Why, then, does parallel programming require so much effort?

There are two reasons for this, one beyond the scope of this thesis and one we can address directly. Anecdotally, the number of “bug-free” lines of code written per day by a parallel developer is far fewer than that of a mainstream developer working in a serial environment. The lack of truly useful parallel debugging environment hampers the development of parallel code, cripples developers, and clogs supercomputing queues with known faulty code executed only to find bugs. This is a serious problem and one beyond our purview.

To mitigate this problem, the total number of lines a developer must write to have a working application must be minimized. Some would say simply standardizing on a particular framework or using a particular code base exclusively is the best way to solve this problem. This is only part of the story. First, no single code base can optimally answer a specialized problem because every single code base has made compromises in design to allow for breadth of scope. To use a library is to compromise.

We offer a different approach altogether. Rather than providing libraries, we provide a mechanism to join libraries together. It is technically possible to use any library with our data structure. As long as Criterion 2.4 is not violated across actions, we believe we can reduce the amount of code necessary to build a working application while providing the developer with tools to optimize the libraries with respect to meshes.

What a developer receives in a library compromise is a suboptimal solution. For instance, using a generic sparse matrix-vector solver on a mesh-based problem will lead to inefficiencies unless the simulation designer writes extra lines of code to reorder the matrix. Some libraries will do this for you at the expense of recreating the mesh in their data structure. Even then, developers have to write hundreds of lines of code that just translates data from one format to another.

Another place developers need to write hundreds of lines of code is in maintaining parallel data structures. This workload is increased when the mesh, any derived matrix, and list of vectors are also maintained in parallel. Our data structure reduces the amount of code that needs to be written by using generic programming and literally storing everything on the mesh. By managing the mesh, all other data are handled automatically. This requires fewer lines of code, which means fewer bugs and faster development time.

We have identified another culprit that extends development time unnecessarily and developed a remedy for its solution. The action interface offers an unambiguous input and output for the various methods used in mesh-based engineering simulations. By this, we mean the mesh exists in memory in a
prescribed fashion before and after the call. There is never any need for a sim-
ulation developer to write code that prepares the mesh specially for the actor
call, nor does the simulation developer need to translate the results. By en-
forcing Criterion 2.4, all mesh transforms happen on the mesh data structure
directly, and we reduce the amount of code required to program a simulation
by hundreds or thousands of lines.

We feel that Criterion 2.4 is the most important of all. While violation of
this Criterion within a method may lead to increased performance, violation
across actors leads to more software to write. This, we feel, is the heart of the
pernicious 10%. Stated more generally, a library takes some data structure $x$
and transforms it to $x'$. Unfortunately, simulation developers have data structure $y$
and need $y'$. Programming in parallel exacerbates the problem since the $y \rightarrow x$
and $x' \rightarrow y'$ transforms need to happen in parallel.

In summary, lines of code are the enemy of productive parallel development.
Consider how often the reader has heard a statement like “I have method $x$
working well, but I am having a tough time getting the result to work with
library $y$.”? In our experience, many parallel software development complaints
sound like this, be it coupling mesh management routines with legacy solvers,
applying new development techniques to existing solutions, or even restarting
from checkpoint backups. We have demonstrated that it is possible to reduce
the amount of coding for some phases of application development of numerical
simulation to one line.

Reconsider the algorithm given in Figure 2.4. We propose that the main
function of actual, relevant, high-performance engineering simulations could
look like this, implemented in no more than a dozen lines of code. Each in-
dividual method will still require hundreds or thousands of lines of code; there
is no escape from the actual coding of the methods. However, by keeping the
methods completely independent of each other, we expect the amount of time
necessary to develop these codes to be dramatically reduced.

2.10 Future Goals

We hope to increase the usability of our data structure in the future. In par-
ticular, we see four areas that need development to enhance the viability of the
data structure: interface development, tool development, library development,
and C/Fortran integration.

By improving in these areas, the data structure described above will be more
attractive to the audience for which it is intended: software developers of large-
scale, mesh-based engineering simulations. In addition, the ease with which
simulations can be constructed should lead to more efficient rapid prototyping
in a parallel environment.

There are many possible actions that can be implemented for the mesh data
structure. For instance, mesh motion, smoothing, and generation generally
have a common interface: take a mesh as input and receive a mesh as output. Transferring solutions between meshes in a coupled simulation is another action that can be introduced. For instance, one popular way to achieve this is to compute a common refinement on the interface between meshes and use this to transfer solutions [71]. This and other methods for solution transfer can use the same action.

Of course, the actors themselves are not useful without methods to use with them. We will demonstrate in the following chapters that development of algorithms using our data structure is simple and efficient. This reduces the total amount of software that must be written, shortening software development time. In the near future, we hope to offer several simple preconditioners and implement several iterative solvers.

To speed adoption, though, we can use existing libraries and write the supporting code necessary to translate between the library data structures and our data structures. For instance, we have written translation code for the ParMetis library [75] so that we can compare our partitioner developed in Chapter 4 to a widely deployed and well understood library. In the near future, we hope to employ mesh smoothing software such as Mesquite [25].

This leads to the final hurdle for widespread adoption: integration with C and Fortran. This will require a tedious reimplementation of the action interface. For each action, each available method must be instantiated and placed into a library. Object handles, integers that point to objects in memory, will be employed to track the mesh, mesh objects, stencils, and other compiler-generated data types. Transferring much of the flexibility of the data structure will prove tricky.

2.11 Summary

Our development of the data structure was motivated in two ways. We first developed a prototype data structure and used it to perform mesh partitioning for large scale supercomputing applications. The most difficult part of the task was transferring data from our data structure to that of the simulation. The lessons we learned in this exercise were applied to the second generation data structure.

Second, we fine tuned our data structure during the development of several parallel applications. This taught us how our most basic assumptions, such as that data in a matrix-vector product must be random access, can be relaxed if the application we develop for can be generalized. In this case, we chose mesh-based simulation. We now present several novel algorithms we developed that helped shape the data structure, starting with the parallel computation of the medial axis.
Approximating the Medial Axis in Parallel

Do not hover always on the surface of things, 
nor take up suddenly, with mere appearances; 
but penetrate into the depth of matters, 
as far as your time and circumstances allow, 
especially in those things which relate to your profession. 
— Isaac Watts

Scientific principle always works! With these X-Ray Specs 
you apparently see through flesh and peek at the bones underneath! 
— Novelty Advertisement (1987)

A critical step in the hybrid mesh partitioner developed in the next chapter is the approximation of the medial axis transform (MAT). Originally defined over three decades ago by Blum [23], the medial axis can be thought of as a skeleton of a compact set in a space. Intuitively, Blum describes the medial axis as the result of a prairie fire. If a fire is ignited along the boundary of a field and the fire burns uniformly, the set of places where the fire runs out of fuel is the medial axis.

The definition of the medial axis transform used in this chapter relies on the concept of a maximal ball.

Definition 3.1. A ball $b(c, r)$ with center $c$ and radius $r$ is said to be maximal on a compact set $S$ in a metric space if and only if $b \subseteq S$ and there does not exist a ball $b'$ such that $b \subset b' \subseteq S$.

In other words, a maximal ball on set $S$ is not contained within another ball contained in $S$. In Euclidean spaces, a maximal ball is tangent to the surface of the containing set $S$ in at least two places. For the rest of this discussion, we assume $S$ is compact.

We formally define the medial axis and MAT as follows.

Definition 3.2. The medial axis of a compact set $S$ is the locus of the centers of all maximal balls on $S$. (Figure 3.1)

Definition 3.3. The medial axis transform of a compact set $S$ is a scalar function defined on the medial axis, whose value at a given point is the radius of the maximal ball for which the point is the center.
In some ways, the medial axis simplifies a figure. The medial axis of an $n$-dimensional figure is made up of “sheets” of dimension $n - 1$ or less. In three dimensions, for instance, the medial axis is generally a set of surfaces, edges, and points and can be referred to as the medial surface. To maintain a mesh neutral discussion, we denote the construct as a medial axis, regardless of the dimension.

The medial axis has applications in the varied fields of computer vision [108], robotics [78], medical imaging and figure segmentation [20, 81], feature extraction [65, 67], image processing [57, 70] and mesh generation [95, 103]. As such, there has been a substantial amount of research into the approximation of the medial axis transform. The next section discusses some notable results from the literature.

### 3.1 Background

Since Blum defined the medial axis, many algorithms have been developed to compute it. Unfortunately, the medial axis is inherently unstable. Small changes in a surface can radically alter the medial axis. Discretization of a domain can introduce sharp features into the surface. These features introduce spurious “hairs” or “sheets” into the medial axis that are not present in the medial axis of the original domain. Therefore, it is insufficient simply to compute the medial axis of the discretized set. A robust algorithm must determine whether portions of the approximate medial axis belong to the medial axis of the underlying object or are spurious artifacts introduced by discretization. As a result, there are many algorithms designed to compute the medial axis.

#### 3.1.1 Previous Work

Foskey and Lin [44] list more than a dozen algorithms and provide a simple classification system for the various algorithms: thinning algorithms, algebraic solutions, surface sampling methods, and distance field computation. Thinning algorithms propagate the surface of a figure based on certain erosion operations. Algebraic solutions directly compute the sheets of the medial axis of a polyhe-
dron. Sampling methods compute the medial axis from a subset of the Voronoi diagram of a set of sample points. Finally, distance field computation algorithms approximate the gradient of the signed distance function and attempt to isolate discontinuities.

Thinning algorithms can be parallelized [42, 43], but they rely on a pixel or voxel description. Unfortunately, any reasonable approximation requires a substantial amount of computation since there are no interpolations in the description or final result.

Algebraic solutions directly compute the points, lines, and surfaces of the medial axis from closed form solutions. The more successful of these algorithms require arbitrary precision arithmetic to compute the medial axis stably. Further, these algorithms require an enormous amount of computing resources to compute the medial axis, even for simple polyhedra [33].

Current sampling methods have several drawbacks. Despite solving a more difficult problem than the previous algorithms, these methods are extremely sensitive to the sampling of the surface. In the worst case, these algorithms can cost $O(n^2)$ where $n$ is the number of sample points [44]. Also, user supplied parameters can dramatically alter the result of the computation. Without a keen knowledge of these methods and how they work, the parameters can be easily misused. Finally, these algorithms are notoriously difficult to parallelize. For more information on these methods, see [11].

Distance field computations rely on approximating the distance of a point in space to the surface of the figure. Some use this information with the Voronoi diagram to determine the location of the medial axis. Others compute an approximation to the gradient of the distance field. These identify discontinuities in the gradient of the distance field to identify the medial axis. For instance, it is possible to link these discontinuities to non-zero divergence integrals over small portions of the domain. Many of these algorithms rely on parameters and thresholds for identification of the medial axis. Some can be adapted to distributed memory environments.

3.1.2 Medial Axis Simplification

Unfortunately, computing the medial axis is an ill-posed problem. Consider a rectangle with a circular “bump” on one side with radius $\epsilon$ as in Figure 3.2. Provided $\epsilon > 0$, there is a “hair” in the medial axis, represented by $h$ in the figure at right. If $\epsilon = 0$, then the hair disappears as in the figure at left. This trivial example shows that the medial axis does not depend on the shape of the surface in a continuous manner. Disheartening as this may be, there is no shortage of shapes, from smooth to exotic, that yield ever more degenerate medial axes [31].

To make computation feasible, algorithm designers make assumptions about the inputs and resulting axes. These simplifications remove, alter, or ignore
Figure 3.2: As $\epsilon \to 0$, $h$ is not altered, but for $\epsilon = 0$, $h$ disappears.

situations that lead to problems. For many applications that use the medial axis, the bump from the previous example serves only to complicate the results. The resulting hair would muddy calculations and, therefore, figures such as those in Figure 3.2 are judged to be problems for the algorithm or are handled in such a way as to make them continuously dependent on the input. These assumptions are a necessary evil of any medial axis computation.

Unfortunately, situations such as these define the problem of the medial axis. Without delving too deeply into the various assumptions made by algorithm designers, many attempt to measure the “stability” of a portion of the medial axis. From the example above, the stability of $h$ is proportional to $\epsilon$. Compared with the other parts of the medial axis, $h$ is unstable. This leads to many thresholds, parameters, and tweaks that users must customize, depending on how their applications stand up to the assumptions made by the designers. These stabilization factors are also a necessary evil of any medial axis computation. The key to a successful algorithm is finding an algorithm with parameters that are easy to choose.

3.1.3 Eikonal Equation

Generating a good algorithm with a simple stabilization parameter is not the only concern of this chapter. Since the medial axis is used in the parallel mesh partitioner developed later, the algorithm must compute the medial axis in a highly parallel fashion.

Calculation of the Delauney triangulation or Voronoi diagram of a set of points is very difficult to implement in a scalable parallel fashion. For this reason, we will rely on computing the distance field through the solution of a simple differential equation. Consider a wave propagating through space starting at the surface of a set at a speed given by some function $f$. Let the function $t(x)$ describe the time the wave passes through the point $x$. As motivated by
Figure 3.3: Wave propagation models illustrate difficulties in computing medial axis.

Siddiqi, et al. [105], the solution to the partial differential equation

$$f(x) ||\nabla t(x)||_2 = 1,$$  \hspace{1cm} (3.1)

with $t(x) = 0$ if $x$ lies on the surface of the set, gives a function $t$ whose value is the time the wave passes through the point $x$. Note that $\nabla t$ is the inward facing normal of the surface when evaluated on the surface. Finally, in the case of determining the medial axis, $f(x) \equiv 1$.

This hyperbolic partial differential equation, called the eikonal equation, is related to wave propagation and can be very difficult to solve. For instance, smooth initial conditions can lead to singularities as the solution evolves. These singularities, or shocks, comprise the medial axis of the surface given by $x$ such that $t(x) = 0$. The Euclidean concepts normal and tangent break down at these singularities.

On the other hand, rarefactions can also occur. As converging characteristics lead to a singularity, diverging characteristics lead to a situation where extremely small areas of the set surface govern larger and larger areas of the domain of $t$. In fact, for sharp, “inside” corners, large swaths of the solution are governed by this feature. Computing the solution to the eikonal equation for these rarefactions is extremely difficult because $\nabla t$ does not exist at such corners. Figure 3.3 shows how wave propagation can indicate some portions of the medial axis and fail to find other portions.

Finally, $\nabla t$ always points toward the medial axis. And since the eikonal equation is hyperbolic, if some region of the domain of $t$ is divergence free, then the region does not have a shock and the medial axis is not in the region.
3.2 A Consistent Algorithm

More general algorithms such as Tight Cocone [37] and Power Crust [8] pose the medial axis problem in terms of a point cloud. For instance, given a set of points, compute the medial axis of the surface described by the points. Unlike these approaches, the one we develop below assumes the existence of a discretization of the compact set $S$. This is a reasonable starting point since, given a point cloud, algorithms are available to compute a surface mesh [5, 6, 7, 36, 68], which can then be volume meshed automatically [45, 103, 104]. In short, the algorithm developed below addresses the most difficult stage of medial axis computation: the computation itself.

The final parallel algorithm is developed through a series of three algorithms. The first algorithm is inefficient, but is easily shown to be consistent in a geometric sense. The second algorithm is efficient and its results are shown to be consistent with the first algorithm, implying that the computed geometry is consistent with the medial axis. The final algorithm is a straightforward parallelization of the second algorithm. For the first two algorithms, simplifying assumptions are made to show consistency. For most meshes, these assumptions are reasonable.

This section develops a simple algorithm based solely on the surface mesh of the discretization of $S$. The algorithm computes points on the medial axis from a simple observation about maximal balls. We then show that the error in the calculation, that is the maximum distance between a computed point and a point on the medial axis of $S$, approaches zero as the discretization of $S$ is refined.

3.2.1 Motivation

The idea behind this algorithm is to compute and successively refine an “upper bound” on the location of the medial axis. Let $S$ be a compact set. The rest of the motivation depends on classifying surface points in $S$.

**Definition 3.4.** A point $x \in S$ is said to be on the surface of $S$ if there is no $\epsilon > 0$ such that $b(x, \epsilon) \subseteq S$. The surface of $S$ is the set of all points in $S$ that satisfy this property. We denote the surface of $S$ by $\partial S$.

Using two points in $\partial S$, a ball $b$ is constructed such that it intersects both points, and there is a maximal ball of $S$, $b_m$, that is contained entirely within $b$. This approach provides an upper bound on the location of the medial axis.

The upper bound developed below depends on computing the normal to a surface, generally defined as $n = \nabla \times f$ in Euclidean spaces. Unfortunately, for many surfaces, there are non-differentiable “features” for which this definition is inadequate (Figure 3.3). The proofs below rely on a more general definition of the normal and tangent.
Definition 3.5. Given a point \( x \in \partial S \), an extended normal of \( \partial S \) at \( x \) is the inward facing normal of any ball \( b \) incident on \( \partial S \) at \( x \) such that \( b \subseteq S \).

Definition 3.6. Given a point \( x \in \partial S \), an extended tangent \( t \) is given by \( t \cdot n = 0 \) for some extended normal \( n \).

Theorem 3.1. For a continuously differentiable surface in Euclidean space, the extended normal is unique and is the normal.

Proof. By definition, any such ball is tangent (in the Euclidean sense) to \( S \) at \( x \). As such, the center of the ball must lie on the normal to the surface. \( \square \)

Theorem 3.2. For a continuously differentiable surface in Euclidean space, an extended tangent at \( x \in \partial S \) is tangent to the surface.

Proof. The proof follows from Theorem 3.1. \( \square \)

Unless otherwise stated, all tangents and normals in this chapter refer to extended tangents and inward facing extended normals. Also, we use the carat (\( \hat{\cdot} \)) to denote a unit vector. These definitions are used to develop the upper bound.

For any point \( x \in \partial S \) with unit extended normal \( \hat{n} \) at that point, the ball \( b(x + \alpha \hat{n}, \alpha) \) is (extended) tangent to \( S \) at \( x \). A ball can be computed tangent to \( S \) at \( x \) and passing through another point \( y \) by solving \( ||x + \alpha \hat{n} - y||_2 = \alpha \) for \( \alpha \). Rearranging this equation gives a function for the radius of the ball:

\[
\alpha(x, \hat{n}, y) = \frac{||y - x||_2^2}{2\hat{n} \cdot (y - x)} = \frac{||y - x||_2}{2 \cos \theta},
\]  

(3.2)

for \( \theta \) the angle between \( \hat{n} \) and \( y - x \).

As mentioned above, a maximal ball is tangent (in the Euclidean sense) to \( S \) in at least two places. We now demonstrate that this is true for the a ball that is extended tangent to \( S \).

Theorem 3.3. The finite extrema of the function defined by (3.2) occur when \( b(x + \alpha \hat{n}, \alpha) \) is tangent to \( \partial S \) at \( y \).

Proof. Taking the directional derivative with respect to \( y \) gives

\[
D_d \alpha = \lim_{\delta \to 0} \frac{\alpha(x, \hat{n}, y + \delta d) - \alpha(x, \hat{n}, y)}{\delta} = \frac{2d \cdot (y - x)n \cdot (y - x) - ||y - x||_2^2 \hat{n} \cdot d}{2||\hat{n} \cdot (y - x)||^2}.
\]

(3.3)

Setting this equal to zero and solving shows that extrema occur when \( d \cdot (c + \alpha \hat{n} - y) = 0 \), that is, whenever \( d \) is an extended tangent of \( \partial S \) at \( y \). \( \square \)

Theorem 3.4. Given a compact set \( S \), \( x \in \partial S \), normal \( \hat{n} \) at \( x \), and

\[
\beta = \min_{y \in \partial S, \alpha > 0} \alpha(x, \hat{n}, y),
\]

(3.4)
**Figure 3.4:** For any $y$ that generates a non-minimal $\beta$, there must be another surface point $w$ that generates a smaller $\beta$.

The ball $b(x + \beta \hat{n}, \beta)$ is maximal on $S$ and $x + \beta \hat{n}$ is on the medial axis of $S$.

**Proof.** First, we demonstrate $b \subseteq S$. Assume there is a point $z \in b$, $z \notin S$ and $z$ not on the surface of the ball. A line segment drawn from $x$ to $z$ must intersect $\partial S$ at some point $w \neq x$ (Figure 3.4). Let $f(\gamma) = ||x + \gamma \hat{n} - w||_2 - \gamma$. It is trivial to show that $f$ is continuous. Since $f(0) > 0$ and $f(\beta) < 0$, there must exist some $0 < \delta < \beta$ such that $f(\delta) = 0$. By definition, $\delta = \alpha(x, \hat{n}, w)$ for $w \in \partial S$. This violates the minimum condition on $\beta$ so $b \subseteq S$.

Next, assume there is some $b' = b(c + \beta' \hat{n}, \beta')$ such that $\beta' = \beta + \epsilon$, $\epsilon > 0$, and $b \subset b' \subseteq S$. Choose $y$ such that $\beta = \alpha(x, \hat{n}, y)$. Let $b'' = b(y, \delta)$ for

$$0 < \delta \leq \beta' - \sqrt{\beta^2 + 2\epsilon(x + \beta \hat{n} - y) \cdot \hat{n} + \epsilon^2}.$$ 

By construction, $b'' \subset b'$. But, $S$ is compact and $y \in \partial S$, so there is some point $v \in b''$ and $v \notin S$. This violates the assumption $b' \subseteq S$. Therefore, $b$ is maximal on $S$. □

**Theorem 3.5.** A point $p$ is on the medial axis if and only if $p = x + \alpha(x, \hat{n}, y)\hat{n}$ and $\alpha(x, \hat{n}, y) = \alpha(y, \hat{m}, x)$ for normal $\hat{m}$ at $y$.

**Proof.** The proof follows from Theorems 3.3 and 3.4. □

According to these theorems, a maximal ball is extended tangent to $\partial S$ in at least two places. As a corollary, any radius computed in this manner will provide an upper bound on the radius of the maximal ball. In other words, $0 < \beta_m \leq \beta$ and the point $x + \beta_m \hat{n}$ is on the medial axis.
3.2.2 Algorithm

Using the upper bound presented in the previous section, it is possible to design an algorithm that consistently computes points on the medial axis. The idea is to approximate (3.4) directly from a discretized mesh of the domain. By using this approximation, it is possible to construct a mesh-neutral algorithm that works for all types of meshes.

The algorithm proceeds in three steps. First, the surface mesh is extracted from the mesh. For an \(n\)-dimensional mesh, the surface is made of \((n-1)\)-dimensional facets. The algorithm finds all of the mesh facets incident on only one cell (Figure 2.2).

Next, for each surface facet, its centroid and inward facing normal are calculated. For an \(n\)-dimensional mesh, \(n\) points \(p_1, p_2, ..., p_n\) are selected from the surface facet and used to construct an \(n \times (n-1)\) matrix

\[
A = [(p_1 - p_n) \ (p_2 - p_n) \ ... \ (p_{n-1} - p_n)].
\]

Then, the complete orthogonal factorization \(QR = A\) is computed, where \(Q\) is an \(n \times n\) orthogonal matrix. The last column of \(Q\) then gives the normal to the facet.

Finally, all pairs of centroids are used to compute the upper bound on the radius of maximal balls. The smallest positive value is kept for each facet and points on the medial axis are computed from the data. Figure 3.5 details the algorithm.

![Figure 3.5: Inefficient algorithm to compute points on medial axis.](image)
3.2.3 Accuracy

Under certain assumptions, the accuracy of this method can be computed directly. We define \( h \) to be some measure of mesh size such as average edge length or average facet volume/area. Related to this measure is a specific measure of mesh resolution. This measure, denoted by \( \sigma \), is the minimum value that satisfies the following assumptions.

**Assumption 3.1.** For every point \( x \in \partial S \), there is a point \( y \) on the surface mesh such that \( ||y - x|| \leq \sigma \).

It is easy to see that, for uniformly discretized meshes, \( \sigma = O(h) \), for \( h \) a measure of mesh refinement. The next assumption imposes a slight limitation on the orientation of faces vis-a-vis the underlying surface.

**Assumption 3.2.** For each surface facet \( f \) with normal \( \hat{n}_f \), there is a point \( x \) on the underlying surface such that \( x + \gamma \hat{n} \) is on the facet for \( |\gamma| \leq \sigma \) and \( \hat{n} \) is an extended normal of \( \partial S \) at \( x \).

**Definition 3.7.** Any point \( x \in \partial S \) that satisfies Assumption 3.2 for some surface facet \( f \) is called a normal reference point.

Basically, the mesh accurately resolves the position of the surface and each facet has a reasonable normal. For instance, for meshes with features resolved by lower-dimensional mesh entities, the assumptions are satisfied. By using the extended normal, even meshes that do not resolve features with mesh objects meet these assumptions. Figure 3.6 shows an example of a cusp with an extended normal equal to the normal of the local discretization.

We can now show that the error in calculation of medial axis points approaches zero as the mesh is refined. We begin by showing that there is no inherent loss of information due to discretization.

**Theorem 3.6.** The error in approximating points on the medial axis is consistent with the error in approximating (3.4).
Proof. For some surface facet \( f \), let \( \mathbf{c}_f \) be the centroid of \( f \) and \( \mathbf{n}_f \) be the unit normal. Let \( \mathbf{x} \) be a normal reference point of \( f \). Let \( \mathbf{p}' = \mathbf{x} + \beta' \mathbf{n}_f \) lie on the medial axis of \( S \) and \( \mathbf{p} = \mathbf{c}_f + \beta \mathbf{n}_f \) be the approximated point. Finally, let \( \mathbf{f} = \mathbf{x} + \gamma \mathbf{n}_f \) lie on \( f \). Then, the error in the approximation can be bounded as
\[
e \leq ||\mathbf{p}' - \mathbf{p}||_2 = ||(\mathbf{f} - \mathbf{c}) + (\beta' - \beta - \gamma) \mathbf{n}_f||_2.
\]

Squaring and applying the Pythagorean theorem yields
\[
e^2 \leq ||\mathbf{f} - \mathbf{c}||^2 + (\beta' - \beta - \gamma)^2 = (\beta' - \beta)^2 + O(h).
\]
Taking the square root bounds the error as
\[
e = |\beta' - \beta| + O(h) \quad \square
\]

Finally, to prove consistency, we demonstrate that the error in \( \beta \), given by \(|\beta - \beta'| \), diminishes as the mesh is refined.

**Theorem 3.7.** The algorithm consistently computes points on the medial axis of the underlying surface.

**Proof.** The proof relies on characterizing points in \( \partial S \) incident on a maximal ball. Let \( f \) be a surface facet of the mesh with centroid \( \mathbf{c}_f \) and normal \( \mathbf{n}_f \). Let \( \mathbf{x}, \mathbf{y} \in \partial S \) such that \( \mathbf{x} \) is a normal reference point of \( f \), and \( \mathbf{y} \) positively minimizes the \( \alpha \) function for \( \mathbf{x} \) and \( \mathbf{n}_f \). Finally, let \( \mathbf{p} = \mathbf{c}_f + \beta \mathbf{n}_f \) be the computed point and \( \mathbf{p}' = \mathbf{x} + \beta' \mathbf{n}_f \) be a point on the medial axis.

By construction, we know that \( ||\mathbf{x} + \beta \mathbf{n}_f - \mathbf{y}||_2 = \beta' \). Further, note that \( ||\mathbf{c}_f + \beta \mathbf{n}_f - \mathbf{y}||_2 = \beta + \delta \) for \( |\delta| = O(\sigma) \). Squaring these equations and subtracting the former from the latter yields
\[
||\mathbf{c}_f - \mathbf{y}||^2 - ||\mathbf{x} - \mathbf{y}||^2 + 2\beta \mathbf{n}_f \cdot (\mathbf{c}_f - \mathbf{y}) - 2\beta' \mathbf{n}_f \cdot (\mathbf{x} - \mathbf{y}) = 2\beta\delta + \delta^2. \quad (3.5)
\]

Substituting \( \mathbf{x} - \mathbf{y} = \mathbf{x} - \mathbf{y} + \mathbf{c} - \mathbf{c} \) and rearranging gives
\[
\beta - \beta' = \frac{2\beta\delta + \delta^2 + ||\mathbf{x} - \mathbf{y}||^2 - ||\mathbf{c}_f - \mathbf{y}||^2 + 2\beta \mathbf{n}_f \cdot (\mathbf{x} - \mathbf{c}_f)}{2\mathbf{n}_f \cdot (\mathbf{c}_f - \mathbf{y})}. \quad (3.6)
\]

It is obvious that the numerator is \( O(h) \) for uniformly discretized meshes. According to the Law of Cosines, the denominator can be bounded by
\[
\frac{||\mathbf{c}_f - \mathbf{y}||^2 - \sigma^2}{\beta} - 2\sigma \leq 2\mathbf{n}_f \cdot (\mathbf{c}_f - \mathbf{y}) \leq \frac{||\mathbf{c}_f - \mathbf{y}||^2 - \sigma^2}{\beta} + 2\sigma. \quad (3.7)
\]

As the mesh is refined, the denominator approaches \( ||\mathbf{c}_f - \mathbf{y}||^2 / \beta \), since refinement reduces \( \sigma \). Because \( S \) is compact, \( \beta \leq \beta_{\text{MAX}} < \infty \). Finally, consider the point \( \mathbf{x} \). Obviously, \( ||\mathbf{c}_f - \mathbf{c}|| \) decreases as the mesh is refined, but \( ||\mathbf{x} - \mathbf{y}|| \) remains constant. Therefore, as \( \sigma \rightarrow 0 \), \( ||\mathbf{c}_f - \mathbf{y}||_2 \rightarrow ||\mathbf{x} - \mathbf{y}||_2 \). Since the denominator approaches some constant greater than \( ||\mathbf{x} - \mathbf{y}||^2 / \beta_{\text{MAX}} \), \( |\beta - \beta'| = O(h) \). \quad \square
3.2.4 Drawbacks

The algorithm given in Figure 3.5 has several drawbacks, not least of which is its inefficiency. Obviously the algorithm requires $O(n^2)$ computations in the number of surface facets. Further complicating matters is the lack of a straightforward parallel implementation of this algorithm. Since there is no natural way to partition the data and work, any parallel implementation would involve storing a copy of large portions of the surface mesh on each compute node, resulting in a less than scalable algorithm. Finally, this algorithm only computes points on the medial axis without computing their topology.

These drawbacks can be overcome by using information in the mesh itself to improve efficiency to $O(n)$ in the number of mesh cells, create a highly parallel algorithm for computing points on the medial axis, and compute the topology simultaneously.

3.3 An Efficient and Consistent Algorithm

The obvious inefficiency in the previous algorithm is the testing of all pairs of surface facets for every minimum $\beta$. For most meshes, the vast majority of the computation is discarded since the computed $\beta$ is probably either negative or much greater than the eventual solution.

This section describes a method of choosing pairs of facets in an intelligent way to generate an upper bound in an efficient manner. At the same time, a lower bound is computed and topological information about the medial axis is gathered.

Further, under certain assumptions, it can be shown that the algorithm is geometrically consistent with the approximation of the previous algorithm, and by extension, with the medial axis of the underlying surface.

3.3.1 Motivation

The algorithm developed uses information inside the mesh, not just the surface mesh. It makes use of a mesh object location to approximate maximal balls and compute on which “side” of the medial axis mesh objects lie. In this way, a line segment on the order of $O(h)$ in length is computed that crosses the medial axis. One end of the line segment is computed using the upper bound technique of the previous section and lies “above” the medial axis. The other end is no longer some point on the surface. Rather, positions of interior mesh objects are used to provide a point closer to the medial axis, but still “below” it.

The medial axis can be alternatively defined as the locus of points that are equidistant from more than one point on the surface of the domain. This can be seen directly from Theorems 3.3 and 3.4. Knowledge of the distance between a point in a set and the surface can give valuable insight into the location of
the medial axis. For this reason, we define a function on $S$ whose value is the
distance of a point $y \in S$ from the surface $\partial S$.

**Definition 3.8.** The interior distance function $d$ is defined on a compact set $S$
as

$$d(y) \equiv \min_{x \in \partial S} \|y - x\|_2, \quad (3.8)$$

for all $y \in S$.

A useful extension to this function returns actual points on the surface closest
to an interior point.

**Definition 3.9.** The surface point function is defined on a compact set $S$ as

$$s(y) \equiv \{x | x \in \partial S, \|y - x\|_2 = d(y)\} \quad (3.9)$$

for all $y \in S$. For convenience, if there is only one point $x$ in $s(y)$, then $s(y) = x$.

Obviously, these functions can be combined to form the medial axis transform. Given some $y \in S$, if $|s(y)| > 1$, then $y$ is on the medial axis of $S$ and the radius of the maximal ball at $y$ is $d(y)$. More importantly, these functions can be combined to determine a lower bound on the location of the medial axis.

**Theorem 3.8.** Given a point $y \in S$ not on the medial axis and a point $x = s(y)$, there is some $\gamma > 1$ such that $\gamma y + (1 - \gamma)x$ is on the medial axis.

**Proof.** Consider some line segment drawn between $x$ and $x'$, both on $\partial S$, such that the line segment is contained entirely in $S$, is normal to $\partial S$ at $x$, and contains $y$. This segment must cross the medial axis, say at point $m$. Obviously, $\|m - x\|_2$ must be greater than or equal to $d(y)$ or $b(m, d(m)) \subseteq b(y, d(y)) \subseteq S$, violating the definition of the medial axis. \qed

In effect, the point $y$ provides a lower bound on the location of the medial axis along a line segment normal to $x$. If we choose a $y'$ such that $s(y) \neq s(y')$ and $y'$ is “close” to $y$, we can provide an upper bound on the location of the medial axis along the same line. For

$$\beta = \alpha(s(y), \frac{y - s(y)}{\|y - s(y)\|_2}, s(y')), \quad (3.10)$$

$$\|y - x\| \leq d(m) \leq \beta.$$ 

Now, the challenge is to determine if the upper bound computed is tight. To this end, we can identify points on the surface incident on the same maximal ball with a simple test.

**Theorem 3.9.** Given $x, y \in \partial S$ with normals $\hat{n}_x$ and $\hat{n}_y$, they are incident on the same maximal ball if and only if $(x - y) \cdot (\hat{n}_x + \hat{n}_y) = 0$. 

74
Proof. Let $b_m$ be a maximal ball on $S$ tangent to $\partial S$ at $x$ and $y$. This implies $\alpha(x, \hat{n}_x, y) = \alpha(y, \hat{n}_y, x)$. Simple algebra shows that if $x$ and $y$ are incident on the same maximal ball, then $(x - y) \cdot (\hat{n}_x + \hat{n}_y) = 0$. The converse follows from Theorem 3.5.

The algorithm in Figure 3.5 identifies points that satisfy a similar relation. Unfortunately, the corresponding function evaluations are not equal, but nearly equal. That is,

$$\alpha(x, \hat{n}_x, y) = \alpha(y, \hat{n}_y, x) + \zeta,$$

for some $\zeta$. Rearranging terms gives a simple test for determining if the computed upper bound is tight:

$$\left| \frac{1}{\hat{n}_x \cdot (x - y)} + \frac{1}{\hat{n}_y \cdot (x - y)} \right| \leq \frac{2\zeta}{\|x - y\|^2}.$$  \hfill (3.12)

In order to recover the output of the previous algorithm by testing points on the interior of $S$, we must characterize $\zeta$. According to Theorem 3.7, both functions evaluate to within $O(h)$ of the minimum defined by Theorem 3.4.

### 3.3.2 Interior Distance and Surface Point Function Evaluation

There are well known algorithms for computing the interior distance function [99, 102]. However, since our medial axis algorithm requires fairly precise knowledge of the value as well as locations on the surface, we have adapted a marching method to evaluate both $d$ and $s$ for points in the interior of the mesh. Specifically, the points are the nodes of the dual mesh or dual nodes. The algorithm has an initialization phase used to set up a queue of dual nodes. In the main loop, the dual nodes in the queue "march" toward the medial axis.

The algorithm begins by computing a limited dual of the mesh: only dual nodes and dual edges are computed. A dual node is located at the centroid of each cell. A dual edge is inserted between dual nodes if the related cells share a facet. Next, the algorithm simultaneously gathers all surface facets and the cells that are incident on them. For each surface facet, the centroid is calculated as well as the average distance between centroids for adjacent surface facets. The distance between each dual node and the associated surface facet centroid is calculated, and the centroid is associated with the dual node. Finally, these dual nodes are placed in a queue.

While the queue is not empty, a dual node is popped from the queue. For all neighbors of this dual node, the distance between the neighbor and associated surface facet centroid is computed and compared with the current minimum distance of the neighbor. If the distance is less than the current distance or the current distance has not yet been calculated, the neighbor is associated with the surface facet centroid of the current dual node and the neighbor’s current
Let \( r \) be a queue

for each surface facet \( f \) in \( M \)

do

\( f.centroid \) ← facet centroid
\( f.normal \) ← inward unit normal of face
\( p \) ← dual node in \( D \) associated with cell incident on \( f \)
\( p.d \) ← \( |p - f.centroid|_2 \)
\( p.surf = f.centroid \)
Push \( p \) onto \( r \)

while \( r \) is not empty

do

Let \( s \) be a queue

for each \( p \in r \)

do

for each neighbor dual node \( q \) of \( p \)

\( d = |q - p.surf|_2 \)

if \( d < q.d \) or \( q.d \) unset

then

\( q.d = d \)
\( q.surf = p.surf \)
Push \( q \) onto \( s \)

end do

end do

\( r \) ← \( s \)

Figure 3.7: Interior distance and surface point approximation computed via marching method.

distance is set to the distance computed. The neighbor is then inserted into the queue. The complete algorithm is given in Figure 3.7.

To make parallel implementation easier, two queues are used. Each iteration consumes one and fills the other. By dividing the work in this manner, a bulk synchronous parallel pattern \[88\] can be employed to perform this marching method efficiently. When there are no more nodes in either queue, the algorithm has completed.

At the end of the algorithm, each dual node has an approximate distance to the surface of the mesh as well as the location used in calculating the point. For clarity, let \( \bar{s} \) be the approximation to \( s \) and let \( \bar{d} \) be the approximation to \( d \).

3.3.3 Accuracy of Interior Distance Function

The accuracy of the interior distance function can be determined for sufficiently smooth surfaces by noting that the normals along the surface change in a continuous fashion. For each surface centroid and associated surface normal, the difference \( \bar{d} - \bar{n} \cdot (p - p.surf) \) gives some indication of the accuracy of the method.

The path taken by the algorithm to compute the minimum distance also plays a role in computing \( p.surf \). Intuitively, centroids “near” the surface will
be paired with their closest centroid on the surface. Conceivably, as paths come together or diverge, the path from a particular facet may disappear due to discretization error.

Consider Figure 3.8. The path that determines data for centroid 4 goes through centroids 1, 2, and 3. Even though centroid 4 should be associated with surface centroid $b$, the algorithm computes the data for centroid $a$. The algorithm chooses $a$ over $c$ simply because the centroid is closer to $a$.

This example demonstrates that there may be error in the computation, but the error is not large and is due to discretization issues. We speculate that mesh quality is a major influence on this error. Even though the example given has a mesh node sharing six cells, the angles between edges is not evenly distributed. The gray node with gray dashed lines gives a mesh that could lead to this problem. Note that the angles are not evenly distributed.

Under two reasonable assumptions, we can argue that the error generated in this fashion diminishes as the mesh is resolved. First, we assume that the length of the dual edges in a path are bounded above and below. Mathematically, the length $l$ is bounded by $l_b \leq l \leq l_a$. Second, we assume that the walks advance a front, as in solving the eikonal equation 3.1: the dual nodes $q$ inserted in the new queue are strictly ahead of the front. Alternatively, we assume that the computed distances are strictly increasing in a path.

The computation is done in a method similar to the so-called method of lines [101]. Characteristics of the eikonal equations are cast from the surface centroids into the interior. Given two surface centroids of adjacent surface facets,
the characteristics diverge if the two two facets are concave. Two facets are concave if a straight line that intersects both is external to the mesh at some point between the two facets. If two neighboring facets are concave, the characteristics diverge. If the two facets are collinear, co planar, etc., the characteristics are parallel in the Euclidean sense. Otherwise, the characteristics converge.

At the beginning of the algorithm, the computed characteristics are a bounded distance apart with bounds derived from \( l_b \) and \( l_a \) defined above. If the characteristics diverge, the distance between them will exceed \( 2l_a \) at some point, meaning that neighboring nodes cannot be closer to a characteristic originating from a different centroid.

Parallel characteristics may exhibit the error shown above. Intuitively, a path cannot wander across many parallel characteristics, perhaps as many as \( 2l_a/l_b \), without violating the bounds on the dual edge length. In this instance, the computation will exhibit error of the order of the length of the dual edges.

For converging characteristics, consider a sphere drawn on the wave front being computed, centered at a characteristic. As the wave front progresses, the number of paths in the computation, and therefore a measure of characteristics, that lie in the sphere, increases. Let the sphere have radius \( l_a \). The density of paths through this sphere increases as the computation progresses. At a certain density, the accuracy of the computed distance increases because the support for the computation is spread across a larger area of the surface. Even though paths disappear, others enter the sphere to keep the computation accurate.

For the overall algorithm to be consistent, as the mesh is refined the related surface centroid must be on the “same side” of the medial axis as the dual node. By the same side we mean a line segment that connects the surface centroid with the point does not cross the medial axis. Intuitively, the dual node can be no more than \( l_b \) past the medial axis before it would be associated with a characteristic originating from a surface facet distant from the one it is associated with.

### 3.3.4 Dual Edge Identification

Once the interior distance and surface point fields have been approximated, the algorithm can consider each dual edge independently and determine whether the dual edge crosses the medial axis. The algorithm proceeds in five steps and is given in Figure 3.9. In the first step, each dual node of the dual edge is projected down to the surface facet to which it is closest (lines 9 and 10).

To speed computation, each surface facet is idealized to an \( n \)-sphere of radius \( h/2 \), half the average distance between the surface centroid and its neighbors. For two-dimensional surface facets, the \( n \)-sphere is a circle. For one-dimensional facets, it is a line segment. If a projected node lies outside the sphere, it is projected onto the surface of the sphere (lines 14–19). Once this is done, unit extended normals are approximated by subtracting each dual node from its
**ISDUALEDGE_MEDIAL(e)**

1. Let $p_1$ and $p_2$ be nodes of edge $e$  
2. $c_1 \leftarrow \bar{s}(p_1)$  
3. $c_2 \leftarrow \bar{s}(p_2)$  
4. Let $\hat{n}_1$ and $\hat{n}_2$ be inward facing normals of related facets  
5. Let $h_1$ and $h_2$ be average distances between surface centroids around $c_1$ and $c_2$  
6. Let $\bar{h}_1$ and $\bar{h}_2$ be average edge lengths of edges from dual nodes  
7. Let $\bar{c}_1 \leftarrow (p_1 - c_1) \cdot \hat{n}_1 \hat{n}_1$  
8. $\bar{c}_2 \leftarrow (p_2 - c_2) \cdot \hat{n}_2 \hat{n}_2$  
9. $t_1 = \bar{c}_1 - c_1$  
10. $t_2 = \bar{c}_2 - c_2$  
11. If $||t_1||_2 > h_1/2$  
12. Then  
13. $\bar{c}_1 \leftarrow c_1 + h_1 t_1 / ||t_1||_2/2$  
14. If $||t_2||_2 > h_2/2$  
15. Then  
16. $\bar{c}_2 \leftarrow c_2 + h_2 t_2 / ||t_2||_2/2$  
17. $\hat{n}_1 \leftarrow (p_1 - \bar{c}_1) / ||p_1 - \bar{c}_1||_2$  
18. $\hat{n}_2 \leftarrow (p_2 - \bar{c}_2) / ||p_2 - \bar{c}_2||_2$  
19. $\alpha_1 \leftarrow \alpha(\bar{c}_1, \hat{n}_1, \bar{c}_2)$  
20. $\alpha_2 \leftarrow \alpha(\bar{c}_2, \hat{n}_2, \bar{c}_1)$  
21. $\bar{p}_1 \leftarrow \bar{c}_1 + \alpha_1 \hat{n}_1$  
22. $\bar{p}_2 \leftarrow \bar{c}_2 + \alpha_2 \hat{n}_2$  
23. $e.m \leftarrow (\bar{p}_1 + \bar{p}_2)/2$  
24. If $||p_1 - e.m||_2 < h_1$ and $||p_2 - e.m||_2 < h_2$  
25. Then  
26. Return true  
27. Return false

Figure 3.9: Dual edge detection algorithm.
eventual projection and normalizing (lines 21 and 22).

The fourth step involves approximating Equation 3.2 for each dual node. The projected points are used for the \( x \) and \( y \) parameters, with the approximate extended normals used as expected (lines 24 and 25). The last step computes the centers of the balls given \( \alpha_1 \) and \( \alpha_2 \). If the centers of the balls are within half of a dual edge of their related dual nodes, the dual edge crosses the medial axis of the surface mesh.

Once the set of dual edges that cross the medial axis of the surface mesh has been computed, the topology can be pieced together simply by collecting the facets associated with the dual edges. The resulting medial axis contains much of the medial axis of the surface \( \partial S \) as well as hairs. These hairs must be filtered out.

### 3.3.5 Identifying Ill-Conditioned Hairs

As mentioned in Section 3.1.2, any medial axis algorithm requires a stabilization factor. Many of the issues surrounding small features are handled by Assumption 3.1. Unfortunately, spurious hairs and sheets in the medial axis arising from the discretization of smooth curves are detected by the algorithm (Figure 3.10). Because these hairs are extremely sensitive to the discretization, or ill-conditioned, they must be filtered out of the final result.

Consider the pentagon given in Figure 3.11. The hair in green is computed from the small portion of surface, given in red. The radius of the maximal balls tangent to the red area change dramatically. Computation of the green hair is, by definition, ill-conditioned.

These hairs are identified as being part of the medial axis because they are part of the medial axis of the surface mesh. The points used for computing the medial axis upper bound are chosen in the plane of the surface facet. The projected dual nodes can be near the boundary of the surface facet and give rise to the situation depicted in Figure 3.12. In this case, the nodes projected on the
Figure 3.11: Hair, in green, governed by very small portion of surface, in red.

Figure 3.12: For dual edges close to set boundary and at low angle to medial hair, algorithm detects medial axis of discretization, not underlying surface.
face have a different inward normal than the nearby $\partial S$. If the normal of $\partial S$ is used in the computation instead of the normal of the facet, the computation of $\alpha_1$ and $\alpha_2$ changes dramatically and, as will be demonstrated, the conditioning of the computation improves substantially.

The classical definition of conditioning, the ratio of the relative change in the output to the relative change in the input, cannot be directly applied to this problem since translating the mesh through space can arbitrarily alter the conditioning of the computation. Rather, we can approximate the absolute forward error $e$ in the computation as

$$e = f(x + \delta x) - f(x) \approx f'(x) \Delta x.$$  \hspace{1cm} (3.13)

For large absolute error, especially if the error is greater than the resolution of the mesh, the computation can be classified as noise. So, for some dual edge length $h$, if

$$h \leq f'(x) \Delta x,$$  \hspace{1cm} (3.14)

the error in the computation is (roughly) greater than the resolution of the mesh.

We can compute the derivative term as a directional derivative in the first argument of $\alpha$ in a plane orthogonal to the approximate characteristic. Since the derivative varies with the direction, we choose the minimum absolute value. The derivative term is given by

$$f' = \min_{m \cdot \bar{n} = 0} \lim_{\eta \to 0} \frac{\alpha(\bar{c}_1 + \eta m, \bar{h}_1, \bar{c}_2) - \alpha(\bar{c}_1, \bar{h}_1, \bar{c}_2)}{\eta} = \tan \theta,$$  \hspace{1cm} (3.15)

for $\theta$ given in Equation 3.2.

The stabilization factor chosen by the user is $\Delta x$. Obviously, $\Delta x$ must be no greater than $h$, the current mesh surface resolution. A change in position by that amount would necessarily encroach on another facet of the surface, changing the normal used in the equation, and invalidating the result of the error calculation. Intuitively, if the center of the surface facet is used in the error computation, $\Delta x$ should be roughly $h/2$, a step to the edge of the idealized surface facet. Substituting these values into Equation 3.14 gives the maximum acceptable tangent of $\theta$ for a particular computation. For this reason, we recommend $1 \leq \tan \theta \leq 2$ for computational meshes. The algorithm for computing the stability of a medial facet/dual edge is given in Figure 3.13.

### 3.3.6 Medial Axis Smoothing

`IsDualEdgeMedial` and `IsDualEdgeStable` identify dual edges, and therefore mesh facets, that make up the topology of the medial axis. The nodes that make up these facets are near the medial axis, but need some smoothing to represent the geometry of the medial axis accurately. The medial facets inherit from
isDualEdgeStable(e, threshold)
1  Let p1 and p2 be nodes of edge e
2  c1 ← s(p1)
3  c2 ← s(p2)
4  Let \( \hat{n}_1 \) and \( \hat{n}_2 \) be inward facing normals of related facets
5  Let \( h_1 \) and \( h_2 \) be average distances between surface centroids around \( c_1 \) and \( c_2 \)
6  Let \( \bar{h}_1 \) and \( \bar{h}_2 \) be average edge lengths of edges from dual nodes
8
9  \( \bar{c}_1 \leftarrow p_1 - (p_1 - c_1) \cdot \hat{n}_1 \hat{n}_1 \)
10 \( \bar{c}_2 \leftarrow p_2 - (p_2 - c_2) \cdot \hat{n}_2 \hat{n}_2 \)
11 \( t_1 = \bar{c}_1 - c_1 \)
12 \( t_2 = \bar{c}_2 - c_2 \)
13
14  if \( ||t_1||_2 > h_1/2 \)
15      then
16      \( \bar{c}_1 \leftarrow c_1 + h_1 t_1/||t_1||_2/2 \)
17  if \( ||t_2||_2 > h_2/2 \)
18      then
19      \( \bar{c}_2 \leftarrow c_2 + h_2 t_2/||t_2||_2/2 \)
20
21  \( \tilde{n}_1 \leftarrow (p_1 - \bar{c}_1)/||p_1 - \bar{c}_1||_2 \)
22  \( \tilde{n}_2 \leftarrow (p_2 - \bar{c}_2)/||p_2 - \bar{c}_2||_2 \)
23
24  \( \theta_1 \leftarrow \) angle between \( \bar{c}_2 - \bar{c}_1 \) and \( \tilde{n}_1 \)
25  \( \theta_2 \leftarrow \) angle between \( \bar{c}_1 - \bar{c}_2 \) and \( \tilde{n}_2 \)
26
27  if \( \tan \theta_1 \geq \) threshold or \( \tan \theta_2 \geq \) threshold
28      then
29      return false
30  return true
31
Figure 3.13: Identification of stable dual edges.
SmoothNode(node)
1. \( F \leftarrow \text{Set of all faces incident on node} \)
2. for each \( f \in F \) do
3. \( t \leftarrow t + f.m \)
4. node = \( t/|F| \)

Figure 3.14: Medial node averaging method for orienting facets taken from mesh.

\[
\text{SmoothNode(node)} \\
1. F \leftarrow \text{Set of all faces incident on node} \\
2. \text{for each } f \in F \\
3. \text{do} \\
4. \quad t \leftarrow t + f.m \\
5. \text{node} = t/|F| \\
\]

Figure 3.15: Complete algorithm to compute medial axis.

the dual edge the medial node \( m \) computed on line 28 of IsDualEdgeMedial.

To smooth the mesh, the nodes in the medial axis are positioned at the average of \( m \) for the incident medial faces. The smoothing algorithm is given in Figure 3.14. It should be noted that the boundary of the smoothed medial axis will not extend to the edge of the computable medial axis. This should not be of grave concern, however, since computing the extremities of a medial axis is prone to computational error.

### 3.3.7 A Consistent Algorithm

The various algorithms described thus far are combined in Figure 3.15, which gives the complete algorithm for computing the medial axis of a mesh. We now
demonstrate that as the mesh is refined, the geometric error of the medial axis approximation tends to zero.

In order to show consistency, let \( x \in \partial S \) be the closest point to \( \bar{c}_1 \) on the discretized surface, let \( y \in \partial S \) be the closest point to \( \bar{c}_2 \) on the discretized surface, and let \( \hat{n} \) be the normal at \( x \) that maximizes \( n = \hat{n} \cdot \bar{n} \). Let \( \beta_1 = \alpha(x, \hat{n}, y) \) and let \( \beta_2 = \alpha(\bar{c}_1, \hat{n}, \bar{c}_2) \).

To show consistency, we must make another assumption.

**Assumption 3.3.** As the mesh is refined, \( n \rightarrow 1 \). Equivalently, the angle between \( \hat{n} \) and \( \bar{n} \) is \( O(h) \).

Intuitively, we assume the paths chosen by the interior distance surface point function converge to the characteristics of the eikonal equation. We begin the proof by demonstrating that using idealized surface facets is consistent with using the underlying discretization.

**Theorem 3.10.** \( |\beta_1 - \beta_2| = O(h) \), for \( h \) a measure of the length of dual edges.

**Proof.** Performing the subtraction and noting \( ||y - x|| - ||\bar{c}_2 - \bar{c}_1|| = O(h) \), we observe that

\[
|\beta_1 - \beta_2| = ||y - x|| \left( \cos \theta_1 - \cos \theta_2 \right) + O(h),
\]

where \( \theta_1 \) is the angle between \( \hat{n} \) and \( (y - x) \) and \( \theta_2 \) is the angle between \( \bar{n} \) and \( (\bar{c}_2 - \bar{c}_1) \). By construction, the angle between \( (y - x) \) and \( (\bar{c}_2 - \bar{c}_1) \) is given by

\[
\sin^{-1} \frac{O(h)}{||y - x||} = O(h).
\]

By assumption, the angle between \( \hat{n} \) and \( \bar{n} \) is also \( O(h) \). Therefore, \( |\theta_1 - \theta_2| = O(h) \). Since \( \theta_1 \) and \( \theta_2 \) are chosen to be strictly greater than zero by the algorithm, the denominator is bounded away from infinity. The proof follows from substituting the difference in the \( \theta \) terms into the Taylor series expansion of the numerator:

\[
\cos \theta_1 - \cos \theta_2 = \left( 1 - \frac{\theta_1^2}{2} + \ldots \right) - \left( 1 - \frac{\theta_2^2}{2} + \ldots \right) = O(h^2).
\]

\( \square \)

**Theorem 3.11.** \( ||(x + \beta_1 \hat{n}) - (\bar{c}_1 + \beta_2 \bar{n})|| = O(h) \), for \( h \) a measure of the length of dual edges.

**Proof.** Squaring the expression and regrouping terms gives

\[
||(x + \beta_1 \hat{n}) - (\bar{c}_1 + \beta_2 \bar{n})||^2 = ||x - \bar{c}_1||^2 + 2||x - \bar{c}_1|| ||\beta_2 - \beta_1|| + \beta_1^2 + \beta_2^2 - 2\beta_1 \beta_2 n.
\]

By construction, \( ||x - \bar{c}_1|| = O(h) \). The proof follows from this and Theorem 3.10. \( \square \)
**Theorem 3.12.** The medial axis computation is consistent with the first algorithm: as the mesh is refined, the points computed lie within $O(h)$ of the medial axis computed by the first algorithm.

**Proof.** As a result of Theorem 3.11, this demonstration is equivalent to demonstrating the distance between $x + \beta_1 \hat{n}$ and the medial axis diminishes with mesh refinement. Let $\hat{m}$ be the inward facing normal of $y$ that satisfies Theorem 3.11 with respect to $\bar{c}_2$. We examine two cases:

**Case I,** $\hat{n} \cdot \hat{m} < 0$: According to Theorem 3.4, the line segment between $y$ and $y + \gamma \hat{m}$ ($\gamma$ the solution to the $\alpha$ function) crosses the medial axis. The closest point to $x$ on this segment is a point on the line segment between the medial axis and $y + \gamma \hat{m}$. Whereas, by the same Theorem, $x + \beta_1 \hat{n}$ must lie beyond the medial axis with respect to $x$. Since the medial axis is continuous, the line segment between $y + \gamma \hat{m}$ and $x + \beta_1 \hat{n}$ must cross the medial axis. If these two points are within $O(h)$ of each other, as specified by the algorithm, then the medial axis must be within $O(h)$ of the midpoint of that line segment.

**Case II,** $\hat{n} \cdot \hat{m} \geq 0$: For convenience, let $p_x = x + \beta_1 \hat{n}$ and $p_y$ be the point computed for $y$ and $\hat{m}$. Furthermore, let these points be chosen to satisfy Theorem 3.11. We note that because $|| p_x - p_y ||_2 = O(h)$, the distance between the projected points onto a segment between $x$ and $y$ is $O(h)$:

\[
|| (p_x - x) \cdot (y - x)(y - x) - (p_y - y) \cdot (x - y)(x - y) ||_2 = O(h). \tag{3.20}
\]

Grouping together like terms and moving to the right-hand side yields

\[
|| (\beta \hat{n} - \gamma \hat{m}) \cdot (y - x) || = O(h). \tag{3.21}
\]

There are two ways to satisfy this expression. First, the dot product could go to zero. However, since $S$ is compact, the cosine term in the dot product will take a minimum value, regardless of refinement. The other way is to shrink the lengths of the vectors as the mesh is refined.

Since $y - x$ is fixed, $|| \beta \hat{n} - \gamma \hat{m} ||_2 = O(h)$. And because $\hat{n} \cdot \hat{m} \geq 0$, the triangle inequality implies $| \beta - \gamma | = O(h)$. By Theorem 3.9, as the mesh is refined, $p_x$ and $p_y$ approach a point on the medial axis computed by the consistent serial algorithm.

Thus we have shown that this algorithm is consistent with the previous algorithm. By extension, this algorithm consistently computes the medial axis of the underlying surface $\partial S$. This algorithm also lends itself to parallelization, as discussed in the next section.

86
3.4 A Parallel, Efficient, and Consistent Algorithm

This section details implementing the medial axis algorithm given in Figure 3.15 in parallel. A close look shows that, as long as centroid and normal data accompany dual nodes and edges, the majority of the algorithm is trivial to parallelize. The IsDualEdgeMedial, IsDualEdgeStable, and SmoothNode algorithms can be used without alteration on a parallel supercomputer.

The ComputeLimitedDualMesh algorithm is used in several other parallel applications such as mesh partitioning and node ordering. A parallel version of this algorithm is also trivial to construct. Each cell creates a node and each facet creates an edge, which can be accomplished in a trivially parallel fashion.

This leaves the InteriorDistanceSurfacePoint algorithm. As mentioned earlier, this algorithm can use a double-buffer strategy to facilitate a bulk synchronous style parallel algorithm [88]. The parallel algorithm is given in Figure 3.16. The first portion of the algorithm is unaltered. The second portion involves moving nodes from one compute node to another, with the additional communication steps given in red.

Line 31 of this algorithm scans for neighbors of local nodes and adds the neighbors to the current queue. In the actual implementation, the specific neighbors (the q not found on line 19) are part of the broadcast, to limit redundant and irrelevant computation. Also, when the data are broadcast, the relevant surface centroids and normals accompany the dual node. With this modest change to this one serial algorithm, the medial axis can be approximated in parallel. The other algorithms can be computed as described or by ghosting dual nodes and the related mesh facets.

The complete parallel medial axis approximation algorithm is given in Figure 3.17. We analyze four parts of this algorithm in the next section: the computation of the dual mesh, the computation of the interior distance and surface point functions, the identification of dual edges, and the smoothing of the medial axis.

3.5 Results

The medial axis algorithm is implemented using the generic data structure presented in the previous chapter. This single implementation is capable of computing the medial axis of any kind of mesh, regardless of dimension. The results we present in this Section use the same meshes given in Figure 2.11.

Like all parallel algorithms, this one is sensitive to the partitioning of the mesh, computed on line 1. The results in this section use a simple Cartesian nested dissection partitioning of the mesh. The partitioner computes the longest Cartesian dimension, finds the middle node along that dimension, cuts the mesh
Let $r$ be a queue

for each surface facet $f$ in $M$

do

$f$.centroid ← facet centroid
$f$.normal ← inward unit normal of face
$p ←$ dual node in $D$ associated with cell incident on $f$
$p.d ← ||p − f$.centroid||_2
$p$.surf = $f$.centroid
Push $p$ onto $r$

do

while $r$ is not empty

Let $s$ be a queue
Let $t$ be a list

for each $p ∈ r$

do

for each neighbor dual node $q$ of $p$

do

if $q$ not on compute node
then
Add $p$ to $t$
Continue

$d = ||q − p$.surf||_2
if $d < q.d$ or $q.d$ unset
then
$q.d = d$
$q$.surf = $p$.surf
Push $q$ onto $s$

$r ← s$

Broadcast $t$

Scan $t$ for neighbors of local nodes and add them to $r$

Figure 3.16: Parallel interior distance and surface point approximation computed via marching method.
Figure 3.17: Complete algorithm to compute medial axis in parallel.

at that node and continues until the desired number of partitions is computed. This partitioning algorithm guarantees perfect load balance but with greater-than-optimal communication volume.

The following experiments were performed on an Apple 2GHz G5 computing cluster. Each node has two processing units, 4GB of RAM, and a high bandwidth, low latency Myrinet network interface.

### 3.5.1 Example Medial Axis Computations

Figure 3.18 shows the approximate medial axis of the green 2.7 million cell mesh depicted in Figure 2.11 computed on 64, 128, 196, and 256 compute nodes. The times to compute the medial axis (not including the partitioning mentioned in line 1) were 18 seconds, 10 seconds, 15 seconds, and 17 seconds respectively. The colors represent the partitions in which the medial axis resides. A detail of the top portion of this mesh can be seen in Figure 3.19. Though computed in parallel, indication of partitioning is suppressed to expose the accuracy of the computation.

### 3.5.2 Effects of Varying the Conditioning Filter

As stated in Section 3.1.2, medial axis algorithms generally require user supplied parameters to help make simplifying assumptions. A successful algorithm will have simple-to-use parameters. The algorithm presented above relies on
Figure 3.18: Medial axis computed on 64, 128, 192, and 256 compute nodes.
Figure 3.19: Detail of medial axis approximation (right) of 450,000 cell mesh (left). Bottom shows cutaway.
a threshold such that when the angle between the normal and a segment connecting to points on the surface exceeds that threshold, the computation is disregarded as being too error prone.

For the mesh shown in Figure 3.19, we computed the medial axis for several values of the threshold parameter in Figures 3.15 and 3.17. In Figure 3.20, as threshold increases from 0.5 to 4, more of the medial axis is resolved. Once the threshold exceeds 2, however, spurious hairs begin to appear in the computation.

Figures 3.21, 3.22, and 3.23 show a close up view of the end of one of the fins. As expected, for \( \tan \theta < 0.5 \) and \( \tan \theta < 1 \), the medial axis is not well resolved near corners. For \( \tan \theta < 2 \) (Figure 3.22 top), the medial axis is almost completely resolved to the sharp corners. The smoothing routine prevents extremal facets, those with an edge on the boundary of the medial axis, from being accurately resolved. In fact, many detected facets are those that have an edge on the boundary of the mesh. There are still some gaps along the edges implying full resolution requires a “looser” threshold. For \( \tan \theta < 4 \) (Figure 3.22 bottom), the medial axis is completely resolved, along with some spurious hairs.

### 3.5.3 Effect of Poor Meshes

The results presented thus far were generated on “high quality” meshes: the cells had good aspect ratios and the volumes of the cells do not wildly vary. For poor meshes, the algorithm still does reasonably well. The mesh depicted in Figure 3.24 has many poor elements and the volumes of the cells vary wildly (by almost five orders of magnitude!). Even with this poor mesh, the medial axis is reasonably well resolved for threshold = 4.

### 3.5.4 Scalability Results in Parallel

The medial axis algorithm can be broken into four parts: partitioning (line 1), dual mesh creation (line 3), interior distance and surface point computation (line 4), and the identification and smoothing of medial facets (line 6-19). We treat partitioning in the next chapter. In this section, we evaluate the scalability of the other three parts. The dual mesh creation portion is absolutely parallel—no communication is necessary to perform this computation. The interior distance and surface point computation requires propagating a wave through the mesh. This propagation is the least scalable portion of the algorithm. The last portion, facet filtering, requires a constant number of global communications in order to ghost the relevant facets for smoothing in parallel.

Computation of the dual mesh is performed completely in parallel: each cell and facet are translated to a dual node and dual edge without communication. To obtain accurate timing information, an MPI.Barrier is placed after the computation. The wall clock time results with respect to the number of compute nodes and number of cells per compute node are given in Figure 3.25.
Figure 3.20: Effects on computed medial axis of varying derivative threshold $\tan \theta$ (left) with cutaway (right).
$\tan \theta < 0.5$

$\tan \theta < 1$

Figure 3.21: Detail of medial axis computation around bottom of fin.
Figure 3.22: Detail of medial axis computation around bottom of fin.
Figure 3.23: Detail of medial axis computation around bottom of fin.

Perhaps the most interesting feature of these plots is the relative peak in wall clock time in computing the dual mesh for the 2.7 million cell mesh using 12 compute nodes (roughly 225,000 cells per compute node). The bottleneck is entirely on one compute node. The other eleven finish the task in time expected by inspection of the plot. This one compute node apparently has a poor ordering of the cells, facets, and nodes, taking roughly twice the time to compute its portion the dual mesh.

The bottom plot in Figure 3.25 hints at the perfectly parallel performance of the algorithm. For similar numbers of cells per compute node, the increase in size of the problem did not seriously affect performance. A twenty times increase in problem size results in less than a three times increase in wall clock time for similar number of cells per compute node.

Using the wall clock timings, we can compute the $E_2$ efficiency for each experiment we performed. Plots of these efficiencies are shown in Figure 3.26. Again we see the poor cache use when twelve compute nodes are employed.

The mesh data structure is memory intensive, using roughly 110 bytes per cell for tetrahedral meshes. Since much of the data associated with each cell is not used in constructing the dual mesh, the computation is extremely sensitive to memory access patterns. We chose not to impose a better ordering on the mesh since, for this computation, the dual mesh was constructed just once. If the dual mesh were constructed several times, then the additional cost of ordering the mesh better could be cost effective.
Figure 3.24: Great variance in cell volume can lead to poorly resolved medial axis ($\tan \theta < 4$).
Figure 3.25: Wall clock time for computing dual mesh.
This example also illustrates how inefficient use of the memory hierarchy can mask parallel scalability. Since there is no communication in the algorithm, it scales perfectly. In terms of Amdahl’s law, there is no serial portion of the computation. There is no theoretical lower bound on the speed of computation. The difference in memory access times masks this facet.

The next part of the process is the approximation of the interior distance surface point field. This computation has a serial thread that runs through it: the propagation of a wave from the surface to the medial axis. The parallelism involves computing the propagation in parallel. Unfortunately, as the algorithm proceeds, the size of the wave, measured in number of dual nodes, decreases. This decreases the available work for each compute node.

As a result, we expect to see an upward trend in the wall clock time per compute node. Further, we expect the number of cells per compute node to increase to maintain scalability. This will make curves higher and more to the right for larger problems when plotted versus cells per compute node. Figure 3.27 depicts both trends.

The plots in Figure 3.28 show efficiencies computed from the wall clock times. The lower plot demonstrates that, roughly, an increase in problem size requires a similar increase in cells per compute node to maintain efficiency. For instance, a 40% efficient computation on a problem with 140,000 cells requires, roughly 1100 cells per compute node. The same efficiency on a 450,000 cell mesh requires around 4,000 cells per compute node. And the same efficiency for a 2,700,000 cell mesh requires around 20,000 cells per compute node.

In other words, increasing the problem twenty-fold requires the number of cells per compute node to increase twenty-fold as well. This is approaching unfavorable scalability. Unfortunately, reducing the serial portion of the computation is difficult because it involves a time-like serial thread.

The final step is construction of the medial axis. Like construction of the dual mesh, this step is extremely parallel. A constant number of global communications is needed to enable the computation. This step requires a traversal through the dual mesh and a traversal through the medial axis mesh. The dual mesh requires much less memory than the underlying mesh. The data used in the calculation is “denser” in some sense than when the dual mesh was constructed, allowing better use of cache. The medial axis mesh is generally much, much smaller than the original mesh.

Figure 3.29 shows the time for various numbers of compute nodes to construct the medial axis on the three meshes. The plots demonstrate the scalability of this algorithm: for similar numbers of cells on a compute node, the algorithm took a similar amount of time regardless of the size of problem.

The efficiency plots are similar to those for the construction of the dual mesh in Figure 3.26. The analysis is nearly the same except that the software can maintain a 50% efficiency. As with building the dual mesh, the majority of the inefficiency can be eliminated using a cache friendly ordering of the computation.
Figure 3.26: Wall clock time for computing dual mesh.
Figure 3.27: Wall clock time for computing interior distance and surface point fields.
Figure 3.28: Wall clock time for computing interior distance and surface point fields.
Figure 3.29: Wall clock time for identifying and smoothing medial facets.
The results are given in Figure 3.30.

Finally, we combine all the data presented so far into four plots for the entire algorithm. According to Gustafson [54] and demonstrated in the previous chapter, if we vary the fractional amount of serial work with the size of the problem, we should see reasonable scalability. The plots in Figure 3.31 show some characteristics of a scalable algorithm. In particular, the time to complete the algorithm for similar numbers of cells per node does not grow too fast as the problem size grows.

In Figure 3.32, we see the relevant efficiency plots. The results are similar to those for approximating the interior distance surface point function. This empirically demonstrates the scalability of the algorithm, at least for these particular meshes.

## 3.6 Future Work

We see three avenues for improving our algorithm. The first involves improving the parallel performance and the other two seek to make this algorithm more of a general purpose medial axis approximation tool. We also have two extended results we would like to demonstrate.

### 3.6.1 Better Communication Pattern

Currently, the communication throughout the medial axis computation is global in nature. Rather than keep track of neighbor lists, the algorithm presumes that there is no intelligent partitioning of the mesh. It relies on `MPI_Allgather` to transfer information in the interior distance and surface point computation, and on the `MPI_Alltoall` in the ghosting of medial facets for the smoothing phase.

Rather than use such expensive collective communication, the algorithm could use overlapping point-to-point communication. Or the implementation could have a template option to choose between the point-to-point communication and global communication. Overlapping communication may require a triple buffer approach, with the third buffer being the data communicated while the other two buffers are operated on. The programmer could choose between the models depending on how well the mesh is partitioned before method invocation.

### 3.6.2 Generalizing to a Point Cloud

Finally, this algorithm should be generalized to a point cloud to be relevant to many applications of the medial axis. Above, we detail a process to recover the medial axis from a surface triangularization using existing software. However, we feel we can generalize this algorithm, at least in serial, to a point cloud.

One way to develop an algorithm to work from a point cloud involves using quad-trees, oct-trees, or their higher-dimensional counterparts. A bounding box
Figure 3.30: Wall clock time for identifying and smoothing medial facets.
Figure 3.31: Wall clock time for entire medial axis computation.
Figure 3.32: Wall clock time for entire medial axis computation.
around the point cloud can be computed, which would then be decomposed into boxes to some resolution. Each point would be in a separate box. This is the mesh that would be used in the algorithm.

Each point would then be considered a centroid, and the normal used in the computation would be computed based on the current dual node and the point. This will generate several distinct axes to varying degrees: the interior medial axis, the exterior medial axis, and surface axes. The surface axes can be recognized by resolution and locality of the axis to the point cloud. This would give rise to a surface. This surface would then be used with the box decomposition to perform another medial axis computation.

This works on the principle that the medial axis computation does not require conformal meshes to work well. Rather, just some semblance of a dual mesh must be computed in order to propagate the wave to solve the eikonal equation.

3.6.3 More General Correctness Proof

We demonstrated that the points computed by the algorithm approach the medial axis, given Assumption 3.3. We would like to demonstrate this assumption more rigorously. For greater acceptance, it should also be demonstrated that the points computed by the algorithm sample the medial axis well.

For instance, let \( b \) be a ball on the medial axis of the underlying domain of radius \( r \), and let \( B_r \) be the set of all such balls. Let an approximate medial mesh be called \( r \)-recovered if \( r \) is the smallest radius such that for all \( b \in B_r \), each \( b \) contains a point in the approximate medial mesh. We would like to demonstrate that as the mesh is refined (\( h \to 0 \)), \( r \to 0 \). In other words, when a mesh is refined in the limit, the computation densely samples the medial axis.

We also wish to demonstrate the correctness of the topology constructed by the algorithm. We feel this proof will lead to better understanding of the filter for edge stability.

3.6.4 Extensive Mesh Testing

Finally, the serial thread in the computation of the interior distance surface point computation is sensitive to the shape of the domain. The meshes we chose are relatively simple in shape. We would like to perform the same experiments given above on meshes that are more complex and larger.

3.7 Summary

Computation of the medial axis is a very tricky problem. Fortunately, many degeneracies vanish when using meshes due to the nature of “good” computation grids. The medial axis has many uses and we employ it next in the development of a mesh partitioner.
Distributed memory supercomputers have two advantages over other types of computers. First, and most obvious, they have the capacity to perform simultaneous computation on distinct data. Efficient use of this advantage is measured in speed-up, isoefficiency functions, and others described in Section 2.7.1. The second advantage is increased memory capacity. Unfortunately, the memory is addressed by compute node and location in RAM. Programmers must explicitly partition and distribute data across the compute nodes in an intelligent fashion. For most mesh-based simulations, programmers generally rely on a mesh partitioning algorithm to perform this task.

The ultimate goal of a mesh partitioner is to reduce the amount of wall clock time a particular engineering simulation requires to complete. Generally, this is accomplished in three steps: identify individual units of computation, analyze the dependencies of these computations, and order the computation so that most of the computational units can be done independently of the others. It is common to use a graph to model the computation, where each node represents a unit of computation and each edge represents a dependency between computational units [60]. As a rule, if a matrix constructed from the mesh is sparse, the graph representing the corresponding matrix computation will be sparse.

Mesh partitioning is a special case of graph partitioning. While the terms are often used interchangeably, graph partitioners are a superset of mesh partitioners. For instance, large sparse matrices not derived from meshes are often partitioned using graph partitioners. In this chapter, we focus solely on mesh partitioners. Many of the assumptions made and heuristics employed will not generalize to the graph partitioning problem.

The reduction of a complex computation to a sparse graph allows a partitioner designer to choose from a myriad of graph-based algorithms with proven theoretical and empirical results. These algorithms attempt to create equal
SimulationAlgorithm()
1 Pre-process mesh
2 Partition mesh
3 while not finished
4 do
5 Generate and solve (non-)linear system
6 Alter mesh according to solution
7 if mesh not good enough
8 then
9 Refine, smooth, or regenerate as necessary
10 Re-partition as necessary
11 Post-process mesh

Figure 4.1: High level algorithm for large-scale engineering simulation given in Figure 2.4.

sized partitions while limiting the number of edges cut by the graph separators. A separator is an imaginary line drawn through a graph that partitions the nodes into two distinct groups. These algorithms are referred to as topological partitioners since only the graph is considered by the algorithm. While these algorithms tend to minimize the overall volume of computation, they neglect other aspects of parallel computation that affect performance, such as the total number of messages and the limited number of network interconnects on each compute node. Even without these considerations, computation of optimal graph partitions is NP-hard [46].

For this reason, the reduction of the mesh partitioning problem to the graph partitioning problem has been called simplistic [59]. Due to the combinatorial complexity of the over-simplified problem, complicating this model seems unlikely to reduce the overall compute time. Depending on the type of simulation, this may or may not matter. Consider a simulation that requires just one mesh partitioning. The overhead in producing the partitioning will be amortized over the subsequent simulation, so it may be insignificant if the simulation runs long enough. A partitioner that takes two hours longer than another but reduces simulation time by one percent requires a simulation to run for longer than two hundred hours to recover the cost.

As pointed out in Figure 4.1, modern simulations often require many partitionings of a mesh. As a simulation evolves, the mesh changes due to the solution computed on line 5. These changes can occur from mesh motion [22, 35], adaptive mesh refinement [85, 97], or other evolutionary processes. As the mesh deforms, the cells in the mesh deteriorate in a numerical sense, becoming skewed or even inverted. Mesh smoothing techniques attempt to fix these cells without changing the topology of the mesh. Sometimes the mesh must be repaired, which involves adding or removing nodes or cells. This changes the distribution of the computation and the interaction of the dependencies, requiring a repara-
tioning. The amount of repair that must be performed varies with the degree of mesh refinement [53]. Therefore, the marginal cost of adding complexity to the graph model of computation cannot be ignored.

A common way to augment the graph model is to add geometric information. In other words, rather than partition a mesh based solely on the dependencies between computational units, a partitioner decides where to separate a mesh based on the geometric location of the nodes. Direct computational dependencies between two entities implies the two entities are “close” to each other relative to the overall size of the domain. Geometric partitioners rely on this fact and compute a mesh separator as a geometric object.

Both topological and geometric partitioners have advantages relative to the other. In this chapter, we present a hybrid approach to mesh partitioning that attempts to combine the advantages of both types into a single partitioner.

4.1 Designing Partitioners

Since computing optimal partitions is hard, a variety of heuristics are employed to reduce the overall computation time. The heuristics compute and optimize quality metrics. These metrics attempt to quantify a particular facet of a computation and reduce the compute time of that facet. For instance, the number of edges cut by a partitioner is a rough estimate of the amount of communication required to perform a computation.

In this section, we explore the statistical correlation between several popular metrics and the speed of computing a matrix-vector multiply. Several meshes are constructed, partitioned, and used to derive matrices. The partitions are measured using the popular metrics and compared with the overall rate of computation.

4.1.1 Experiment Design

In Chapter 2, we explored matrix-vector multiplication on a single compute node with a simple discretization of the Laplacian operator $\nabla^2$ in two and three dimensions. Below, we use the same discretization on various three dimensional domains to understand better how these design metrics affect the run-time of parallel matrix-vector multiplication.

When evaluating finite difference stencils in parallel, there are two options for organizing the computation: sharing nodes and ghosting nodes. With shared node schemes, each compute node calculates the values of all nodes in memory in parallel and exchanges the data associated with shared nodes with appropriate neighbors. The incoming data are combined with computed data to arrive at a common value for each shared node. With ghost node schemes, each compute node stores extra nodes so that all other nodal data can be computed in parallel. At the beginning of the matrix-vector multiply algorithm, each compute node
Each scheme offers the ability to “overlap” communication with computation. Overlapped communication, as opposed to synchronized communication, lets the compute node continue computation without waiting for an explicit acknowledgement of receipt by the other party in the communication. For instance, the matrix-vector multiply algorithm in Figure 4.1.1 demonstrates one way to overlap communication of data in a ghost node scheme. In this way, the impact of the communication on the overall runtime is minimized.

Using this algorithm we performed a multitude of experiments. Each experiment entailed computing the average time to compute a matrix-vector multiply on a particular number of compute nodes for matrices derived from each of three meshes partitioned eight different ways. For instance, one experiment consists of timing results on \( p \) compute nodes for twenty-four different mesh/partitioning combinations. The average time of many matrix-vector multiplies was taken for each combination. The partitioners used included coordinate bisection, exact Cartesian nested dissection (with two distinct parameters), the rough partitioning described in Section 2.8.1, the ParMetis k-way graph partitioner, the ParMetis geometric k-way graph partitioner (with two distinct parameters), and the ParMetis space-filling curve partitioner.

The experiments were performed on two types of cube and a mesh we call a sliced cube (Figure 4.3). For each experiment, a cube was generated so that each compute node would have a fixed number of nodes, roughly. For instance, an experiment that ran on \( p \) compute nodes and required approximately \( n \) nodes per compute node generated a cube with \( \left( \lfloor \sqrt[3]{np} \rfloor \right)^3 \) nodes. This mesh was chosen since coordinate bisection will produce near optimal partitions of this mesh. This mesh was then rotated and distorted so that coordinate bisection would fail to generate small separators. This second mesh was the same size as the original cube.

Finally, cells in the second mesh were removed in order to produce “fingers,” slender protrusions that often introduce difficulty in mesh partitioning. Each
of the slices were two or three cells thick and separated by one cell. Figure 4.3 shows an example of a sliced cube with five slices on the left and eleven slices on the right.

For each mesh, the dual graph was partitioned. Several popular partitioning metric statistics were collected. For each metric, the average, variance, and maximum were computed. The metrics were the number of dual nodes per compute node, the number of dual edges cut on each compute node, the number neighbors each compute node must send data, and the number of disjoint partitions on each compute node.

A large number of experiments were run varying the number of compute nodes from 2 to 200 and varying the number of cells per processor from 1,000 to 100,000. Not all experiments were successful due to time constraints on mesh generation and memory constraints on partitioning. However, a great deal of data were collected in the 1,000 to 25,000 nodes per processor range.

### 4.1.2 Finding Correlations

The correlation between each statistic and average matrix-vector multiply time is given in Table 4.1. First, it is reassuring to note that the number of compute nodes used has very little correlation with the run-time of matrix-vector multiply, roughly 1.2% of the variance in matrix-vector run-time can be explained by the number of compute nodes. Also, the average number of dual nodes per compute node is the most important factor in determining matrix-vector multiplication speed. Therefore, if the mesh is partitioned in a somewhat intelligent way, the most important factor in determining the time of matrix-vector multiplication is the size of the mesh.

The second statistic that correlates most with matrix-vector multiply runtime is the maximum number of dual nodes on a single compute node. It turns
Table 4.1: Importance of various indirect measures on computation rate (products per second).

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Correlation coefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average number of dual nodes</td>
<td>0.53</td>
</tr>
<tr>
<td>Maximum number of dual nodes</td>
<td>0.42</td>
</tr>
<tr>
<td>Maximum number of cut dual edges</td>
<td>0.30</td>
</tr>
<tr>
<td>Average number of cut dual edges</td>
<td>0.29</td>
</tr>
<tr>
<td>Average number of neighbors</td>
<td>0.27</td>
</tr>
<tr>
<td>Maximum number of neighbors</td>
<td>0.26</td>
</tr>
<tr>
<td>Variance of number of cut dual edges</td>
<td>0.25</td>
</tr>
<tr>
<td>Variance of number of neighbors</td>
<td>0.21</td>
</tr>
<tr>
<td>Maximum number of patches</td>
<td>0.16</td>
</tr>
<tr>
<td>Variance of number of patches</td>
<td>0.15</td>
</tr>
<tr>
<td>Average number of patches</td>
<td>0.15</td>
</tr>
<tr>
<td>Variance of number of dual nodes</td>
<td>0.13</td>
</tr>
<tr>
<td>Number of compute nodes</td>
<td>0.11</td>
</tr>
</tbody>
</table>

Table 4.2: Importance of various indirect measures on computation rate (rows per second).

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Correlation coefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of neighbors</td>
<td>-0.62</td>
</tr>
<tr>
<td>Average number of neighbors</td>
<td>-0.61</td>
</tr>
<tr>
<td>Variance of number of neighbors</td>
<td>-0.53</td>
</tr>
<tr>
<td>Number of compute nodes</td>
<td>-0.46</td>
</tr>
<tr>
<td>Maximum number of cut dual edges</td>
<td>-0.46</td>
</tr>
<tr>
<td>Average number of cut dual edges</td>
<td>-0.42</td>
</tr>
<tr>
<td>Variance of number of cut dual edges</td>
<td>-0.40</td>
</tr>
<tr>
<td>Maximum number of patches</td>
<td>-0.24</td>
</tr>
<tr>
<td>Variance of number of patches</td>
<td>-0.23</td>
</tr>
<tr>
<td>Average number of patches</td>
<td>-0.22</td>
</tr>
<tr>
<td>Variance of number of dual nodes</td>
<td>-0.22</td>
</tr>
<tr>
<td>Maximum number of dual nodes</td>
<td>-0.17</td>
</tr>
<tr>
<td>Average number of dual nodes</td>
<td>-0.02</td>
</tr>
</tbody>
</table>

Table 4.1: Importance of various indirect measures on computation rate (products per second).

out that the maximum value also correlates with the average number of dual nodes because the partitioners chosen avoid extremely poor partitionings. One way to reduce this bias is to consider the number of vector entries computed per second per node rather than the number of matrix-vector operations per second.

The times were divided by the average number of nodes—not dual nodes—per compute node. This new measure is relative only to the overall rate of computation independent of the size of the problem. The correlation coefficients of the various statistics to the new measurement of rate are given in Table 4.2.

This model is uncorrelated with average number of dual nodes per compute node with a shared variance of less than 0.04% (four basis points). In other words, less than four parts in ten thousand of the variance in the relative rate of
matrix-vector multiply (number of vector entries computed per second per node) can be attributed to the size of the problem. And, since the rough statistics of load imbalance, the variance and maximum number of dual nodes per compute node, are highly correlated with the average number of dual nodes, load balancing is not a major factor in the variance of the relative rate of the matrix-vector multiply. According to this experiment, roughly 5% of the variance of the rate of matrix-vector multiply is correlated with load balancing.

However, if we control for size of problem in a different fashion, we see a somewhat contradictory result. In Figure 4.4, the black + figures are matrix-vector multiply times for meshes of varying sizes. As can be seen, the meshes can be roughly divided into groups based on the average number of dual nodes per compute node. Within each group, the correlation coefficient of the maximum number of dual nodes and the speed of the matrix-vector multiply are somewhat correlated. Roughly one quarter of the variance of the time is explained by load imbalance.

When the using the same groupings, the correlation of the maximum number of dual nodes per compute node to the relative rate of the matrix-vector product is unchanged. This implies the apparent correlation between load balance and rate of computing products is small, less important than other factors in the computation. When controlling for overall problem size, load balance is not a major factor in performance.

One surprising result is the importance of the number of neighbors a compute node must send data to. In Figure 4.5, there is a well defined negative trend between the number of neighbors and the average number of vector rows...
computed per second on each processor. In fact, a little more than 35% of the variance in the rate can be attributed to the variance in the average number of neighbors.

It turns out that for cut dual edges and average neighbors, the maximum and average are almost perfectly correlated (Table 4.3). The dual edge and average neighbor statistics correlate to some extent, the variance in each explaining roughly a quarter of the variance with the other.

Another interesting result is the negative correlation between the number of compute nodes in the computation and the relative computing rate for each dual node. As the number of nodes increases, the rate of the computation decreases. Since we have controlled for problem size, we would expect there to be a sizeable correlation. Otherwise, the isoefficiency function of matrix-vector multiplication would be constant.

One final observation concerns the number of disjoint patches of mesh on each partition. When controlling for problem size, almost 5% of the variance in time can be explained by the variance in the number of disjoint patches. However, the majority of this sample does not have any disjoint partitions. If

Table 4.3: Correlation coefficient matrix for cut dual edges and number of neighbors.

<table>
<thead>
<tr>
<th>Avg cut dual edges</th>
<th>Max cut dual edges</th>
<th>Avg neighbors</th>
<th>Max neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.99</td>
<td>0.52</td>
<td>0.51</td>
</tr>
<tr>
<td>0.99</td>
<td>1</td>
<td>0.51</td>
<td>0.49</td>
</tr>
<tr>
<td>0.52</td>
<td>0.51</td>
<td>1</td>
<td>0.99</td>
</tr>
<tr>
<td>0.51</td>
<td>0.29</td>
<td>0.99</td>
<td>1</td>
</tr>
</tbody>
</table>
Figure 4.6: As the sample space becomes smaller, disregarding good meshes, negative correlation between maximum number of patches and computation rate increases.

we exclude these, there is a noticeable increase in the absolute value of the correlation coefficient.

Let $\alpha$ be the minimum of the maximum number of disjoint patches in a partitioned mesh to be included in a sample $S_\alpha$. To be included in $S_\alpha$, at least one partition must have at least $\alpha$ disjoint patches. For $\alpha = 1$, the correlation between the maximum number of disjoint partitions and relative computation rate decreases to $-0.25$. In this set, all samples in which all partitions are connected are disregarded. For $\alpha = 27$, a minimum coefficient is achieved: $-0.38$. This indicates the number of disjoint partitions may not be very important unless the mesh is “highly” concave. Figure 4.6 shows a plot of correlation coefficient versus $\alpha$.

4.1.3 Summary

Identifying differences in partitioners is difficult. Consider the data in Figure 4.7. In black are the matrix-vector product times for the rough partitioner used in Chapters 2 and 3. This partitioner mostly balances load and has just a couple of neighbors. The communication profile is horrible. Compare this to the data in red. These are very well balanced partitions generated with ParMetis. While it is obvious that the rough partitioner would be considered very bad, the results are generally not so bad. In fact, as the number of nodes per compute node increases, any difference between the partitioners disappears.

This result can also be seen in the correlation coefficients detailed above. In either set of coefficients, no single criterion explained a majority of the variance
Figure 4.7: Differences between partitioners can be difficult to detect. Averages of columns given by solid line.

seen in the data. And, since the data do not vary greatly, teasing out the importance of one metric over another is quite difficult. We have demonstrated some metrics are more important than others. For instance, minimizing the total number of communications appears to be more important than minimizing the volume of communication.

4.2 Mesh Partitioning Algorithms

Computing an optimal partitioning is difficult. Rather than attack the very difficult problem of computing a $k$-way partitioning, many algorithms attempt to compute a 2-way partitioning. A mesh (or surrogate) is split in two, or bisected. Each piece is bisected in turn until the mesh has been partitioned into the desired number of pieces. Obviously, the relative sizes of the pieces is important to the load balance of the computation. For instance, if there are $m$ cells in one of the bisected pieces and $m \mod k \approx k/2$, then the eventual partition may have poor load balance. Coordinate nested dissection [18] and space filling curves [123] are algorithms that ensure $m \mod k \approx 0$.

Algorithms that optimize load balance tend to use heuristics to reduce the amount of communication incurred by mesh partitioning. Algorithm designers make an assumption that mesh objects (edges or facets) indicate the amount of communication necessary to perform a computation. For instance, in finite difference meshes, each edge indicates which nodes contain data that are required for computation of the value at each node. The greater the number of edges cut in a partition, the greater the overall communication volume in the
computation. To varying degrees, algorithms use this information to reduce the total amount of communication required by a simulation.

For example, topological partitioners rely on connectivity information. Spectral methods, examples of purely topological partitioners, attempt to minimize the number of cuts by computing the eigenvectors of the connectivity matrix \([61, 62, 94]\). These algorithms reduce communication volume at the expense of load balance.

Coordinate nested dissection and recursive spectral dissection represent two extremes of computing a bisection of a mesh. Geometric partitioners, such as coordinate nested dissection, are fast and guarantee load balance. Topological partitioners, such as spectral dissection, reduce communication but are slow. Most partitioners attempt to combine aspects of each.

Meshes, especially meshes with good computational characteristics, have some topological uniformity. This allows algorithm designers to use geometric aspects to approximate topological characteristics and vice versa. Geometric algorithms use these assumptions to improve the communication profile of a computation. On the other hand, topological partitioners use these assumptions to improve partitioner execution time.

4.2.1 Geometric Partitioners

There are many different types of geometric partitioners. However, we are mainly concerned with those that make a particular assumption about the topology of the mesh: nodes and edges are evenly distributed. More precisely, the area of a cutting plane and the volume of communication are related by some invariant \(d\). We call this the Smooth Mesh Assumption.

Under this assumption, a bisector of a mesh will be a straight line (or plane, or other subspace normal to a vector). For instance, Cartesian nested dissection algorithms \([58, 115]\) identify cutting planes orthogonal to a Cartesian direction by first identifying a small range of the mesh in that direction that ensures load balance then fine tune the cut by minimizing the number of edges cut as the plane is swept across the small range.

Another geometric partitioner attempts to find the smallest separator, regardless of orientation. In inertial bisection algorithms, the vector used to compute the cutting plane is the principal inertial axis of mass distribution \([111]\). Since the inertial axis is the direction in which the mesh is “longest,” the smallest diameter of the mesh should be orthogonal to it. There have been many refinements to these and other algorithms, each of which has particular advantages and drawbacks.

For meshes that satisfy the smoothness assumption, these methods are optimal. For instance, a right parallelepiped made of cells with nearly constant volume can be partitioned extremely well with these partitioners. Not only is the total volume of communication reduced, but the total number of communi-
cations is small.

To understand how minimizing the total number of communications affects the overall performance of a mesh simulation, consider the total time ($t_{\text{tot}}$) it takes to complete the transmission of a message from one compute node to another. This total time can be broken into two parts, the latency of the network ($t_{\text{lat}}$) measured in seconds and the throughput of the network ($t_{\text{thru}}$) measured in bytes per second. If a compute node needs to send $b$ bytes to another compute node, then it will take $t_{\text{tot}} = t_{\text{lat}} + b/t_{\text{thru}}$ seconds to complete the message transfer. Reducing the number of edges cut by a partitioner reduces $b$. Reducing the number of messages a compute node needs to send for each step of the simulation reduces the impact of $t_{\text{lat}}$.

Interesting meshes, however, are non-convex, have irregular surfaces, and contain varying cell sizes depending on the resolution of the desired computation. This can lead to larger communication volume and underutilized parallelism. Consider the partitioning of the non-convex domain shown in Figure 4.8. The partition depicted on the right has three disjoint sub-meshes. Any local matrix derived from this mesh would be block diagonal,

$$\mathbf{A}_{\text{local}} = \begin{bmatrix} \mathbf{P}_1 & 0 & 0 \\ 0 & \mathbf{P}_2 & 0 \\ 0 & 0 & \mathbf{P}_3 \end{bmatrix}. \quad (4.1)$$

Since this matrix is obviously reducible, there is a loss of potential parallelism in the computation.

### 4.2.2 Topological Partitioners

Also called combinatorial techniques, topological partitioners mainly treat the computation purely as a graph. Any geometric information used by these algorithms speeds computation or aids parallel implementation. For instance, the popular MeTiS software library [74] uses a topological algorithm to partition the graph.
This algorithm is a multilevel algorithm in that the graph is **coarsened**, simplifying the problem. Nodes are combined based on connectivity until the partitioning algorithm can efficiently compute a bisection. Then, the mesh is refined as further bisections are computed. This algorithm can be computed in parallel, but at a significant cost compared to a partitioner that uses some geometric information to compute coarse grids [75].

Topological partitioners tend to be slower than geometric partitioners and many ignore the total number of messages transmitted between partitions. For smaller simulations, this is not a problem due to the surface effect discussed in Section 2.7.2. As the number of compute nodes increases for a particular domain, the average number of neighbors will also increase.

### 4.2.3 Other Partitioners

Much recent work in matrix partitioning has focused on improving the model of computation. For instance, the use of bipartite graphs and hypergraphs [60] can be used to model communication costs between partitions more accurately. Unfortunately, these models add a level of complexity to the computation. Computations that are simple on a graph become more complex with some of these models. For instance, computing the minimum spanning tree for hypergraphs is NP-complete [117]. Models such as these do not assume a locality guaranteed by a mesh and solve a more general problem.

### 4.3 Hybrid Partitioning Strategy

Each type of partitioner has relative strengths over the other. A geometric partitioner is fast, easy to parallelize, and produces partitions that require a small number of messages. Topological partitioners, on the other hand, produce connected partitions that minimize message size. Topological partitioners also generate better partitions for meshes with irregular shape, varying cell sizes, *etc.*

As a topological algorithm partitions the mesh, the pieces generated at each level of the recursion are smoother than the pieces in higher levels of recursion. After a few levels of recursion, the resulting partitions are much more likely to be partitioned well by a geometric partitioner. This observation motivates our hybrid algorithm. A topological partitioner is used to separate a complex mesh into simpler pieces. These simpler pieces are then partitioned by a geometric partitioner. We refer to the two phases as **coarse topological partitioning** and **fine geometric partitioning**.

In theory, any topological partitioning algorithm can be used for the coarse partitioner. Suppose a particular simulation needs a mesh partitioned into $k$ pieces. One way to accomplish this would be to compute $t$ topological partitions ($t \ll k$) and then partition each of these pieces $k/t$ ways. The optimal $t$
depends on the smoothness of the mesh. It is difficult to determine a good $t$
automatically.

To this end, we have developed a topological partitioner designed to generate
“smooth” partitions quickly. Non-smooth meshes generally have features that
prevent geometric partitioners from achieving optimal results. These features
can be identified by examining the medial axis of the surface of the mesh [113].
Rather than use a sophisticated feature detection algorithm, we employ a simple
one. While this algorithm may not find small features due to noise, it will
identify the larger features that cause problems for a geometric partitioner.
For instance, if a mesh is not simply connected, then the medial axis will not
be simply connected. Geometric partitioners may not identify a potentially
advantageous separation in the mesh, causing a large increase in communication
volume.

The algorithm proceeds in four phases. First, the medial axis is computed.
Then, the features of the medial axis are identified in parallel. Next, the mesh
is partitioned by these features, with individual cells assigned to a feature based
on locality to a medial axis feature. Finally, each of these features is parti-
tioned using a geometric partitioner. Since the first and last step are described
elsewhere, we describe steps two and three below. Rather than muddle the al-
gorithm description with the particulars of parallel implementation, we present
the topological partitioner in serial first.

\subsection{Serial Coarse Topological Partitioning}

The medial axis is a useful tool for identifying features in a structure. In partic-
ular, large features are easy to detect. Consider the example given in Figure 4.9.
The medial axis can be partitioned by looking for nodes that are incident on
three or more edges. In general, given an $n$ dimensional mesh with an $n-1$
(topological) dimensional medial axis, looking for $n-2$ dimensional mesh objects
with three or more incident facets will delineate the boundaries of a feature. In
Figure 4.9, the medial axis of this two-dimensional object has been divided into
thirteen distinct features, indicated by colors in the diagram.

This observation leads to a simple algorithm for identifying the facets in a
medial axis that belong to the same feature. If two facets (an $n-1$ dimensional
mesh object) have an $n-2$ dimensional mesh object in common, they are called
neighbors, and the $n-2$ dimensional object is called a connector. If a connector
is incident on exactly two facets, then the two facets are in the same feature.
Figure 4.10 gives an algorithm to group facets in the medial axis into features.
The input to IDENTIFYMEDIALFEATURES is the output from the medial axis
algorithm described in the previous chapter.

\begin{theorem}
The time complexity of IDENTIFYMEDIALFEATURES is $O(f)$,
where $f$ is the number of facets in the medial axis.
\end{theorem}
Figure 4.9: Medial axis features identified by intersection of three or more facets.

Figure 4.10: Algorithm to identify features.

```
IDENTIFY_MEDIAL_FEATURES(M)
1 feature ← 1
2 for each facet f ∈ M
3 do
4    if f.my_feature = 0
5      then
6          ADD_FACET_TO_FEATURE(f, feature)
7          feature ← feature + 1

ADD_FACET_TO_FEATURE(f, feature)
1 if f.my_feature = feature
2   then
3     return
4
5 f.my_feature ← feature
6 for each neighbor facet n of f
7 do
8   e ← connector between n and f
9   if e incident on two facets
10      then
11         ADD_FACET_TO_FEATURE(n, feature)
12
```

Figure 4.10: Algorithm to identify features.
Proof. The feature number of each facet is queried at most twice for each connector incident on the facet during recursion and once during the for loop in the main algorithm. Since the facets have the same structure as the facets in the mesh, each facet is incident on a small number of connectors, generally three or four for computational meshes. The assignment of a facet to a feature occurs only once.

The algorithm produces very good results for large portions of the medial axis. When run on the 450,000 cell mesh used in previous chapters, the large features of the medial axis are detected. Figures 4.11 and 4.12 demonstrate that the algorithm resolves features on the medial axis for each of the interior “fins” of the mesh as well as the surrounding cylinder.

As expected, several spurious features are detected. These are related to sharp corners in the mesh. In Figure 4.12, the hole at the top of the mesh (above) has sharp corners that are resolved as two distinct features, portrayed as white and brown (below).

By contrast, the sharp corner along the bottom of the mesh is resolved as several different features. In Figure 4.11, the different features are colored vermilion, brown, lavender, and others. The reason this is not resolved as a single feature is the choice of twiddle factor in computing the medial axis. Gaps in this feature occur because some facets of the medial axis are not resolved. In the end, this is not an issue with the partitioner since these features will be subsumed into the feature representing the cylinder, shown in green in the figures.

In Figure 4.13, many of these types of features are resolved. The green feature is the cylinder, the magenta feature is one of the fins of the star grain of the mesh. Along the top of this feature (and the other fins) are myriad little features, each just one or two facets in size. These corner features are governed by surface facets that are too close to each other to compute the medial axis accurately by the algorithm in the previous chapter. As with the bottom corner features, each of these will be included with the magenta feature.

These spurious features can be filtered out by computing the number of cells associated with each feature. Intuitively, a cell is associated with a medial feature if a surface normal passing through the cell intersects the feature. We developed a greedy algorithm to assign cells to features. The features are sorted by an estimate of the feature size. Then, starting with the largest feature, a wave is propagated from the feature until the wave exceeds the distance between the medial axis and the surface. Due to discretization error, not all dual nodes are classified through this process, so each feature is grown until all the cells are classified. The size of the feature is then computed by counting how many dual nodes are assigned to the feature.

Computation of the medial axis may be highly ill-conditioned. For this reason, the size of the medial feature cannot be used to estimate the size of the
Figure 4.11: Mesh (top) and features of medial axis detected by algorithm (bottom).
Figure 4.12: Cut-away view of mesh (top) and features of medial axis detected by algorithm (bottom).
Figure 4.13: Detail of fin in star grain.
Let \( r \) be a priority queue on dual node distances

for each surface facet \( f \) in \( M \)

\[
\text{do}
\]

\[
f\.\text{centroid} \leftarrow \text{facet centroid}
\]

\[
f\.\text{normal} \leftarrow \text{inward unit normal of facet}
\]

\[
p \leftarrow \text{dual node in } D \text{ associated with cell incident on } f
\]

\[
p\.d \leftarrow ||p - f\.\text{centroid}||_2
\]

\[
p\.\text{surf} \leftarrow f\.\text{centroid}
\]

\[
p\.\text{count} \leftarrow 0
\]

Push \( p \) onto \( r \)

\[
\text{while } r \text{ is not empty}
\]

\[
\text{do}
\]

\[
p = r\.\text{Pop()}
\]

for each neighbor dual node \( q \) of \( p \)

\[
\text{do}
\]

\[
d = ||q - p\.\text{surf}||_2
\]

if \( d < q\.d \) or \( q\.d \) unset

\[
\text{then}
\]

\[
q\.d \leftarrow d
\]

\[
q\.\text{surf} \leftarrow p\.\text{surf}
\]

\[
q\.\text{count} \leftarrow p\.\text{count} + 1
\]

Push \( q \) onto \( r \)

Figure 4.14: Updated marching method to compute number of cells between facet and surface, with additions shown in red.

corresponding mesh feature. However, by counting the number of cells between a facet on the medial axis and the surface of the mesh during the medial axis computation, we can get a rough estimate of the number of cells between the medial axis and the surface. In order to accomplish this, we alter the marching algorithm given in Figure 3.7.

The algorithm approximates the distance from a dual node to the surface of the mesh. It does so by marching along dual nodes in as direct a route as possible. If the algorithm keeps track of the number of dual nodes—which is, by definition, the number of mesh cells—then each facet can store both the size of a maximal ball and (roughly) the number of cells between the medial facet and the surface. This trivial addition does not significantly affect the total running time of the algorithm. Figure 4.14 gives the new algorithm.

This minor change in the marching method allows the estimation of feature size with a single iteration over the medial facets. Once completed, the features are sorted by size in decreasing order and a wave is propagated from the medial axis back toward the surface. This will provide a more accurate estimate of the feature size and begin the actual partitioning process. We use the interior distance and surface feature algorithm in Figure 4.14 as a template to accomplish
Figure 4.15: Interior distance and surface point approximation computed via marching method.

this task.

Given a feature, the set of dual edges represented by the facets is iterated over. For each node associated with each edge, if the node has not been assigned to a partition, it is assigned to the partition associated with the feature and pushed onto a queue. While there are nodes in the queue, a node \( n \) is popped from the front. For each neighbor \( m \) of \( n \), let \( p \) be a point on the feature such that \( ||m - p||_2 \) is minimized over all points on the feature. If \( ||m - p|| < \alpha \), for \( \alpha \) the radius of the maximal ball at \( p \), then \( m \) is assigned to the current partition and pushed onto the queue.

Exact computation of the point \( p \) may be prohibitively expense. Rather, we employ the same heuristic used in the marching method in the previous chapter: the point on the medial axis estimated to be closest to \( n \) is used in the calculation. When a node is first inserted into the queue, the medial point of the facet represented by the dual edge of the node is used as the current closest approximation. The algorithm to collect dual nodes into a feature is given in Figure 4.15.

By prioritizing the largest features in the medial axis, dual nodes that may
Figure 4.16: Algorithm to complete rough partitioning of dual mesh.

be assigned to two or more features are assigned to the larger of the features. This reduces the impact of corners, since many of the dual nodes that would be assigned to corners will also be assigned to another feature.

Once the dual node assignment is complete, some dual nodes may not be assigned to a feature. These dual nodes are assigned to neighboring features using a marching algorithm (Figure 4.16). Since the majority of a feature, especially a large feature, will be identified during the march from the medial axis to the surface, these straggling nodes should not constitute a large portion of the mesh and can be added to any feature to which they are adjacent.

The mesh is almost partitioned by feature. Figure 4.17 shows an example of a such a rough partitioning. Note that cells along sharp corners have been grouped into features. In this example, the twiddle factor in the medial axis computation was reduced compared with the examples in Figures 4.11, 4.12, and 4.13. Notice that the sharp corner around the bottom of the mesh is resolved as only three distinct features instead of the many features in the previous examples.

It is now trivial to compute an estimate of the actual size of a feature. If a feature does not have enough dual nodes to be a partition, it is merged with a neighbor. In the dual mesh, the number of edges between these small features and each of their neighbors is used to determine which neighboring partition it is merged with, opting for the maximum number shared. The features are then merged with the neighbor to which they are most connected. The serial feature

```plaintext
RECLAIMLOSTDUALNODES(D)
1 Let s be a queue
2 for each node n ∈ D
3 do
4 if n not in a feature
5 then
6 Push n onto s
7 while s not empty
8 do
9 Let t be a queue
10 for each n ∈ s
11 do
12 if n has a neighbor m assigned to a feature
13 then
14 Assign n to that feature
15 else
16 Push n onto t
17 end
18 s ← t
```

be assigned to two or more features are assigned to the larger of the features. This reduces the impact of corners, since many of the dual nodes that would be assigned to corners will also be assigned to another feature.

Once the dual node assignment is complete, some dual nodes may not be assigned to a feature. These dual nodes are assigned to neighboring features using a marching algorithm (Figure 4.16). Since the majority of a feature, especially a large feature, will be identified during the march from the medial axis to the surface, these straggling nodes should not constitute a large portion of the mesh and can be added to any feature to which they are adjacent.

The mesh is almost partitioned by feature. Figure 4.17 shows an example of a such a rough partitioning. Note that cells along sharp corners have been grouped into features. In this example, the twiddle factor in the medial axis computation was reduced compared with the examples in Figures 4.11, 4.12, and 4.13. Notice that the sharp corner around the bottom of the mesh is resolved as only three distinct features instead of the many features in the previous examples.

It is now trivial to compute an estimate of the actual size of a feature. If a feature does not have enough dual nodes to be a partition, it is merged with a neighbor. In the dual mesh, the number of edges between these small features and each of their neighbors is used to determine which neighboring partition it is merged with, opting for the maximum number shared. The features are then merged with the neighbor to which they are most connected. The serial feature
Figure 4.17: Rough feature partitioning.
PartitionFeature(Mesh, k)
1   // Identify features in the mesh
2   M ← COMPUTEMEDIALAXIS(Mesh)
3   IDENTIFYMEDIALFEATURES(M)
4
5   // Partition dual mesh by feature
6   D ← Dual mesh of Mesh
7   Let F be an array of sets of medial facets
8   for each f ∈ M
9       do
10          Insert f into F[f.my_feature]
11       Sort F by set size, descending
12       for each set s ∈ F
13          do
14             ASSIGNDUALTOFEATURE(s, d)
15             RECLAIMLOSTDUALNODES(D)
16       for each feature with fewer than k dual nodes
17           do
18              Merge feature with most connected neighbor
19

Figure 4.18: Complete topological partitioning algorithm.

partitioning algorithm is given in Figure 4.18.

4.5 Parallel Coarse Topological Partitioning

The implementation of this algorithm in parallel is relatively straightforward. Through use of dual node ghosting and the bulk synchronous parallel pattern, all but one of the previous algorithms can be implemented with little alteration. Unfortunately, computing the medial axis features in parallel is more complex.

One way to parallelize this step is for each compute node to perform feature detection on its portion of the medial axis. Each compute node chooses feature identifiers (ids) distinct from all others by ensuring all feature ids modulo the number of compute nodes is the rank of the compute node. For instance, if there are n compute nodes, compute node i will choose feature numbers f such that i = f mod n. Since each compute node generates a set of feature ids disjoint from all others, there are no collisions. Once IDENTIFYMEDIALFEATURES completes, the common feature identifiers must be coalesced. For instance, in Figure 4.19, nineteen features are detected on three compute nodes when there are in fact only thirteen.

To complicate matters, a single feature may be disjoint on a particular compute node. Suppose a mesh has just one feature. On two compute nodes, the feature may be detected as three separate features, one part on the first and two parts on the second. The red feature at the bottom right of Figure 4.20 is an example of just such a feature. At top left, the feature is computed as many
Figure 4.19: On three compute nodes, thirteen actual medial features are detected as nineteen.

different features that must be combined together.

To accomplish this, we must create a map between the feature numbers. Since the feature numbers are determined by the connectivity of the medial axis, a map can be constructed during the AddFacetToFeature algorithm. This map will indicate which nodes on remote compute nodes are in which local feature. The change to the algorithm is given in Figure 4.21.

The feature identifier unification process requires just two communication steps. For the first step, a point-to-point communication between compute nodes and their neighbors disseminates the data collected in PAddFacetToFeature. This is used to map local feature numbers to a neighborhood map of feature numbers. The second step is a global communication in which the neighborhood maps are broadcast so that a global map can be constructed. The algorithm is given in Figure 4.22.

The next two parts of the algorithm, AssignDualToFeature and ReclaimLostDualNode, are parallelized in the same fashion as PInteriorDistanceSurfacePoint in the medial axis computation (Figure 3.16). During the inner loop, neighboring facets not in the current partition are stored in a queue and broadcast at the bottom of the outer loop. The final step, merging small features, is accomplished by adding a ghost layer to the dual mesh, computing the size of each feature and number of edges between features in parallel, and an MPI_Allgather collective communication to distribute all the data. To ensure edges are not double counted, an edge between a ghost node and a local node is counted only on the compute node where the local node has a smaller identifier than the ghost. Once this computation completes, each compute node has the
Figure 4.20: Features detected independently on distinct compute nodes (top) are resolved into single features. Bottom left shows facets colored by partitioning. Bottom right shows facets colored by feature.

```plaintext
PAddFacetToFeature(f, feature)
1 Let g be an associative map
2 if f.my.feature = feature
3 then
4   return
5
6 f.my.feature ← feature
7 for each neighbor facet n of f
8   do
9     e ← connector between n and f
10    if e incident on two facets
11       then
12         if other facet on another compute node
13           then
14             g[other facet] ← feature
15           else
16             AddFacetToFeature(n, feature)
17   end
18 return g
```

Figure 4.21: Simple addition to AddFacetToFeature to track which nodes on remote compute nodes are in which local feature.
**UnifyFeatureIDs**$(g)$

1. Let $m$ be an associative array
2. Send $g$ to neighbors
3. for each $f, \text{feat}$ pair received
   4. do
   5. Let $f'$ be the local copy of $f$
   6. $m[f'] \leftarrow \min(f'.\text{feature}, \text{feat})$
   7. MPI_Allgather the associative array $m$
4. for each $m'$ received
   5. do
   6. for each $a \Rightarrow b$ pair in $m'$
   7. do
   8. $m[a] = \min(m[a], b)$

Figure 4.22: Global feature identifier unification algorithm.

**MergeSmallFeatures**$(D, k)$

1. Let $f$ be an associative array
2. Let $c$ be an associative array
3. Ghost a layer of dual nodes on $D$
4. for each node $n \in D$
   5. do
   6. $f[n.\text{feature}]++$
   7. for each edge $e \in D$
   8. do
   9. Let $n_1$ and $n_2$ be the nodes on edge $e$ with $n_1$ having the smaller id
   10. if $n_1$ is local
       11. then
       12. $c[n_1.\text{feat}, n_2.\text{feat}]++$
   13. MPI_Allgather $f$ and $c$
14. Combine all received $f$ and $c$ with $c$
15. for each $\text{feat} \in f$
   16. do
   17. if $\text{feat} < k$
       18. then
       19. Merge $\text{feat}$ with most connected neighbor given by $c$

Figure 4.23: Parallel merge feature algorithm.
same data for determining which features to merge. Any ties in the merging process are broken first by feature size (the merged feature going to the larger neighbor) then by feature identifier (smaller has priority).

Putting all of these together gives the final parallel topological partitioning algorithm. Once the dual mesh has been partitioned, a geometric partitioner is used to divide each feature into the desired number of pieces. Figure 4.24 gives an example of a mesh partitioned this way.

4.6 Future Work

As part of broadening the appeal of the data structures mentioned in Chapter 2, we have included hooks to the ParMetis library. We plan to time matrix vector products of the finite element stiffness matrix arising from the mesh used extensively in this chapter for one and three degrees of freedom at each node. Further, we will obtain other complex domains with which to compare times for matrix vector products. To ensure direct comparison, each partition will number the cells using RCM as suggested in Chapter 2.

Timing a matrix-vector multiply provides a direct measure of the efficiency of the partitioner. Indirect measurements, including statistics such as variance of partition size, number of dual edges cut, number partition neighbors, and volume of total communication, will also be computed.

As pointed out in Section 4.1.3, determining the relative efficacy of this hybrid partitioner to others will be difficult. However, for certain problem domains, this partitioner should have benefits beyond reducing the impact of communication on the computation. This sort of rigorous testing will require many different types of meshes as well as a substantial amount of computation time.

4.6.1 Implementation Refinement

Beyond testing the partitioner, we plan to improve the implementation of some of the algorithms. For instance, the walk back from the medial axis to the surface can be improved by overlapping computation with communication as suggested in Section 3.6.1.

There are several portions of the code that can be optimized. For instance, tightening the ghost management in the implementation will reduce both the amount of work and communication in the overall computation. Once these inefficiencies have been removed, we will also perform scalability tests and measure overall performance of the partitioner.

4.6.2 Improved Feature Partitions

As pointed out above, a geometric partitioner works optimally when several conditions are met. In order to improve the results from the geometric partitioner, we propose a post-processing phase of the feature partitioning. Several
Figure 4.24: Example of mesh partitioned with hybrid partitioner. Discolorations are artifacts of the rendering process.
of the mesh manipulations could reposition the dual nodes spatially so that they are more evenly distributed through the mesh. One way to do this would be to sort the nodes into buckets such that each bucket contains the same number of nodes. Each bucket can then be made uniform on the mesh, inducing a transform. Applying this transform to the dual nodes would generate a more uniform distribution of nodes.

Another idea is to align the mesh along the Cartesian axes so that inertial dissection routines would be equivalent to coordinate dissection routines. This can be done by computing a bounding box around the feature and rotating the bounding box accordingly. This also induces a transform which can then be applied to the nodes.

More advanced refinements include detecting and unrolling tori, “straightening” a smoothly curved feature, and other transforms designed to make the resulting features appear more rectilinear and regular. By approaching the problem in this fashion, the geometric partitioning begins to resemble a topological partitioner more and more, but still retaining the features of both.

4.7 Remarks

Two trends in supercomputing indicate the probable future of computing. The first, dubbed Moore’s Law [100], is the exponential growth of the number of transistors on a chip and the logarithmic decay of the cost of manufacturing them. For years, this growth translated into faster compute nodes. Due to thermodynamic considerations, the speed of these compute nodes has hit a plateau. To improve performance, chip manufacturers have created chips with multiple compute nodes. This implies that the amount of RAM per compute node will begin to decrease.

The second trend is the increase in the number of compute nodes on a computer. The use of a network hierarchy and the cheapness of components means these computers appear to be limited by power consumption alone. The computers are getting larger but the amount of memory per compute node may be shrinking.

We draw one conclusion from this: managing meshes will become a completely parallel undertaking. A mesh will be generated in parallel, operated on in parallel, and visualized in parallel. For large scale simulations, the mesh will never be serialized. Hence, mesh partitioning should be viewed entirely as a refinement operation. And, like all refinement algorithms, they should be measured in terms of relative performance increase versus operational cost.

For instance, as a simulation evolves, the performance of the iterative solver degrades. The mesh is repaired, negatively affecting the performance of the matrix-vector multiply operation. The mesh is partitioned to improve performance. As such, there should be work in performing local repairs. For instance, expensive topological partitioning of a small part of the mesh may make sense...
if the mesh is already partitioned relatively well.
Toward a Parallel Stable Direct Mesh Solver

The worst enemy of life, freedom and the common decencies is total anarchy; their second worst enemy is total efficiency.
— Aldous Huxley

Adversity has the effect of eliciting talents, which in prosperous circumstances would have lain dormant.
— Horace

At the heart of most engineering simulations is the equation \( Du = f \) where \( u \) and \( f \) are functions in some Hilbert space and \( D \) is an operator. \( D \) is often linearized (as in Newton’s method) and discretized (as in the finite element method) to arrive at a system of simultaneous linear equations

\[ Ax = b. \]  

(5.1)

Most discretization methods render \( A \) sparse, meaning each row or column has a relatively small number of non-zeros. For mesh-based discretizations, each row or column is computed based on the dependencies of a degree of freedom associated with a single mesh entity. Because mesh entities have a restricted number of adjacencies, each row or column has a small number of non-zeros, independent of the number of rows or columns in the matrix. For non-trivial meshes, the resulting matrix is necessarily sparse.

For self-adjoint \( D \), the resulting matrix \( A \) is symmetric and definite. These types of matrices can be solved very efficiently both iteratively (using the conjugate gradient method) or directly (using Cholesky factorization). The harmonic operator \( \nabla^2 \) is the canonical example of such an operator.

Unfortunately, for many interesting problems, \( A \) may be nonsymmetric or indefinite. These systems can be solved iteratively (using GMRES or BiCG) or directly (using Gaussian elimination). This chapter deals with the direct solution of these matrices.

In order to compute the solution to the simultaneous equations, Gaussian elimination computes two triangular matrix factors of a square matrix: \( A = \)
\( LU \), where \( L \) is a lower unit triangular matrix, and \( U \) is an upper triangular matrix. Solving the system using these matrices is simple, requiring \( \mathcal{O}(n^2) \) computations for \( n \) the number of columns in the matrix.

Gaussian elimination is *unstable*—it can fail to solve a system of equations even when a solution exists. The algorithm computes each diagonal entry \( u_{i,i} \) of \( U \) as the ratio of some number to the associated diagonal entry of \( A \), \( a_{i,i} \).

For the trivial example

\[
\begin{bmatrix}
0 & 1 \\
1 & 0 \\
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1 \\
\end{bmatrix}
=
\begin{bmatrix}
b_0 \\
b_1 \\
\end{bmatrix},
\]

(5.2)

Gaussian elimination will fail to find the solution \( x = [b_1 \ b_0]^T \). One way to stabilize Gaussian elimination is to interchange, or *pivot*, matrix rows to avoid dividing by zero. One pivoting strategy is to swap the diagonal entry with the entry of largest absolute value below the diagonal. This method, called *partial pivoting*, improves stability by limiting the rate at which entries in the upper factor can grow [112, 118].

Gaussian elimination is typically described by triply nested loops, but the computation can be organized in other ways. One such way is to take advantage of the recursion pattern set up by a mesh partitioner. Solvers of this nature are referred to as multi-frontal elimination algorithms. True multi-frontal elimination algorithms recursively produce factors from block arrow matrices. Consider such a block arrow matrix

\[
M = \begin{bmatrix}
U_1 & 0 & B_1 \\
0 & U_2 & B_2 \\
B_3 & B_4 & B_5
\end{bmatrix},
\]

(5.3)

for \( U_1 \) and \( U_2 \) upper triangular. At each level of recursion, these methods calculate the upper factor of the matrix using block Gaussian elimination:

\[
U_M = \begin{bmatrix}
U_1 & 0 & B_1' \\
0 & U_2 & B_2' \\
0 & 0 & U_S
\end{bmatrix},
\]

(5.4)

for \( U_S \) the upper factor of the Schur complement of \( M \). This sort of recursion naturally fits with nested dissection mesh partitioning, since \( U_1 \) and \( U_2 \) correspond to the upper factors of each partition and \( U_S \) is the upper factor of the computed mesh separator.

For many mesh-based problems arising from self-adjoint, definite operators, these methods stably compute the upper factor of the underlying matrix. Unfortunately, the elimination of \( B_3 \) and \( B_4 \), and the computation of the upper factor of the Schur complement, \( U_S \), for other types of problems may become unstable. The instability can be overcome by pivoting rows or columns of the matrix.
Unfortunately, pivoting violates the strict recursive definition of multi-frontal methods. Consider two separate recursion levels of the algorithm on the matrix

\[
M = \begin{bmatrix}
U_1 & B_1 \\
U_2 & B_2 \\
B_3 & B_4 & B_5 \\
D_3 & & & \\
\end{bmatrix}
\begin{bmatrix}
U_3 & C_1 \\
U_4 & C_2 \\
C_3 & C_4 & C_5 \\
D_4 & D_5 \\
\end{bmatrix}.
\] 

(5.5)

During the elimination of the upper left arrow matrix, rows may require pivoting. For any pivoting algorithm to be correct, rows in the \(D_1\) matrix must also be pivoted. Since these matrices are out of the scope of the current level of recursion, multi-frontal methods cannot implement pivoting without searching for rows and columns not in the current context.

Implementation of pivoting schemes in parallel complicates matters further. For instance, consider the matrix in Equation 5.3. Assume \(U_1\) and \(U_2\) are on separate compute nodes. The \(B\) matrices represent the boundary layer between the compute nodes. Depending on ordering of the elimination algorithm, pivoting in \(B_3\) affects the matrices \(B_4\) and \(B_5\). If the compute nodes share the work of eliminating the \(B\) matrices, then some interprocessor communication must occur to coordinate the compute nodes.

This chapter develops a multi-frontal method with optimal search for rows and columns to be pivoted. Further, the algorithm schedules elimination of boundary matrices in such a way that simultaneously reduces the amount of communication necessary to perform partial pivoting and provides a way to coordinate the remaining communication in parallel for best performance.

5.1 Node Numbering, Fill, and Mesh Partitioning

Sparse matrices do not necessarily have sparse factors. In fact, the number of non-zeroes in the factors is very sensitive to the ordering of the rows and columns of the matrix. As scaled rows are subtracted from one another in the computation, non-zeroes are introduced. These non-zeroes are called fill. The amount of work necessary to compute a solution to a system of equations increases as fill increases. Unfortunately, computing an ordering that minimizes the amount of fill is NP-complete [121].

Several heuristics are used to reduce the amount of fill incurred during elimination. For instance, minimum degree algorithms [47] reduce fill by eliminating rows with a smaller impact on the sparsity of the factors before those with a larger impact. This simple description belies the complexity behind these
Another fill-reducing ordering is nested dissection. Just like the partitioning pattern described in the previous chapter, a separator is identified in the matrix such that the remaining rows and columns form a reducible matrix. The separator is reordered to the bottom right of the matrix, giving the matrix an “arrowhead” profile as in Equation 5.3. Once the mesh is partitioned, the nested dissection heuristic used to partition the mesh is continued on each compute node to order the mesh objects.

5.2 Multifrontal Methods

Multifrontal elimination methods and nested dissection algorithms work well together. The nested dissection algorithm continually identifies separators until the remaining disjoint mesh pieces are trivial. This divide and conquer strategy creates an elimination tree. An elimination tree is a recursion tree with the separators as tree nodes and mesh objects not in a separator at leaves. Multifrontal elimination methods process the separators during a post-order traversal of the tree. Each node in the tree is dependent only on its children. In fact, the name multi-frontal is derived from the elimination tree nodes being operated on independently and the solution being obtained on multiple “fronts” simultaneously.

Using clever numbering, these methods can be implemented using dense kernels for symmetric definite systems. Organizing the computation in this fashion is possible because symmetric definite systems do not require pivoting for stable computation. As mentioned in Section 2.1.2, dense kernels generally perform better than their sparse counterparts due to cache effects and lack of indirect addressing.

Multifrontal solvers for sparse nonsymmetric or indefinite matrices must augment the computation somewhat to ensure the algorithms compute factors correctly [9, 34, 55]. UMFPACK, for instance, augments any “front” that requires a pivot with columns that have a similar non-zero pattern so that pivots can occur entirely within the front. Unfortunately, these methods require a search through the matrix to identify the columns to add to the current front. Next we will see that the mesh data structure described in Chapter 2 allows pivoting without searching for matrix entries, even on a distributed memory supercomputer.

5.3 Pairwise Pivoting in Parallel

Parallel implementation of direct matrix solvers requires entries of the matrix to be stored across the memory of a distributed supercomputer. The partitioning of a mesh determines which compute node stores which diagonal entry. Of-
diagonal entries can be stored by row, by column, or by connectivity. A matrix ordered for multi-frontal elimination stored by connectivity on two compute nodes has the form

$$M = \begin{bmatrix} A_0 & B_0 \\ A_1 & B_1 \\ C_0 & C_1 & C_s \end{bmatrix},$$

such that matrices with subscript 0 are on compute node 0, the matrices with subscript 1 are on compute node 1, and $C_s$ is shared between them. Each of these matrix decompositions may require communication to perform pivoting.

Unfortunately, deciding which rows require pivoting when using partial pivoting is $P$-complete [114]. In general, there is no non-deterministic algorithm that will predict which rows in a matrix will be pivoted except computing the factors explicitly. The need to pivot is unpredictable. Scheduling unpredictable phenomena in a distributed memory, SPMD environment is tricky.

Unpredictable behavior can be handled efficiently using interrupts: whenever a pivot requires communication, all required compute nodes are interrupted to perform the pivot. The cache coherence protocols of shared memory, symmetric multiprocessors, and multi-core architectures allow for a type of passive interrupt. Should one compute node need to communicate with others, the local cache lines of the other compute nodes is marked dirty, triggering a read from memory if those data are needed again.

Explicit message-passing architectures, on the other hand, offer limited support for this type of interrupt. Using MPI\_Put and MPI\_Get with sparse data structures or associative containers is very complex at best. For instance, these functions do not provide an interface for updating opaque associative containers. On these systems, pivoting must be explicitly scheduled.

One popular method for achieving this parallelism is called **pairwise pivoting** [109]. The approach of pairwise pivoting is to eliminate entries of the matrix by row instead of by column. The rows can be eliminated in a pipelined fashion [39] similar to parallel elimination of entries using Givens rotations [32, 93]. For instance, in the dense matrix in Figure 5.1, each box with the same number can be eliminated in parallel, even if elimination requires pivoting.
\[
M = \begin{bmatrix}
M_{1,1} & 0 & M_{1,3} \\
0 & M_{2,2} & M_{2,3} \\
M_{3,1} & M_{3,2} & M_{3,3}
\end{bmatrix}
\]

Figure 5.2: Resulting matrix structure after nested dissection. \(M_{1,1}\) and \(M_{2,2}\) have this same structure.

### 5.4 Stable Multifrontal Elimination

The result of nested dissection partitioners and node numbering algorithms is a nested block arrow matrix as in Figure 5.2. Due to this recursive structure, we need to describe how to eliminate only \(M_{3,1}\), \(M_{3,2}\) and the strict lower triangular portion of \(M_{3,3}\), a process we call \(M_{3,*}\) elimination.

The first step of the multi-frontal elimination algorithm is to eliminate \(M_{1,1}\) and \(M_{2,2}\) by recursively calling the elimination routine. Since these two matrices are not dependent, they can be solved independently of each other in any order. This parallelism is easy to exploit. Also, we can assume that after the recursive calls to the elimination algorithm return, the matrices are in the form given by Equation 5.3. The next step is to eliminate the \(M_{3,*}\) matrices.

Rather than eliminate each column in turn, our stable multi-frontal algorithm eliminates by rows. This allows row pivoting, since everything to the left of the current entry being eliminated is zero. If the current entry being eliminated is of larger absolute value than the diagonal entry above it, the rows are interchanged and the algorithm continues.

After determining the pivot, the elimination proceeds as expected, with the row with the corresponding diagonal entry being scaled and subtracted from the current row. Due to the pivoting, the scaling factor has absolute value at most one. This ensures that entries in the upper factor will not grow too quickly, destabilizing the elimination process [109]. The complete algorithm is given in Figure 5.3. To start, the algorithm is invoked as \texttt{MultiFrontalElimination}(\(A, A\)).

There are two lines of the algorithm that require searching through \(A\) for appropriate actions. First is the \texttt{SwapRows} algorithm (line 16) not depicted in the Figure. The second is the location of entries in rows \(c\) and \(r\) for appropriate updating (line 18). These operations can be optimized using a non-contiguous memory Standard Template Library container to store the upper triangular factor being computed.

### 5.4.1 Implementation Details

Any nested dissection ordering generates an elimination tree [84] of stencils. These stencils describe how particular mesh objects can be translated to a row or rows of a matrix. This elimination tree is used as a surrogate for the matrix \(A\) in the description above. Along with this elimination tree, the mesh objects
that contain the values for the right-hand side are included so that the unit lower triangular Gaussian factor need not be explicitly constructed.

The result of the elimination phase is the upper triangular factor of $A$. This factor is organized using the Standard Template Library map and vector containers. The vector provides $O(1)$ random and iterative access to each row. Each map provides $O(\log n)$ random access in the number of entries in the row and $O(1)$ iterative access. To facilitate pivoting, the upper triangular factor is stored as a vector of pointers to maps.

To ensure that the implementation has optimal asymptotic performance, the algorithm given in Figure 5.3 must be rephrased to use iterators. Figure 5.4 gives details of the iterator based algorithm. The functions begin, end, and increment (++) are used to access the entries of the elimination tree node, stencil, and upper triangular factor.

The implementation has the familiar triple nested loop, just expressed in terms of iterators. Except for the recursive function calls, each line is completed in constant time. Asymptotically, this algorithm is as efficient as an array-based triple nested loop implementation of Gaussian elimination.

### 5.4.2 Sources of Parallelism

There are two sources of parallelism in this algorithm. First, and most obvious, is the divide-and-conquer approach to elimination mentioned above. Since each
MFEIMPLEMENTATION($EliminationTreeNode, U, b$)
1 for each child $e$ of $EliminationTreeNode$
2 do
3 \hspace{1em} MFEIMPLEMENTATION($e, U, b$)
4
5 $s' \leftarrow EliminationTreeNode\.begin$
6 while $s' <> EliminationTreeNode\.end$
7 \hspace{1em} do
8 \hspace{2em} $s \leftarrow$ new map($s'$)
9 \hspace{2em} $U.push(s)$
10 \hspace{2em} $c \leftarrow s.begin$
11 \hspace{2em} while $c <> s.end$
12 \hspace{3em} do
13 \hspace{4em} if $s[c] > U[c] - > $ BEGIN
14 \hspace{5em} \hspace{1em} \text{PivotRow($s,U[c]$)}
15 \hspace{4em} then
16 \hspace{5em} \hspace{1em} $c' \leftarrow U[c].begin$ \hspace{1em} // First non-zero in row
17 \hspace{5em} update $\leftarrow s[c]/c'.value$
18 \hspace{5em} while $c' <> U[c].end$
19 \hspace{6em} do
20 \hspace{7em} \hspace{1em} $s[c'] = s[c'] - update \times c'.value$
21 \hspace{7em} $c' + +$
22 \hspace{7em} $b[s.row] \leftarrow b[s.row] - update \times b[s.col]$
23 \hspace{6em} \hspace{1em} Remove $c$ from $s$ \hspace{1em} // $s$ has a constant number of entities
24 \hspace{6em} $c + +$
25 \hspace{6em} $s + +$

Figure 5.4: Elimination implementation using constant time access of STL containers.
mesh partition is in one-to-one correspondence with an elimination tree node, it is easy to exploit this parallelism.

The other source of parallelism is not as obvious. When close to the leaves of the elimination tree, the remainder of the algorithm still runs in an “embarrassingly parallel” fashion. As the algorithm nears the root of the elimination tree, another source of parallelism can be exploited. The middle loop of the triple nested loop can be pipelined.

Without loss of generality, assume that two compute nodes are executing the algorithm simultaneously. Each of the child nodes of the elimination tree can be handled in a round-robin fashion. Once the children are eliminated, each compute node will enter the outermost loop simultaneously (line 5 of Figure 5.4).

At the outset of the second loop, one of the compute nodes will generate a token for the first row to be eliminated. The other will wait for the token to start computation. When the first compute node has completed the first row, the first row token is passed to the second compute node and the first compute node generates a second row token. In this way, \( p \) compute nodes can theoretically eliminate \( n \) rows in \( O(n + p) \) time.

Assuming there are no pivots, the token consists of a vector to be applied to \( M_{3,3} \) and the current right-hand side. A little math demonstrates that row \( i \) of \( M_{3,3} \) is updated by

\[
m'_{i,*} = m_{i,*} - u_i^T M_{1,3},
\]

for \( m_{i,*} \) the \( i \)th row of \( M_{3,3} \) and \( u_i^T M_{1,3} \) part of the token. The second compute node applies this update to \( M_{3,3} \) and continues elimination. Since \( M_{1,3} \) is stored in a vector of map pointers, the dot product can be evaluated efficiently. Pivoting adds a minor complication to the process.

5.4.3 Parallel Pairwise Pivoting

At the start of the elimination algorithm, we assume that the mesh objects that give rise to elements in the mesh are distributed according to a nested dissection algorithm. In other words, excluding the first call to MFEIMPLEMENTATION, the data are split in two with half of the data on one set of compute nodes and the other half on another set of compute nodes.

We develop the algorithm by first considering the simplest case of parallel computation: two compute nodes. This description provides insight into the complete algorithm, since the algorithm follows the same pattern as nested dissection.

Two Compute Nodes

If the matrix resulting from the elimination tree were formed, then, referring to Figure 5.2, the matrices \( M_{1,1}, M_{1,3}, \) and \( M_{3,1} \) would be on one compute node;
\( M_{2,1}, M_{2,3}, \) and \( M_{3,2} \) would be on the other; and \( M_{3,3} \) shared between them. We assign responsibility for elimination to the compute nodes accordingly, with \( M_{3,3} \) solved by the second compute node.

When the elimination algorithm completes, the output is an upper triangular matrix, ready for back substitution. To reduce communication during this phase of computation, each row of the matrix should be on one and only one compute node. Without pivoting, this behavior occurs naturally, but, when pivoting occurs, the algorithm may need to exchange rows between the two compute nodes.

This communication happens only if the first compute node needs to pivot. By eliminating by rows, the algorithm ensures that all entries to the left are zero in both rows to be pivoted. Therefore, when pivoting on the second compute node, the first compute node would be exchanging rows of zeros. Since the second compute node eliminates the shared matrix \( M_{3,3} \), when using two compute nodes, we expect most pivots in MFEImplementation to require no communication.

If the first compute node needs to pivot rows, the zeros in what would be \( M_{1,2} \) are replaced by a row in \( M_{3,2} \). To complicate matters, pivoting also affects portions of the matrix not described by the current elimination tree node being operated on. Here, we rely on the mesh data structure to find these portions of the matrix and move the data as necessary.

Before the data are sent, the values are placed in the vector of maps data structure. Then the map corresponding to the pivoted row is packaged and exchanged between compute nodes. To simplify this process, line 8 of the MFEIMPELEMENTATION algorithm extracts the entire row associated with the stencil rather than just the portion in the current scope of elimination. Since each row of the matrix is processed only once at line 8, redundant data extraction will not occur.

**Generalized Pivoting Scheme**

The token passing scheme extends to multiple compute nodes eliminating rows simultaneously. Consider Equation 5.5. Matrices \( D_3 \) and \( D_4 \) can be expanded to demonstrate how their elimination can be computed in parallel:

\[
\begin{bmatrix}
U_1 & B_1 \\
U_2 & B_2 \\
B_3 & B_4 & B_5 \\
D_{3,1} & D_{3,2} & D_{3,*}
\end{bmatrix}
\begin{bmatrix}
U_3 & C_1 \\
U_4 & C_2 \\
C_3 & C_4 & C_5 \\
D_{4,3} & D_{4,4} & D_{4,*}
\end{bmatrix}
\begin{bmatrix}
D_{1,1} \\
D_{1,2} \\
D_{1,*} \\
D_{2,3} \\
D_{2,4} \\
D_{2,*} \\
D_{5,*}
\end{bmatrix}, \quad (5.8)
\]
where $D_{i,j}$ indicates the portion of $D_i$ located on compute node $j$ and $\ast$ indicates
the matrix is shared. It is evident from this matrix partitioning that compute
node 1 can eliminate a row in $D_{3,1}$, pass the token to compute node 2, and
continue elimination. Compute node 2 can eliminate $D_{3,2}$ and $D_{3,\ast}$ and pass
the token to compute node 3, and so on.

The pivoting algorithm works in a similar fashion as above. Instead of
a point-to-point communication between two compute nodes during a pivot,
there is a scatter communication between relevant compute nodes rooted at the
compute node that needs to pivot. If compute node 1 pivots, then the data are
broadcast to 2, 3, and 4. If compute node 3 pivots, only 4 needs to be updated.

Separator Elimination

The last data to be coordinated are in the $D_{5,\ast}$ matrix from Equation 5.8.
By extension of the two compute node algorithm, compute nodes 2 and 3 are
responsible for eliminating this matrix. The rows are renumbered so that the
computation can be pipelined via dense pairwise parallel elimination [39]. The
complete multi-frontal Gaussian elimination with pairwise pivoting algorithm is
given in Figure 5.5.

5.5 Pivoting Study

The algorithm outlined above presents two possible pitfalls with regard to length
of computation. First, the algorithm potentially increases the number of row
pivots, thereby increasing the volume of communication necessary to perform
the elimination. Any algorithm that potentially increases the number of row
pivots during elimination may also increase fill. This increase in fill must be
kept small to prevent a significant further increase in computation.

To test the pivoting behavior of the algorithm, we factored a matrix derived
from the Helmholtz equation,

$$-\nabla^2 u = \kappa^2 u. \quad (5.9)$$

The Laplacian operator was discretized using the standard centered finite dif-
ference approximation given in Equation 2.4,

$$\frac{\partial^2 u}{\partial x_i^2}(x) \approx \frac{u(x + h e_i) - 2u(x) + u(x - h e_i)}{h^2}. \quad (5.10)$$

For $\kappa^2 = 0$, the resulting matrix is diagonally dominant and requires no pivoting
for computing the solution. In three dimensions, as $\kappa^2 \to 6$, the condition
number of the matrix approaches infinity.

We chose two different domains on which to test our method. The first, a
notoriously difficult domain for multi-frontal methods, is a cube. The mesh is
PivotRowSend$(s, U[c], \text{token})$
1 Scatter $U[c]$
2 $t \leftarrow$ map from gather operation
3 $t \leftarrow t - \text{token}^TU$[appropriate range]
4 $s \leftarrow U[c]$
5 $U[c] \leftarrow t$

PivotRowRecv$(s)$
1 $t \leftarrow$ Map from scatter operation
2 Send back $s$ in gather operation
3 $s \leftarrow t$

PMFIEImplementation$(\text{EliminationTreeNode}, U, b)$
1 for each child $e$ of $\text{EliminationTreeNode}$
2 do
3 MFEImplementation$(e, U, b)$
4 $s' \leftarrow \text{EliminationTreeNode}$.BEGIN
5 while $s' <> \text{EliminationTreeNode}$.END
6 do
7 $s \leftarrow$ new map$(s')$
8 $U$.PUSH$(s)$
9 $c \leftarrow s$.BEGIN
10 if on compute node 1
11 then
12 $\text{token}_c \leftarrow$ new vector
13 else
14 Wait for $\text{token}_c$ and listen for pivot
15 Received $\text{token}_c$
16 $U[s] \leftarrow U[s] - \text{token}_c$
17 while $c <> s$.END
18 do
19 if $s[c] > U[c][c] - >$ BEGIN
20 then
21 PivotRowSend$(s, U[c], \text{token}_c)$
22 $c' \leftarrow U[c]$.BEGIN
23 $\text{update} \leftarrow s[c]/c'.value$
24 $\text{token}_c[e] \leftarrow \text{update}$
25 while $c' <> U[c]$.END
26 do
27 $s[c'] = s[c'] - \text{update} \times c'.value$
28 $c' + +$
29 $b[s.row] \leftarrow b[s.row] - \text{update} \times b[s.col]$
30 $c + +$
31 $\text{token}_c \leftarrow \text{token}_c^TU$[range of $c$], $b[s.row]$
32 send $\text{token}_c$ to next compute node
33 $s + +$

Figure 5.5: Parallel Gaussian elimination with partial pivoting.
structured with $n^3$ nodes, $n$ in each dimension. The separator at the root of the elimination tree computed by nested dissection grows as $O(n^2)$. The matrix representing this separator, $M_{3,3}$ in Figure 5.2, is dense. Thus, $O(n^4)$ entries of the matrix must be eliminated.

The second mesh we chose is a right parallelepiped, or brick, with $n$ nodes along eight edges and $10n$ nodes along the remaining four. A mesh of this shape has a much smaller separator at the root of the elimination tree compared to a cube. Though this separator still grows as $O(n^2)$, the matrix representing the largest separator for a cube has more than twenty times the number of non-zeroes than the comparable matrix derived from the brick. Problems arising from meshes with similar aspect ratios (such as the 2.7 million element mesh in Figure 2.11) are possible candidates for multi-frontal solutions.

We performed serial experiments on four meshes, two cube meshes with 3375 ($15^3$) nodes and 29791 ($31^3$) nodes, and two brick meshes with 3430 ($7 \times 7 \times 70$) nodes and 33,750 nodes ($15 \times 15 \times 150$). To construct each elimination tree, we used the exact Cartesian nested dissection algorithm [115] mentioned in previous chapters to order the nodes to reduce fill.

We then computed the triangular factors using two methods. First, to illustrate partial pivoting, we used the MATLAB sparse matrix solving package, which in turn calls UMFPACK [34]. The MATLAB routine returns three matrices, $L$, $U$, and $P$ such that $PA = LU$. We compute the number of non-zeroes in $L + U$ and the number of pivots required by UMFPACK from $P$ using the algorithm in Figure 5.6.

These experiments were carried out on two different computers, each running
a version of Microsoft Windows and each having three gigabytes of memory. For some matrices, the elimination method failed due to insufficient memory. In general, for matrices with more than 30 million non-zeroes, the eliminations failed.

The second method used our data structure and the pairwise pivoting strategy outlined above. We counted the number of eliminated matrix entities, corresponding to the number of non-zeroes in $L - I$. Then, we counted the number of non-zeroes in the computed upper factor. We also kept track of the number of pivots computed by the algorithm. The eliminations were performed on an Apple G5 running Linux with four gigabytes of RAM. No memory limitations were encountered.

We performed our experiment varying $\kappa^2$ from 0 to 5.75. For $\kappa^2 = 5$, we obtained results for partial pivoting that correlate with the rest of the results presented in this chapter. On the other hand, we observed phenomenal results for pairwise pivoting. Generally, the elimination required a small fraction of the number of pivots, generating factors roughly equivalent in non-zeroes to the definite case of $\kappa^2 = 0$. This is due to the cancellation provided by a matrix with entries in the set $\{-1, 0, 1\}$ and the order of elimination. Since these results are due to the values in the matrix rather than to the problem itself, they are omitted.

There is an obvious link between $\kappa^2$ and the number of pivots in partial pivoting. We note that as $\kappa^2 \to 6$, the elimination of the matrix requires more pivots. In Figure 5.7, except for one outlier, we see that the relative number of pivots required for stable elimination is independent of mesh size or shape. Most likely, this outlier is an example of the cancellation mentioned above, with the pivoting strategy choosing updates that reduce fill.

On the other hand, Figure 5.8 demonstrates a different behavior for pairwise pivoting. Since pairwise pivoting may require multiple pivots of the same row, measuring the number of pivoted rows will underestimate the amount of pivots. For this reason, we counted the total number of pivots. In this case, the number of pivots for the smaller problem approach some value. For the larger problem, the maximum number of pivots is reached and then the number of pivots declined as $\kappa^2$ increased.

Generally, the number of non-zeroes in the factors varies directly as the number of pivots. Except for instances of cancellation, the number of non-zeroes increases as sparse rows are subtracted from one another. Figure 5.9 demonstrates this relationship for both partial pivoting and pairwise pivoting.

The Helmholtz problem, when sufficiently well conditioned, is suited to iterative solution [41]. Consequently, we are particularly concerned with the performance of the multi-frontal algorithm with respect to matrices with poor condition numbers, to provide a viable alternative to iterative methods. The top plot in Figure 5.10 demonstrates that as the matrix conditioning deteriorates, the relative performance of the pairwise pivoting strategy improves. The
Figure 5.7: As $\kappa^2$ approaches 6, more and more rows require pivoting.

Figure 5.8: For pairwise pivoting, number of pivots quickly reaches a maximum.
Figure 5.9: As number of pivots increases, so does number of non-zeroes in factors.
dashed red line roughly indicates the size of the largest factors that could be computed with the MATLAB $lu$ function on computers with three gigabytes of RAM.

The bottom plot indicates how well our data structure uses memory. While the data we collected cannot be used for a direct comparison of memory usage, we believe our data structure uses memory more efficiently than compressed sparse column, the format preferred by UMFPACK. The reason is simple: UMFPACK requires two very large contiguous arrays, one for the matrix data and one for the row data.

For a matrix with thirty million non-zeroes, the data are stored in a 229 megabyte array, a 114 megabyte array, and an array of negligible size. Windows, by convention, allows users access to only half of the RAM of the computer. Even though the data require less than a quarter of available RAM, the memory manager may not be able to allocate contiguous blocks due to memory fragmentation. Our approach obviates the need for contiguous blocks, thereby allowing more efficient use of memory, even if we require more of it.

5.6 Future Work

5.6.1 Improve Solver Efficiency

The current solver implementation requires some further work to improve efficiency. For instance, rows are copied explicitly during a pivot, rather than having pointers manipulated as described above. The interface can also be refactored to improve efficiency. For instance, an explicit matrix row is maintained to ease implementation. Since the matrix row need not be used directly, this can be taken out of the computation altogether.

5.6.2 Pairwise-Static Pivoting Strategy

Partial and pairwise pivoting may increase fill beyond the point of usability. There are pivoting strategies employed to reduce the impact of pivoting on fill, particularly that of static pivoting [82]. The idea is to perform a row or entry scaling instead of pivoting if the pivot would introduce an unacceptable amount of fill. This approach is not the best for all problems but could substantially reduce run time for appropriate problems.

5.6.3 Parallelization of Separator Elimination

Many multi-frontal algorithms eliminate the final separators in parallel, generally by assigning the rows in a round-robin fashion. The separator itself can be reordered to reduce fill, albeit in a limited sense because the separator is generally dense. In this case, the reordering will be used to partition the matrix
Figure 5.10: As number of pivots increases, so does number of non-zeroes in factors.
across the entire communicator so that the elimination can be pipelined using pairwise pivoting.

A simple calculation shows the total work would be $O(n^4/p + p)$. For a brick of 10 million nodes in the same aspect ratio as described above, the number of nodes in the largest separator would be ten thousand. If ten thousand compute nodes are used in the calculation, then each compute node would need to eliminate roughly ten thousand entries. This gross calculation indicates the viability of multi-frontal methods for large scale parallel computations, given the appropriate domain.

5.6.4 Implementation in Parallel

Finally, we need to undertake the technically daunting task of implementing this algorithm in parallel. The data collected in our experiments and the serial implementation indicate that this algorithm may be useful for the direct solution of highly ill-conditioned systems, at least for some aspect ratios. We have deferred parallel implementation until we had sufficiently studied the probable parallel performance of the pivoting strategy.
Some scientists are driven by the noble ideal of research, science for science’s sake. For others, solving the problem is motivation enough because interesting problems often beget other interesting problems. No matter the scientist, they follow the same, rote method. They take a statement of undetermined validity and set about demonstrating its truth or falsity.

Theoreticians use rigor, logic, and a predefined set of rules to come to an undeniable conclusion. Empiricists design experiments to demonstrate that a statement is not true or possibly not false. Generally, there is a feedback loop with results from one driving the experiments of the other. As computers become larger, the disconnect between the theoreticians and empiricists narrows. Computational scientists are increasingly required to perform both roles. They derive models, demonstrate that the models are mathematically sound, then implement the models to compare against real world phenomena.

Computers are fast becoming tools of empiricists, and not just to make plots and typeset documents. The acceptance of computer analysis software in engineering has spilled over into nearly all experimental fields of science. Computers show great promise in extending the limits of experiments beyond observable phenomena. As in any scientific endeavor, the key to success is to leverage the work of others.

With this in mind, we considered large-scale, mesh-based engineering simulations. There are innumerable solvers, methods, and techniques used in engineering simulations. Some are general purpose, and some are refined for very narrow cases. There is active research in nearly every aspect of these simulations.
This presents another problem, beyond the technical challenge of performing the simulation: organizing the code for the most effective use of the computer.

We set about the rote method given above. Our statement is simple: Organizing the computation around actions on the mesh provides a robust means for developing simulation codes. One possible experiment is obvious: design a mesh data structure and perform simulations with it. Unfortunately, a task of such enormous size was beyond our resources. Thus, we went with a smaller scale experiment: design a mesh data structure and perform science with it.

6.1 Three Motivations

The purpose of an application programming interface is to allow developers to write software. Due to the specialized nature of large-scale mesh-based parallel engineering simulation, efficiency is not the only goal. In particular, our design was motivated by this and two other factors: risk mitigation and reduced coding time.

6.1.1 Risk Mitigation

Production software must work because opportunities to use large supercomputers are scarce. If it does not work well, the entire computing allotment may be swallowed by fixing bugs. Research software, on the other hand, is by definition novel, meaning it is relatively untested. Many researchers corral their efforts into one large simulation in the hope of generating results that prove their theories and software work. Programming supercomputers for complex simulations is a risky venture indeed.

Like all risks, there should be a hedge against failure. Failures are to be expected, but their effects can be minimized. Due to the collaborative nature of these simulations, they are designed as a series of actions performed by methods. One way to minimize the risk is to categorize the components of the simulation into three groups: untried, unproven, and proven. An untried component has never been used in the computing environment, be it computer architecture, problem size, or other factor in the simulation. An unproven component has demonstrated success on one or several similar simulations, but not the current simulation. And a proven component is widely regarded as “working.”

To broaden the appeal of the simulation, encourage collaboration, and foster scientific progress, priority is given to untried components followed by unproven ones. To guarantee success, each task of the simulation should have at least one component in each group. As components fail, they are replaced with more stable ones.

For instance, a simulation may rely on partitioning a mesh in parallel. In this case, there should be an untried partitioner (such as the hybrid partitioner in Chapter 4), an unproven partitioner (such as the exact Cartesian nested
dissection partitioner in [115], and a proven partitioner (such as ParMetis [75]). Should the untried partitioner fail due to coding errors or fault in design, the simulation can use the unproven partitioner. Should the unproven partitioner be flawed for this particular simulation, the proven partitioner is employed. If the proven partitioner fails, then science has demonstrated the need for more research on the task.

Our approach to this problem was not simply to have swappable components in a simulation. Rather, it was to minimize the number of key strokes required to perform the swap. How many key strokes does it take? For native methods, a complete swap takes a handful of keystrokes: change the method in the action in the code and recompile. For non-native applications, a wrapper must first be written, then the swap occurs as in a native switch.

We can demonstrate this by switching between the conjugate gradient method and Cholesky factorization in a serial application by changing just part of one line of code. Similarly, we can swap between ParMetis, coordinate dissection, and exact Cartesian nested dissection for partitioning the dual of a mesh by changing only part of one line of code. We expect similar results for exchanging types of elements, solution transfer algorithms, mesh refinement protocols, and the like.

6.1.2 Reduced Coding Time

All well-designed simulations allow for a relatively seamless component exchange. That does not imply that the simulation is easy to maintain. Due to the scarcity of supercomputer time, programmers are often forced to “kludge” shortcomings of software for a temporary fix. Unfortunately, a temporary fix is temporary only if it does not work. That is, if the fix works, then the priority given to replacing the kludge is reduced substantially. It has been our experience that the most nefarious kludges appear at the seams.

One reason for this is the bottom-up approach taken to implement simulations. Models are chosen that dictate solvers. The solvers in turn dictate data structure organization. The data structures dictate how components must fit together. In theory, this approach should generate solid code. But, solid code is far from guaranteed.

Software development is an iterative process in which each decision limits the programmer’s choices thereafter. The earlier a decision is made in the implementation process, the greater impact it will have. For this reason, the part of the code that requires the biggest decisions should be implemented first. Otherwise, a seemingly innocent design decision in implementing one solver may cause large problems in all other solvers. For instance, choosing how cells are represented in memory is a decision that should not be made by a programmer writing a solver.

\[1\text{We consider this a highly qualified success}\]
Thus, we adopted a top-down approach to design. As stated above, this greatly reduces the amount of code necessary to glue components together. Does this necessarily reduce overall coding time? Have we traded macroscopic code for reinventing the wheel? Organizing the code in this fashion requires reimplementation of matrix-vector multiply, GMRES, and a host of other components that exist in extremely optimized libraries. Or, alternatively, we must write a wrapper.

However, this top-down approach is also advantageous to coders who write code on the call stack between the solvers and the large-scale simulation outer loop. A large-scale simulation can be thought of as a (loosely) coupled set of smaller simulations. For instance, a fluid-structure code may contain three different simulations, one for structural simulation, one for fluid simulation, and one for the interaction of the two. These simulations should also be easier to write.

Finally, by restricting our code to mesh-based simulations, we have reduced the parallel programming overhead for solver writers. Using associative containers allows a more natural coding style with less-error prone indirection. Even though the wheel must be reinvented, the process should be shorter.

6.1.3 Increased Efficiency

At the beginning of this thesis, we pointed out a certain unnaturalness of modern simulation design. Mathematically, a mesh-based approximation gives rise to a matrix that is then solved. Gropp, et al. demonstrated that using a vanilla sparse solver on these problems may not use the compute node efficiently. Rather, the matrix should be ordered to use cache efficiently. They recommended an ordering derived from the mesh, reverse Cuthill-McKee. From this result, we conclude that any linear solver can be improved with a tighter coupling to the mesh.

We take this process one step further. For instance, when multiplying with matrices arising from finite difference stencils, a plain matrix-vector multiply multiplies values by $-1$ before adding them to results. We demonstrated we can optimize out these multiplications and simply subtract the results, substantially reducing the amount of computation necessary to perform the matrix-vector multiplication.

There is also promising research in reducing the number of floating point operations in the assembly of a finite element stiffness matrix [120]. These results can be used directly with our data structure to reduce the amount of computation required to perform an element-by-element matrix-vector multiply.

Finally, the data structure is designed to use memory better. Most mesh data structures and solvers rely on contiguous memory allocation. Use of associative containers instead of contiguous memory allows the operating system greater freedom in allocating space. As graphically demonstrated in Chapter 5, our
data structure can perform in situations where standard direct solvers may run out of memory.

6.2 A Science of Science

The second half of our experiment is performing science. This is an empirical experiment. At best, we can conclude science may be viable and at worst that it is not. Since empirical observation is measurement, we present a simple metric to quantify our results: how many successful experiments we conducted. While we do not believe this will be the case universally, our data structure performed very well in each experiment.

In this thesis, we presented six experiments carried out with the data structure. Some are theoretical, some are empirical, and some are computational. Some were in parallel and some were in serial. In all cases, we obtained results. And, as mentioned above, we obtained results when some high performance libraries were incapable of completing due to memory limitations. In one application not mentioned in this thesis, our data structure succeeded in partitioning a mesh for which a popular library failed entirely.

One striking result is the actual performance of the matrix-vector multiply routine compared to the theoretical maximum performance of a vanilla matrix-vector multiply routine. As part of that experiment, we implemented a routine to compute a reverse Cuthill-McKee ordering of the mesh. We implemented this routine in less than an hour and it can be used on any type of mesh. While our performance may be atypical, it is compelling anecdotal evidence that our hypothesis is true.

6.3 A Path to Acceptance

The mark of truly useful software is ease of integration. Our data structure faces three hurdles with respect to being widely used. First, it is remarkably different from other data structures used in large-scale simulation. The gap between sequential containers such as arrays and associative containers is substantial. It is technically possible to implement the mesh interface using sequential arrays, but at a substantial cost in programming. One way around this is gradually to integrate the functions into simulation and translate back and forth as necessary.

The second hurdle is the lack of support for many necessary parts of a large-scale simulation. Implementing wrappers is one way to solve this conundrum. Another is to develop gradually a complete simulation using the data structure through the effort of a team of programmers. Either case is taking a long view of the data structure and both are worth pursuing.

The final hurdle is legacy programming languages. The mechanism used to implement parametric polymorphism is not widely supported in either C or
Fortran\textsuperscript{2}. To use these languages across platforms, a wrapper must be called to a C++ library compiled with the appropriate template chosen. Admittedly this is awkward. For instance, in Fortran, a partitioner would be invoked by

```fortran
CALL MPART ( MHAND )
```

for \texttt{MHAND} a handle to the mesh. This function invokes a C++ function

```c
void mpart ( int m )
{
    MESH_TYPE m = *(void *)m;
    m.dual_partition<cnd> ();
}
```

where \texttt{MESH\_TYPE} is a macro defined as the canonical mesh type. In this case, to change the partitioner, one must change the C++ function and recompile the file. Other alternatives are equally unattractive.

These difficulties are not major impediments for wider acceptance. The software we developed is not the only way to implement these ideas. As programming languages evolve, parametric polymorphism will become more widespread and better supported.

### 6.4 Reflections on the Medial Axis

The medial axis is subtle. There are myriad tests for determining whether points are on the medial axis or if regions contain the medial axis. Still, accurate computation of the medial axis as feature size becomes small relative to sampling density evades most techniques. For our purposes, these small features are immaterial and do not need to be resolved.

The problem, of course, is ill-conditioning. A small patch of surface can contain all tangent points of a maximal ball. If one chooses surface points close to these tangent points, the radius of the new maximal ball, also called the feature size, can change dramatically. Surfaces can be fiendishly devised to make features arbitrarily ill-conditioned.

Computing the medial axis from a sampling of the surface makes the problem more difficult. By definition of conditioning, the sampling density implies a certain accuracy around any ill-conditioned feature. Our method attempts to rule out ill-conditioned features by a simple heuristic. If the surface patch used to compute a point on the medial axis is too small, ignore the result if the angle between normals is too small.

Perhaps a more direct conditioning estimate would result in better resolution of features. Hairs introduced by the surface mesh discretization are necessarily

\textsuperscript{2}The Fortran 2003 and 2008 specifications do allow some interoperability between templated C++ and Fortran
ill-conditioned. Also, assigning an estimated condition number to the approximate medial axis would be a good indication of the accuracy of the computation. For applications of the medial axis that can control sampling, estimating the condition number can allow a better sampling for more efficient computation of the axis.

As for our application, mesh partitioning, we do not need to resolve the complete medial axis. The large features detected by the partitioner are generally well conditioned. On the other hand, applications such as sharp corner feature detection may benefit from accurate resolution of the medial axis for small, well-conditioned features.

6.5 Importance of Mesh Partitioning

As a rule of thumb, compute nodes can generate data orders of magnitude faster than they can communicate it. For computations with highly dependent parts, such as matrix-vector multiply, the quality of the partitioning affects performance. Since it is a hard problem, we may never see an optimal solution. Fortunately, it is easy to generate a reasonable partitioning of the mesh, say computing a partitioning by coordinate nested dissection. For the sake of argument, assume coordinate nested dissection is the fastest way to generate a reasonable partitioning.

From this reasonable partitioning, all other algorithms can be measured. They can be measured in terms of how long they take and how much they improve the efficiency of matrix-vector multiplication. The ratio of these values determines how many matrix-vector computations are necessary to recoup the cost of the running the partitioner. Combining these two facts and information about the problem, a simulation designer can make an informed choice of partitioner.

Partitioning is not a one-size-fits-all problem. Rather, it requires a compromise between competing efficiencies. Depending on the problem, one type of partitioner may be better suited than another. Also, the domain plays an important role in how the partitioner affects efficiency. All of these factors indicate the need for many different mesh partitioning algorithms.

6.6 Direct Methods

Direct solutions on meshes arising from three-dimensional simulations may never be possible in general. Even though mesh separators grow as $O(n^2)$, implying dense solution of a matrix with $O(n^8)$ non-zeroes, direct solution should not be discounted.

Particular problems may be solved more efficiently using a direct method as opposed to an iterative method. As the matrix condition number increases, so
does the cost of an iterative method. For matrices with large condition numbers, iterative methods may stall, never producing a result. If the mesh is of the right shape, not dense or extremely elongated, direct methods may be the only way to arrive at a solution.

Furthermore, the use of pairwise pivoting instead of partial pivoting may limit the number of non-zeros in the resulting factors, especially as the condition number increases. Much more work needs to be undertaken to confirm this hypothesis.

6.7 Future Directions

Throughout, we have documented future improvements to the data structure and algorithms. Some are rote and obvious, while others indicate a particular direction for future research. For instance, we can improve the data structure through performance tweaks. One way to improve performance would be to implement faster associative arrays to be specifically used for high performance computation on meshes. Other improvements include increasing asymptotic efficiency of several algorithms.

Even though we have demonstrated a substantial benefit to using our data structure through its use in the pursuit of science, we have uncovered many other problems that need to be addressed. These problems lie on the path toward implementation of multi-physics simulations. This is, indeed, our final goal, and it is still too far away to plan for directly. Rather, our next goal is the implementation of an engineering simulation. This includes mesh actions such as mesh motion, mesh repair, and preconditioned iterative methods.
References


Author’s Biography

Bill Cochran was born and raised in North Carolina. He spent his formative years in the eastern part of the state and moved to High Point in 1984. He graduated from North Carolina School of Science and Math in 1992 and briefly attended The Johns Hopkins University. There he met his wife Kristine.

He completed a bachelor’s degree in computer science at Wake Forest University in 1999. There, he discovered his passion for scientific computing and decided to pursue this as a career. He entered the computer science Ph.D. program at the University of Illinois in 2000. While there, he and his wife were blessed with their son Evan Christopher.

When Bill is not working on engineering simulation, he fancies himself an entrepreneur. He holds several patents related to data security and occasionally tries to start an e-business.

Bill knows how to brew strong ale and makes the best hamburgers. He is pretty handy with a pool cue and devastating on a foosball table. If you are looking for a fun backyard barbeque or a little challenge on a table, you kind find him in Champaign for just a few more weeks.