Automated Parallel Solution of Unstructured PDE Problems

Guy E. Blelloch[1], Anja Feldmann[1], Omar Ghattas[2], John R. Gilbert[5], Gary L. Miller[1], David R. O'Hallaron[1], Eric J. Schwabe[3], Jonathan R. Shewchuk[1], Shang-Hua Teng[4]

[1]School of Computer Science
[2]Department of Civil Engineering
Carnegie Mellon University
Pittsburgh, PA 15213-3891

[3]Department of EECS
Northwestern University
Evanston, IL 60208

[4]Department of Mathematics and LCS
Massachusetts Institute of Technology
Cambridge, MA 02139

[5]Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304

CR Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems; G.1.0 [**Numerical Analysis**]: General; G.2.2 [**Discrete Mathematics**]: Graph Theory

General Terms: Algorithms

Additional Key Words and Phrases: Partial Differential Equations, finite element methods, graph partitioning, parallel processing.

Introduction

Many physical phenomena in science and engineering can be modeled by partial differential equations (PDEs). When these equations are complex or are posed on irregularly shaped domains, they usually do not admit closed-form solutions. A numerical approximation of the solution is thus necessary. Computational science and engineering has emerged as an activity that applies the tools of numerical analysis, computer science, and applied mathematics to creating computer models of natural and man-made systems, with the goal of gaining a better understanding of the chemical, electrical, fluid mechanical, magnetic, solid mechanical, or thermal phenomena exhibited by these systems.

The most popular methods for numerically approximating the solution of linear or nonlinear PDEs replace the continuous system by a finite number of weakly coupled linear or nonlinear algebraic equations. This process of *discretization* associates an equation (or set of equations) and an unknown (or set of unknowns) with each of a finite number of points in the problem domain. The accuracy of the numerical approximation improves as the number of points increases. However, the complexity of solving the algebraic system is typically superlinear in the number of points, and more time is spent solving the system than is spent modeling and interpreting the results as this number increases. Some problems are intractable on current sequential computers, and scientists and engineers have become interested in using parallel computers to attack them. PDE problems for which high performance computing is required include global climate change, aerospace vehicle design, cardiovascular blood flow, combustion, weather forecasting, air pollution, and earthquake-induced ground motion.

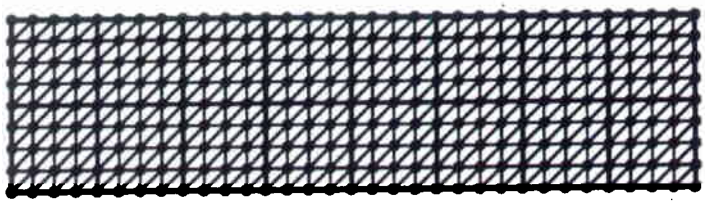

Figure 1: A structured discretization of a rectangle partitioned for sixteen processors.

Coupling among unknowns results from physical adjacency of points; hence, when points are laid down in a uniform fashion, the structure can be exploited to yield efficient parallel solvers. The domain is divided into similarly shaped subdomains, which are allocated to separate processors. Processors communicate in a regular fashion at subdomain boundaries. Figure 1 shows a simple *structured* PDE mesh divided into sixteen subdomains.

On the other hand, many problems are defined on irregularly shaped domains, and resist regular discretization. Others exhibit phenomena that occur on widely differing spatial or temporal scales. In this case, uniform distribution of points is inefficient, because the spacing of points must reflect the areas that require the finest resolution. The excess points in other areas lead to unnecessary computational effort.

Consider the *unstructured* mesh in Figure 2, which depicts a cross-section through the Kanto Basin, a layered valley within which the city of Tokyo sits. The problem is to predict the surface ground motion due to a strong earthquake. The discretization is finer in the top layers of the valley, reflecting the much smaller wavelength of seismic waves in the softer upper soil, and becomes coarser

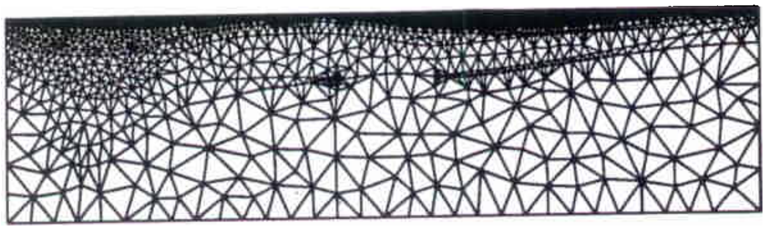

Figure 2: Unstructured discretization of Los Angeles Basin

with increasing depth, as the soil becomes stiffer and the corresponding wavelength increases. A structured discretization of the same problem would employ a uniform distribution of points, the density being dictated by the uppermost layer. Although the data structures, storage requirements, and parallelization of structured meshes are considerably simplified, the resulting several-orders-of-magnitude increase in number of unknowns is unacceptable. Other multiscale phenomena include turbulence modeling, plasticity, transonic aerodynamics, multi-phase flow through porous media, and crack propagation. Unstructured discretization of these problems can result in greater efficiency; unfortunately, it is difficult to map an unstructured mesh to a parallel computer.

Most parallel PDE solvers in the literature are of limited scope. Because of the difficulties, few parallel PDE solvers use truly unstructured meshes. Furthermore, most parallel PDE solvers are built to solve a specific problem (e.g. Navier-Stokes equations). Our philosophy is that if parallel computers are to contribute to scientific progress, it is necessary to build general-purpose tools that allow researchers in engineering and science to model the complexities of real domains.

This article describes Archimedes, an automated system for solving partial differential equations on geometrically complex domains using distributed memory supercomputers. The tasks of such a system are manifold. First, Archimedes discretizes the domain being modeled by generating an unstructured mesh that fills the region. Then, the domain is partitioned into separate subdomains, which are placed onto individual processors. Communication is routed between these processors. Finally, code is generated to solve a PDE in parallel. We discuss how we have automated each of these tasks, and describe our efforts to bring state-of-the-art computing to state-of-the-art engineering.

While the methodology employed by Archimedes is applicable to other unstructured discretization methods such as the finite volume method or collocation, we focus on the finite element method (FEM) because of its generality and widespread use.

Finite Element Methods

This section describes the structure of the equations that result upon application of finite element methods. Readers wishing a proper treatment of FEM should consult standard texts such as Becker, Carey, and Oden [2] for an introduction, and Strang and Fix [22] for mathematical analysis.

Consider a heat conduction problem posed on the two-dimensional domain of Figure 3, a metal plate. The problem is to find the steady state temperature $u(x, y)$ of the plate, given that the plate is exposed to specified heat sources. The physical behavior of this system is modeled by a partial

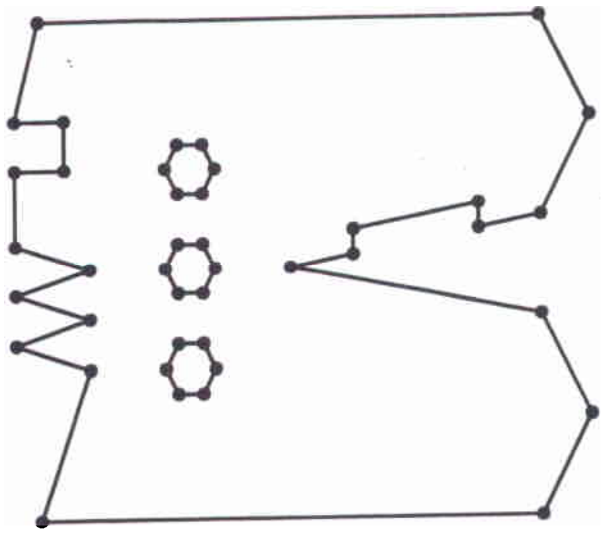

Figure 3: A metal plate with holes.

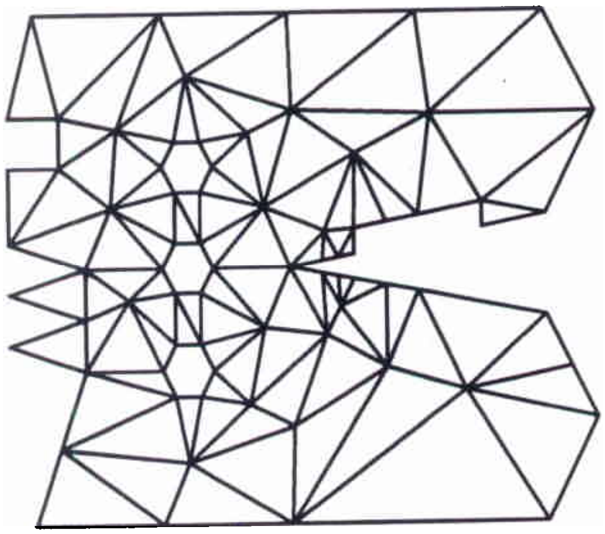

Figure 4: Finite element mesh.

differential equation[1]. We replace this continuous problem with a discrete approximation. Finite element methods achieve this goal by partitioning the domain into convex polygons or polyhedra, called *finite elements.* Elements are typically triangles or rectangles in two dimensions, and tetrahedra or rectangular blocks in three dimensions. With each element we associate points, called *nodes*, at which temperatures are sought. The simplest element is the linear triangle, which possesses three nodes, one at each corner. A *finite element mesh* is a collection of elements (and nodes) covering a domain. Figure 4 shows a mesh of linear triangles covering the metal plate.

By considering the effects of the differential equation over each element, we construct an approximate system of linear equations of the form

$$\mathbf{K}\mathbf{u} = \mathbf{f} \tag{1}$$

where $\mathbf{u} = [u_1, u_2, \ldots, u_n]^T$ denotes the temperatures at each of the n mesh nodes, the *force vector* $\mathbf{f}$ denotes the heat sources at each mesh node, and the $n \times n$ *stiffness matrix*[2] $\mathbf{K}$ describes the interactions among the nodes due to heat conduction. Once we have solved for $\mathbf{u}$, we can interpolate between the nodal values to produce an approximate continuous solution $\tilde{u}(x, y)$, which is piecewise linear on the mesh.

The structure of $\mathbf{K}$ is illustrated in Figure 5, which shows a simple mesh composed of nine elements and nine nodes. K_{ij} is nonzero only if there is at least one element which contains both nodes i and j. Hence, K_{24} is nonzero, K_{37} is zero, and K_{ii} is nonzero for any i. This implies that for large meshes, $\mathbf{K}$ is very sparse. The stiffness matrix $\mathbf{K}$ is computed by summing the effects of the differential equation over each individual element. Hence, K_{24} receives contributions from elements a and b, and K_{66} from elements f and g. The structure of $\mathbf{f}$ is similar; for example, f_6 receives contributions from f and g.

[1]The heat equation is $-\nabla \cdot [k(x,y)\nabla u(x,y)] = f(x,y)$, subject to boundary conditions, where $f(x,y)$ represents internal heat sources, and $k(x,y)$ is the thermal conductivity of the material.

[2]The terminology for $\mathbf{K}$ and $\mathbf{f}$ comes from solid mechanics, and has become widespread.

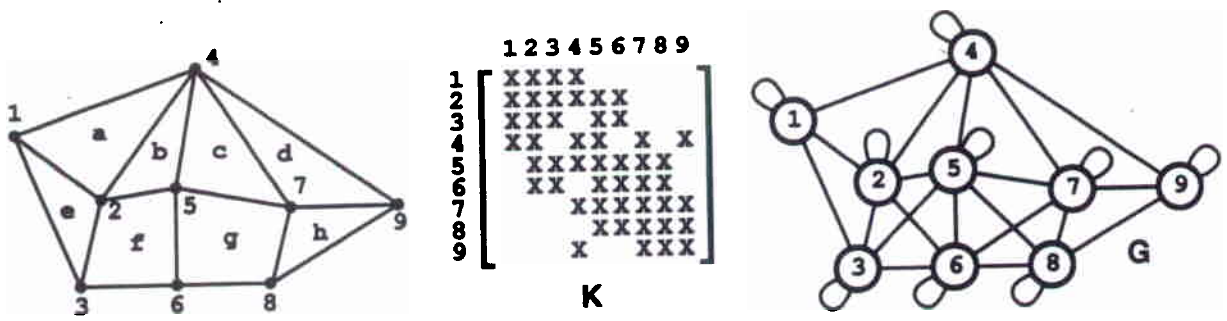

Figure 5: A finite element mesh and its associated stiffness matrix **K** and finite element graph G. X's indicate nonzero matrix entries.

Given a finite element mesh, the associated *finite element graph* G is defined such that its vertices are the nodes of the mesh, and there is an edge between any two vertices which belong to a common element (see Figure 5). G is isomorphic to **K**, in the sense that $\mathbf{K}_{uv} = 0$ if $(u, v) \notin G$.

Solving the linear equation (1) accounts for most of the computation and interprocessor communication in solving the PDE. The linear system may be solved either by a direct method such as Gaussian elimination [12] or by an iterative method such as conjugate gradients [9]. In this article, we shall consider only iterative methods, which are much easier to parallelize than direct methods. Specifically, we shall discuss how to perform in parallel the sparse matrix-vector product in the inner loop of most iterative solvers. We will use the finite element graph G to help us parallelize this operation.

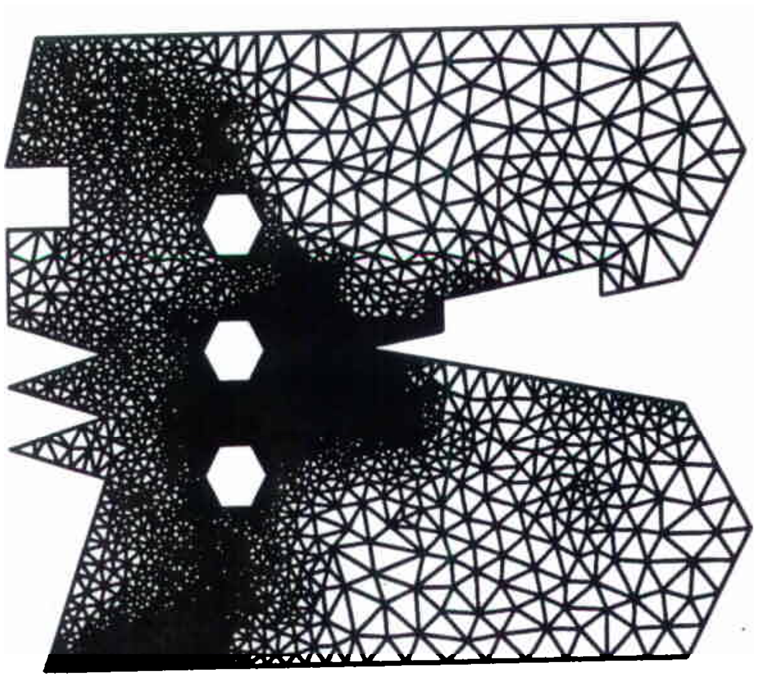

Figure 6: Refinement of the original mesh.

If the approximate solution $\tilde{u}$ is not as close to the exact solution as desired, a better solution can be found by refining the mesh and repeating the solution process. A mesh is *refined* by dividing some or all of its elements into a larger number of smaller elements, as in Figure 6. As the mesh is refined, the approximate continuous solution $\tilde{u}$ approaches the exact solution u. Numerical analysts have developed ***a priori* error estimates** which show that the *discretization error* $\|u - \tilde{u}\|$ is reduced as the size of the largest edge is reduced. However, as we have explained, a constant element density is not always desirable. Hence, there are also ***a posteriori* error estimates** which estimate the discretization error on each element after solving the problem, thereby providing a guide to refinement. Figure 6 was generated from *a posteriori* error estimates based on a stress simulation on the plate in Figure 4.

Beyond this example, finite element methods and our techniques can be extended to three-dimensional, time-dependent, higher-order, and nonlinear problems.

Parallel Finite Element Systems

Archimedes

At Carnegie Mellon we have built Archimedes, an automated system for solving partial differential equations over complex domains using distributed memory supercomputers. Archimedes is intended to allow researchers in engineering and science to solve real physical problems on unstructured meshes.

Figure 7 diagrams the structure of Archimedes. A user provides two things: a description of the geometry of the problem domain, and an algorithm for performing finite element calculations which solve some physical problem.

The problem geometry is the shape of the domain that will be modeled. In modelling earthquake ground motion, this would be the shape of a valley and the surrounding rock. In flight vehicle aerodynamics, this would be the shape of the surface of the vehicle and surrounding air. Archimedes generates a mesh that fills the region of interest.

If we wish to use a parallel machine to perform the finite element simulation, Archimedes must partition the domain into subdomains, one per processor. More specifically, it partitions the finite element mesh into separate pieces. Each piece should be approximately the same size, ensuring a good load balance. Each piece should have a relatively small boundary, minimizing the parallel communication.

Next, we want a one-to-one mapping of subdomains to processors. Depending on the parallel architecture, we may also need to route communication channels between processors, and perhaps even schedule the order in which communication is done. Archimedes uses placement and routing heuristics to perform these tasks.

Along with the problem geometry, the user writes an algorithm for constructing and solving the finite element system for a specific problem. For instance, to solve our heat conduction problem, a user would write an algorithm to compute $\mathbf{K}$ and $\mathbf{f}$, and solve for $\mathbf{u}$. The algorithm is written in a C-like syntax with built-in primitive operations tailored for finite element methods. This algorithm is machine-independent, and can be written without knowledge of the parallel machine's underlying communication mechanisms. Archimedes compiles the algorithm into code for a specific machine.

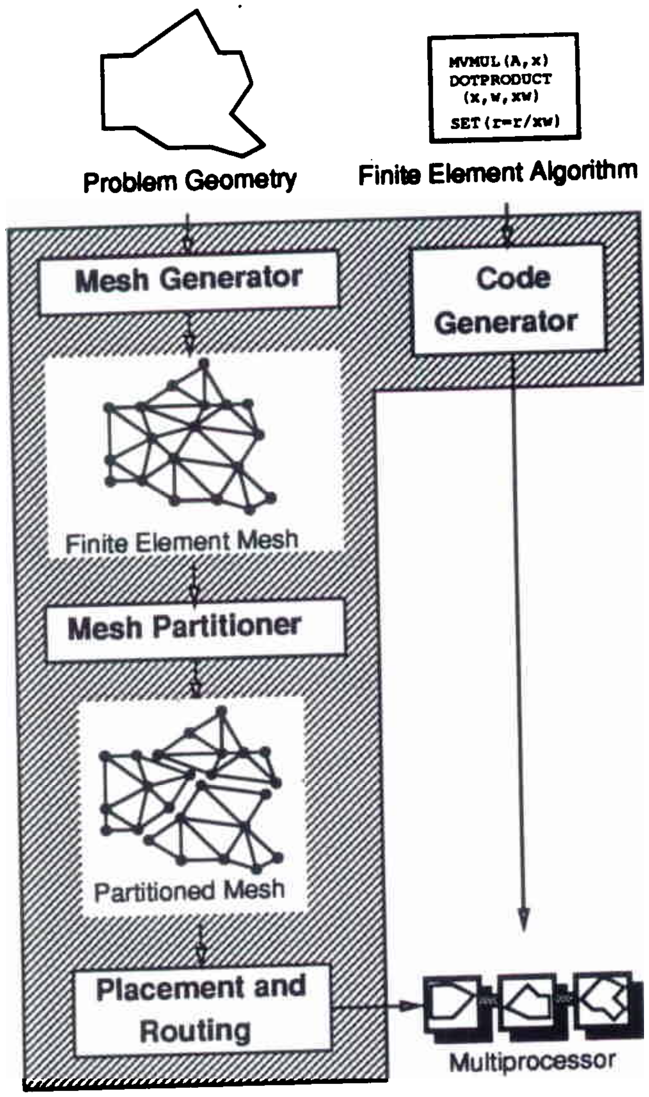

Figure 7: Structure of Archimedes.

If the algorithm is designed to generate *a posteriori* error estimates, this output can be fed back into the mesh generator for the purposes of mesh refinement, and the entire process is repeated.

Other Systems

There are several other systems designed for parallel PDE solving. We describe only a few selected systems in order to compare the differences in approach.

//ELLPACK [13] (pronounced "Parallel Ellpack") is a complete system for the solution of elliptic PDEs, including nonlinear and time-dependent problems. //ELLPACK has a high-level language interface and offers a variety of discretization methods and linear solution modules. It includes an extensive graphical user interface for specifying problem geometry, decomposing the region into subdomains, and analyzing solutions. Archimedes has followed a path similar to //ELLPACK, but with emphasis on working with unstructured meshes.

A different approach is taken by Saltz and his co-workers in the *PARTI compiler* [23]. This is a parallelizing Fortran D compiler with runtime support for computation on unstructured meshes. A programmer specifies the structure of an unstructured mesh implicitly as an ordinary Fortran loop which makes indirect array references.

The central PARTI primitives are inspector/executor pairs. The *inspector* loop examines (at runtime) the data references which will be made by a processor, and computes which data needs to be fetched from or stored on other processors. The *executor* loop uses this information to perform the actual computation. To minimize communication, PARTI includes a runtime partitioner which, like Archimedes' partitioner, distributes arrays across the processors. PARTI currently uses a spectral partitioning algorithm due to Pothen, Simon, and Liou [20].

Parallelizing Sparse Matrix-Vector Products

Choosing a Data Distribution

As previously mentioned, we consider only iterative methods of solving the linear system $\mathbf{Ku} = \mathbf{f}$. Most iterative methods have as their central primitive the problem of computing the product of a large sparse matrix $\mathbf{K}$ and a vector $\mathbf{x}$. For example, the iterative step of the conjugate gradient method consists of several vector operations and a single sparse matrix-vector multiplication. Note that over the course of an iterative method the nonzero structure of the sparse matrix $\mathbf{K}$ remains fixed (isomorphic to the finite element graph G), even though the values of the nonzero entries of $\mathbf{K}$ may change over time (for instance, in nonlinear problems).

To compute $\mathbf{y} = \mathbf{Kx}$ on a set of processors, we must consider the data distribution by which vectors are stored. Suppose that each element of the mesh is assigned to one processor, and each mesh node resides on one or more processors, depending on what elements contain that node. The vectors $\mathbf{x}$ and $\mathbf{y}$ are stored in a distributed fashion according to this mapping. If a node i resides on several processors, the values x_i and y_i are replicated on those processors. The matrix $\mathbf{K}$ is distributed so that K_{ij} resides on any processor on which nodes i and j both reside.

The multiplication $\mathbf{y} = \mathbf{Kx}$ is performed in two steps. First, each processor computes a local matrix-vector product over the subgraph of G that resides on that processor. Second, processors that share mesh nodes communicate and combine their nodal y values into correct global values for each node.

We find a data distribution by embedding the finite element graph G into a graph H, which represents the underlying interconnection network of the target machine. (The vertices of H are the set of processors, and the edges of H are the set of communication wires connecting the processors.) For our purposes, an *embedding* of a graph G into a graph H consists of a mapping from the vertices of G to the vertices of H, together with a mapping from each edge of G to a path (possibly of zero length) in H between the images of its endpoints. Each vertex of G is mapped to one or more vertices of H, and typically, many vertices of G are mapped to each vertex of H.

There are three primary measures of the efficiency of an embedding. One is its *load*, which is the maximum number of edges of G mapped to any single vertex of H. (To compute a sparse-matrix vector product, each processor does an amount of work roughly equal to the number of edges it is assigned.) The goal of keeping load low is to balance the parallel work among processors. The other two measures are its *dilation*, which is the maximum over all edges in G of the lengths of their images in H, and its *congestion*, which is the maximum over all edges in H of the total number of edges in G whose corresponding paths contain that edge. The goal of keeping dilation and congestion low is to minimize the total time required for communication.

The relative importance of dilation and congestion for a particular machine will depend on its communication mechanism and method of routing. For instance, when store-and-forward routing is used, it is crucial to minimize the number of "hops" that a message takes, and the reduction of dilation is a primary concern. On the other hand, if the machine in question uses circuit-switched routing then the length of the routing paths is not nearly as important as the number of paths that share a single edge at the most severe bottleneck, so congestion becomes more important.

Finding an Embedding

It is not easy to find an efficient embedding of a finite element graph G to a graph H. To simplify the problem, we break it into several stages.

First, we *partition* the mesh by cutting it into one piece for each processor. In an ideal partition, the pieces will be equal in size (to balance the load), and will have small boundaries (to minimize the amount of information which must be communicated between processors).

We define the *quotient graph* G' derived from G and the partition. G' is a graph with a supervertex for each piece of the partition, and an edge between two supervertices if they need to communicate with each other (because they share mesh nodes).

Second, we *place* the quotient graph G' on the target machine H. This placement is a one-to-one mapping of supervertices to processors. Finally, we *route* G' by finding a mapping of edges in G' to paths in H.

It is not necessary (nor necessarily optimal) to break the embedding problem into these stages. Better partitioning might be possible with knowledge of the target machine, and better placement might be possible if routing is considered simultaneously. However, finding an optimal embedding is difficult, and some simplifying assumptions are necessary. More importantly, we separate partitioning from placement and routing because partitioning is machine-independent, whereas placement and routing depend upon the communication mechanisms of the target machine.

Partitioning

Archimedes' partitioner is based on an algorithm due to Miller, Teng, Thurston, and Vavasis [19], which finds provably good separators for a large class of geometrically defined graphs. A *separator* is a small set of vertices or edges whose removal divides the graph into roughly equal pieces. Unlike most partitioners, which use only the combinatorial structure of the finite element graph, the Miller et al. algorithm exploits the geometric structure of the mesh as well. One of the main advantages of constructing Archimedes as a unified system is that the geometric information—the node coordinates and the node/element relationships—is readily available at all stages of the computation, including the partitioning and placement steps.

Archimedes also uses the geometry of the mesh to make the partitioner itself more efficient, by a technique called *geometric sampling*. The idea is to first choose a random sample of the input nodes, then solve the partitioning problem over the sample. Thus the problem size is reduced, while the underlying geometric structure of the mesh ensures the quality of the result. Geometric sampling allows us to do most of the work on sets of only a few hundred or a few thousand points, even if the mesh itself has millions of nodes.

The Miller et al. algorithm runs in randomized linear time. For an n-node mesh of bounded aspect ratio in d dimensions, it produces (with high probability) a separator of size $O(n^{1-1/d})$. This is the best possible guarantee, in a worst-case asymptotic sense. In practice, our program generates much better partitions than the theoretical worst-case bounds predict. The partitions are competitive with more expensive methods such as spectral bisection [20], which does not exploit the mesh geometry.

The partitioning algorithm divides the mesh into two pieces. We use the algorithm recursively to get as many pieces as processors. Figure 8 illustrates our metal plate partitioned into sixty-four pieces. Figure 9 illustrates the corresponding quotient graph G'.

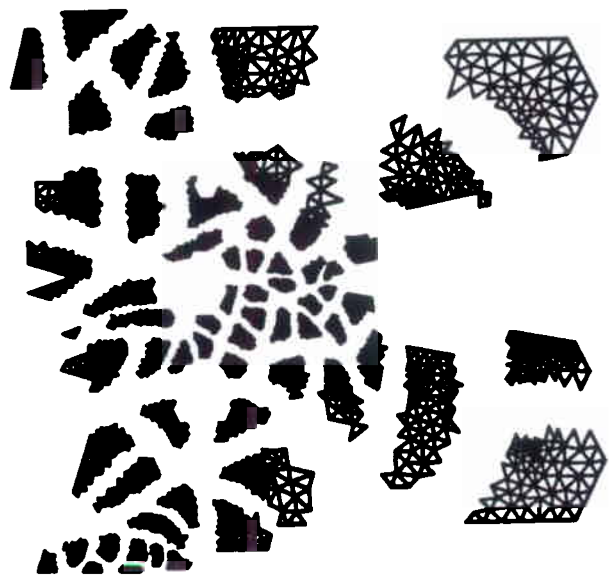

Figure 8: Partitioned metal plate for sixty-four processor machine.

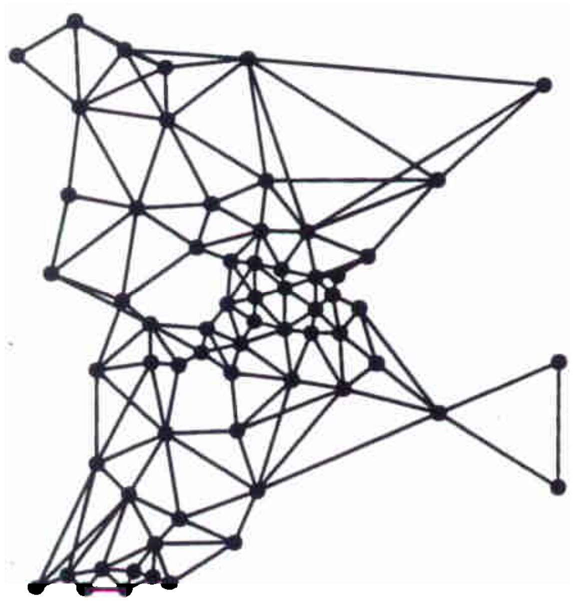

Figure 9: Sixty-four node quotient graph G' for partitioned metal plate.

Outline of the Partitioning Algorithm

The partitioning algorithm is described in greater detail by Miller et al. [19] and Gilbert et al. [8]. Archimedes' partitioner, which we present here, divides the elements among the processors; formally, it finds an edge separator for the geometric dual of G.

Algorithm (Partition a Mesh in $\mathbb{R}^d$)
Input: Mesh G in $\mathbb{R}^d$, with Φ the set of elements of G.

1. Choose a random sample S of constant size from the vertices of G;
2. Map S conformally onto the surface of the $d+1$-dimensional unit sphere, in such a way that every (d-dimensional) hyperplane through the origin partitions S approximately evenly;
3. Choose hyperplanes randomly until one is found which intersects relatively few edges of G. The intersection of this hyperplane with the unit sphere is a *great circle*;
4. Map this great circle back to $\mathbb{R}^d$ to obtain a d-dimensional circle that partitions G (as well as Φ) approximately evenly;
5. Partition Φ into two subsets, Φ_1 and Φ_2. Each element is placed into Φ_1 or Φ_2, depending on whether its center is mapped to the interior or the exterior of the circle in $\mathbb{R}^d$. From Φ_1 and Φ_2, form the subgraphs G_1 and G_2 of G. (If a vertex or edge is part of both an element in Φ_1 and an element in Φ_2, then that vertex or edge is in both subgraphs);
6. Partition G_1 and G_2 recursively.

The main computation is in Steps 2 and 3. Step 2 computes a map that "spreads" the input points (in $\mathbb{R}^d$) onto the unit sphere in $\mathbb{R}^{d+1}$ in such a way that every great circle divides the points approximately evenly. The key to this is a *center point* computation. Step 3 then finds a separator by choosing a great circle that not only separates the graph evenly, but also intersects only a small number of edges.

Center Points

The mapping in Step 2 from $\mathbb{R}^d$ to the unit sphere in $\mathbb{R}^{d+1}$ has three parts. First, the input points are projected stereographically onto the $d+1$-sphere. Second, a special point called a *center point* is found for the projected points. Third, the sphere is mapped conformally onto itself in a way that moves the center point to the origin.

A center point for a set of n input points in $\mathbb{R}^{d+1}$ is a point c such that every d-dimensional hyperplane through c has at most $d/(d+1)\cdot n$ points on each side. A center point exists for every set of input points, and can be found by linear programming. However, this method is much too slow to be practical; thus Archimedes uses a heuristic due to Miller and Teng (and analyzed by Clarkson et al. [6]) that finds extremely good approximate center points in linear time.

Geometric sampling comes into play in that the center point is based not on the entire input set, but only on a small randomly chosen subset S. The subset size depends on the dimension but not the size of the input mesh. In theory, the subset size is proportional to $d\log d$; in practice, we use several hundred points in two dimensions and about a thousand points in three dimensions.

Splitting Circles

In theory, a randomly chosen great circle on the unit $d+1$-sphere will divide the conformally mapped points about evenly while cutting only $O(n^{1-1/d})$ edges. In practice, not every great circle is a good circle. We therefore generate a number of great circles, searching for one that gives a balanced partition with few edges cut.

Our original method used circles chosen uniformly at random. Recently, however, Gremban [10] has suggested using moments of inertia of the conformally mapped points to bias the random choice. The idea is to take advantage of the fact that the mapped points are not uniformly distributed on the unit $d+1$ sphere. A circle in a (hyper)plane perpendicular to the principal axis of the moment of inertia of these points is likely to cut fewer edges than a different circle. Preliminary results indicate that this biased distribution can save a factor of 5 or 10 in the number of random circles examined.

Geometric sampling makes this step more efficient in two ways. First, we can filter candidate circles against the sample S and quickly reject those that do not split evenly enough. Second, with Gremban's inertial method, we can compute the moments of inertia based only on S.

Placement and Routing

The result of partitioning is the quotient graph G', which describes the communication pattern of the matrix-vector multiplication. G' should be embedded into the parallel architecture H such that the communication time is minimized. The notion of a good embedding thus depends on the target machine.

We consider two communication models. With *message passing*, all routing decisions are made by the message passing system, and these decisions are hidden from the application. We also consider a connection-based model called the *ConSet model.* The set of interprocessor connections we need is broken up into a small number of *phases*, only one of which is active at a time. A phase consists of a subset of connections which can simultaneously be held active by the available hardware resources; the application controls which phase is instantiated at any given time. All connections belonging to the active phase are themselves active. For a detailed discussion see Feldmann, Stricker, and Warfel [7].

The advantage of message passing is that no explicit routing need be done. The advantage of the ConSet model is that, because the structure of G' is known in advance, a *communication compiler* can choose routes and schedule communication so as to minimize network congestion.

We discuss below our target machines, the iWarp and the Connection Machine CM-5, and how to embed G' into them. Our heuristics are inspired by algorithms from VLSI gate array layout [17].

The iWarp system

Our first target machine is the iWarp [4]. The iWarp is a distributed memory parallel computer whose processors are connected by a two-dimensional torus. The iWarp component is a VLSI chip that contains a *processing agent* and a *communication agent.* At the heart of the communication agent is a limited set of communication resources called *queues.* These queues can be dynamically chained together to form *pathways*, which are point-to-point connections formed between pairs of processors using wormhole routing. Data traveling along a pathway passes from processor to processor automatically, with low latency, without disturbing the computations on intermediate processors. A pathway consumes a queue on each processor it *touches*. These ideas are illustrated in Figure 10.

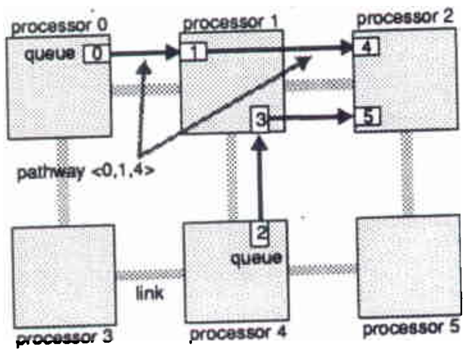

Figure 10: iWarp communication structures. Processor 0 can send data to processor 2 without interfering with the computation on processor 1.

Because the number of queues per processor is limited, in general not all desired connections can be routed simultaneously. One way to address this problem is to use a simple message passing system: a pathway is dynamically set up directly between the source and destination, the data is transmitted, and the pathway is taken down to free up queues to be used for other messages. We have also implemented ConSet communication on the iWarp, which requires explicit routing but offers better performance.

iWarp Placement and Routing

The characteristics of the iWarp affect the efficiency of embeddings. Dilation is of minor importance because of the low latency of the pathways. Congestion is an important measurement because all pathways sharing a communication bus are multiplexed over that link. In addition to these traditional measurements, we must address the constraint that a limited number of pathways can touch any particular processor at any particular point in time. This makes the *vertex congestion* of our embeddings, defined for each vertex in the target graph H as the total number of paths in H (that correspond to edges in G') that are incident to that vertex, of paramount importance.

For placement, as with partitioning, it is to our advantage to take into account the geometric information given in the original problem. We assign each node of G' a coordinate in two- or three-dimensional space by computing the center of mass of each subgraph of G. If our problem is three-dimensional, we project G' to two dimensions to match our processor topology. One simple approach to placing two-dimensional graphs on the iWarp is to simply halve the set of vertices repeatedly with alternating vertical and horizontal cuts, and map the vertices to the torus in the natural way. We further improve the placement with local hill-climbing search.

To route for the ConSet model, we must address two problems. First, each edge of G' must be assigned to one phase, during which its connection will be active. Second, each phase's set of connections must be routed with a fixed number of queues and with minimal congestion. We solve both problems together using sequential routing techniques based on weighted shortest path algorithms.

Our sequential routing algorithm is outlined below. It starts by trying to use a small number of phases and increases the number of phases until it is successful. All connections are sorted, from longest to shortest, by distance (in H) from source to destination. Each vertex of H (in each phase) is assigned an initial cost of one. We define the cost of a path between two vertices in H to be the sum of the cost of the vertices along the path. Connections are routed one by one, each along a shortest path from all possible phases. Each time a connection is successfully routed, the cost of each vertex along the path is increased. (The increase is nonlinear, and the cost of a vertex whose queues are all used is infinite.)

If all connections can be routed this way, we have found a solution, but not necessarily a good one, because the quality depends on the order in which connections are routed. Whether or not all connections are successfully routed, the communication compiler tries to improve the routing with a "ripup and reroute" stage, wherein routes are repeatedly selected to be ripped up, and the vertex costs are updated; then the connections are rerouted. During this stage it is often possible to find a route for a previously unroutable connection.

```
Algorithm (Route Connections on the iWarp)
Input: $G'$.
  1. $k =$ minimum possible number of phases, based on degree of $G'$;
  2. sort edges in $G'$ by connection length (longest to shortest);
  3. while not successful do
     (a) use $k$ tori (copies of $H$)
     (b) initialize cost of each vertex in each copy of $H$ to 1
     (c) for each edge in $G'$ do
             find shortest path in all $k$ tori
             update vertex costs in that torus
     (d) for $i = 0$ to some empirically chosen number do
             ripup and reroute a connection
     (e) $k := k + 1$
```

When routing is complete, a schedule is generated which describes a good order in which to exchange messages during each phase. The automatic scheduler is described by Feldmann et al. [7].

Figure 11 illustrates the quotient graph of Figure 9, placed and routed on a 64-processor iWarp by our algorithms. We have tried other heuristics for placement and routing, including simulated annealing and linear programming, but they have not performed as well.

Placement on the Connection Machine CM-5

Our next target machine is the Connection Machine CM-5 [24]. The CM-5 is a distributed memory parallel computer whose computational nodes are connected by a fat-tree network [16]. The network supports message-based, point-to-point communication, along with efficient primitives for synchronization, broadcast, and scan. The network topology, a fat-tree, is best suited for communication patterns with hierarchical locality. The network is organized such that groups of 4 processors have

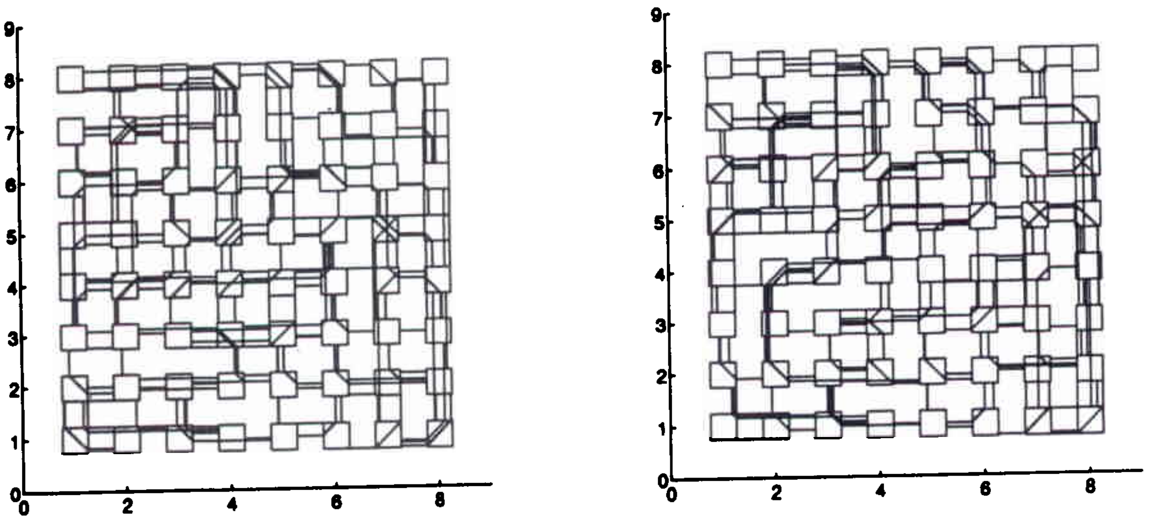

Figure 11: Interprocessor connections on a 64-processor iWarp for the quotient graph G'. Two phases are required.

the highest bandwidth among themselves, groups of 16 processors have a slightly lower bandwidth, and anything past a 16 processor boundary has yet a lower bandwidth.

This hierarchical locality of a fat-tree is well suited for our recursive partitioner as the separators give a hierarchical decomposition. While partitioning G, we derive a binary *cut tree* T, which is the recursion tree of the partitioning algorithm. The leaves of T represent the vertices of G'. We can collapse alternate levels of T to obtain a 4-ary tree whose structure matches that of the fat-tree network, and embed this tree into the fat-tree in the natural way. Because the network hardware takes care of the routing, we do not need to specify the embedding of the edges.

Code Generation

Language Abstractions

We illustrate Archimedes' language with several excerpts from code written to solve the heat equation.

Archimedes' language has several distributed data types in addition to the usual C data types. Distributed data types include vectors and sparse matrices.

Parallelism is expressed through the **FORNODE** and **FORELEM** loop constructs. These process each node, or each element, in parallel. No data dependences between loop iterations are allowed, except operations which combine the results with an associative, commutative operator. Consider this simplified Archimedes code, which calculates the value of the stiffness matrix needed to solve $-\nabla \cdot k\nabla u = f$ over a 3D region, where k and the unknown u are functions of the spatial coordinates x, y, and z:

```
MATRIX K;
NODEFLOAT boundary, u0, f;
```

```
float block[4][4];

MATRIXZERO(K);
NODEFLOATZERO(f);
FORELEM(i) {
  GAUSS_TETRAHEDRON(i, k_function, block);
  ASSEMBLEMATRIX(block, i, K);
}
FORNODE(i) {
  if (boundary[i] != 0.0) {
    DIRICHLET(u0[i], i, K, f);
  }
}
```

Here, **K** is declared to be a sparse matrix, and **boundary**, $\mathbf{u}_0$, and **f** are vectors. *block* is an array used to store the stiffness matrix of a single element. The identifier *k_function* denotes a user-supplied C function which computes an arbitrary function of x, y, and z.

The matrix **K** and vector **f** are first set to zero. Then, Archimedes iterates through the elements in parallel, and performs a numerical integration (by Gaussian quadrature) over the volume of each tetrahedral element. Each integration returns a 4×4 element stiffness matrix, which expresses the interactions between the corners of the tetrahedron. The values in this small matrix are *assembled* into the (large, sparse) stiffness matrix **K**. This assembly process should be thought of as a simple matrix addition, where the small matrix is padded to the size of the large matrix by the addition of rows and columns (filled with zeroes) for mesh nodes not in the tetrahedron.

Because the matrix assembly process involves only addition, the elements can be processed in any order. If a node is the corner of tetrahedra on several different processors, then the node is duplicated across processors, so the elements can be processed in parallel.

The final loop modifies the matrix to set boundary conditions. The vector *boundary* specifies which nodes are on the boundary of the region. At these points, we set the Dirichlet boundary condition $\mathbf{u} = \mathbf{u}_0$.

None of the code thus far requires any communication. The main communication primitive is the sparse matrix-vector product. This performs two functions: each processor performs a local matrix-vector product on its subgraph; then, processors that share nodes communicate with each other to ensure they have the correct value for each node. As we have explained, the communication may be divided into several phases.

For the simplest explicit methods, this is the only communication primitive we need; all other computation can be done locally. For iterative solvers, we also need reduction operators such as vector norm and dot product. Because many problems can be solved with only these two communication primitives, a simple version of Archimedes is easy to port.

Consider the inner loop of a simplified conjugate gradient code [9] to solve $\mathbf{Ku} = \mathbf{f}$.

```
NODEFLOAT(u, p, w, pw, residual);

while (norm > EPSILON * bnorm) {
  beta = norm / oldnorm;
  SET(p = residual + beta * p);
  MVPRODUCT(K, p, w);
  DOTPRODUCT(p, w, pw);
  alpha = norm / pw;
  SET(u = u + alpha * p);
  SET(residual = residual - alpha * w);
  oldnorm = norm;
  DOTPRODUCT(residual, residual, norm);
}
```

Here, the SET operation performs mixed scalar/vector arithmetic, without any need to communicate. The MVPRODUCT operation forms the product $\mathbf{w} = \mathbf{Kp}$, and the DOTPRODUCT operation forms the products $\mathbf{pw} = \mathbf{p} \cdot \mathbf{w}$ and $\mathbf{norm} = \mathbf{residual} \cdot \mathbf{residual}$, each requiring communication.

Proposed Optimizations

The use of a FEM-specific language opens the door for certain program transformations which improve the speed of the application. Here, we present some optimizations we plan to implement. Although they are not yet a part of the code generator, they are based on well-understood compiler design techniques, and will be simple to add.

The most important optimizations are those which reduce communication. If a system of partial differential equations is being solved, each iteration of the inner loop of the linear solver will perform several matrix-vector products. The communication for these can usually be performed all at once. Hence, the code generator breaks each MVPRODUCT into two stages — a local computation stage, and a communication stage. The communication stages are performed simultaneously after all computation stages have completed. On a message passing multiprocessor, we might improve communication time by combining them into a single message. With ConSet on iWarp, we need only instantiate each phase once. Either way, execution time is reduced.

This optimization takes a subtler form if we compute a vector of the form $\mathbf{u} = \mathbf{Ax} + \mathbf{By}$. In this case, we can delay the communication step until after the local matrix-vector products and the addition have been performed. Hence, only half as much data will be routed between processors.

Mesh Generation

Properties of Good Meshes

Meshes are often designed by conformally mapping a structured mesh (such as a square mesh) onto a region. This approach gives the user little leeway to choose how the density of a mesh varies, and is difficult to apply to complex geometries, such as airplane bodies. Unstructured mesh generators are more flexible, but difficult to design. A good survey of mesh triangulation methods is provided by Bern and Eppstein [3].

It is not difficult to automatically find a triangulation for an arbitrary two-dimensional region. It is trickier to find a tetrahedralization for an arbitrary three-dimensional region. The real difficulty, though, is that a mesh generator should avoid producing elements with large aspect ratios — "skinny" triangles and tetrahedra, so to speak — because angles close to 180° cause a large discretization error [1], and very small angles cause the stiffness matrix **K** to be ill-conditioned [5]. A good mesh generator will also minimize the number of nodes and elements in the mesh. Finally, it ideally should be able to refine previously generated meshes.

Archimedes' Mesh Generator

Archimedes includes a two-dimensional mesh generator with all of the above properties. The algorithm is due to Ruppert [21], and is built on the Delaunay triangulation primitives of Guibas and Stolfi [11]. Ruppert has proven that the algorithm can discretize two-dimensional regions with straight-line boundaries using triangles with angles no smaller than 20° (except where there are small input angles; these cannot be improved). This also has the effect of bounding the largest angle below 140°. In practice, the algorithm produces relatively few nodes, as Figure 4 illustrates.

Ruppert's algorithm starts by constructing a *Delaunay triangulation* of the input nodes. A Delaunay triangulation is a triangulation which has the property that its smallest angle is maximized (compared to any other triangulation of the same nodes). To improve the mesh further, additional nodes are added one by one, and the triangulation is incrementally updated each time.

Nodes are added for two reasons. If the triangulation does not conform to the input boundaries, nodes are added to force it to do so. If a triangle has an angle smaller than 20°, a node is added at the triangle's circumcenter. This changes the triangulation so that the skinny triangle is no longer present.

Nodes can be added for a third reason: to refine triangles which are too large. Figure 6 was generated from Figure 4 as follows: Archimedes solved a stress simulation on the original mesh, and generated *a posteriori* error estimates on each element of the mesh. Based on the error estimates, it determined a maximum desired triangle area on each element. Archimedes' mesh generator refined the existing mesh so that no triangle has area greater than the maximum allowed.

There are no known satisfactory algorithms for three dimensional mesh generation on arbitrary regions. Archimedes has only rudimentary 3-D mesh generation; we are actively studying this area.

Matrix-Vector Product Performance

We tested Archimedes on a three-dimensional 22,000 node mesh on a 64 processor iWarp machine. With a highly optimized kernel, each iWarp processor can perform a local symmetric sparse matrix-vector multiply at 5.22 MFLOPS. Of the time required to perform a global multiplication, 20% of the total time is spent communicating, so the net speed per processor is 4.15 MFLOPS. The communication time is significant; this is the reason for our careful efforts in partitioning, placement, and routing.

[Note to referees: we have run the same test on a CM-5, and achieved 7.75 MFLOPS on one processor with one vector unit. Our communication measurements are still tentative, and we are not yet permitted to publish them. We will include them in the final draft.]

Future Directions

The code generator's repertoire could be expanded in many directions. We plan to consider various parallel linear solvers [12]. We will pay special attention to multigrid methods [18], domain decomposition methods [14] related to nested dissection [15], and methods of preconditioning that are based on the geometry of the mesh. We are also investigating the feasibility of adding primitives for unstructured finite differences and boundary element methods.

Another of our goals is to parallelize as much of Archimedes as is practical. It should be straightforward to parallelize mesh partitioning. With more effort, it may also be possible to parallelize mesh generation.

Our greatest impediment to further work is the difficulty of generating high-quality three dimensional meshes for arbitrary geometries. It is also important to have an easy way to discretize regions with curved boundaries.

Our most ambitious goal is adaptive time-dependent finite element methods. Suppose that we wish to refine our mesh every few time steps to keep up with changing conditions. It is too slow to generate and partition a new mesh each time. We need to develop algorithms which can efficiently modify an unstructured mesh during run-time. Several serial algorithms have been developed [25], but we know of no parallel ones.

References

[1] Babuška, I., and Aziz, A. On the angle condition in the finite element method. *SIAM J. Numer. Analysis 13*, 2 (1976), 214–227.

[2] Becker, E.B., Carey, G.F., and Oden, J.T. *Finite Elements: An Introduction.* Prentice-Hall, 1981.

[3] Bern, M., and Eppstein, D. Mesh Generation and Optimal Triangulation. In *Computing in Euclidean Geometry.* Du, D.-Z., and Hwang, F.K., eds. World Scientific Publishing Company, 1992.

[4] Borkar, S., Cox, G., Kung, H.T., Levine, M., Moore, W., Susman, J., Urbanski, J., Cohn, R., Gross, T., Moore, B., Peterson, C., Sutton, J., and Webb, J. Supporting Systolic and Memory Communication in iWarp. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990.

[5] Carey, G.F., and Oden, J.T. *Finite Elements: Computational Aspects.* Prentice-Hall, 1984.

[6] Clarkson, K., Eppstein, D., Miller, G.L., Sturtivant, C., and Teng, S.-H. Approximating Center Points with Iterated Radon Points. In *Proceedings of 9th ACM Symposium on Computational Geometry*, 1993.

[7] Feldmann, A., Stricker, T.M., and Warfel, T.E. Supporting Sets of Arbitrary Connections on iWarp through Communication Context Switches. In *Proceedings of the 1993 ACM Symposium on Parallel Algorithms and Architectures*, 1993.

[8] Gilbert, J.R., Miller, G.L., and Teng, S.-H. A Geometric Approach to Mesh Partitioning: Implementation and Experiments. Technical Report, Xerox Palo Alto Research Center, to appear.

[9] Golub, G.H., and Van Loan, C.F. *Matrix Computations.* The Johns Hopkins University Press, 1989.

[10] Gremban, K.D., Miller, G.L., and Teng, S.-H. Moment of Inertia and Graph Separators. Manuscript, CMU and MIT.

[11] Guibas, L., and Stolfi, J. Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams. *ACM Transactions on Graphics 4*, 2 (1985), 74–123.

[12] Heath, M.T., Ng, E., and Peyton, B.W. Parallel Algorithms for Sparse Linear Systems. In *Parallel Algorithms for Matrix Computations.* SIAM, 1990.

[13] Houstis, E.N., Rice, J.R., Chrisochoides, N.P., Karathanasis, H.C., Papachiou, P.N., Samartzis, M.K., Vavalis, E.A., Wang, K.Y., and Weerawarana, S. //ELLPACK: A Numerical Simulation Programming Environment for Parallel MIMD Machines. In *Proceedings of 1990 International Conference on Supercomputing.*

[14] Keyes, D.E., and Gropp, W.D. A Comparison of Domain Decomposition Techniques for Elliptic Partial Differential Equations and their Parallel Implementation. *SIAM Journal on Scientific and Statistical Computing 8*, 2 (1987), s166–s202.

[15] Khaira, M.S., Miller, G.L., and Sheffler, T.J. Nested Dissection: A Survey and Comparison of Various Nested Dissection Algorithms. Technical Report CMU-CS-92-106R, Carnegie Mellon University, 1992.

[16] Leiserson, C.E., Abuhamdeh, Z., Douglas, D., Feynmann, C., Ganmukhi, M., Hill, J., Hillis, W., Kuszmaul, B., St. Pierre, M., Wells, D., Wong, M., Yang, S., and Zak, R. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1992.

[17] Lengauer, T. *Combinatorial Algorithms for Circuit Layout.* John Wiley & Sons, 1990.

[18] McBryan, O.A., Frederickson, P.O., Linden, J., Schuller, A., Solchenbach, K., Stuben, K., Thole, C.A., and Trottenberg, U. Multigrid Methods on Parallel Computers — A Survey of Recent Developments. *Impact of Computing in Science and Engineering 3*, 1 (1991), 1–75.

[19] Miller, G.L., Teng, S.-H., Thurston, W., and Vavasis, S.A. Automatic Mesh Partitioning. In *Graph Theory and Sparse Matrix Computation.* George, A., Gilbert, J.R., and Liu, J.W.H., eds. Springer-Verlag, 1993.

[20] Pothen, A., Simon, H.D., and Liu, K.-P. Partitioning Sparse Matrices with Eigenvectors of Graphs. *SIAM Journal on Matrix Analysis and Applications 11*, 3 (1990), 430–452.

[21] Ruppert, J. A New and Simple Algorithm for Quality 2-Dimensional Mesh Generation. Technical Report UCB/CSD 92/694, University of California at Berkeley, 1992.

[22] Strang, G., and Fix, G. *An Analysis of the Finite Element Method.* Prentice-Hall, 1973.

[23] Sussman, A., Saltz, J., Das, R., Gupta, S., Mavriplis, D., Ponnusamy, R., and Crowley, K. PARTI Primitives for Unstructured and Block Structured Problems. *Computing Systems in Engineering 3*, 1–4 (1992), 73–86.

[24] Thinking Machines Corporation. *The Connection Machine CM-5 Technical Summary*, 1991.

[25] Zeng, L.F., and Wiberg, N.-E. Spatial Mesh Adaptation in Semidiscrete Finite Element Analysis of Linear Elastodynamic Problems. *Computational Mechanics 9* (1992), 315–332.