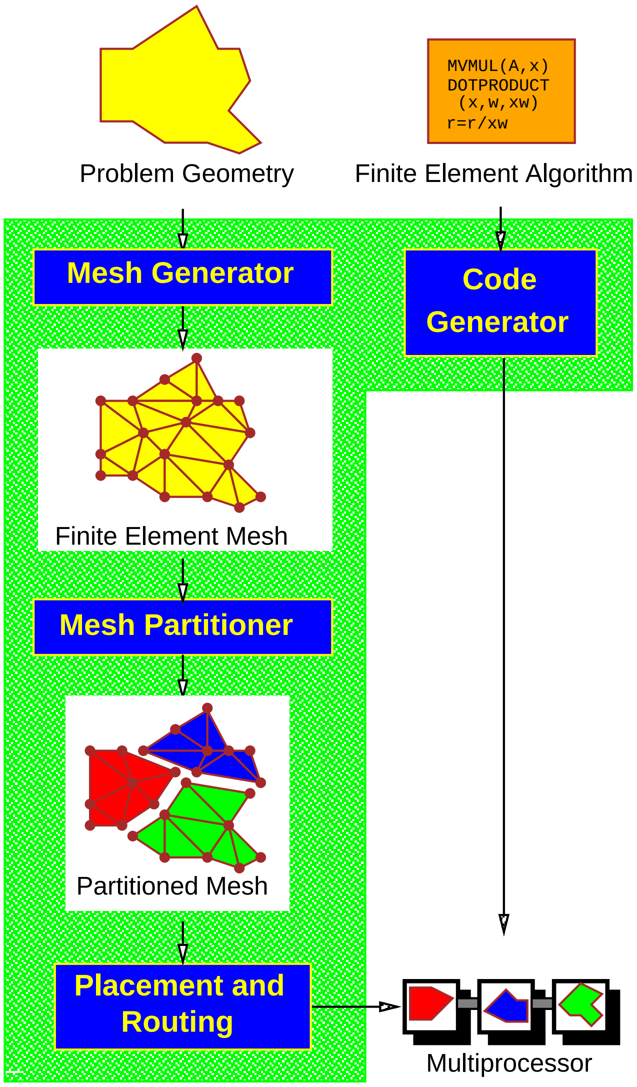

Figure 8: Structure of Archimedes.

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. Once a mesh is provided, Archimedes performs the partitioning, placement, and routing steps as necessary for a given architecture, using algorithms that will be described in the next few sections.

Along with the problem geometry, the user writes a sequential 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.

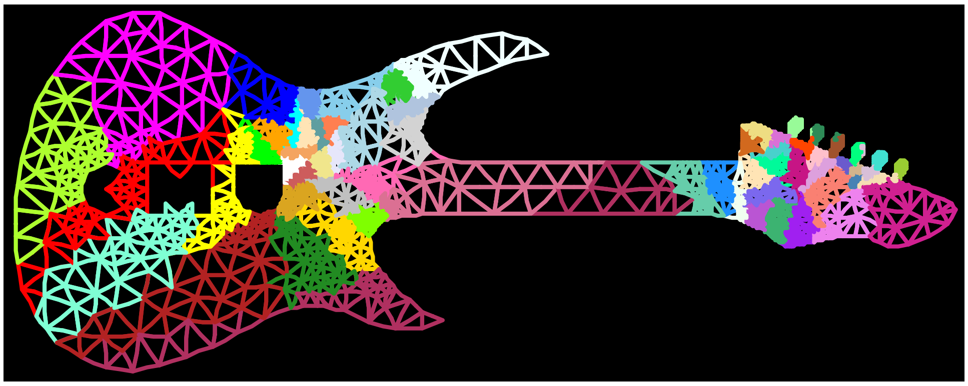

Figure 10: Partitioned guitar for sixty-four processor machine.

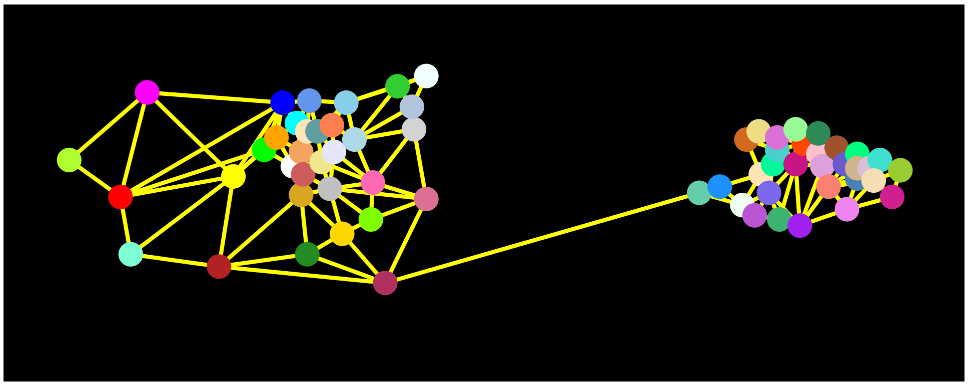

Figure 11: Sixty-four vertex quotient graph E' for partitioned guitar.

Partitioning

The two most common types of partitioning algorithm in the literature are known as *recursive geometric bisection* (RGB) and *recursive spectral bisection* (RSB). Both methods bisect a graph into two equal pieces, while attempting to minimize the size of the cut. Both methods are "recursive" because they first bisect a graph, then bisect each of the two resulting pieces, and so on; they therefore yield a number of subdomains that is a power of two.

RSB [15] bisects a graph by exploiting spectral properties of its combinatorial structure. However, RSB does not take into account the positions of the graph's vertices in space. While RSB usually produces excellent partitions in practice, it tends to be slow, as it relies on expensive eigenvector calculations. RGB uses the spatial coordinates of the vertices rather than the combinatorial structure of the graph, and typically executes more quickly. 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' partitioner is an advanced recursive geometric bisection algorithm due to Miller, Teng, Thurston, and Vavasis [14], which finds provably good separators for a large class of geometrically defined graphs. A *separator* is a small set of edges whose removal divides the graph into roughly equal pieces. For an n-vertex graph, it produces (with high probability) a separator of size $O(\sqrt{n})$ in two dimensions, or

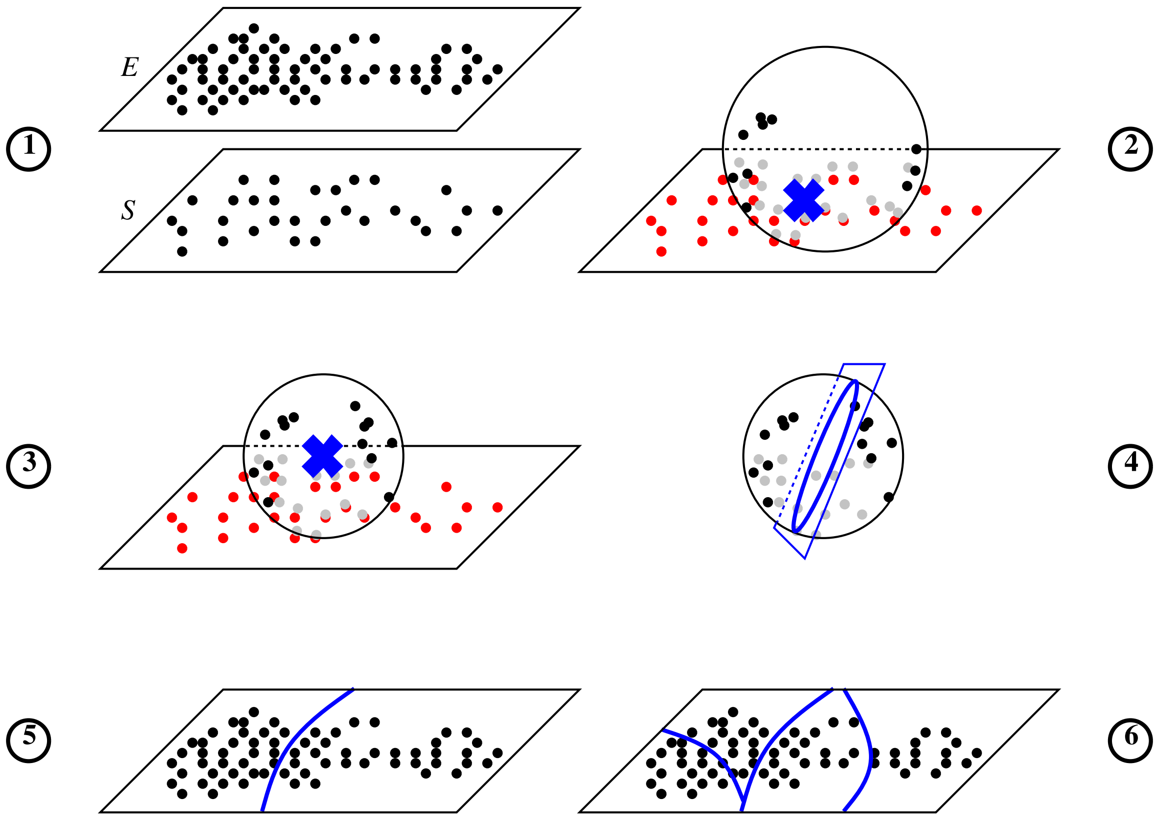

Figure 13: Illustration of partitioning algorithm. 1. A random sample S is chosen. 2. Points are projected to the surface of a ball. The large X indicates the center point of the projected points. 3. The projection is modified to move the center point to the center of the ball. 4. A random hyperplane splits the sample. The intersection of the plane and the surface of the ball form a great circle (bold). 5. The great circle is mapped to a circle in the plane. (Only a small section of the circle is shown here.) 6. Subdomains are bisected recursively.

6. Partition E_1 and E_2 recursively.

Two aspects of the algorithm deserve deeper examination: center points and the random choice of hyperplanes.

By definition, a center point for a set of n input points in $d+1$ dimensions is a point c such that every d-dimensional hyperplane through c has at most $n \cdot d/(d+1)$ points on each side. A center point exists for any set of points, and can be found by linear programming. However, this method is much too slow to be practical; thus Archimedes uses a heuristic that finds good approximate center points in linear time. Clarkson et al. [8] show that the center point calculation is a good approximation for E even though it is based on the small random sample S. The sample 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.

In theory, a randomly chosen hyperplane passing through the center of the ball has a 50% or better chance of cutting only a small number of edges ($O(\sqrt{n})$ in two dimensions, $O(n^{2/3})$ in three). In practice, not

Figure 14: iWarp communication structures. Pathways are constructed by linking together queues. Processor 0 can send data to processor 2 without interfering with the computation on processor 1.

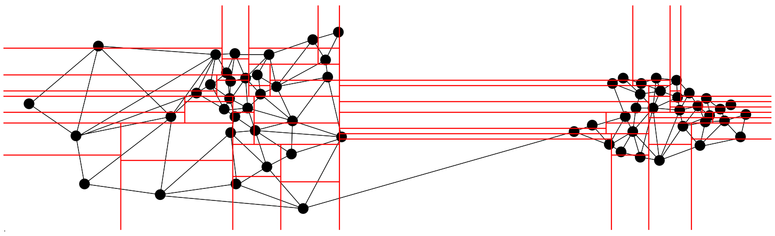

Figure 15: A succession of horizontal and vertical cuts defines the placement of the quotient graph on an 8×8 grid.

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. We shall address routing for ConSet in detail.

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 measure 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 connections routed through (or into) 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 E' a coordinate in two- or three-dimensional space by computing the center of mass of each subdomain (as illustrated in Figure 11). If our problem is three-dimensional, we project E' to two dimensions to match our processor topology. One simple approach to placing a two-dimensional graph on the iWarp, or any two-dimensional mesh topology, is to simply halve the set of vertices repeatedly with alternating vertical and horizontal cuts (Figure 15), and map the vertices to the torus in the natural way. We further improve the placement with local hill-climbing search: we try swapping pairs of subdomains, and accept the swap if it reduces the sum of all connection lengths.

To route for the ConSet model, we must address two problems. First, each edge of E' 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 per processor and with minimal congestion. We solve both problems together using sequential routing techniques based on weighted shortest path algorithms.

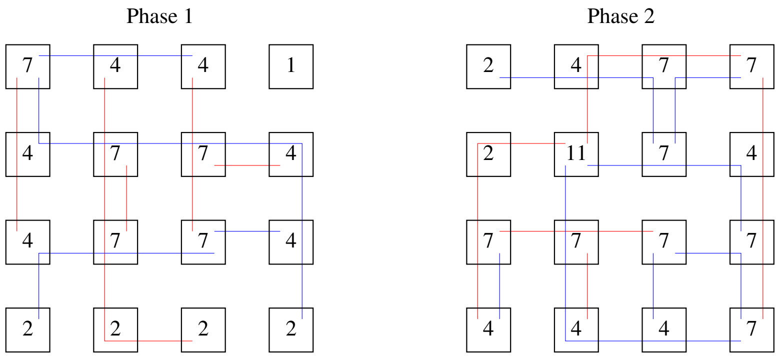

Figure 16: Connections are routed by finding the path of least cost from among the communication phases. The cost of each processor is shown; the cost of a processor increases nonlinearly with the number of connections through that processor.

Our sequential routing algorithm begins by sorting all connections, 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 (Figure 16). 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, often along less congested routes. During this stage it is often possible to find a route for a previously unroutable connection.

The question remains, how do we know how many communication phases are required? The routing algorithm is fast enough that we simply begin by trying to route with one phase; if routing is not successful, the number of phases is increased until the algorithm succeeds.

When routing is complete, a schedule is generated that describes a good order in which to exchange messages during each phase. The placement, routing, and scheduling heuristics are all described in detail by Feldmann et al. [9].

Figure 17 illustrates the quotient graph of Figure 11, 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

Another target machine is the Connection Machine CM-5 [20]. The CM-5 is a distributed memory parallel computer whose computational nodes are connected by a fat-tree network [12]. The network supports message passing communication, as well as efficient primitives for synchronization, broadcast, and scan.