A Separator-Based Framework for Automated Partitioning and Mapping of Parallel Algorithms for Numerical Solution of PDEs

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

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

[3]Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304

Abstract

This paper is a report on ongoing work in developing automated systems for the partitioning, placement, and routing of data that is necessary for the efficient parallel solution of large problems in scientific computing, specifically the numerical solution of partial differential equations. Many of these problems have as an iterated inner loop the formation of the product of a large sparse matrix and a vector of variables. This problem of sparse matrix-vector multiplication has an underlying combinatorial graph structure that can be exploited. Using geometric information from the original problem, we can partition this combinatorial graph using provably good two- or three-dimensional graph separators (depending on the dimension of the problem). The resulting partitions into subproblems have good load balancing properties and a relatively small amount of communication between subproblems. In order to develop effective heuristics for the placement of these subproblems on the available processors and the routing of messages between them, we must also carefully consider the characteristics of the target architectures. The first parallel machine we are considering is the iWarp system. The novel communication mechanism of the iWarp system allows us to draw an analogy between our placement and routing problem and certain area minimization problems in the field of VLSI circuit layout, giving us an additional collection of insights and heuristics which can be brought to bear on our problem.

*This research was supported in part by the Defense Advanced Research Projects Agency (DOD) monitored by DARPA/CMO under Contract MDA972–90–C–0035, the Air Force Office of Scientific Research under Contract F49620–92–J–0131, the Natural Sciences and Engineering Research Council of Canada under a 1967 Science and Engineering Scholarship, and the National Science Foundation under Grants CCR–9016641 and CCR–8713489.

1 Introduction

There is widespread interest in the engineering and scientific communities in using parallel computers to solve larger problems than were previously feasible. Many problems in computational science and engineering can be modeled by partial differential equations. Numerical approximation of these equations over the spatial domain of interest by finite difference, finite element, or finite volume methods gives rise to algebraic systems with large, sparse coefficient matrices. Often, however, the nonzero structure is irregular, as in problems with geometric complexity or differing scales of resolution (typical of problems in solid and fluid mechanics). In these cases, achieving maximum performance on distributed memory parallel computers (which generally are limited in both memory and communication resources) requires careful partitioning of problems along with efficient placement of and routing between the resulting subproblems. Such domain-based problems have an underlying combinatorial graph with useful properties derived from its geometric structure, among them the existence of small separators which can be used to partition the problem. However, both the placement of subproblems on the processors of a parallel computer and the routing between them require detailed knowledge of the target machine if maximum performance is to be obtained.

Multiplying a sparse matrix by a vector is a frequently-used inner loop in many PDE solvers, such as Krylov subspace methods (e.g. conjugate gradients) and other iterative methods. There is a natural combinatorial graph representation of this problem which reduces the communication requirements of this multiplication to exchanging data across all edges of the graph. The unstructured meshes associated with finite element calculations have underlying graphs with embeddings in two or three dimensions already defined, which will in general have small separators that may be used recursively to partition these graphs into as many subgraphs as desired. The separator properties of the underlying graph in two or three dimensions will give bounds on the load balancing characteristics and communication requirements of the resulting partition.

As was stated earlier, effective performance in placement and routing requires careful attention to the properties of the target machines. Our first target is the iWarp system at Carnegie Mellon [3, 4], whose novel communication mechanism adds some interesting elements to the problem of placement and routing. Its communication mechanism, which uses circuit-switched logical pathways multiplexed over each physical channel, changes the measures we must consider when choosing a set of routing paths. In particular, the maximum congestion of a set of routing paths is of far greater importance than its maximum dilation. This allows us to draw a strong analogy between the problem of placement and routing and established work in the area of circuit layout [6, 9], and bring insights and techniques from this area to bear on our problem. In addition, geometric information from the original problem is helpful in the placement of subproblems. Preliminary experiments suggest that even relatively simple heuristic approaches to placement and routing for these graphs that take advantage of their geometric structure perform nearly twice as well as the current placement and routing packages on iWarp [12] in terms of the quantity of routing resources required to route a given underlying graph. We are also planning an implementation on the Connection Machine CM-5 [1, 8].

2 A Combinatorial Graph Representation of Sparse Matrix-Vector Multiplication

The problem of multiplying a large sparse matrix by a vector is ubiquitous in scientific computation. Together with inner products and elementwise vector operations (e.g., addition, multiplication by scalars), it is one of the basic primitives used in iterative numerical methods, and is among the most communication-intensive. For example, the basic iterative step of the conjugate gradient method for finite element calculations consists of several inner-product calculations followed by one multiplication of a sparse matrix by a vector (and some simpler elementwise operations). It is important to note at this point that although the nonzero entries of the matrix may change for nonlinear problems, the nonzero structure of the matrix is fixed.

We now consider the communication requirements for calculating the product y of a sparse $n \times n$ matrix A and an n-element vector x. Assume that there are n processors, and each processor i stores a single element x_i of the vector. In order to calculate y_i, processor i must receive the values x_j from each processor j such that $A_{ij} \neq 0$. The value x_j must be multiplied by A_{ij}, which can be thought of as an edge weight, at either the source or the destination. Thus the set of routing paths which we must implement on the network of processors includes an edge from processor j to processor i whenever $A_{ij} \neq 0$. Since in our application areas the nonzero structure of A is typically symmetric (even if A is asymmetric), this set of routing paths is just the undirected graph on n vertices whose adjacency matrix is A. We will refer to this graph as G. This reduces the problem of calculating the product of a sparse matrix and a vector to that of exchanging data across all edges of G.

Given G, the problem of performing an inner product

$x \cdot y$ is relatively simple, because each processor i holds the values of both x_i and y_i. All that is required is for the ith processor to calculate $x_i y_i$, and then for the resulting values to be accumulated over the n processors using a spanning tree in G. Elementwise operations on the vectors require no communication, but distribution of scalars for scalar-vector multiplication will require a broadcast, which can use the same spanning tree. These two primitives, data exchange and accumulation/broadcast, are sufficient to implement most iterative methods. Since the data exchange graph contains the spanning tree for the accumulation/broadcast operation, we need only consider the implementation of the communication needed for the sparse matrix-vector multiplication.

3 Graph Partitioning Using Separators

3.1 Partitioning of Sparse Matrix-Vector Multiplication

In general, the sparse matrix-vector multiplications used in scientific computing can have millions of variables, which is far more than the number of processors in current parallel machines. Therefore we partition the problem into subproblems and map the subproblems to the available processors, while allowing for the necessary communication between subproblems.

The most natural way to partition a graph G is to remove some subset of its edges, leaving some connected components of vertices remaining. These connected components are assigned to processors. This is called *vertex-partitioning*, because each vertex is assigned to a single processor.

Current finite element implementations suggest a different partitioning approach. A finite element mesh (such as in Figure 2) is a partitioning of space into elements. An element is a polygon (usually a triangle or rectangle) in two dimensions, or a polytope (usually a tetrahedron or rectangular prism) in three dimensions. The mesh is made up of nodes connected by edges in two dimensions, or by faces in three dimensions. Physical quantities are associated with each node; for instance, the vector x, whose length n is the number of nodes in the mesh, might specify the displacement of each node. Nodes within a single element have a direct physical effect on each other; thus, A_{ij} is nonzero if and only if nodes i and j are in some common element e.

Given a finite element mesh M, a corresponding graph G is generated by creating one vertex in G for each node in M, and adding an edge to G for each pair of nodes which share a common element. Thus, if M is composed of triangular elements, G will be identical to M. If M is composed of square elements, G will look similar to M, but with the addition of edges between opposite corners of squares.

It is possible to partition G by cutting between mesh elements and storing a copy of each vertex along the cut in each bordering block of the partition. This is called *element-partitioning*, because each element is assigned to a single processor, whereas some vertices are assigned to several processors.

In order to write an efficient implementation there are two measures we must keep in mind. First, we must keep the sizes of the partition blocks relatively balanced — note that for this problem, the amount of local work done on a processor is the number of edges contained in its partition block, not the number of vertices. Second, we wish to keep the boundaries of the partition blocks as small as possible, as this corresponds to the amount of interprocessor communication we must perform. In the next subsection we will discuss how graph separators allow us to address both of these concerns.

If we have partitioned G into p blocks, where p is the number of available processors, then we can define the *p-vertex minor* G' derived from G and the partition. G' is a weighted graph with a supervertex for each block of the partition, and an edge between two supervertices if their corresponding blocks need to communicate with each other (i.e., they either have a cut edge between them or contain copies of the same vertex, depending on the partitioning approach). The weight on an edge of G' is the number of values which must be communicated between those two supervertices in order to perform a multiplication. The weighted minor G' is what we must place and route on the target machine.

3.2 Implementation of Separators in Two and Three Dimensions

Miller, Teng, Thurston and Vavasis [10] give provably good separators in two or three dimensions for the sort of combinatorial graphs generated by these problems. These separators can be used recursively to obtain a partition of the graph with both good balance among block sizes (corresponding to processor load) and a relatively small number of cut edges or duplicated vertices (corresponding to data which must be routed), depending on the method of partitioning.

We have implemented the random linear time separator algorithm of Miller et al. [10] in MATLAB. The input to the algorithm is a pair (G, xyz), representing a graph embedded in two or three dimensions. G is the graph we derive from the finite element mesh M, and xyz is the coordinates of the vertices of G (taken from M).

A detailed description of the graph partitioning algorithm can be found in [11]. The following is an outline of the basic steps.

Algorithm (Vertex-Partition a Graph in $\mathbb{R}^d$)
Input: G, xyz in $\mathbb{R}^d$.

1. Choose a random sample S of constant size[a] from xyz;
2. Map S conformally onto the unit sphere in $\mathbb{R}^{d+1}$ in such a way that every hyperplane through the origin partitions S approximately evenly.
3. Choose hyperplanes randomly until a good split is found;
4. Map the great circle induced by the hyperplane back to $\mathbb{R}^d$ to obtain a sphere that partitions G (as well as xyz) approximately evenly;
5. The edges that cut this sphere form an edge separator. This set of edges can be found using the structure of G; (If a vertex separator is desired, it can be computed from the edge separator above. The structure of G can be used to optimize the separator size.)
6. Partition G into two subgraphs, G_1 and G_2. Each vertex is placed into G_1 or G_2, depending on whether it is mapped to the interior of the exterior of the sphere in $\mathbb{R}^d$;
7. Recursively partition G_1 and G_2.

[a] See [11] for the theoretical justification. According to our experiments, 400 to 800 points work well in two dimensions, and 600 to 1100 points work well in three dimensions.

Element-partitioning can be done similarly.Each element is placed into G_1 or G_2 depending on, for instance, where the center of the element falls.

Two important aspects of the partitioning algorithm make it suitable for efficient practical implementation:

- *Geometric Sampling*: Graphs derived from large-scale computational problems may have millions of vertices. Geometric sampling is a technique that reduces the problem size and simultaneously guarantees a provably good approximation of the larger problem. The basic idea is to first choose a random sample of the input, then solve the partitioning problem over the sample. The underlying geometric structure of the mesh ensures the quality of the partition.

- *Exploiting both Geometric and Combinatorial Structures*: The geometric structure of the mesh not only ensures the quality of the approximation, but also enables efficient implementation of some key subroutines. One of the goals of the algorithm is to find a balanced partition while minimizing the cost of the cut. The geometry of the mesh is used to map the problem into a canonical form so that the randomized selection can be performed efficiently. Then the combinatorial structure of the graph is used to derive and optimize the final solution.

We have experimented with various examples, and the numerical results are encouraging. With the help of some heuristics to speed up the geometric transformation and the local optimization, the program is fast (even in the MATLAB environment, which is interpretive and lacks code optimization). In practice, our program generates partitions much better than the theoretical results predict. The partitions are competitive with such previous methods as those based on an expensive eigenvector computation [13].

4 Parallel Implementation of Matrix-Vector Product

Here, we consider how a matrix-vector product $y = Ax$ might be implemented for a vertex-partitioned graph and an element-partitioned graph. We assume that A, x, and y are distributed over the processors according to the partition of G. Recall that G represents the sparsity structure of A. We will show that element-partitioning has distinct advantages.

After we have partitioned G into p blocks, let G_c be the subgraph of G corresponding to the block that is mapped to processor c by the placement algorithm (see Section 5). Let G_a and G_b be two such graphs, whose corresponding supervertices in G' are adjacent.

4.1 Vertex-Partitioned Problems

Suppose G has been vertex-partitioned. Then, for each vertex u in G_a, processor a stores the value x_u and the nonzero entries of row u of A. In order to calculate y_u, processor a must have the entry of x for each vertex adjacent to u in G. In other words, processor a must allocate storage for every vertex adjacent to G_a in G, as well as for every vertex in G_a. Thus, processors a and b share two "layers" of vertices at the interface between G_a and G_b.

The multiplication is performed in two phases. The first phase is a communication phase in which every processor receives needed entries of x from its neighbors in graph G', so that each processor has up-to-date entries for every vertex it uses. The second phase is a computation phase in which each processor p calculates y_u for each vertex u in G_p.

4.2 Element-Partitioned Problems

Now, suppose instead that G has been element-partitioned. Then, for each vertex u in G_a, processor a stores the entry x_u. For each edge (u, v) in G_a, processor a stores the value A_{uv} if a is the only processor storing both u and v. Otherwise, the value A_{uv} is split between several processors.

To understand how the matrix A is stored, it is necessary to have some understanding of how the sparse matrices used in finite element methods are generated. Typically, a separate calculation is made for each element. Any element whose vertices include both u and v will make an additive contribution to A_{uv}. The process of summing the contributions of individual elements to determine the value of A is called *assembly*.

Figure 1 is an example of element-partitioning. Suppose that the vertices s and t are each in both G_a and G_b (remember that in an element partition, vertices on the boundary are contained in both subgraphs). Suppose that edge (s, t) is part of two elements: e in G_a and element e' in G_b. Then, the contribution of e to A_{st} is stored on processor a, and the contribution of e' to A_{st} is stored on processor b. Let ${}^pA_{st}$ denote the contribution to A_{st} stored on processor p. Then, $A_{st} = \sum_p {}^pA_{st}$. We say that ${}^pA$ is *partially assembled*, because it has been assembled everywhere except at processor boundaries. Clearly, we could *fully assemble* A by communicating the shared values between processors and summing; but as we shall see, a fully assembled A would be less useful.

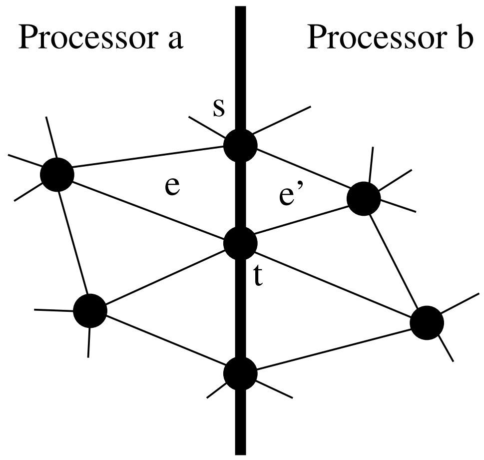

Figure 1: An element-partitioned mesh.

Again, the multiplication is performed in two phases; however, the computation phase is first. Each processor p calculates a local matrix-vector product of the form ${}^py = {}^pAx$.

The communication phase follows. For each vertex u shared by two or more processors, these processors exchange their respective entries for ${}^py_u$, and take the sum. This gives us correct values because $y = Ax = (\sum_p {}^pA)x = \sum_p {}^pAx = \sum_p {}^py$. We see that vectors, like matrices, have both partially assembled and fully assembled forms; and the product of a partially assembled matrix and a fully assembled vector is a partially assembled vector. (There is no similar property for fully assembled matrices.)

4.3 Comparison of Partitioning Approaches

When using the element-partitioning approach, processors a and b share only one "layer" of vertices at the interface between G_a and G_b, so less storage space is required for a vector than when using the vertex-partitioning approach. The sparse matrix A, which dominates memory usage, requires less storage in the element-partitioning approach if A is symmetric, and requires less storage in the vertex-partitioning approach if A is asymmetric (for reasons we omit here). The element-partitioning approach makes initial matrix generation much easier, because A is built element-by-element, so no communication is required.

So far, these are small differences. There are two far greater advantages to the element-partitioning approach. First, in nonlinear problems where the matrix A changes over time, A can be recomputed without communication.

Second, the element-partitioning approach allows assembly of a vector to be delayed, and thus may reduce the amount of communication which must be done. For instance, suppose one wishes to calculate $z = Ax + By$, where A and B are sparse matrices with the same sparsity structure. (This arises in the solution of second-order differential equations.) The products Ax and By each produce a partially assembled vector. These two partially assembled vectors can be summed locally to form a third partially assembled vector; this vector, if necessary, can be assembled to form the result z. Note that only one assembly, and thus only one communication step, is needed for two matrix-vector products. This is called the *delayed assembly optimization*.

Because of the communication reduction and speed advantages of matrix generation and the delayed assembly optimization, we strongly prefer the element-partitioning approach.

4.4 Code Generation Issues

Our goal is a code generator for finite element methods, that takes as its input the mesh M and a machine-independent description of a finite element algorithm, and automatically partitions the problem and generates multiprocessor code to execute the algorithm.

A code generator can handle the delayed assembly optimization in two ways. One is to determine at compile time when vector assembly should take place. This requires data dependence analysis, and cases will arise where it is not applicable because the behavior of an algorithm is not always predictable. The other solution is to defer all assembly decisions until runtime. When an operation requires a fully assembled vector as input, it checks the vector provided and calls an assembly subroutine if necessary. However, this creates another problem: inner products and elementwise multiplication require that at least one of their operands is fully assembled. If two partially assembled vectors are provided, one must be chosen arbitrarily for assembly, with no knowledge of which would be the optimal choice. We have not yet surveyed enough different finite element problems to resolve this issue.

We have not yet implemented a code generator. We have, however, implemented code for the iWarp multiprocessor to solve a second-order time-varying equation used for earthquake simulation, with encouraging results.

5 Placement and Routing

5.1 Background

Let H be a graph which represents the interconnection network of the target machine. Once we have partitioned G into subgraphs that correspond to subproblems to be solved locally, and identified which subproblems will have to communicate and how much information they will have to send to each other, we must embed the resulting p-vertex weighted graph G' into H. To implement the exchange of data across each edge of G, we must communicate between each pair of adjacent vertices in the p-vertex graph G' an amount of data proportional to the weight of the edge between them.

For our purposes, an *embedding* of a weighted graph $\mathcal{G}$ into a graph $\mathcal{H}$ (as distinguished from an embedding of a graph in two- or three-dimensional space) is a mapping of the vertices of $\mathcal{G}$ to the vertices of $\mathcal{H}$, corresponding to an assignment of subproblems to processors, together with a mapping of the edges of $\mathcal{G}$ to paths in $\mathcal{H}$, corresponding to routing paths between processors which need to communicate. We are only considering one-to-one vertex mappings because our first target machine does not support multi-tasking and our partitioning approach already guarantees us load balancing. In general, the two most important measures of the efficiency of such an embedding are its *dilation*, which is the maximum length of the paths in $\mathcal{H}$ corresponding to edges in $\mathcal{G}$, and its *congestion*, which is the maximum over all edges in $\mathcal{H}$ of the total weight of the edges in $\mathcal{G}$ whose corresponding paths contain that edge. (Since we are only considering one-to-one mappings, the *load* of our embeddings will always be one.)

The goal of keeping dilation and congestion low is to minimize the total time required for communication, so the relative importance of these measures for a particular machine is dependent on its communication mechanism and routing method (e.g., store-and-forward versus circuit-switched routing). When we consider different machines, we also expect to discover additional measures of the efficiency (or even feasibility) of embeddings that depend on specific features of their architectures.

5.2 The iWarp System

Our first target machine is the iWarp system. The iWarp system is based on a single chip VLSI processor [3, 4] developed jointly by Carnegie Mellon and Intel Corporation. The iWarp component contains a *computation agent*, and a *communication agent* for transferring data to and from other iWarp processors. Computing rates up to 20 Megaflops and 20 MIPS per cell are possible in the computation agent. The communication agent can sustain a bandwidth of 320 Megabytes/second per cell with a latency of 0.2 microseconds per hop. iWarp systems are 2-dimensional tori of iWarp processors, ranging in size from 4 to 1024 processors. All implementations are done on a prototype 64-processor system at Carnegie Mellon.

iWarp processors communicate with neighboring processors over structures called *logical channels*, which can be chained together to form *pathways* [4]. Although the physical connection pattern of the iWarp is a 2-dimensional torus, logical channels and pathways allow for logical connection patterns such as rings, trees, or even hypercubes [14]. A network of logical channels can be created statically at compile time or created and destroyed dynamically at runtime. Each iWarp processor can support at most 20 logical channels incident to it at any point in time. (Note that a logical channel is a unidirectional communication medium.) Since pathways are handled completely by the communication agent the delay at each processor along such a pathway is very small, and data communicated thus does not interrupt the computation agent.

From these characteristics of iWarp systems we can now discuss some measurements of the efficiency of embeddings. Dilation is of minor importance because of the low latency of the pathways. Congestion is a very important measurement because all pathways sharing a certain physical connection are multiplexed over that link, and the total available bandwidth is limited to 40 Megabytes/second per physical channel. In addition to these traditional measurements we have to face the fact that each iWarp processor can only support up to 20 logical channels at any point in time. Therefore if we cannot limit the number of logical channels needed to be at most 20, then we have to use dynamic setup of pathways, which is slower than the static setup of pathways. 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 since it determines which style of pathway setup to use. (We are currently investigating whether there are more efficient ways to implement precomputed routing patterns which require more than 20 logical channels per processor.)

We have implemented code for the iWarp system to solve a second-order time-varying equation used for earthquake simulation, with encouraging results. This code uses a mesh which has been manually element-partitioned to allow the use of static pathways.

Code for the direct solution of a three-dimensional finite element problem is currently running on the iWarp, but it was designed for one specific block-structured three-dimensional grid. The next step is to modify this code to handle more general and useful irregular grids in two and three dimensions.

5.3 Approaches to Placement and Routing

Because of the communication mechanism of the iWarp system, the congestion of our embeddings are more important than their dilation to the amount of time it will take to communicate the needed messages. This is analogous to some of the issues considered in Gate Array layout (see [9]), where the placement problem reduces to an assignment problem. In the Gate Array layout problem, each gate in the netlist of the circuit has to be assigned to a set of cells on the master that will implement the gate. In our placement problem, each subtask needs to be assigned to a processor. The routing problem between gates in the Gate Array layout problem is characterized by the fact that the routing channels on the master have a fixed width (often the same over the whole chip). This is in contrast to standard-cell and full-custom layout, where the channel widths can be chosen by the layout algorithm. Routing a connection in the layout problem is analogous to embedding a graph edge in the processor network. The limited number of logical channels is analogous to the fixed uniform channel width in the Gate Array layout problem. We use the analogy between these problems to bring some ideas from layout algorithms to our heuristics for embedding the p-vertex weighted graph of subproblems into the graph representing the interconnection network of the target machine. Placement corresponds to the mapping of vertices to vertices, and routing to the mapping of edges to paths.

For placement, it is clearly to our advantage to take into account as much of the geometric information given in the original problem as possible. After all, the first step in layout is often to come up with an embedding of the graph in the plane, and G is given to us already embedded in two- or three-dimensional space. One simple approach to placing two-dimensional graphs in the grid is to simply halve the set of vertices repeatedly with alternating vertical and horizontal cuts, and map the 64 vertices to the torus in the natural way. On the other hand, we can take advantage of the structure of the graph in a deeper way by using the recursion tree from the partitioning algorithm to map the p vertices of the graph of partition blocks to the vertices of the torus. There are many other heuristics for this placement problem, including simulated annealing and distance minimization, which we continue to investigate.

There are also a variety of approaches which can be taken for routing the needed connections. The simplest are one- and two-bend routing schemes with or without randomization, but algorithms using weighted shortest paths to minimize congestion seem to work considerably better. By rerouting paths which pass through areas of high congestion and varying the vertex weights and cost functions in appropriate ways, routings with very good balance of congestion across the network can be obtained. Local optimization heuristics yield further improvements. We are currently experimenting extensively with these methods.

Preliminary experiments have shown that it is not likely that we will be able to embed even the weighted graphs which are derived from partitions of two-dimensional problems within the current numerical limits that would allow us to implement our communication as a single set of static paths. This does not bode well at all for three-dimensional problems, where the average degree of the weighted graphs is nearly double. However, in two dimensions it seems that in general we can divide up the communication into two sets of statically routable paths (as it turns out, simply putting one orientation of each bidirectional edge into each set will work). Two sets may suffice for three dimensions as well, but this requires more study. iWarp researchers are currently studying the question of how efficiently one can switch between precomputed sets of routes such as these when the communication phases are tightly synchronized.

5.4 The Connection Machine CM-5

Our next target machine is the Connection Machine CM-5 [1, 8]. The CM-5 is made up of SPARC microprocessors with optional vector units which are connected by three different interconnection networks: the data network (organized as a fat-tree [7] with processors at the leaves), the control network, and the diagnostic network. The data network provides a high-bandwidth network (up to 20 Megabytes/sec at each processor) that supports message-based, point-to-point communication. Sending a message over the network requires no special routing decisions by the user. The fat-tree network is best suited for communication patterns with hierarchical locality. Communication patterns that take advantage of locality run up to four times faster than random patterns with the same total throughput. The control network supports the data-parallel programming model as well as efficient primitives for synchronization, broadcast and scan.

To solve PDEs, we need to embed the p-minor graph G' into the data network. Because the network hardware takes care of the routing via a randomized algorithm [5], we do not need to specify the embedding of the edges, but only need to map the vertices of G' to the target fat-tree in a way that will enable the hardware to find efficient routings.

Because fat-tree topologies do not have the same bandwidth on all their communication links, a general guideline to optimize the efficiency of the routing of a certain communication pattern is to keep all point to point communication as local as possible. This means that it is important to keep the number of messages passing through any switch proportional to the available capacity. The quality of a placement of G' on a fat-tree F is measured by how well its communication requirements match the actual capacity limitations of the fat-tree.

It is clear how to use the separator results to obtain a natural and fairly efficient placement. During partitioning of 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 of the data network, and embed this tree into the fat-tree in the natural way. Due to the properties of the geometric separators and the fat-tree network, this mapping should yield reasonably good results. We are interested in seeing how locality concerns affect the performance of our implementation.

We intend to analyze the time spent on computation and communication (once the optional vector units are available to us), and compare the efficiency and simplicity of the CM-5's data-parallel and MIMD programming models against each other and the iWarp MIMD implementation.

6 A Step-by-Step Example

On the following pages we illustrate the partitioning, placement, and routing of an unstructured finite element mesh with roughly 5000 nodes on an 8×8 torus. The mesh was designed to predict stresses in a cracked plate, and because of the singularity in stress at the crack tip, is hundreds of times more dense at the center than near the edges. The purpose of this small example is to show the behavior of our partitioning method on an unstructured mesh with a large variation in resolution.

Figure 2 illustrates the original unstructured mesh M for the problem. Note that the mesh is so refined near the center that the figure is completely black. Figure 3 is the underlying graph G of M, after having been vertex-partitioned into 64 blocks. Due to the fineness of the original mesh, many of the partition blocks are indistinguishable. Figure 4 illustrates the 64-vertex minor graph G' derived by collapsing all the vertices in each block of the partition to a single supervertex. Finally, Figure 5 shows a placement and routing of G' on an 8×8 torus. Edges extending past the boundary of the grid wrap around to the other side.

7 Future Directions

The ultimate goal for this project is a fully automatic code generator for finite element problems, which as discussed earlier, will take as input a finite element mesh and a high-level machine-independent description of the algorithm, and will automatically partition the problem and generate code which it will place and route on the target machine. (The problem of generating finite element meshes from the PDEs in question is an interesting problem as well, but we do not address it here.)

Many of the tools needed for such a code generator currently exist, at least in preliminary form: Programs for graph partitioning, automated placement and routing, and parallel computation of sparse matrix-vector product on iWarp processors. The next step is to continue to integrate these tools and improve them.

We are reimplementing the graph partitioning algorithm in C, to improve its portability and to allow better integration with our other modules. We are also considering a data-parallel implementation in NESL [2], for possible use with exceedingly large meshes.

Regarding placement and routing on iWarp systems, we must either develop methods for code generation which can orchestrate communication in graphs which are not statically routable, or modify our partitioning code to guarantee that the resulting minor graphs will be statically routable. Another possibility is to implement statically routable subgraphs of the needed communication graphs. Then, some messages will need to be routed over two consecutive pathways in order to reach their destinations. This approach seems particularly promising.

References

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

[2] G. E. Blelloch. "NESL: A Nested Data-Parallel Language." Technical Report CMU-CS-92-103, Carnegie Mellon University, 1992.

[3] S. Borkar et al. "iWarp: an Integrated Solution to High-Speed Parallel Computing." In *Proceedings of Supercomputing '88*.

[4] S. Borkar et al. "Supporting Systolic and Memory Communication in iWarp." In *Proceedings of the 17th Annual ACM International Symposium on Computer Architecture*, 1990.

[5] R. I. Greenberg and C. E. Leiserson. "Randomized Routing on Fat-Trees." In *Randomnes and Computation*, Vol. 5 of *Advances in Computing Research*, JAI Press, 1989.

[6] F. T. Leighton. *Complexity Issues in VLSI*. MIT Press, 1983.

[7] C. E. Leiserson. "Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing." *IEEE Trans. Computers*, C-34(10):892–901, 1985.

[8] C. E. Leiserson et al. "The Network Architecture of the Connection Machine CM-5." To appear in *Proceedings of the 4th ACM Symposium on Parallel Algorithms and Architectures*, 1992.

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

[10] G. L. Miller, S.-H. Teng, W. Thurston, S. A. Vavasis. "A Unified Geometric Approach to Graph Separators." In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, 1991.

[11] G. L. Miller, S.-H. Teng, W. Thurston, S. A. Vavasis. "Automatic Mesh Partitioning." To appear in *Proceedings of the 1992 Workshop on Sparse Matrix Computations: Graph Theory Issues and Algorithms*, Institute for Mathematics and its Applications.

[12] D. R. O'Hallaron. "The ASSIGN Parallel Program Generator." Technical Report CMU-CS-91-141, Carnegie Mellon University, 1991.

[13] A. Pothen, H. D. Simon, K.-P. Liu. "Partitioning Sparse Matrices with Eigenvectors of Graphs." Technical Report RNR-89-009, NASA Ames Research Center, 1989.

[14] T. M. Stricker. "Supporting the Hypercube Programming Model on Mesh Architectures (A Fast Sorter for iWarp Tori)." To appear in *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1992.