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, 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 res-

olution (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.

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.

[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.

- *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, $A_{st} = \sum_p {}^pA_{st}$.