Paper submitted to:
Third National Symposium on
Large-Scale Structural Analysis for
High-Performance Computers and Workstations

Computational Results for Parallel Unstructured Mesh Computations[†]

Mark T. Jones
Computer Science Department
University of Tennessee
Knoxville, TN 37996
e-mail: jones@cs.utk.edu
phone: (615) 974-4406
FAX: (615) 974-4404

Paul E. Plassmann
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
e-mail: plassman@mcs.anl.gov
phone: (708) 252-4213
FAX: (708) 252-5986

Abstract: The majority of finite element models in structural engineering are composed of unstructured meshes. These unstructured meshes are often very large and require significant computational resources; hence they are excellent candidates for massively parallel computation. Parallel solution of the sparse matrices that arise from such meshes has been studied heavily, and many good algorithms have been developed. Unfortunately, many of the other aspects of parallel unstructured mesh computation have gone largely ignored.

We present a set of algorithms that allow the *entire* unstructured mesh computation process to execute in parallel—including adaptive mesh refinement, equation reordering, mesh partitioning, and sparse linear system solution. We briefly describe these algorithms and state results regarding their running-time and performance.

We then give results from the 512-processor Intel DELTA for a large-scale structural analysis problem. These results demonstrate that the new algorithms are scalable and efficient. The algorithms are able to achieve up to 2.2 gigaflops for this unstructured mesh problem.

[†] *This work was supported in part by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38. In addition, the first author received support from the 1994-1995 UTK Professional Development Award Program.*

1. Introduction.

Unstructured finite element mesh strategies have proven enormously successful in structural analysis as well as in many other disciplines. A structured mesh has regular interconnection patterns between elements of the mesh over the entire mesh—examples of structured and unstructured meshes are given in Figure 1. An unstructured mesh allows much more flexibility in specifying a geometry and refining select areas of the mesh for more accurate results. By selectively refining portions of the mesh rather than the entire mesh, increased accuracy can be achieved at a reduced cost for many computations [21].

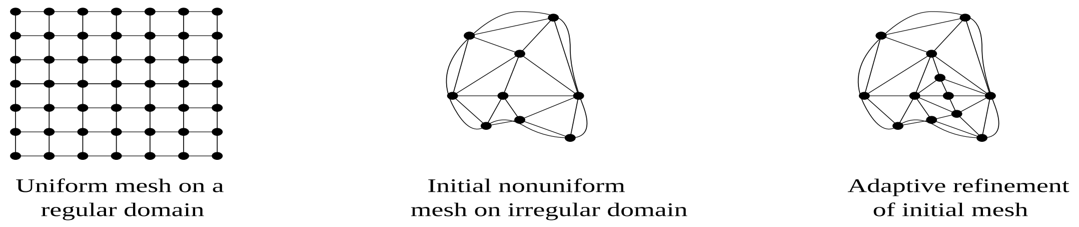
Uniform mesh on a regular domain
Initial nonuniform mesh on irregular domain
Adaptive refinement of initial mesh

FIG. 1. *Examples of structured and unstructured meshes*

Unstructured mesh strategies are particularly appropriate for large-scale structural analysis, where the geometries may be complex and the mesh elements can number in the millions. Distributed-memory parallel computers such as the Intel DELTA or a network of RISC workstations offer a cost-effective tool for solving such problems. However, many difficult algorithmic and implementation issues must be addressed to make effective use of this resource.

It has often been observed that the dominant computational cost in unstructured mesh calculations is the solution of the sparse linear systems derived from this mesh. For this reason, much effort has been invested in developing the parallel algorithms and software for general, sparse linear systems. Software developed for distributed memory computers includes the `BlockSolve` package for the iterative solution of symmetric systems [10], the CAPSS project for direct methods [8], and PETSc, which contains parallel iterative methods for nonsymmetric systems [7].

This paper describes parallel algorithms that work together to solve three basic problems:

- Mesh refinement: adaptive refinement and de-refinement of an initial mesh to provide accuracy at a reduced cost;
- Mesh partitioning: partitioning of meshes into equally sized, well-separated regions for assignment to processors; and
- Sparse matrix solution: assembly and solution of the linear systems that arise from unstructured mesh problems.

The algorithms described are scalable; that is, if the problem size on *each* processor remains fixed and the number of processors increases, then the efficiency of the algorithms will remain fixed. For example, if the algorithm is achieving an operation rate of 50 megaflops on 10 processors on a mesh with 100 elements, then the algorithm should achieve 100 megaflops on 20 processors on a mesh with 200 elements.

The meshes described in this paper are all triangulations of surfaces in two or three dimensions. The algorithms are not restricted to triangles or surfaces; however, the software for mesh refinement is currently restricted to triangles.

The assumption is made that an initial triangulation, T_0, of the structure is given. The parallel generation of conforming triangulations for complex geometries is an important topic, but will not be addressed here.

Given T_0 the following algorithm adapted from [21] is used to solve the problem. This basic framework can be used for many types of structural analysis; in this paper the problem chosen for demonstration is linear static displacement. An example of the basic operation of this algorithm is given in Figure 2.

```
$i = 0$
partition $T_0$ among the processors
assemble the stiffness matrix, $K_0$, and load vector, $f_0$
solve $K_0 u_0 = f_0$
estimate the local error for each triangle in $T_0$
**while** the maximum error on any triangle is larger than the given tolerance **do**
    based on error estimates, determine a set of triangles, $S_i$, to refine
    refine the triangles in $S_i$ and other triangles as needed to form $T_{i+1}$
    partition $T_{i+1}$ among the processors
    assemble a sparse matrix, $K_{i+1}$, and load vector, $f_{i+1}$
    solve $K_{i+1} u_{i+1} = f_{i+1}$
    estimate the local error for each triangle in $T_{i+1}$
    $i = i + 1$
**endwhile**
```

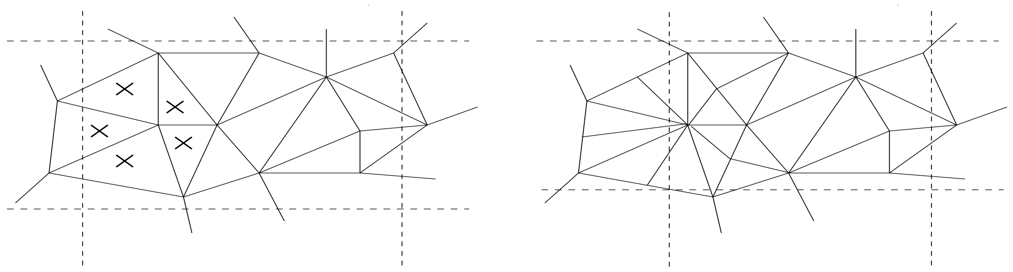

FIG. 2. *An example of the basic operation of the parallel unstructured mesh strategy. On the left is T_i for a processor and its neighbors with triangles marked for refinement. On the right is T_{i+1} for the same processor and its neighbors.*

In Section 2, a provably fast parallel algorithm for mesh refinement is given. A parallel algorithm that yields provably good partitions is given in Section 3. Parallel matrix assembly and solution algorithms are given in Section 4. Computational results for these algorithms for a large-scale structural analysis problem are given in Section 5. These results demonstrate the algorithms are scalable, operating at rates of more than 2 gigaflops.

2. Adaptive Mesh Refinement.

Rather than using a structured mesh with grid points evenly spaced on a domain, adaptive mesh refinement techniques place more grid points in areas where the solution is changing rapidly. The mesh is adaptively refined and de-refined during the computation according to local error estimates.

Many researchers have examined the adaptive construction of these nonuniform meshes. Typically, one begins with an initial mesh and selectively refines that mesh based on local error estimates until a final mesh is constructed that satisfies a given error tolerance.

In this paper adaptive refinement of triangular meshes by simple bisection is considered. Other possible approaches and more detail are given in [17]. Simple bisection has excellent properties; it generates conforming, graded meshes that preserve the element quality of the initial mesh. For a mesh to be conforming, the intersection of any two triangles in the mesh must be a single vertex, a line segment connecting two vertices, or the empty set. In order to be considered a graded mesh, adjacent triangles should not differ dramatically in area. Finally, all angles in the mesh must be bounded away from 0 and π. The latter requirement is necessary because the discretization error in a finite element approximation has been shown to grow as the maximum angle approaches π [1]. Small angles are to be avoided because the condition number of the matrices arising from mesh elements has been shown to grow as $O(\frac{1}{\theta_{\min}})$, where $\theta_{\min}$ is the smallest angle in the mesh [5].

2.1. A Parallel Bisection Algorithm. The bisection algorithm bisects triangles across the largest edge (dividing the largest angle), with selective divisions across a smaller edge (termed simple bisection). This has been shown to yield triangulations whose smallest angle is bounded by at worst one-half the smallest angle in the initial mesh [23]. The algorithm is given in Figure 3.

```
$i = 0$
$Q_i$ = the set of triangles marked for refinement
$R_i = \emptyset$
while $(Q_i \cup R_i) \neq \emptyset$ do
        bisect each triangle in $Q_i$ across its longest edge
        bisect each triangle in $R_i$ across a nonconforming edge
        all nonconforming triangles embedded in $Q_i$ are placed in $R_{i+1}$
        all other nonconforming triangles are placed in $Q_{i+1}$
        $i = i + 1$
endwhile
```

FIG. 3. *The bisection algorithm*

Obviously, the refinement could propagate through many initially unmarked triangles before finishing. Rivara, however, has shown that this loop will terminate in a finite number of iterations, say L_P iterations [22]. Rivara also has shown that each triangle in the resulting conforming mesh, T_{i+1}, embeds 1, 2, 3, or 4 triangles of T_i.

During the execution of the algorithm, no side of a triangle will have more than one nonconformity. We give an example of the propagation in Figure 4.

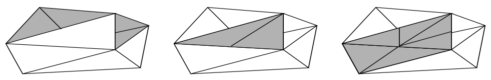

FIG. 4. *From left to right, the process of the bisection algorithm. In the initial mesh the shaded triangles are refined; subsequently the shaded triangles are refined because they are not compatible.*

The refinement algorithm is formulated mainly within the context of the dual graph to the mesh, which we define as follows. Let $V = \{v_i \mid i = 1 \ldots n\}$ be the set of vertices in the mesh and $T = \{t_a \mid a = 1 \ldots m\}$ be the set of triangles. Let $G = (V, E)$ be the graph associated with the mesh, where $E = \{e_{i,j} = (v_i, v_j) \mid v_i, v_j \in t_a\}$. Let $D = (T, F)$ be the dual graph associated with the mesh, where $F = \{(t_a, t_b) \mid e_{i,j} \in t_a, t_b\}$.

For this discussion assume there are as many processors as triangles and that t_a is assigned to processor p_a. Each processor, p_a, must keep track of the neighbors of t_a in D. The algorithm must be synchronized so that this neighbor information is correct. The management of the neighbor information in G for the refinement algorithm is straightforward and will not be discussed here. In order to keep the data structures coherent, two different processors may not create vertices at the same location when bisecting a triangle on their processor. For example, in Figure 5 note the two processors creating two copies of the vertex V at the same location. In the same figure, a possibility is shown for outdated neighbor information to be propagated; triangle U_1 may believe that triangle W is its neighbor rather than triangle W_1 if triangles U and W are simultaneously refined.

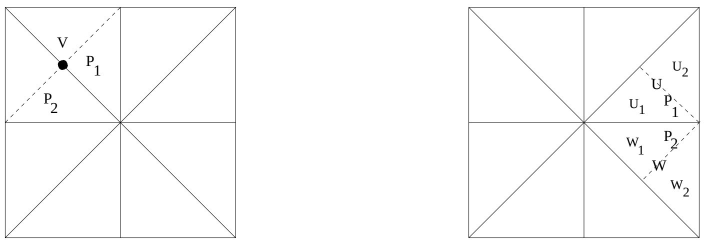

FIG. 5. *On the left, two processors creating a vertex at the same location; on the right, a possible corruption of neighbor information*

In order to avoid these synchronization problems, the new algorithm determines a sequence of independent sets of triangles in the dual graph and refines the triangles in these sets in parallel. The complete algorithm, given in [17], takes into account additional triangles to be refined to obtain a conforming mesh. The crux of the algorithm is the Monte Carlo rule used to determine the independent set. An independent set, I, is chosen at step i by the rule: $t_a \in I$ if for each of its neighbors, t_b, in D, if (a) t_b is not $\in Q_i \cup R_i$ or (b) $\rho(t_a) > \rho(t_b)$, where the $\rho(t)$ are independent random numbers. This rule ensures that no two adjacent triangles are refined simultaneously on different processors, ensuring that the algorithm will execute correctly. The complete parallel algorithm is shown in Figure 6.

In [17] it is shown that under the P-RAM execution model, this algorithm has an expected runtime of $EO(\frac{\log Q_{max}}{\log\log Q_{max}}) \times L_P$ where $Q_{max} = \max_i \mid Q_i \mid$ and L_P is the number of levels of propagation. This bound implies that the running time of the algorithm will increase *very* slowly as the size of the grid increases, thus the algorithm has the potential to perform in a scalable fashion. However, because the P-RAM model ignores many of the communication costs in real parallel computers, the bound is not a guarantee of such behavior.

```
$i = 0$
Based on local error estimates, a set of triangles, $Q_0$, is marked for refinement.
Each triangle, $t_j$, in $Q_0$ is assigned a random number, $\rho(t_j)$
$R_0 = \emptyset$
**While** $(Q_i \cup R_i) \neq \emptyset$ **do**
    $Q_{i+1} = \emptyset$
    $R_{i+1} = \emptyset$
    **While** $(Q_i \cup R_i) \neq \emptyset$ **do**
        Choose an independent set in $D$, $I$, from the triangles in $(Q_i \cup R_i)$
        Simultaneously bisect each of the triangles in $I$
            embedded in $Q_i$ across its longest edge
        Simultaneously bisect each of the triangles in $I$
            embedded in $R_i$ across a nonconforming edge
        For each new triangle, $t_j$, a new random number, $\rho(t_j)$, is chosen
        Each processor containing two triangles now sends one of the
            triangles with neighbor information to a new processor
        Each processor containing a bisected triangle tells its
            neighbors about the bisection
        $R_{i+1} = R_{i+1} \cup$ Any triangles embedded in $Q_i$ made nonconforming
        $Q_{i+1} = Q_{i+1} \cup$ All other triangles made nonconforming
        $Q_i = Q_i - (I \cap Q_i)$
        $R_i = R_i - (I \cap R_i)$
    **Endwhile**
    $i = i + 1$
**Endwhile**
```

FIG. 6. *Parallel algorithm for refinement*

The distributed-memory implementation is based on this algorithm, but allows for many triangles and vertices to be stored on each processor. The same neighbor information is maintained; that is, each triangle knows the location of all of its neighbors at any given time. In addition, in this implementation each processor stores copies of all the triangles to which its triangles are adjacent. These copies yield a savings in communication at a cost of some space. More details of this distributed-memory implementation are given in [17].

3. Mesh Partitioning. As grid points are adaptively added to and deleted from the mesh, one must determine good partitionings of these points onto processors. A good partition ensures that grid points are evenly distributed to the processors in a way that minimizes interprocessor communication costs. The latency and transmission communication costs may be minimized by respectively minimizing the number of partition neighbors and the number of links crossing the partition boundary. For uniform meshes a good partitioning of grid points may be determined a priori by simple constructions. For unstructured adaptive meshes, however, the partitioning cannot be predetermined because it changes with each new refinement of the mesh.

The orthogonal recursive bisection (ORB) algorithm, sometimes called recursive coordinate bisection (RCB), is a simple yet effective graph partitioning algorithm for certain types of graphs for which geometric coordinates are known for the vertices. The authors' unbalanced recursive bisection (URB) algorithm is an improvement on the ORB algorithm that behaves better in practice and for which results on partition quality can be proved. Often, the vertices, V, of the mesh are partitioned into p subsets, V_i, where each V_i is assigned to a processor of a p-processor parallel computer. This is the approach taken here.

Both the URB and ORB algorithms give partitions with no load imbalance; this is simply the nature of the algorithms. However, only for the URB algorithm can bounds be found on the maximum number of neighbors of any partition *independent* of the number of processors. The maximum number of neighbors of a partition is an indication of the maximum number of messages that any one processor must send. If this number is not bounded independently of the number of processors, then as the number of processors increases, some processors may be sending more and more messages. This is clearly not scalable behavior. Other important bounds on URB can also be proved but are given elsewhere [16].

The ORB algorithm as described in [2], given in Figure 7 with an illustration of execution in Figure 8, partitions the vertices according to their physical coordinates while ignoring the edges between vertices. The ORB algorithm has many practical virtues [2] [18] [25] including ease of implementation, inexpensive execution cost, and ease of parallelization. In order to simplify the presentation, the ORB algorithm is given here for the two-dimensional case, but it is easily generalizable to three dimensions.

The URB algorithm is a generalization of the ORB algorithm. The generalization is based on evaluating the *aspect ratio* (a.r.) of the rectangles in the partition,

$$a.r. = \max(\frac{h}{w}, \frac{w}{h}) \ , \tag{1}$$

where h is the height of the rectangle and w is the width of the rectangle. Rather than strictly alternating cut directions and forcing the number of vertices to be divided into two equal sets, the cut is chosen that yields the smallest maximum aspect ratio

```
initial call: ORB($V$,$p$,0)

Procedure ORB(V,p,dir)
if (p == 1) then
    $V$ is marked as a final partition
    return
endif
if (dir == 0) then
    Partition $V$ into $(V_1, V_2)$ such that $| V_1 |=| V_2 |$ and the
        maximum $x$ coordinate of the vertices in $V_1$ is less
        than the minimum $x$ coordinate of the vertices in $V_2$
    dir = 1
else
    Partition $V$ into $(V_1, V_2)$ such that $| V_1 |=| V_2 |$ and the
        maximum $y$ coordinate of the vertices in $V_1$ is less
        than the minimum $y$ coordinate of the vertices in $V_2$
    dir = 0
endif
call ORB($V_1$,$p/2$,dir)
call ORB($V_2$,$p/2$,dir)
End Procedure
```

FIG. 7. *The ORB algorithm in the x-y plane*

of the two resulting rectangles. The URB algorithm is given in Figure 9, with an illustration of the execution in Figure 10.

3.1. Dynamic Repartitioning. One can take advantage of the gradual changes to the mesh that occur during mesh refinement. After each mesh refinement step, the mesh need not be partitioned from scratch; the old partitioning can be used to generate a new partitioning that is only slightly different. This approach has two advantages: (1) the time for partitioning may be reduced, and (2) rather than moving all of the vertices to new processors, only a small percentage of the vertices may need to be moved if the partitioning is updated rather than generated from scratch.

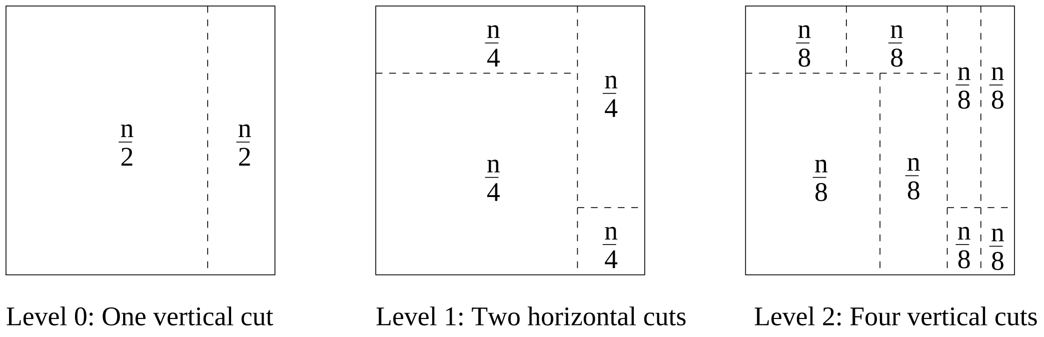

FIG. 8. *Possible sequence of cuts for the ORB algorithm*

```
initial call: URB($V$,$p$)

Procedure URB(V,p)
if (p == 1) then
    $V$ is marked as a final partition
    return
endif
Partition $V$ into $(xV_1, xV_2)$ such that $| xV_1 |= k\frac{n}{p}$, where $k$ is an integer,
    the maximum $x$ coordinate of the vertices in $xV_1$ is less
    than the minimum $x$ coordinate of the vertices in $xV_2$, and
    the cut yields the best a.r. over $1 \leq k \leq p-1$
Partition $V$ into $(yV_1, yV_2)$ such that $| yV_1 |= m\frac{n}{p}$, where $m$ is an integer,
    the maximum $y$ coordinate of the vertices in $yV_1$ is less
    than the minimum $y$ coordinate of the vertices in $yV_2$, and
    the cut yields the best a.r. over $1 \leq m \leq p-1$
if ($X$-cut yields better a.r.) then
    call URB($xV_1$,$k$)
    call URB($xV_2$,$p-k$)
else
    call URB($yV_1$,$m$)
    call URB($yV_2$,$p-m$)
endif
End Procedure
```

FIG. 9. *The URB algorithm in the x-y plane*

The ORB and URB algorithms are particularly amenable to such updating [2]. One can simply move the cuts of the old partitioning to reflect the refined mesh. If this does not result in an acceptable partition, then the mesh can be partitioned from scratch. For example, assume that the mesh was originally partitioned into two sets of 50 grid points and then 10 grid points were added to the left partition during refinement. During partition updating, the original cut would be moved to the left until the mesh was partitioned into two sets of 55 grid points. This strategy can be recursively carried out to compute an updated partitioning. To determine the acceptability of this updated partitioning for the URB algorithm, one can check the aspect ratio of the partitions generated; if they are unacceptably high, then a new partition can be generated from scratch.

4. Assembly and Solution of Sparse Linear Systems. The sparse matrix assembly algorithm is designed to cooperate with the refinement algorithm. At the end of a refinement step, each processor has the vertices that it is responsible for as well as copies of every triangle connected to these vertices. This information is all that the processor needs to construct all of the columns of the sparse matrix associated with its vertices: no communication needs to take place during the assembly process. This savings in communication expense and code simplicity is achieved at the cost of

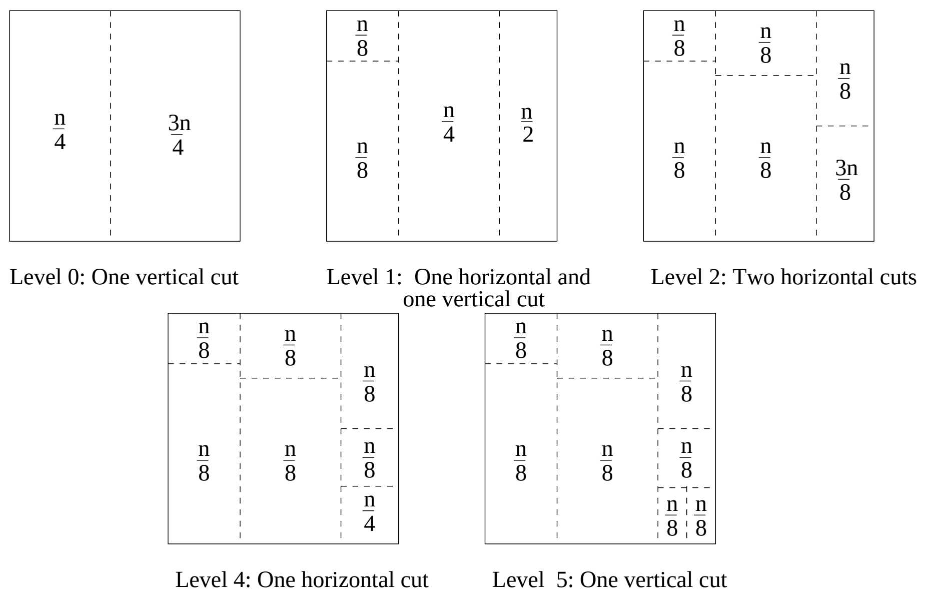
n/4
3n/4
Level 0: One vertical cut
n/8
n/8
n/4
n/2
Level 1: One horizontal and one vertical cut
n/8
n/8
n/8
n/8
n/8
3n/8
Level 2: Two horizontal cuts
n/8
n/8
n/8
n/8
n/8
n/8
n/4
Level 4: One horizontal cut
n/8
n/8
n/8
n/8
n/8
n/8
n/8
n/8
Level 5: One vertical cut

FIG. 10. *Possible sequence of cuts for the URB algorithm*

duplicating finite element evaluations because copies of some triangles may exist on more than one processor. We do not investigate this tradeoff here.

The assembly routine comprises two phases: (1) determining the nonzero structure of the matrix and (2) evaluating the finite elements and inserting them into the allocated structure. Because the mesh changes only in select locations, the entire structure of the matrix does not change at every step. The assembly algorithm updates the structure of matrix to reflect changes made by the refinement algorithm; the entire matrix is not determined from scratch at each step. The evaluation of all the finite elements, however, is done at every step because for many problems (e.g., nonlinear problems) the values in the matrix may change even though the sparsity structure may not.

The `BlockSolve` package, a collection of parallel iterative methods, is employed to solve the sparse linear systems [10]. The iterative solver chosen from `BlockSolve` is an incomplete matrix factorization used as a preconditioner for the conjugate gradient algorithm [20]. This general-purpose preconditioner performs well for many structural analysis problems. For 2-D structural analysis problems such as the one in Section 5, the number of iterations required for convergence to a solution is expected to be proportional to $\sqrt{n}$, where n is the number of unknowns — similar to the results in [3].

The scalable implementation of this preconditioned conjugate gradient algorithm is straightforward with two exceptions that we discuss below. Each processor is responsible for columns of the matrix and the unknowns that correspond to the grid points on that processor.

This implementation comprises three components: the conjugate gradient algorithm, the matrix by vector multiplication, and the incomplete matrix factorization and triangular matrix solution required for the preconditioner. The parallel implementation of the first two operations is relatively simple. The only communication required by the conjugate gradient algorithm is a global sum for the inner product computation.[1] In the matrix by vector multiplication, a single communication step is required to communicate the nonlocal elements of the vector to those processors that need them. Aside from these minor communication steps, both these operations are perfectly parallel.

However, two main obstacles impede the efficient parallel implementation of an iterative solver based on this preconditioning. First, the triangular linear system solutions do not exhibit a high degree of parallelism for standard matrix orderings [19]. Second, it is not sufficient to achieve scalable performance; one must also achieve good computation rates on each processor. In high-performance RISC chips (such as the Intel i860) the best performance is obtained by algorithms that exhibit good data locality and minimize indirect addressing. The following two subsections, 4.1 and 4.2, describe the methods used to overcome each of these obstacles and to obtain scalable, efficient performance on parallel computers such as the Intel DELTA.

4.1. The Scalable Inversion of Triangular Systems. The triangular linear system solution is the central problem in the parallelization of the standard iterative methods. For example, it is involved in the application of an SOR or SSOR iteration, in addition to preconditioners derived from an incomplete factorization.[2] The traditional serial approach to solving a triangular linear system employs a "natural" ordering of the variables. Unfortunately, a scalable parallel implementation of this approach is impossible because the dependencies in the solution of triangular systems make this computation inherently sequential.

However, a reordering of the preconditioning matrix based on a coloring of the graph associated with the matrix does allow for its scalable solution. The reordered triangular system solution is scalable because the number of sequential communication steps is proportional to the chromatic number of the graph [24], which is essentially a function of the local graph structure, and independent of the size of the graph. In Figure 11 an example of a multicoloring ordering is given for a regular grid that requires four colors.

For less regular problems for which one does not have a priori knowledge of an optimal graph coloring, a graph coloring heuristic must be used. The authors have developed and implemented a parallel graph coloring heuristic based on finding a sequence of independent sets that generates colorings similar those found by sequential graph coloring heuristics [15].

This combination of graph coloring heuristics and incomplete matrix factorization is effective for a range of structured and unstructured finite element and finite difference problems [19]. In addition, recent theoretical results have shown that one does not see the dramatic increase of the number of iterations required for convergence with "many-color" orderings that one sees with the red/black ordering for model

[1] For a discussion of the conjugate gradient algorithm see [6].

[2] The scalable computation of the incomplete factors can be accomplished in the same fashion as the triangular linear system solution.

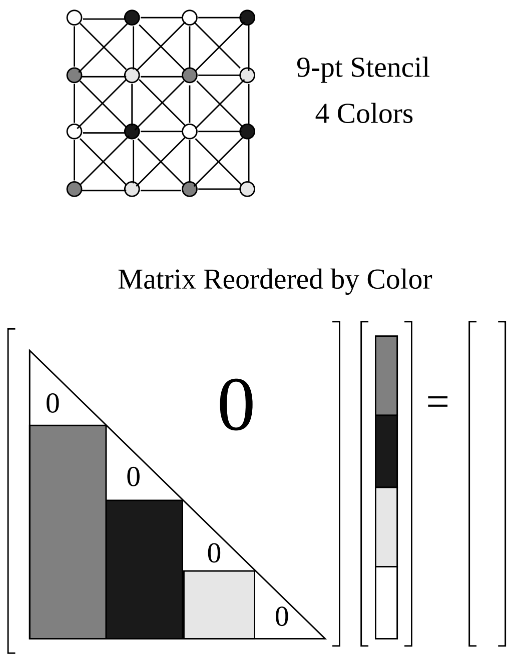

Fig. 11. *The adjacency graph corresponding to a nine-point stencil requires four colors. An ordering of the variables of the corresponding linear system allows for the solution of a triangular system of the same structure to be solved in four major parallel steps: one step for the unknowns corresponding to each color, followed by interprocessor communication to update the right-hand side.*

problems [11].

4.2. Graph Reductions. As discussed previously, it is not sufficient to achieve scalable performance; one must also use each processor efficiently. For example, a standard implementation of a sparse matrix-vector multiplication does not exhibit good data locality and uses a large amount of indirect addressing. To improve locality and minimize indirect addressing, one can take advantage of the special local structure inherent to many finite element problems. For example, large, dense cliques exist in these graphs and can be easily recognized. Operations involving these cliques can utilize dense Level 2 and 3 BLAS. In addition, many rows of the sparse matrix have identical structure, but differing nonzero values. This structure can be exploited to significantly reduce the amount of indirect addressing. Note that these ideas have been used with dramatic effect in direct sparse factorization for several years.

It is often observed that the sparse systems arising in many applications have a great deal of special local structure, even if the systems are described as "unstructured." Illustrations of some of this local structure, and how it can be identified, are given in the following sequence of figures.

In Figure 12a is a depiction of a subsection of a graph that arises from a two-dimensional, linear, finite-element model with three degrees of freedom per vertex.

The three degrees of freedom are denoted by the three dots at each vertex; the linear elements imply that the twelve degrees of freedom sharing the four vertices of each face are completely connected. In the figure only edges between the vertices are shown; these edges represent the complete interconnection of all the vertices on each element, or face.

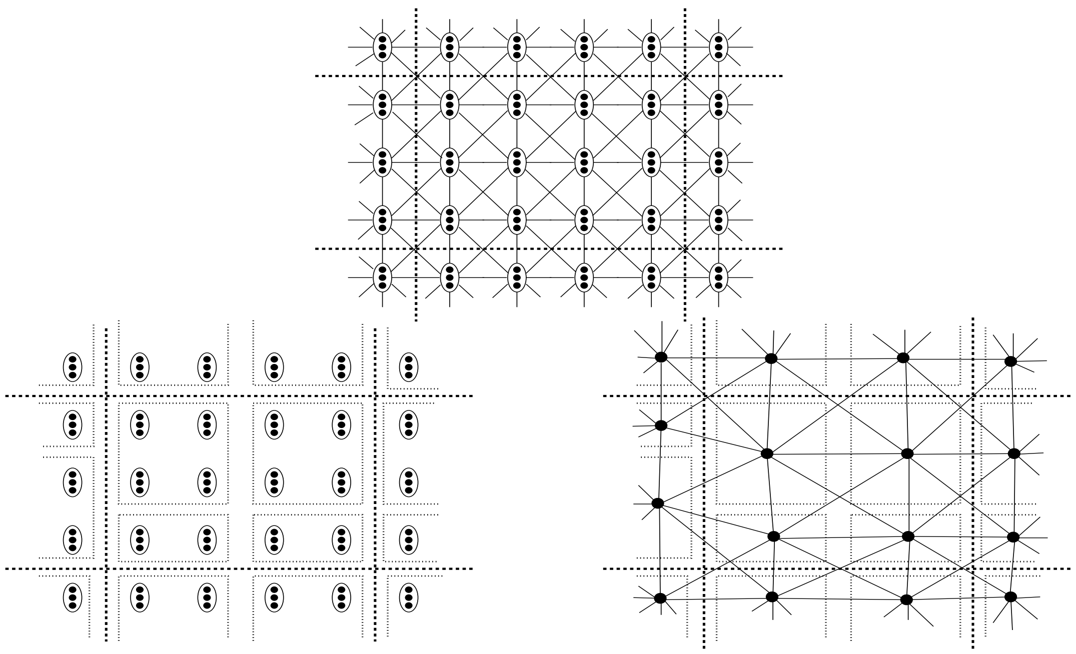

FIG. 12. *(a) The top figure is a subgraph generated by a two-dimensional, linear finite element model with three degrees of freedom per vertex. The partitioning shown by the dotted lines yields an assignment of the vertices in the enclosed subregion to one processor. (b) The left figure is a partition of the vertices into cliques. (c) The right figure is a quotient graph given the clique partition in the left figure.*

The dashed lines in the figure represent a partitioning of the grid; assume that the vertices in the central region are all assigned to one processor. Several observations follow on the local structure of this subgraph. First, note that the adjacency structure of the vertices at the same geometric location (i.e., the nonzero structure of the associated variables) is identical; denote such vertices *identical vertices*. Schreiber and Tang [24] noted that a coloring of the graph corresponding to the vertices results in a system with small dense blocks, of order the number of degrees of freedom per vertex, along the diagonal. This observation can also be used to decrease the storage required for indexing the matrix rows because the structures are identical.

Consider a further graph reduction based on the local clique structure of the graph. In Figure 12b the dotted lines show one possible way the vertices assigned to the partition and its neighbors can be partitioned into cliques.[3] Denote such a partition by Q. If one associates a super-vertex with each clique, then the quotient

[3] Of course , the quotient graph reduction is not limited to the choice of a maximal clique partition; any local partition of the subgraph assigned to a processor can be used to generate the reduced graph. Several alternatives are discussed in [14].

graph G/Q can be constructed based on the rule that there exists an edge between two super-vertices v and w if and only if there exists an edge between two vertices of their respective partitions in G. The quotient graph constructed by the clique partition shown in Figure 12b is shown in Figure 12c.

When the matrix is reordered according to such a clique decomposition, the matrix has large, dense blocks along the diagonal that allow for the use of the higher-level dense BLAS. In addition, by coloring the quotient graph rather than the original graph, the number of colors needed is greatly reduced. Thus, rather than simultaneously solving the diagonal submatrices associated with each color, the processors now simultaneously solve block diagonal submatrices associated with each color.

Finally, note that the efficient determination of identical nodes, and a local maximal clique decomposition, is straightforward. Because the adjacency structure of the vertices assigned to a processor is known locally, no interprocessor communication is required, and a greedy heuristic can be used to determine a clique partition.

5. Computational Results. The computational experiments given in this section demonstrate that

- the refinement algorithm is scalable,
- the partitioning algorithm yields scalable partitions,
- the refinement and partitioning algorithms are relatively inexpensive, and
- the matrix assembly and solution algorithms are scalable and efficient.

The experiments were run on the 512-processor Intel DELTA. The DELTA is a 16×32 mesh of Intel i860 microprocessors in which interprocessor communication takes place using message-passing. The algorithms were implemented in the C language with extensions for message passing.

A single large-scale structural analysis problem was chosen to demonstrate the behavior of the algorithms in as simple a manner as possible. The algorithms have been executed individually on other, very different problems for which similar results have been achieved [4] [12] [13] [14] [18].

The structure of interest is a thin, hollow sphere with four triangular holes equally spaced over each hemisphere. An initial triangular mesh representing this geometry is given in Figure 13.

The sphere is constrained around the south pole, and a force is applied around the north pole toward the south pole; the displacement of the structure at equilibrium is then solved for at every mesh point. The finite element used is a triangular shell element [9] with quartic basis functions. The local error estimator for each triangle is the norm of the strain vector integrated over the triangle. The initial mesh is refined until every triangle satisfies a specified error tolerance. Such a refined mesh is given in Figure 14. The partitioning of this mesh is given in Figure 15.

To demonstrate the scalability of the algorithms, we generated a sequence of problems by choosing the error tolerance such that the final mesh in each problem was roughly twice as large as the final mesh in the preceding problem. If in this sequence, twice as many processors are assigned to a problem as the preceding problem, then the number of vertices/triangles per processor will remain constant over the entire

FIG. 13. *The initial geometry of the test problem*

problem sequence. Note that each of the problems, shown in Table 1, begins with an initial triangulation and that the numbers given in the table are for the final mesh in *each* problem.

TABLE 1
The sequence of test problems

Problem Name	Number of Processors	Number of Vertices	Number of Triangles	Number of Equations	Number of Nonzeros
SPHERE16	16	6,570	1,280	32,850	1,741,550
SPHERE32	32	13,938	2,728	69,690	3,706,270
SPHERE64	64	24,626	4,840	123,130	6,567,390
SPHERE128	128	53,802	10,648	269,010	14,418,730
SPHERE256	256	111,058	26,516	555,290	29,820,870
SPHERE512	512	209,922	41,776	1,049,610	56,478,730

FIG. 14. *The refined, deformed geometry of the test problem. High strain is indicated by red, moderate strain by green, and low strain by blue.*

FIG. 15. *The partitioning of the vertices among 64 processors for the refined test problem. Each partition is indicated by a different color.*

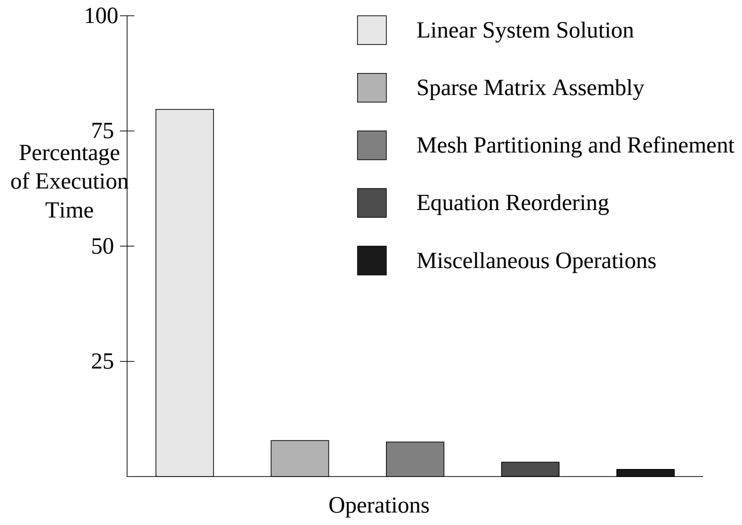

FIG. 16. *The percentage of execution time required by certain operations on 512 processors for SPHERE512*

5.1. Refinement and Partitioning Results. Relative to the time required for matrix assembly and solution, the cost for the refinement and partitioning algorithms is small. In fact, one observes in Figure 16 that the percentage of execution time required for these algorithms is approximately 7.5%. Further, the time for refinement alone is less than 0.2% of the total execution time.

The efficiency of the refinement algorithm is only slightly decreased as the number of processors increases. Both the maximum and average number of triangles refined per processor per second are given in Figure 17. Because some processors typically refine more triangles than other processors, the refinement rate most indicative of the performance of the algorithm is the maximum number of triangles refined per second per processor. At 512 processors, the refinement algorithm is still operating at 63% of the 16-processor rate, for a total speedup of 20 out of a possible 32.

As noted above, the total time for partitioning the graph after each refinement step was less than 7.5%, indicating that the overhead for parallel processing is quite small. Moreover, the partitioning algorithm generated scalable partitions. Each partition was connected to an average of between 6 and 7 other processors regardless of the total number of processors. As will be shown in the next subsection, these good partitionings result in scalable performance for the matrix assembly and solution algorithms.

5.2. Matrix Assembly and Solution Results. If the matrix assembly and equation reordering algorithms are scalable, then their execution costs will remain roughly constant as the number of processors increases. As we observe in Figure 18,

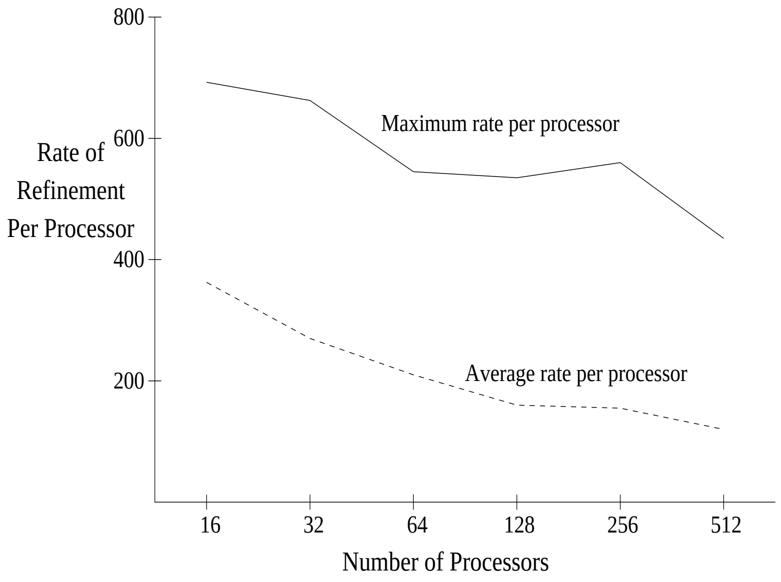

FIG. 17. *The maximum and average number of triangles refined per second per processor as a function of the number of processors*

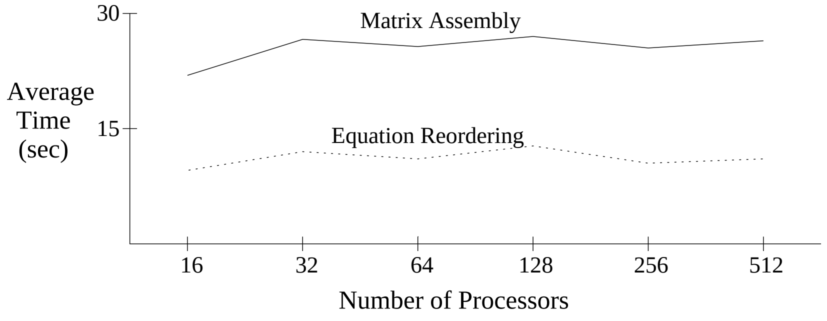

FIG. 18. *The average time for matrix assembly and equation reordering for each problem*

this is indeed the case.[4] Moreover, we note that on 512 processors, the matrix assembly routine achieves a useful floating point operation rate of approximately 2.1 gigaflops; operations that are duplicated on other processors are not counted.

The parallel iterative algorithm for matrix solution performs in a scalable, efficient fashion as well. As is seen in Figure 19, the total gigaflop rate scales as the number of processors increases up to a total of 2.2 gigaflops for 512 processors. Note that not only is the performance scalable, but it is efficient as well: each processor is executing at a rate of over 4 megaflops, a very good rate for sparse operations on the Intel i860 microprocessor. Also note that the growth in the number of iterations required as a function of the number of equations scales as expected: the number of iterations is

[4] Note that the time for equation reordering includes the time to scale the matrix by the diagonal and predetermine the communication pattern of the linear system solution routines.

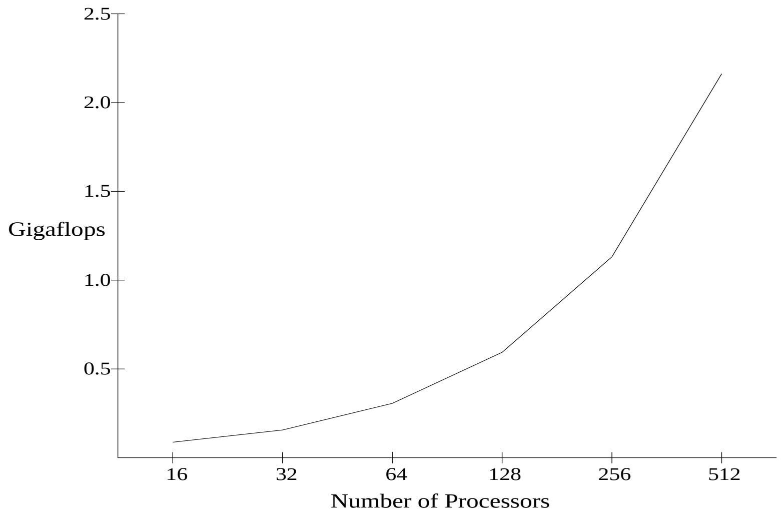

FIG. 19. *The number of gigaflops as a function of the number of processors*

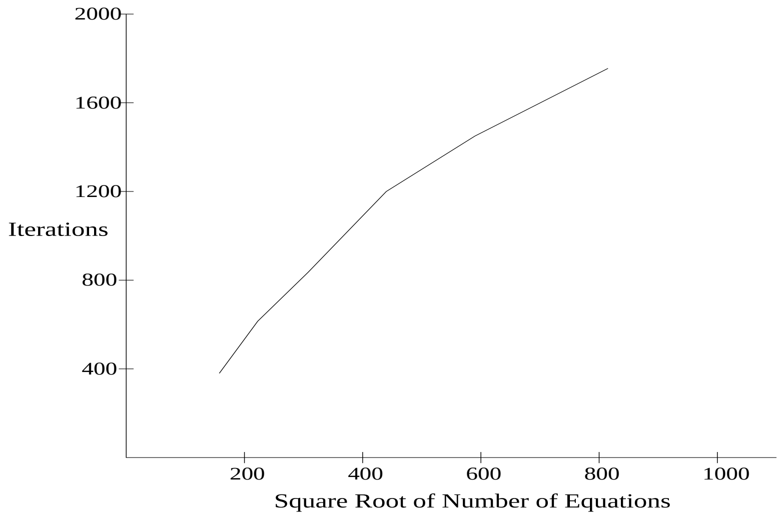

FIG. 20. *The number of iterations as a function of the square root of problem size*

proportional to the square root of the number of equations (see Figure 20) [3].

6. Summary and Future Work. A parallel algorithm for the refinement of unstructured meshes was given and computational results were described that demonstrated the scalability and efficiency of this algorithm. A partitioning algorithm for such meshes was given and shown to generate partitions of high quality for large numbers of processors. Finally, sparse matrix assembly and solution algorithms were given that cooperate with the refinement and partitioning algorithms. These sparse matrix algorithms were shown to be scalable and operate at a rate of up to 2.2 gigaflops on the Intel DELTA parallel computer. The combination of the algorithms was shown to be an effective scalable method for a large-scale structural analysis problem.

Future work includes integrating a parallel generalized eigensolver based on the block Lanczos algorithm into the code, as well as incorporating other software packages for solving linear systems and partitioning meshes. In addition, the refinement software will be enhanced to allow for tetrahedral meshes. The performance and utilization of this code on a high-speed network of RISC workstations are also of great interest to the authors.

REFERENCES

[1] I. Babuška and A. K. Aziz, *On the angle condition in the finite element method*, SIAM Journal of Numerical Analysis, 13 (1976), pp. 214–226.

[2] M. J. Berger and S. H. Bokhari, *A partitioning strategy for nonuniform problems on multiprocessors*, IEEE Transactions on Computers, C-36 (1987), pp. 570–580.

[3] T. F. Chan and H. C. Elman, *Fourier analysis of iterative methods for elliptic boundary value problems*, SIAM Review, 31 (1989), pp. 20–49.

[4] L. A. Freitag, M. T. Jones, and P. E. Plassmann, *New techniques for parallel simulation of high-temperature superconductors*, in Proceedings of the Scalable High-Performance Computing Conference, Los Alamitos, Calif., 1994, IEEE, pp. 726–733.

[5] I. Fried, *Condition of finite element matrices generated from nonuniform meshes*, AIAA Journal, 10 (1972), pp. 219–221.

[6] G. H. Golub and C. F. V. Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, 1983.

[7] W. D. Gropp and B. F. Smith, *Simplified Linear Equation Solvers Users Manual*, Tech. Rep. ANL-93/8, Argonne National Laboratory, Argonne, Ill., Mar. 1993.

[8] M. T. Heath and P. Raghavan, *Distributed solution of sparse linear systems*, Tech. Rep. UIUCDCS-R-93-1793, University of Illinois, Feb. 1993.

[9] T. Hughes, *The Finite Element Method*, Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

[10] M. T. Jones and P. E. Plassmann, *BlockSolve v1.0: Scalable library software for the parallel solution of sparse linear systems*, ANL Report ANL-92/46, Argonne National Laboratory, Argonne, Ill., 1992.

[11] ——, *The effect of many-color orderings on the convergence of iterative methods*, in Proceedings of the Copper Mountain Conference on Iterative Methods, SIAM LA-SIG, 1992.

[12] ——, *Solution of large, sparse systems of linear equations in massively parallel applications*, in Proceedings of Supercomputing '92, IEEE Computer Society Press, 1992, pp. 551–560.

[13] ——, *Computation of equilibrium vortex structures for type-II superconductors*, The International Journal of Supercomputer Applications, 7 (1993), pp. 129–143.

[14] ——, *The efficient parallel iterative solution of large sparse linear systems*, in Graph Theory and Sparse Matrix Computation, A. George, J. Gilbert, and J. W. Liu, eds., vol. 56 of The IMA Volumes in Mathematics and its Applications, Springer-Verlag, 1993, pp. 229–245.

[15] ——, *A parallel graph coloring heuristic*, SIAM Journal on Scientific Computing, 14 (1993), pp. 654–669.

[16] ——, *Bounds on partition quality for orthogonal recursive bisection*, Preprint MCS-P465-0894, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1994.

[17] ——, *Parallel algorithms for adaptive mesh refinement*, Preprint MCS-P421-0394, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1994.

[18] ——, *Parallel algorithms for the adaptive refinement and partitioning of unstructured meshes*, in Proceedings of the 1994 SHPCC, IEEE, 1994, pp. 726–733.

[19] ——, *Scalable iterative solution of sparse linear systems*, Parallel Computing, 20 (1994), pp. 753–773.

[20] T. A. Manteuffel, *An incomplete factorization technique for positive definite linear systems*, Mathematics of Computation, 34 (1980), pp. 473–497.

[21] W. F. Mitchell, *A comparison of adaptive refinement techniques for elliptic problems*, ACM Transactions on Mathematical Software, 15 (1989), pp. 326–347.

[22] M.-C. Rivara, *Mesh refinement processes based on the generalized bisection of simplices*, SIAM Journal of Numerical Analysis, 21 (1984), pp. 604–613.

[23] I. G. Rosenberg and F. Stenger, *A lower bound on the angles of triangles constructed by bisecting the longest side*, Mathematics of Computation, 29 (1975), pp. 390–395.

[24] R. Schreiber and W.-P. Tang, *Vectorizing the conjugate gradient method.* Unpublished information, Department of Computer Science, Stanford University, 1982.

[25] R. D. Williams, *Performance of dynamic load balancing algorithms for unstructured mesh calculations*, Concurrency: Practice and Experience, 3 (1991), pp. 457–481.