PARALLEL ALGORITHMS FOR ADAPTIVE MESH REFINEMENT*

MARK T. JONES AND PAUL E. PLASSMANN†

Abstract. Computational methods based on the use of adaptively constructed nonuniform meshes reduce the amount of computation and storage necessary to perform many scientific calculations. The adaptive construction of such nonuniform meshes is an important part of these methods. In this paper, we present a parallel algorithm for adaptive mesh refinement that is suitable for implementation on distributed-memory parallel computers. Experimental results obtained on the Intel DELTA are presented to demonstrate that, for scientific computations involving the finite element method, the algorithm exhibits scalable performance and has a small run time in comparison with other aspects of the scientific computations examined. It is also shown that the algorithm has a fast expected running time under the PRAM computation model.

Key words: adaptive mesh refinement, distributed memory computers, parallel algorithms, sparse matrices, unstructured mesh computation

AMS(MOS) subject classifications: 65F10, 65F50, 65Y05, 68Q20, 68Q22

1. Introduction. Adaptive mesh refinement techniques have been shown to be very successful in reducing the computational and storage requirements for solving many partial differential equations [12]. Rather than use a uniform mesh with grid points evenly spaced on a domain, adaptive mesh refinement techniques place more grid points in areas where the local error in the solution is large. The mesh is adaptively refined and/or unrefined during the computation according to local error estimates on the domain. This technique is much more efficient than the use of uniform meshes when the solution is changing much more rapidly in some areas than in others.

The adaptive construction of these nonuniform meshes is a crucial part of adaptive mesh solution methods and has been examined by many researchers, for example, [3], [12], [13], [14], [15], [16], [17], [18], and [20]. Typically, one begins with an initial mesh conforming to a particular geometry. This mesh is selectively refined, based on local error estimates, to construct a mesh that satisfies a certain error tolerance. Most research has focused on meshes composed of simplicial elements: line segments in one dimension, triangles in two dimensions, or tetrahedra in three dimensions. This paper focuses primarily on two-dimensional simplicial meshes. However, the algorithms and analyses presented here are applicable to other dimensions and to nonsimplicial meshes.

In this paper, we present a new parallel algorithm for the adaptive construction of nonuniform meshes. This algorithm is well suited for implementation on medium-grained distributed-memory parallel computers such as the Intel DELTA. The algorithm is based on the simplicial bisection algorithm given by Rivara [17]. Our algorithm is scalable in that it has an expected run time that is a *very* slowly growing function of the triangles in the mesh.

To demonstrate the performance of this algorithm, we present experimental results obtained on the Intel DELTA. The results demonstrate that, for practical scientific calculations, the algorithm exhibits scalable performance and a run time that is much smaller than the other computations necessary for the entire solution method.

* This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

† The address of the first author is Computer Science Department, University of Tennessee, Knoxville, TN 37996. The address of the second author is Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439.

This paper is organized as follows. In §2 we review methods for adaptive mesh refinement. In §3 we present our algorithm and analyze its expected run time under the Parallel Random Access Machine (PRAM) computation model. A medium-grained distributed-memory version of the algorithm is detailed in §4. We discuss our experimental results from the Intel DELTA in §5. Finally, in §6 we summarize this research and discuss possible future work.

2. Adaptive Refinement Methods. The finite element method has proven to be extremely effective in the computation of approximate solutions to partial differential equations (PDEs). Our focus is on adaptive, or local, refinement strategies for generating finite element meshes. This approach can obtain much more accurate solutions to these problems than a uniform mesh with the same number of elements.

The computation of an approximate solution to a PDE consists of three main tasks: (1) the construction of the finite element mesh, (2) the assembly of a sparse linear system, and (3) the solution of this linear system. Although we are not explicitly discussing the last two tasks, they must be kept in mind. In particular, any method of local error estimation requires the approximate solution on a given element mesh.

With a parallel implementation, we must remember that it is essential that any adaptive mesh algorithm be integrated with parallel algorithms for the matrix assembly and the solution of the resulting linear system. In addition, with an adaptive strategy the assignment (or partitioning) of elements and vertices to processors must be updated with each modification of the mesh to ensure the *continued* efficient execution of the matrix assembly and linear system solution.

$k = 0$
Solve the PDE on T_k
Estimate the error on each triangle
while the maximum error on a triangle is larger than the given tolerance **do**
    Based on error estimates, determine a set of triangles, S_k, to refine
    $\star$ Divide the triangles in S_k, and any other
        triangles necessary to form T_{k+1}
    Solve the PDE on T_{k+1}
    Estimate the error on each triangle
    $k = k + 1$
endwhile

FIG. 1. *A framework for the adaptive solution of PDEs*

Consider the general adaptive mesh algorithm [12] shown in Figure 1. We begin by assuming that we have an initial element mesh given by the triangulation T_0 consistent with the geometry of the problem domain. Our attention is focused on the step in the algorithm where the current mesh T_k is adaptively refined (the step denoted by the $\star$ in Figure 1).

Suppose that some arbitrary subset of triangles, S_k, of T_k is marked for refinement. We have developed and implemented parallel algorithms for constructing a new mesh T_{k+1} that satisfies the required changes in the mesh. To keep our presentation clear and brief, we assume that the set S_k contains triangles marked only for refinement, not for unrefinement. However, our software is able to unrefine triangles that have been previously refined.

The refinement of the mesh must maintain several important properties, given that finite element approximations are used. First, we require that each mesh T_k be *conforming* (or *compatible*). That is, the intersection of any two triangles in T_k should be a single vertex, a line segment connecting two vertices, or the empty set. A side of a triangle is called $\frac{1}{s+1}$-*nonconforming* if it has $s > 0$ vertices between any two endpoints. A triangle is called *compatible* if none of its sides are $\frac{1}{s+1}$-nonconforming. Examples of conforming and nonconforming meshes are given in Figure 2. If the mesh is conforming, then only one basic type of finite element is necessary. Otherwise, several special element types are required, and/or a more complicated matrix assembly.

Note, however, that the use of triangles does not restrict one to linear finite elements; one can use higher-order basis functions in a triangulation.

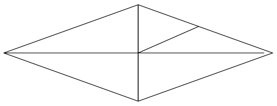

FIG. 2. *On the left, a conforming mesh; on the right, a nonconforming mesh*

A second requirement is that the mesh T_k be *graded* (or *smooth*). That is, adjacent triangles should not differ dramatically in area. A nonsmooth mesh could result in the finite element approximation being very far from the continuous solution.

A final requirement is that all angles in the mesh be bounded away from 0 and π. The latter condition is necessary because the discretization error in a finite element approximation has been shown to grow as the maximum angle approaches π [1]. We would like to avoid small angles 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. Related Work. A number of mesh refinement algorithms have been shown to maintain the mesh properties given above. In this section we briefly review the three most widely used of these refinement methods. To begin, we note that there are two methods used to subdivide a triangle: *bisection* and *regular refinement*. In bisection, a vertex of the triangle is connected to the midpoint of the opposite side of the triangle, as in Figure 3, forming two triangles of equal area. In regular refinement, the midpoints of the sides of the triangle are connected, as in Figure 3, to form four similar triangles.

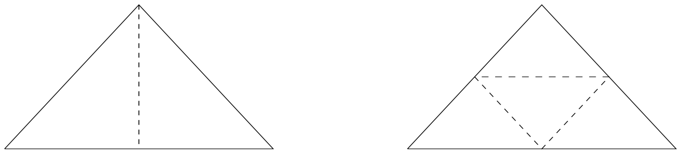

FIG. 3. *On the left, a triangle divided with bisection; on the right, a triangle divided by using regular refinement*

The *regular refinement* algorithm of Bank, Sherman, and Weiser [3] has been used very successfully in the software package PLTMG [2]. Triangles are divided by using regular refinement and temporary bisections of selected triangles to make the mesh conforming. The bisected triangles are merged before the mesh is refined again. By merging the bisected triangles at each level, the method guarantees that each triangle in T_{k+1} either is similar to a triangle in T_0 or is a bisection of a triangle similar to a triangle in T_0. Clearly, the angles in T_{k+1} are bounded away from 0 and π.

The mesh refinement algorithm 5.6 of Rivara [17] uses bisections of triangles across the largest edge (dividing the largest angle) and selective divisions across smaller edges. This approach has been shown to yield triangulations, T_k, whose smallest angle is bounded by at worst one-half the smallest angle in T_0 [19]. A detailed discussion of this algorithm is given in the following subsection.

The *newest-node* algorithm of Sewell is also based on bisection, but without the restriction on bisecting the longest edge [20]. In this algorithm, a triangle is always bisected by using its newest node. The propagation inherent in the bisection and regular refinement algorithms is avoided by refining triangles only in pairs. However, because of the pair restriction, it is possible that a triangle may never be able to be refined. A modification proposed by Mitchell [12] eliminates this problem by ensuring that every triangle is one of a pair of triangles that can be refined. Unfortunately, this modification requires the recursive refinement of triangles adjacent to unrefinable triangles. This refinement gives rise to a propagation similar to the bisection and regular refinement algorithms.

Mitchell compared these three methods in a series of numerical experiments and found that it was difficult to choose a consistently superior algorithm [12]. In addition, he found that all three methods were superior to using uniform refinement except on smooth problems. Given the similar performance of the three methods, we choose to discuss the bisection algorithm in detail in this paper for three reasons: (1) it is simpler from an implementation standpoint than the regular refinement algorithm; (2) it manifests the propagation inherent in the bisection, regular refinement, and Mitchell's modification of the newest-node algorithms and, therefore, demonstrates the ability of our algorithm to handle such propagations; and (3) it does not have the potential for having "unrefinable" nodes as in the unmodified newest-node algorithm. We note, however, that our algorithms are applicable to all three algorithms. Only a simple modification to our parallel implementation is required to implement the unmodified newest-node algorithm.

Williams [21] has developed a *voxel database* approach to the parallel mesh refinement problem. Our approach, which we present in the following section, differs from his approach in that we have explicit parallel runtime bounds. In addition, we have designed our approach to yield data structures that are more suitable to the assembly of the sparse linear systems that arise from these meshes as well as the solution of these sparse linear systems by sophisticated iterative and direct sparse factorization methods.

$i = 0$
$Q_i = S_k$ { *Q always denotes triangles not yet refined* }
$R_i = \emptyset$ { *R always denotes children of refined triangles* }
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 incompatible triangles embedded in $\cup_{j=0}^{i} Q_j$ are placed in R_{i+1}
    All other incompatible triangles are placed in Q_{i+1}
    $i = i + 1$
endwhile

FIG. 4. *The bisection algorithm*

2.2. The Bisection Algorithm. In Figure 4 we present the *bisection* algorithm. This algorithm is slightly altered, for ease of presentation, from Algorithm 5.6 as presented by Rivara in [17]. However, this modified algorithm yields the same final mesh as the original algorithm presented by Rivara.

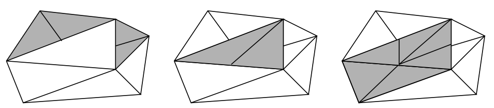

FIG. 5. *The process of the bisection algorithm is shown from left to right. In the initial mesh, the shaded triangles are refined; subsequently the shaded triangles are refined because they are not compatible.*

To illustrate the bisection algorithm, in Figure 5 we give an example of the propagation. Note that the refinement could propagate through unmarked triangles not adjacent to marked triangles before finishing. Rivara, however, has shown that this loop will terminate in a finite number of iterations. We denote this number of iterations by L_P. In general, L_P depends on the characteristics of the mesh being refined. Rivara also has shown that each triangle in T_k embeds

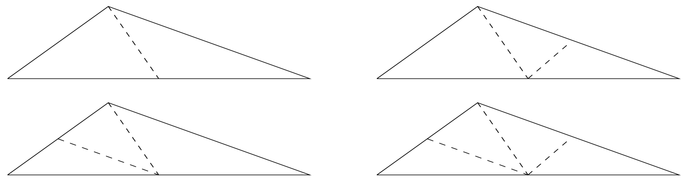

FIG. 6. *The possible divisions of a single triangle in the bisection algorithm*

1, 2, 3, or 4 triangles of the resulting compatible mesh, T_{k+1}. We show the possible 2, 3, or 4 resulting triangles in Figure 6. We formalize the following useful result from [17].

THEOREM 2.1. *During the execution of the bisection algorithm, no side of a triangle may be divided more than once.*

Proof: We first note that bisection of a triangle in R_i does not divide the side of another triangle because these bisections are done using a nonconforming edge. Therefore, an edge can be divided only when a triangle, as member of a set Q_i, bisects its longest edge. If a triangle is in Q_i, its children are excluded from Q_m for steps $m > i$ and thus may not divide their sides.

At the start of the algorithm the mesh is conforming. From the above discussion, an edge can be divided only when one of the original triangles in this mesh is in one of the sets Q_i. But the bisection of two original triangles using the same edge can divide an edge only once: the first bisection divides the edge, the second bisection makes the edge conforming. Therefore, an edge may be divided at most once by the bisection algorithm. □

3. Parallel Adaptive Refinement. In this section, we present a parallel algorithm for adaptive refinement that correctly implements the bisection method. We illustrate the key aspect of this algorithm: the synchronization necessary for the correct parallel execution of the bisection algorithm. Finally, we give an analysis of the algorithm under the PRAM computation model.

First, we need the following definitions. 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. We assume that the initial and final meshes are conforming. However, intermediate meshes can be nonconforming; hence, we allow for these nonconforming elements in our definition. Let $G = (V, E)$ be the graph associated with the mesh, with edges $E = \{(v_i, v_j) \mid \exists\, t_a \in T \text{ with } v_i, v_j \in t_a\}$. Let $D = (T, F)$ be the dual graph associated with the mesh, where $F = \{(t_a, t_b) \mid \exists\, e \in E \text{ with } e \in t_a, t_b\}$.

The refinement algorithm will be formulated within the context of the dual graph. To begin the PRAM analysis, we assume that at any given time we have as many processors as we have triangles and that triangle t_a is assigned to the processor p_a. For the analysis that follows, the specific PRAM computational variant does not make a difference; one may assume that the Concurrent Read, Exclusive Write (CREW) PRAM model is used. Recall that the PRAM computational model assumes that processors communicate through a common shared memory. The CREW variant assumes that different processors are able to read data from same location at the same time, but that only one processor at a time can write data to a location. The essential aspect of the PRAM model that we use in our analysis is that a processor can access data computed on another processor and stored in shared memory in constant time.

Some synchronization must be managed during the execution of the algorithm to maintain the correct neighbor information in both the graph G and the dual graph D as they are modified. Thus, each processor p_a must keep track of the current neighbors of t_a in D. We note that the correct neighbor information for G can be constructed in a straightforward way from D.

To illustrate the synchronization required for the correct execution of the parallel algorithm, we note the two ways that neighbor information can be corrupted. First, two different processors must not create vertices at the same location when bisecting their triangles. If two vertices are created at the same location, a postprocessing step must be included to merge these vertices. We eliminate the need for postprocessing by proper synchronization. In Figure 7 we see an example of two processors, P_1 and P_2, creating two vertices at the same location. Second, we must ensure

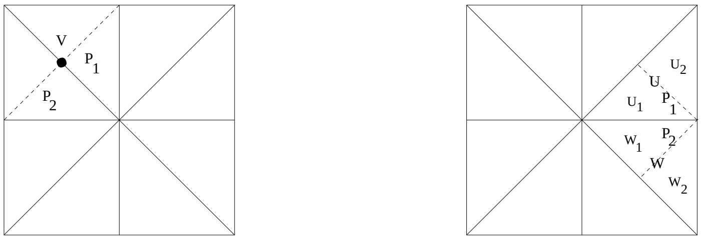

FIG. 7. *On the left, two processors creating a vertex at the same location; on the right, a possible corruption of neighbor information*

that outdated neighbor information is not propagated. For example, in the same figure we see that triangle U_1 may believe that triangle W is its neighbor, rather than triangle W_1, if triangles U and W are refined simultaneously.

The key observation is that both of these synchronization problems can be avoided if only triangles from an independent set in D are refined simultaneously. An independent set, I, is a subset of triangles of T such that no two triangles from I are adjacent in D. Once these triangles are refined and neighboring triangles are notified, another independent set can be chosen for refinement based on the correctly updated neighbor information. In the following subsection, we consider two possible approaches for computing these independent sets in parallel.

3.1. Two Methods for Computing Independent Sets in Parallel. For the purpose of the running time analysis for the refinement algorithm, we review two approaches for computing the independent sets. Both of these approaches require that the graph be of bounded degree— which is true for the problem we consider. The first approach uses an assignment of random numbers to the vertices of a graph to obtain a sequence of independent sets that is a slowly growing function of the size of the graph. The second approach is to compute a graph coloring and use this coloring to generate the independent sets. The advantage of the coloring approach is that we can guarantee that the number of colors, and thus the number of independent sets, is independent of the size of the graph for a bounded degree graph. However, the computation of this coloring requires the use of the first random number approach; therefore, the coloring is useful only if it is used enough times to justify the initial expense.

First we consider the use of independent random numbers to generate the independent sets. Suppose we wish to compute the sequence of independent sets for the set of triangles $T^{'}$, a subset of T, in the corresponding subgraph $D^{'} = D(T^{'})$. For each triangle t_a in $D^{'}$ we assign a distinct, independent random number $\rho(t_a)$. We choose an independent set I from $T^{'}$ according to the following rule: $t_a \in I$ if for each of its neighbors t_b in D, we have that either (a) $t_b \notin T^{'}$ or (b) $\rho(t_a) > \rho(t_b)$. We then update the set of triangles under consideration by deleting the independent set: $T^{'} \leftarrow T^{'} \setminus I$. We are now free to generate the next independent set in the sequence using the same rule. This process continues until $T^{'}$ is the empty set. The expected number of independent sets is given by the following lemma.

LEMMA 3.1. *Let $D^{'}$ be a bounded degree, undirected graph with n vertices. Suppose each vertex t in $D^{'}$ is assigned a unique independent random number $\rho(t)$. Consider the sequence of independent sets generated by the above rule. The expected number of these independent sets is bounded by $EO(\log n/\log\log n)$.*

Proof: This bound is a consequence of Corollary 3.5 in [9]. □

We now consider a second approach for obtaining the sequence of independent sets. First, note that any valid coloring of the graph $D = (T, F)$ can be used to generate the required independent sets. Recall that the function $\sigma : T \rightarrow \{1, \ldots, s\}$ is an s-coloring of D, if $\sigma(t_a) \neq \sigma(t_b)$ for all edges $(t_a, t_b) \in F$. Thus, a sequence of s independent sets can be generated from an s-coloring of D by assigning all triangles of the same color to one of the sets.

To efficiently compute this coloring, we use the parallel greedy heuristic presented in [9]. An outline of this heuristic is presented in Figure 8. The independent sets required for this heuristic can be generated by using the random number method described above. The greedy step in the heuristic is the color assignment; the smallest consistent color for t is the smallest color not assigned to a neighbor of t.

$T^{'} \leftarrow T$
While $T^{'} \neq \emptyset$ **do**
    Choose an independent set I from $T^{'}$
    Color I in parallel by choosing the smallest
        consistent color $\sigma(t)$ for each $t \in I$
    $T^{'} \leftarrow T^{'} \setminus I$
enddo

FIG. 8. *Outline of a parallel greedy coloring heuristic*

The advantage of using a coloring to generate the independent sets is that for a bounded degree graph the maximum number of colors is independent of the size of the graph. We include this well-known result as the following lemma.

LEMMA 3.2. *Consider a bounded degree graph, D, of maximum degree Δ. The parallel greedy coloring heuristic computes an s-coloring of D with $s \leq \Delta + 1$.*

Proof: Every vertex t is colored in the greedy heuristic by assigning it the smallest consistent color. Since, at worst, every neighbor of t is a different color, the maximum color assigned t is the degree of t plus one. Thus, the maximum color assigned by the greedy heuristic to any vertex in D is $\Delta + 1$. □

In sum, we have available two methods for generating the sequence of independent sets required for the parallel refinement algorithm. For the following PRAM running time analysis, it turns out that the best running time bound is obtained by maintaining a coloring of the dual graph comprising the triangles to be refined. However, in practice, the overhead associated with maintaining the coloring is not advantageous. Hence, the first approach is used in the practical algorithm presented in §4.

3.2. A PRAM Adaptive Refinement Algorithm. In Figure 9 we present a PRAM algorithm that avoids the synchronization problems discussed above, by simultaneously refining triangles from independent sets in D. Note that the independent sets used for refinement are also used to update the coloring. This update is required because the dual graph is modified after the bisection of a triangle. In the remainder of this section, we show that this algorithm avoids the two possible synchronization problems and has a fast run time.

We assume that the initial dual graph, D, is of bounded degree. In fact, because the triangulation of a surface is of primary interest, we assume that each triangle edge is shared by at most two triangles in the initial triangulation. In this case we have that maximum degree of the initial conforming mesh is three. The fact that D has bounded degree not only is useful in the following runtime proof, but also is useful in practice. Design of data structures and software is simplified if the maximum number of neighbors in the graph is bounded by a small constant. We now show that degree of any intermediate, nonconforming dual graph is bounded by at most twice the initial maximum degree.

LEMMA 3.3. *The dual graph, D, is of bounded degree at all times during the execution of the algorithm. In fact, the degree of a vertex in D never exceeds six. As a result, the maximum number of colors required at all times during the execution of the algorithm to color D is seven or less.*

Proof: From Theorem 2.1 each triangle edge is divided at most once. Therefore, a triangle can at most double the number of its neighbors. By Lemma 3.2, if the maximum degree of the dual is six, at most seven colors will be required during the execution of the refinement algorithm. □

Because the degree of the dual graph remains bounded, we note that the work assigned to each processor during one pass through the inner loop of this algorithm can be done in constant time under the PRAM computational model. We now show that the two possible corruption problems discussed above cannot occur.

$i = 0$
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 unique random number, $\rho(t_j)$
The subgraph $D(Q_0)$ is colored by the parallel greedy coloring heuristic
$R_0 = \emptyset$
While $(Q_i \cup R_i) \neq \emptyset$ **do**
    $W_i = Q_i$
    While $(Q_i \cup R_i) \neq \emptyset$ **do** $\{$*inner loop*$\}$
        Choose an independent set in D, I, from $(Q_i \cup R_i)$
        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
        Each new triangle, t_j, is assigned the smallest consistent
            color, $\sigma(t_j)$, and a new processor
        Each processor owning a bisected triangle updates this
            information on processors owning adjacent triangles
        $Q_i = Q_i \setminus (I \cap Q_i)$
        $R_i = R_i \setminus (I \cap R_i)$
    Endwhile
    $R_{i+1} =$ All incompatible triangles embedded in $\cup_{j=0}^{i} W_j$
    $Q_{i+1} =$ All other incompatible triangles
    $i = i + 1$
Endwhile

FIG. 9. *Parallel algorithm for refinement*

LEMMA 3.4. *Neighbor information in the dual graph is correctly updated during the execution of the refinement algorithm.*
Proof: The proof is by induction. We assume that the initial neighbor information is correct and that the neighbor information is correct following step $i-1$. If t_a is being refined at step i, by the properties of the independent set none of its neighbors in D are being refined. The triangles, t_{a_1} and t_{a_2}, resulting from bisection of t_a have correct information about their neighbors. The former neighbors of t_a can, therefore, be notified of the refinement of t_a and be given the correct information about their new neighbors. Thus, following step i of the refinement algorithm the modified neighborhood information for D is correct. $\square$

LEMMA 3.5. *No two vertices will be created at the same position during the execution of the refinement algorithm.*
Proof: Again, the proof is by induction. We assume that all vertices are unique initially and following step $i-1$. For a vertex to be created at the same position at step i by two different processors, one of two situations must occur: (1) two processors must simultaneously refine the same edge, or (2) a processor must refine a previously refined edge because it has not been notified that a vertex has been created on that edge. The first condition is prevented by the definition of the independent set—no adjacent triangles are refined simultaneously. The second condition is prevented by the correct notification of neighbor information in D ensured by Lemma 3.4. $\square$

Finally, we give a bound on the expected running time of the refinement algorithm.

THEOREM 3.6. *Recall that L_P is the number of loop iterations in the serial bisection algorithm in Figure 4. The algorithm given in Figure 9 terminates in a finite number of steps and has an expected run time under the PRAM computational model of $EO(\frac{\log|Q_0|}{\log\log|Q_0|}) + O(L_P)$.*

Proof: First, we consider the expected running time to compute the initial coloring of $D(Q_0)$. By Lemma 3.1 this time is $EO(\frac{\log|Q_0|}{\log\log|Q_0|})$.

Next, we consider the running time for the inner loop of the refinement algorithm at step

i. Define the graph, $D_i = (S_i, F_i)$, where S_i is the set of triangles $Q_i \cup R_i$ to be refined at this step. The set F_i is the subset of edges (t_a, t_b) from F with $t_a, t_b \in S_i$. By Lemma 3.3, we know that D_i is always a bounded degree graph. Also by Lemma 3.3 the number of colors required, and thus number of independent sets, is bounded by a constant. Hence the work assigned to any processor in the inner loop (the bisection of its triangle, updating the coloring, and the neighbor notification) takes time bounded by a constant independent of the mesh size.

Finally, we must show that it takes L_P iterations of the outer loop to form a conforming mesh. Clearly, every triangle that becomes incompatible at step i is refined at step $i+1$, just as in the sequential algorithm in Figure 4. Thus, the number of iterations of the outer loop in each algorithm is identical, L_P.

Hence, the total expected running time for the entire algorithm is bounded by $EO(\frac{\log|Q_0|}{\log\log|Q_0|}) + O(L_P)$. $\square$

We close this section with several notes about this running time analysis. First, since typically in this context the initial mesh to be refined was obtained from a previous level of refinement, the initial coloring step would not be required. Instead, a coloring of the mesh could be maintained between levels of refinement. Hence, the PRAM running time of the algorithm would be $O(L_P)$.

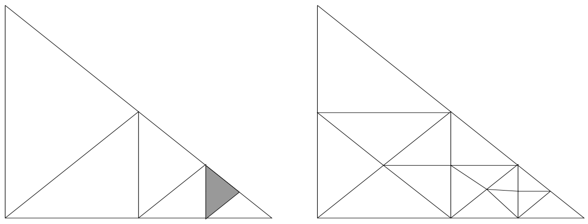

FIG. 10. *On the left, the shaded triangle is marked for refinement; on the right, the resulting conforming mesh after refinement. Note that the refinement has propagated through every triangle in the mesh but one.*

Finally, we close the analysis with some comments about L_P, the length of propagation. It is easy to construct a worst case example where $|Q_0| = O(1)$ and $L_P = O(|T|)$. For example, in Figure 10 we give an example that can be generalized to illustrate this worst case behavior. However, another way of looking at this example is to assume that this particular mesh was generated from previous refinements starting from a single triangle. In this case, the *average* length of propagation, L_P, over all levels of refinement, is actually constant. Furthermore, as we note with the experimental results presented in §5, the average number of independent sets required to obtain a conforming mesh appears to bounded by a small constant, independent of the size of the mesh. Thus, we believe that this possible worst case behavior of L_P is not the ominous problem that it appears it could be in a practical implementation.

One possible way to avoid this propagation is to modify the bisection algorithm by including a variation of the green edges used in the regular refinement approach. The modification requires a two green edge rule for the case when a triangle has two incompatible sides. If there are three incompatible sides, one can either do a regular refinement or use the bisection rule that divides all three sides. To do a level of refinement, one removes the green edges from the previous level of refinement, marking all triangles that had green edges in addition to triangles marked for refinement by the usual rule. All marked triangles are then bisected by using the longest-edge rule. Finally, green edges are added to make the mesh conforming. We did not use this approach because the refinement propagation was not observed to be a problem in our experiments; however, this modification of the refinement algorithm would have $L_P = 1$. One disadvantage of this modification is that it would require the additional software overhead to add and delete the green edges from the adaptive mesh data structures.

4. Distributed-Memory Implementation. For use on a practical parallel computer, we must modify the PRAM algorithm analyzed in the preceding section. Rather than assigning a single triangle or vertex to each processor, we assign a set of vertices and triangles to each processor. The vertices V are partitioned into disjoint subsets V_j, where processor j owns the subset V_j, and we have that $V = \bigcup_{j=1}^{p} V_j$. We choose to partition the vertices rather than the triangles because we have found that it makes the finite element evaluation, mesh refinement, and sparse matrix assembly and solution (if necessary) more straightforward and efficient. Based on the partitioning of V, we determine a partitioning of $T = \bigcup_{j=1}^{p} T_j$ into disjoint subsets where processor j owns the subset T_j. In practice, one can construct these sets so that at least one vertex of a triangle in T_j is in the set V_j.

For communication purposes, each processor, j, stores the set of triangles $\bar{T}_j = T_j \cup adj_D(T_j) \cup T(V_j)$, where $adj_D(T_j)$ is the set of triangles adjacent to a triangle in T_j in the dual graph D, and $T(V_j)$ is the set of triangles containing a vertex in V_j. In addition, processor j stores the set of vertices $\bar{V}_j = V(\bar{T}_j)$, where $V(\bar{T}_j)$ is the set of vertices contained by all triangles in $\bar{T}_j$.

Given the sets $\bar{V}_j$ and $\bar{T}_j$, processor j has all the information necessary to evaluate all finite elements that have vertices in V_j, assemble complete rows and/or columns of a sparse matrix associated with each vertex in V_j, and perform the parallel refinement algorithm (yet to be specified) on the triangles in T_j. We illustrate these sets for some processor j in Figure 11. In this figure we have partitioned the vertices by the geometric cuts represented by the orthogonal dashed lines. The vertices in the interior of the four dashed lines have been assigned to processor j—the set V_j—and are shown as filled vertices. The set $\bar{V}_j$ is the set of unfilled and filled vertices, T_j is the set of shaded triangles, and $\bar{T}_j$ is the set of unshaded and shaded triangles.

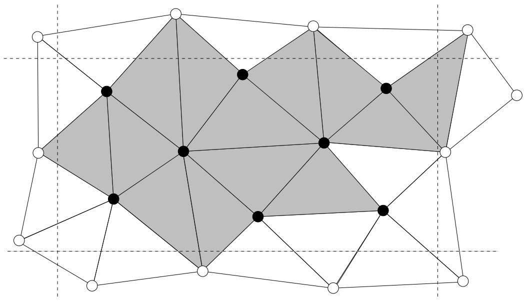

FIG. 11. *An illustration of the sets $\bar{V}_j$ and $\bar{T}_j$ maintained on processor j. The set of vertices assigned to processor j, V_j, is shown as the set of filled vertices. The set of shaded triangles is the set T_j. The union of the set of shaded and unshaded triangles is $\bar{T}_j$; the union of the set of filled and unfilled vertices is $\bar{V}_j$.*

In Figure 12 we present a practical version of the PRAM algorithm given in Figure 9. The algorithm ensures that vertices are not created at the same location and that the sets $\bar{V}_j$ and $\bar{T}_j$ on each processor j are correct. Note that, in this modified algorithm, if a triangle or vertex is created on processor j, processor j is its owner. The inner and outer loops in the PRAM algorithm have been combined into a single loop for greater efficiency; the separate loops allowed for a clearer presentation of the runtime bounds, but that is not necessary in this section.

For this algorithm, independent sets are chosen according to a slightly different rule from the rule used in the PRAM algorithm. The triangle t_a is in I_j if, for each of its neighbors t_b in D, one of the following hold: (a) $t_b \notin (Q \cup R)$, (b) $t_a, t_b \in T_j$, or (c) $\rho(t_a) > \rho(t_b)$. This modification allows two triangles on the same processor to be refined on the same step. The computation of the independent sets requires no communication because each processor has all the necessary information in $\bar{T}_j$ for this computation. Communication of the random numbers is not necessary if the seed given the pseudo-random number generator used to determine $\rho(t_a)$ is based solely on a. It is possible that a random number generator could assign the same number to neighboring triangles. In this case, the tie can be broken by using the global number a assigned to the triangle. Thus, the only communication necessary in the algorithm is the notification of

Based on local error estimates an initial set of triangles, Q,
is marked for refinement
Each triangle, $t_a \in Q$, is assigned a unique random number, $\rho(t_a)$
$R = \emptyset$
$W = \emptyset$
While $(Q \cup R) \neq \emptyset$ **do**
Choose an independent set in D, $I = \bigcup_{j=1}^{p} I_j$, from the
triangles in $(Q \cup R)$, where $I_j = I \cap T_j$
$W = W \cup I$
Each processor, j, bisects the triangles in I_j
embedded in Q across its longest edge
Each processor, j, bisects the triangles in I_j
embedded in R across a nonconforming edge
For each new triangle, t_b, a new random number, $\rho(t_b)$, is chosen
Each new triangle, t_b, created on processor j is added to T_j
Each new vertex, v_k, created on processor j is added to V_j
For each triangle, $t_a \in I_j$ notification of bisection is sent to each processor
l for which $((adj_D(t_a) \cap T_l) \neq \emptyset)$ or $((V(t_a) \cap V_l) \neq \emptyset)$
Each processor receives notification and updates its sets $\bar{V}_j$ and $\bar{T}_j$
$R = (R \setminus (I \cap R)) \cup$ Any incompatible triangles embedded in W
$Q = (Q \setminus (I \cap Q)) \cup$ All incompatible triangles not in R
Endwhile

FIG. 12. *A practical parallel algorithm for refinement*

bisections and the global reduction required to determine whether $(Q \cup R) \neq \emptyset$. For further efficiency, the notification messages can be packed so that each processor receives at most one message from another processor during each time through the while loop.

Because the modified algorithm in Figure 12 uses essentially the same synchronization scheme presented in §3, collisions are avoided, and neighbors in D on separate processors are not simultaneously bisected. Thus, we have the following theorem.

THEOREM 4.1. *All changes made by other processors to the triangles/vertices in the sets $\bar{T}_j$ and $\bar{V}_j$ on each processor j are received so that these sets are kept updated throughout the algorithm in Figure 12.*

Proof: The sets T_j and V_j are updated correctly because only processor j can bisect triangles in this set or create new vertices in this set. Any changes to triangles or vertices in $\bar{T}_j$ and $\bar{V}_j$ attributable to changes in triangles or vertices in $adj_D(T_j)$ and $V(T_j)$ are directly communicated in the algorithm.

The remaining portion of $\bar{T}_j$, attributable to $T(V_j)$, is accounted for because, if $t_k \in T(V_j)$ is bisected on another processor, then $V(t_k) \cap V_j \neq \emptyset$, and notification of this bisection will be sent and received.

Finally, to show that the vertex neighbor information is correct, we note that the neighbor information is correct on the subgraph of D induced by $\bar{T}_j$. Thus, the neighbor information contained in the subgraph of G induced by $\bar{V}_j$ must also be correct because $\bar{V}_j = V(\bar{T}_j)$. □

5. Experimental Results. In this section computational results are presented that demonstrate that the parallel refinement algorithm is scalable and that its execution time is negligible compared with that of other computations required to solve a PDE.

The parallel refinement algorithm is implemented as a subroutine library that can be called by an application program. Chameleon [6] is used to achieve portability across several architectures, including the Intel DELTA, which is the focus of this section. Note that in addition to the refinement algorithm, the subroutine library also includes a similarly constructed, parallel unrefinement algorithm. Because the unrefinement algorithm is necessary in many applications, including one of those used here, and its performance is similar to the refinement algorithm, results from it are included here as well. Results are presented for the parallel refinement algorithm for two different two-dimensional PDEs: Poisson's equation and the equations for linear elasticity. These problems are solved on two different geometries.

5.1. Test Problems. Our first set of test problems models Poisson's equation

$$\begin{aligned} \nabla^2 \phi(x) &= f(x),\ x \text{ in } S, \\ \phi(x) &= 0,\ x \in \partial S, \end{aligned} \tag{5.1}$$

where S is a square domain and a linear finite element approximation is used. The function $f(x)$ is a Gaussian charge distribution centered at a point (S_x, S_y) inside the domain. The mesh is selectively refined according to the energy norm [12] until the estimate of the local error on each triangle is less than a specified tolerance. Further, the point (S_x, S_y) is moved several times, and a new solution/mesh is found from the old solution/mesh. This movement requires significant mesh refinement around the new charge position and unrefinement around the old position while the remainder of the mesh remains relatively constant.

The second problem considered is the linear elasticity equations for the plane stress problem, given (without inclusion of a load) as

$$\begin{aligned} \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} &= \frac{1+\nu}{2}\left(\frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 v}{\partial x \partial y}\right), \\ \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} &= \frac{1+\nu}{2}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 u}{\partial x \partial y}\right), \end{aligned} \tag{5.2}$$

where u and v are the x and y displacements, respectively. These equations are solved on a rectangular region with a central hole. One side of the region is constrained to have zero displacement, and a constant traction is applied to the opposite side. Again, linear finite elements are used to approximate these equations. The mesh is selectively refined according to the energy norm until the local error estimate for each triangle is less than a specified tolerance. The linear systems are solved by using the same code used for the Poisson problem.

In each problem set, the initial coarse mesh has approximately 200 nodes except when running on 128 and 256 nodes; in these cases the initial coarse meshes are approximately 2 and 4 times larger, respectively. For each of these problems, by carefully choosing the maximum tolerance for the local error estimator, one can determine the maximum number of vertices in the solution meshes. The following two problem sets have been constructed such that the final solution mesh for each successive problem has roughly twice as many vertices/triangles as in the previous problem. Information about the two problem sequences is given in Tables 1 and 2.

TABLE 1
A sequence of test problems based on the Poisson problem

Name	Maximum Number of Vertices in the Adaptive Mesh	Maximum Number of Triangles in the Adaptive Mesh	Ratio of Area of Largest Triangle to Smallest Triangle
POISSON1	2,673	5,268	256
POISSON2	5,176	10,260	512
POISSON3	10,238	20,330	512
POISSON4	20,296	40,412	1,024
POISSON5	40,292	80,294	1,024
POISSON6	80,116	159,872	2,048
POISSON7	159,758	318,948	2,048
POISSON8	318,796	636,882	4,096
POISSON9	636,738	1,272,344	4,096

TABLE 2

A sequence of test problems based on the linear elasticity equations for the plane stress problem

Name	Maximum Number of Vertices in the Adaptive Mesh	Maximum Number of Triangles in the Adaptive Mesh	Ratio of Area of Largest Triangle to Smallest Triangle
ELASTIC1	1,460	2,767	156
ELASTIC2	2,798	5,382	419
ELASTIC3	5,535	10,768	512
ELASTIC4	10,736	21,043	1,677
ELASTIC5	21,325	42,041	2,048
ELASTIC6	41,930	82,985	4,096
ELASTIC7	83,308	165,387	6,443
ELASTIC8	165,182	328,594	16,384
ELASTIC9	329,087	655,691	12,886

Tables 3 and 4 give the number of refinement steps required for each problem during the solution process. A refinement step consists of finding an approximate solution to the PDE on the current mesh, T_k, by solving the sparse linear system arising from the finite element model, computing estimates for the local error at each triangle, and then refining T_k according to these estimates to obtain the conforming mesh, T_{k+1}. One observes that, not unexpectedly, it takes more mesh refinement steps to construct the larger meshes. In addition, the number of iterations through the loop in the algorithm in Figure 12 is given. The number of iterations should be at least L_P and perhaps a slowly growing function of the mesh size because we use the random number rule to generate the independent sets. One notes that the number of loop iterations needed is a slowly growing function of the number of processors and problem size. This result indicates that one can, in general, achieve scalable performance, as may be expected from Theorem 3.6.

TABLE 3

Number of refinement steps and loop iterations for the sequence of Poisson test problems

Name	Number of Processors	Number of Refinement Steps	Average Number of Loop Iterations
POISSON1	1	14	1.64
POISSON2	2	14	2.14
POISSON3	4	17	2.24
POISSON4	8	18	2.17
POISSON5	16	19	2.47
POISSON6	32	20	2.40
POISSON7	64	23	2.57
POISSON8	128	24	2.46
POISSON9	256	23	2.35

For the POISSON problem set, the charge location was moved twice; this movement meant that at two solution steps the mesh was not only refined around the charge, but also unrefined around the old charge position. However, there were still more refinement operations/steps than unrefinement operations. No unrefinement was necessary in the ELASTIC problem set; the load function was unchanged.

5.2. Mesh Partitioning and Linear System Solution. Although the primary focus of this paper is the adaptive refinement algorithm, it is important to examine the performance of the algorithm both individually and from the context of the complete problem solution. Thus, we

TABLE 4
Number of refinement steps and loop iterations for the sequence of linear elasticity test problems

Name	Number of Processors	Number of Refinement Steps	Average Number of Loop Iterations
ELASTIC1	1	6	3.17
ELASTIC2	2	8	3.13
ELASTIC3	4	9	3.56
ELASTIC4	8	9	4.22
ELASTIC5	16	11	3.91
ELASTIC6	32	11	4.45
ELASTIC7	64	12	5.08
ELASTIC8	128	12	5.00
ELASTIC9	256	12	4.83

have included the mesh partitioning, data movement after repartitioning, matrix assembly, and sparse linear solution in our experimental results. We have chosen approaches for these aspects that we feel are efficient and practical to provide a comparison of the relative costs of all aspects of the problem solution.

A good partitioning of the vertices and elements is necessary for the refinement algorithm to perform efficiently. A good partitioning ensures that grid points are evenly distributed to the processors in a way that minimizes communication costs and balances the computational load. 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. The work load can be balanced by assigning equal work load estimates to processors.

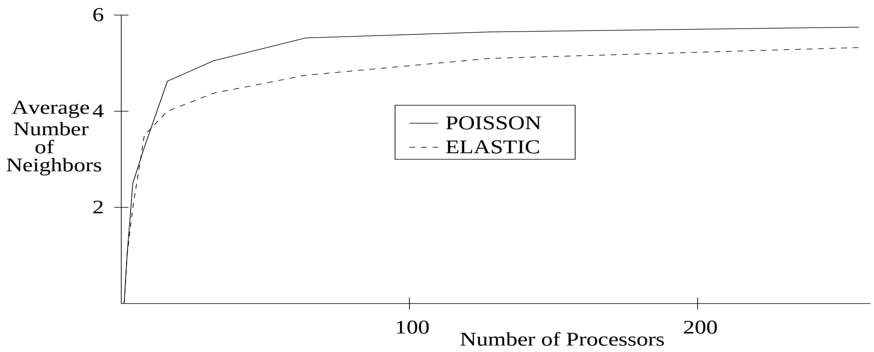

FIG. 13. *The average number of partitions to which each partition is adjacent in the final mesh*

Many good partitioning methods are available; a geometric partitioning heuristic [11] was chosen for this work. The goal of this heuristic is to maintain partitions that have a good geometric aspect ratio, the ratio of the partition's height to its width. This approach attempts to minimize the geometric length of each partition's boundary relative to its area and, consequently, minimize the amount of required communication.

In Figure 13 we show the average number of partition neighbors in the final meshes in the problem sequences. This information gives some sense of the number of processors each processor shares triangles with and must, therefore, exchange information with. In Figure 14 we show the maximum number of neighbors for a partition in the final mesh. If the maximum number of neighbors were too large relative to the average number of neighbors, this imbalance would represent a communication bottleneck. These results indicate that the communication imbalance between processors is not a significant problem. Figure 15 shows the percentage of the

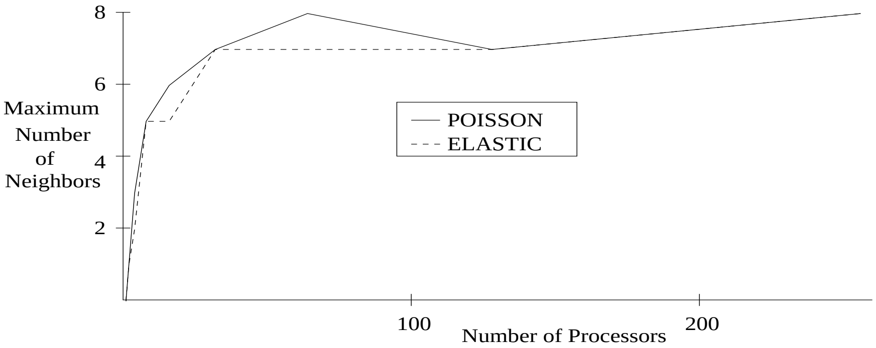

FIG. 14. *The maximum number of partitions to which a partition is adjacent in the final mesh*

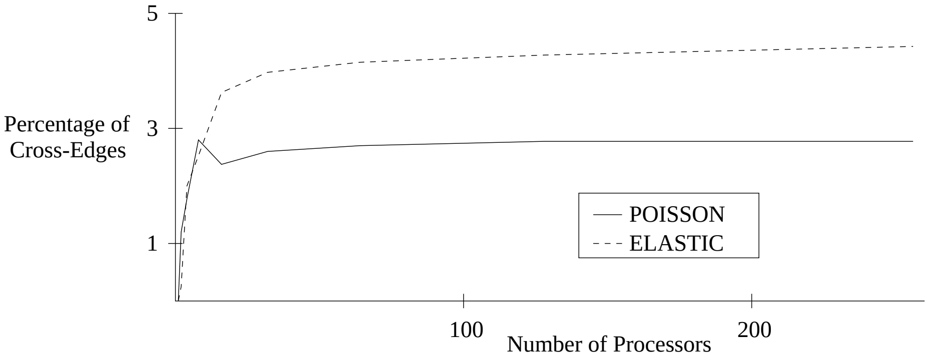

FIG. 15. *The percentage of cross-edges in the final mesh*

total triangle edges that have endpoints on two different processors. This data gives some sense of the number of triangles each processor has that must be coordinated with another processor. Note that these values initially rise rapidly, as one would expect, until approximately 16 processors are in use. For larger numbers of processors, these values increase very slowly. These figures demonstrate that the partitioning heuristic behaves in an essentially scalable manner.

The dominant computational task required to solve these problem sets is the solution of the linear systems. To solve these systems we use a parallel implementation of the conjugate gradient method preconditioned by an incomplete Cholesky factorization that is included in the BlockSolve software library [7]. This sparse solver software achieves good processor performance by extracting dense local submatrices (similar to a supernode approach) and using higher-level dense BLAS during the matrix multiplies and triangular system solves required by the conjugate gradient iteration. The matrix is ordered by using a block multicoloring to achieve scalable parallel performance [8].

Since a preconditioner based on an incomplete factorization is used, it is necessary to explicitly assemble the linear system. For the solution of the linear system we require that the unknowns and the corresponding matrix rows be partitioned among the processors. Because the unknowns are associated with mesh vertices, it is natural to partition vertices rather than elements and use this vertex partitioning for the linear system solution. Because the finite-elements used in these test problems are all of the same order and the vertex degrees do not vary dramatically, the number of nonzeros in row of the assembled sparse linear system is roughly constant. Thus, a partitioning that assigns equal numbers of vertices to processors does a good job at balancing the work load during the sparse linear systems solution. If the number of nonzeros in the matrix rows varied significantly, vertices could easily be weighted and a partitioning computed that assigned equal weights to processors to achieve load balancing.

As noted in the preceding section, if a vertex is assigned to a processor, the refinement algorithm maintains the information associated with any element that contains that vertex. Therefore, the matrix assembly can be accomplished without any interprocessor communication. The trade-off is that elements having vertices on more than one processor will be evaluated redundantly. One could use the partitioning of the elements to avoid this redundant element evaluation. However, these shared element matrices would have to be communicated between processors. For low-order elements, such as the ones used in these test problems, the redundant evaluation approach is probably more economical, since the element evaluation is cheap relative to the time required for interprocessor communication.

The linear systems are solved by running the conjugate gradient algorithm until the relative residual norm is less than a fixed tolerance of 10^{-5}. The starting vector for conjugate gradient iteration on the new refinement level is obtained by using the element basis functions to interpolate the previously computed solution onto the finer mesh. The use of the interpolated solution can dramatically decrease the number of iterations required by the conjugate gradient method. For examples, only 18 and 22 iterations are required respectively to solve the final two linear systems for the ELASTIC9 problem; the times required to solve these systems are shown in Figure 19. We note that approach is similar to the cascadic finite element method proposed by Deuflhard [4] except that the initial system is not solved exactly and a fixed tolerance is used to terminate the iterative method. For these problems the iteration error tolerance was chosen to be at least an order of magnitude smaller than the energy norm tolerance used for refinement. An empirical comparison of the problem solutions obtained with differing iteration tolerances showed that the iteration tolerance used was at least two to three orders of magnitude smaller than the minimum tolerance required to obtain well converged, asymptotic solutions. Thus, one could use a smaller iteration tolerance to slightly reduce the time required to solve the linear systems.

5.3. Experiments. The experiments were run on up to 256 nodes of the Intel DELTA. The DELTA parallel computer is a mesh-connected, 16×32 array of Intel i860 microprocessors.[1] In all of the experiments, the reported times are given in seconds. The operations rates indicate the number of bisections and vertex deletions (note that vertex deletions correspond to unrefinement and constitute a small percentage of the total) per second.

To demonstrate the scalability of the new algorithm and its implementation, we designed the sequence of problems in each test set to increase in size as the number of processors is increased. Specifically, the size of the final mesh (the number of vertices or the number of elements) divided by the number of processors the problem is run on is chosen to be roughly a constant. This fact can be seen in Tables 1 and 2, which show how many processors each problem was run on and the size of the final meshes.

Each of the test problems is refined in localized regions of the mesh; therefore, some processors have more refinement work than others. This load imbalance is reflected in Tables 5 and 6, which give the average number of operations per processor per step and the average of the maximum number of operations on a single processor per step. The average number of operations falls as the number of processors increases; this decrease results because more refinement steps are taken to achieve the same number of vertices per processor in the final mesh. The average maximum number of operations increases because, as the mesh size increases, more refinement is concentrated in the same size area in which a limited number of processors are working. Recall that the entire mesh is repartitioned after each refinement step. Thus, this concentration of new elements is continuously redistributed to processors with fewer elements.

However, even given these handicaps, the results demonstrate that the algorithm performs quite well. Figure 16 shows the average number of refinement operations per second per processor as a function of the number of processors. If refinement were occurring uniformly on all processors, one could expect this rate to be nearly constant; however, in the test problems, as in most practical problems, this is not the case. Figure 17 shows a more interesting rate, the maximum

[1] Note that because of constraints on the amount of time available to us on the DELTA, the 512-processor case was not run. We believe, however, that the results convincingly demonstrate the effectiveness of the parallel refinement algorithm.

TABLE 5
The number of refinement operations for the Poisson problem sequence

Name	Number of Processors	Average Number of Operations per Processor	Average of the Maximum Number of Operations per Processor
POISSON1	1	201	201
POISSON2	2	193	214
POISSON3	4	160	205
POISSON4	8	150	284
POISSON5	16	142	416
POISSON6	32	134	535
POISSON7	64	116	500
POISSON8	128	111	573
POISSON9	256	116	691

TABLE 6
The number of refinement operations for the linear elasticity problem sequence

Name	Number of Processors	Average Number of Operations per Processor	Average of the Maximum Number of Operations per Processor
ELASTIC1	1	214	214
ELASTIC2	2	164	166
ELASTIC3	4	149	208
ELASTIC4	8	147	272
ELASTIC5	16	120	251
ELASTIC6	32	119	296
ELASTIC7	64	108	284
ELASTIC8	128	107	326
ELASTIC9	256	107	318

number of refinement operations per second on an individual processor. One would expect this rate to remain constant, or nearly so, if the algorithm is perfectly scalable.

With the POISSON problem set, one sees very little degradation in the maximum refinement rate. One might expect some degradation resulting from the increasing number of neighbors each processor must exchange information with as the number of processors increases. However, with the POSSION problem set, the increase in the number of neighbors is offset by the rapidly increasing maximum number of operations per processor (given in Table 5). With the ELASTIC problem set, one observes the expected degradation because the maximum number of operations per processor is increasing only moderately. Prior to reaching 16 processors, the maximum rate of refinement is rapidly changing because of the increasing communication requirements as the number of processor neighbors and the percentage of cross-edges increases. After reaching 16 processors, the number of processor neighbors and the percentage of cross-edges stabilizes, and one sees approximately a 20% degradation in the rate of refinement from 16 to 256 processors.

Results given in Figure 18 demonstrate that, for a reasonably complex set of problems, the time to solve the linear systems dominates the time to refine the mesh for any number of processors. In fact, the total refinement time is always less than 4 percent of the total execution time. Note that a linear system is solved after each level of refinement. So, for example, the total execution time shown for 128 processors in Figure 18 includes the assembly and solution of 13 sparse linear systems. The time represented by the white region in the bar graph is composed almost entirely of the sum of the times required for the repartitioning of the mesh after each level of refinement. This partitioning time includes the time to move vertices, triangles, and the data

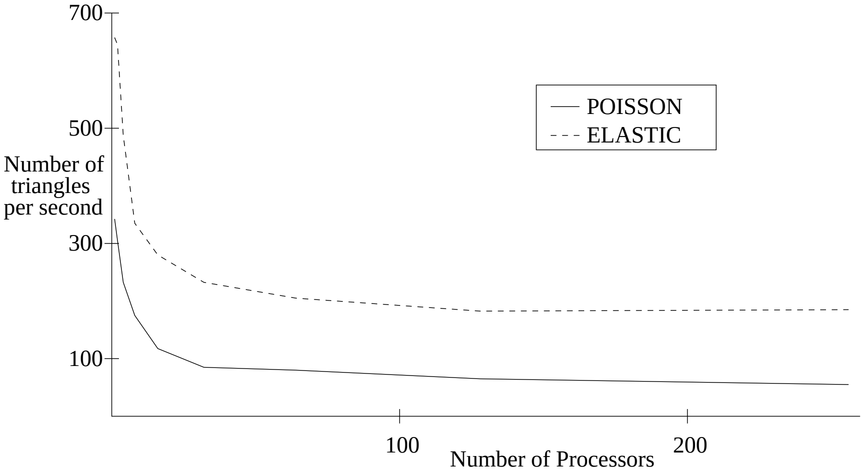

FIG. 16. *The average number triangles refined per processor per second*

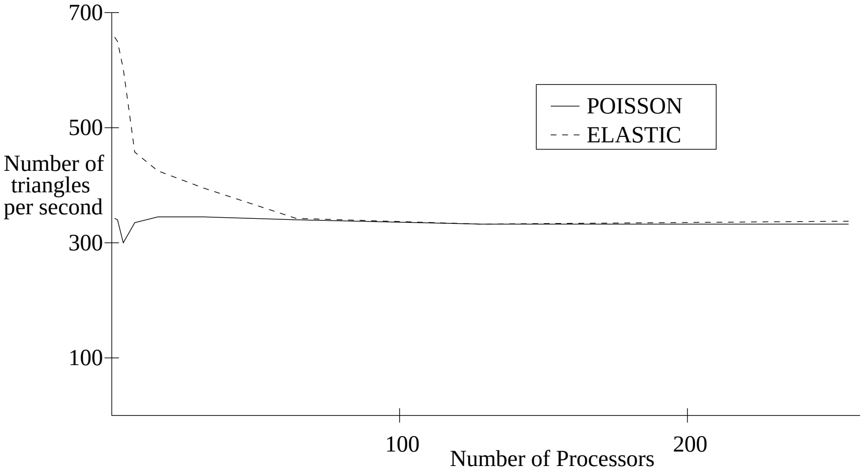

FIG. 17. *The maximum number of triangles refined per second on an individual processor*

associated with them between processors. We note that the implementation of our partitioning heuristic is preliminary; we believe that these times can be significantly reduced.

To examine the total running time in more detail, we consider one problem, ELASTIC9, run on 256 processors. In Figures 19 we show the time required to solve the linear system and the number of nonzeros in the assembled matrix as a function of the refinement level. Initially the matrix size is doubling after every level of refinement, since most triangles are bisected at each refinement step. However, for the last several refinement levels only small areas of the mesh are being refined. As a result, the interpolated solution from the previous mesh is an excellent initial guess to the solution on the refined mesh, and only a small number of conjugate gradient iterations are required to obtain a solution that satisfies the specified tolerance for the relative residual.

For the same problem, ELASTIC9 run on 256 processors, we show in Figure 20 the time required to refine the mesh as a function of the refinement level. In the figure we show the number of vertices that have been added to the mesh at that level of refinement. Note that refinement time continues to increase after the number of vertices reaches a maximum. This effect can be explained by noting that the areas of the computational domain on which refinement is occurring become confined to fewer processors as the mesh is refined. Recall from Figure 17 that it is the *maximum* rate of refinement on a processor that is constant. Thus, because the refinement is occurring on a smaller number of processors, the average rate of refinement is worse at the higher levels of refinement. This behavior explains the decrease in the average rate of refinement as a

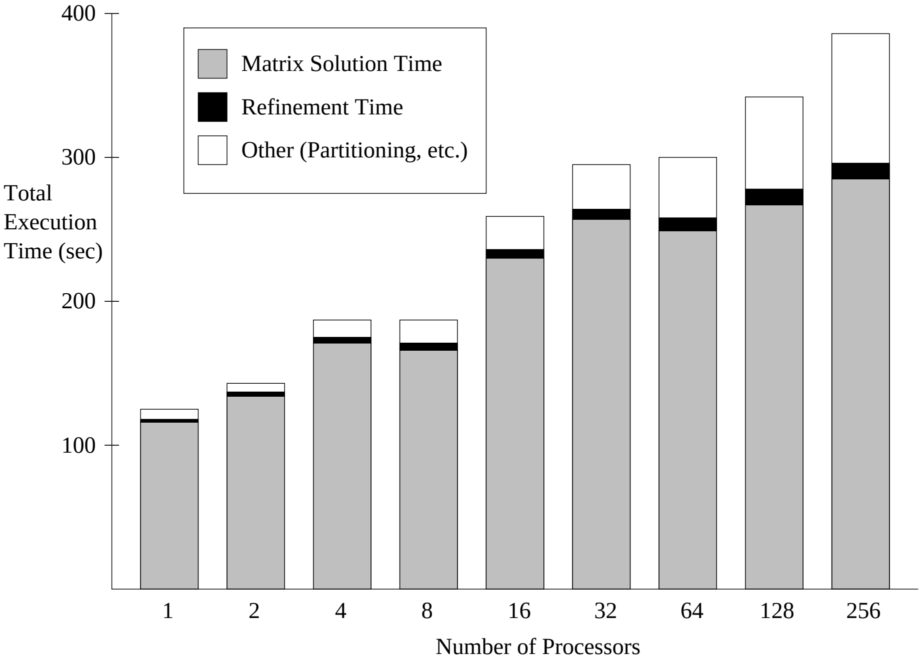

FIG. 18. *A comparison of the refinement times with the times required for all other aspects of the problem solution for the linear elasticity problem sequence*

function of the number of processors as shown in Figure 16.

6. Concluding Remarks. We have described a parallel algorithm for the adaptive refinement of meshes. This algorithm was shown to run in provably fast time under a PRAM model of computation. In addition, we described an efficient method of implementation for this algorithm on a practical, distributed-memory parallel computer. We then gave results for two problems that demonstrate the scalable nature of this algorithm.

The results given in this paper are for a two-dimensional triangular mesh. The use of independent sets for parallel synchronization, however, generalizes to the three-dimensional case as well as other refinement algorithms. The next logical step in this work is to develop theoretical results for three-dimensional tetrahedralizations as well as a practical, parallel implementation for three dimensions. In addition, we note that the use of higher-order basis functions is straightforward in this methodology; in fact, we include this functionality in the current parallel implementation [10].

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] R. E. Bank, *PLTMG: A Software Package for Solving Elliptic Partial Differential Equations. Users' Guide 6.0*, SIAM Publications, Philadelphia, Penn., 1990.

[3] R. E. Bank, A. H. Sherman, and A. Weiser, *Refinement algorithms and data structures for regular local mesh refinement*, in Scientific Computing, R. Stepleman et al., ed., IMACS/North-Holland Publishing Company, Amsterdam, 1983, pp. 3–17.

[4] P. Deuflhard, *Cascadic conjugate gradients for elliptic partial differential equations: Algorithm and numerical results*, in Proceedings of the 7th International Conference on Domain Decomposition Methods, AMS, 1993, pp. 29–42.

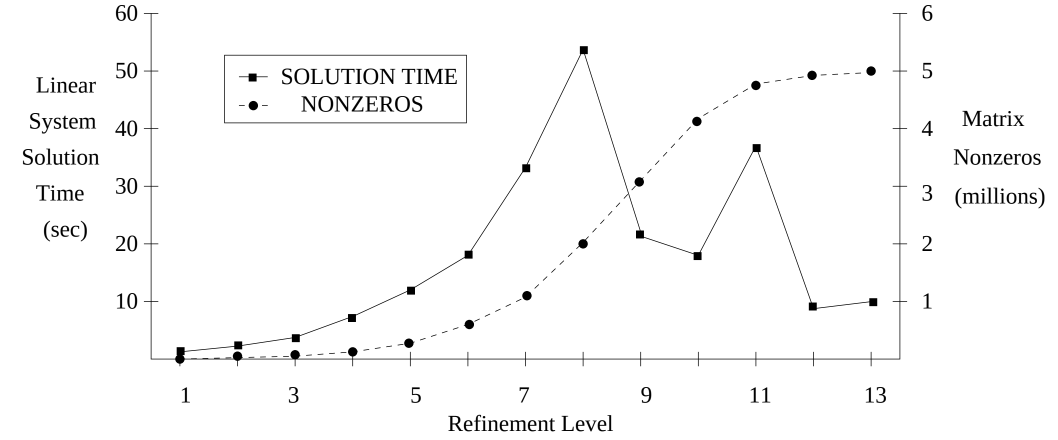

FIG. 19. *The time required to solve the linear systems and the number of nonzeros in the assembled linear system at each level of refinement for problem ELASTIC9 run on 256 processors of the Intel DELTA*

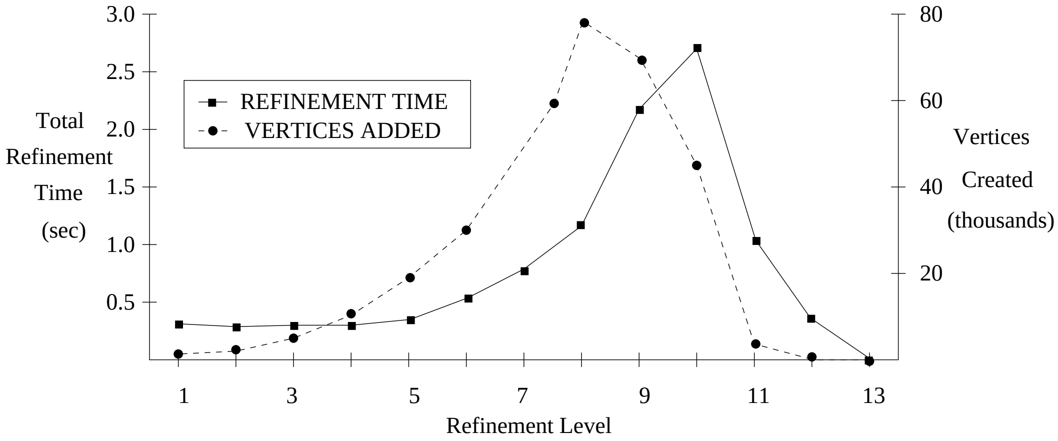

FIG. 20. *The refinement times and the number of vertices added at each level of refinement for problem ELASTIC9 run on 256 processors of the Intel DELTA*

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

[6] W. D. GROPP AND B. F. SMITH, *Users Manual for Chameleon Parallel Programming Tools*, ANL Report ANL-93/23, Argonne National Laboratory, Argonne, Ill., 1993.

[7] 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.

[8] ———, *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.

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

[10] ———, *Computational results for parallel unstructured mesh computations*, Computing Systems in Engineering, 5 (1994), pp. 297–309.

[11] ———, *Parallel algorithms for the adaptive refinement and partitioning of unstructured*

meshes, in Proceedings of the 1994 SHPCC, IEEE, 1994, pp. 726–733.

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

[13] R. V. Nambiar, R. S. Valera, K. L. Lawrence, R. B. Morgan, and D. Amil, *An algorithm for adaptive refinement of triangular element meshes*, International Journal for Numerical Methods in Engineering, 36 (1993), pp. 499–509.

[14] W. C. Rheinboldt and C. K. Mesztenyi, *On a data structure for adaptive finite element mesh refinements*, ACM Transactions on Mathematical Software, 6 (1980), pp. 166–187.

[15] M.-C. Rivara, *Algorithms for refining triangular grids suitable for adaptive and multigrid techniques*, International Journal for Numerical Methods in Engineering, 20 (1984), pp. 745–756.

[16] ——, *Design and data structure of fully adaptive, multigrid, finite-element software*, ACM Transactions on Mathematical Software, 10 (1984), pp. 242–264.

[17] ——, *Mesh refinement processes based on the generalized bisection of simplices*, SIAM Journal of Numerical Analysis, 21 (1984), pp. 604–613.

[18] ——, *Selective refinement/derefinement algorithms for sequences of nested triangulations*, International Journal for Numerical Methods in Engineering, 28 (1989), pp. 2889–2906.

[19] 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.

[20] E. G. Sewell, *A finite element program with automatic user-controlled mesh grading*, in Advances in Computer Methods for Partial Differential Equations III, R. Stepleman, ed., IMACS, New Brunswick, 1979, pp. 8–10.

[21] R. Williams, *A dynamic solution-adaptive unstructured parallel solver*, Report CCSF-21-92, Caltech Concurrent Supercomputing Facilities, California Institute of Technology, Pasadena, Calif., 1992.