Parallel Algorithms for the Adaptive Refinement and Partitioning of Unstructured Meshes *

Mark T. Jones
Computer Science Department
University of Tennessee
Knoxville, TN 37996

Paul E. Plassmann
Mathematics & Computer Science Division
Argonne National Laboratory
Argonne, IL 60439

Abstract

The efficient solution of many large-scale scientific calculations depends on adaptive mesh strategies. In this paper we present new parallel algorithms to solve two significant problems that arise in this context: the generation of the adaptive mesh and the mesh partitioning. The crux of our refinement algorithm is the identification of independent sets of elements that can be refined in parallel. The objective of our partitioning heuristic is to construct partitions with good aspect ratios. We present run-time bounds and computational results obtained on the Intel DELTA for these algorithms. These results demonstrate that the algorithms exhibit scalable performance and have run-times small in comparison with other aspects of the computation.

1 Introduction

Adaptive mesh refinement techniques have been shown to be very successful in reducing the computation and storage requirements for determining approximate solutions to many partial differential equations (PDEs) [9]. Rather than using a uniform 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* during the computation according to local error estimates on the domain. This technique is much more efficient than the use of structured meshes when the solution is changing much more rapidly in some areas than in others.

In this paper, we consider only two-dimensional simplicial meshes (i.e., a mesh of triangles). However, our algorithms are applicable to higher dimensions. Two basic algorithms exist for the refinement of triangles: (1) bisection, and (2) regular refinement. The bisection algorithm in its simplest form bisects the longest edge of a triangle to form two new triangles with equal area [10]. The regular refinement algorithm divides a triangle into four similar triangles or, during a cleanup phase, into two triangles [1]. Both algorithms must finish with a conforming mesh; in a conforming mesh the edge of a triangle cannot contain a vertex other than its endpoints.

In both methods, the refinement of a single triangle usually causes a *propagation* of refinement to other mesh elements. This propagation ensures that the final mesh is graded and conforming. The two methods differ in how they handle this propagation. In Figure 1, we show the operation of both of the algorithms as refinement occurs. The shaded triangles are triangles that have just been refined. The sequence of meshes from left to right shows how the refinement propagates to neighboring triangles.

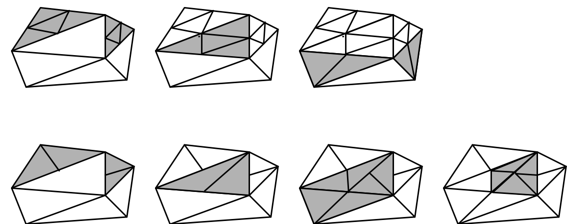

Figure 1: The refinement of a mesh using regular refinement (top row) and bisection of the longest side (bottom row). Follow the mesh sequences from left to right. The shaded triangles have just been refined in that step.

Both of these algorithms have been shown to perform well on a variety of problems; it is difficult to make a choice between the two [9]. We have chosen to discuss and implement the bisection algorithm be-

*This work was supported in part by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38.

cause of its simplicity; however, it is possible to modify the algorithms given in this paper for use with regular refinement.

Implicit in a discussion of parallel algorithms for mesh refinement on distributed-memory computers is the problem of partitioning the mesh to processors. Clearly, the performance of the parallel refinement algorithm depends critically on the partitioning used and on how this partitioning is adjusted as the mesh is refined. We have developed and implemented a new parallel partitioning heuristic called unbalanced recursive bisection (URB). The goal of this heuristic is to maintain partitions with good geometric aspect ratios; this property helps to minimize interprocessor communication during both refinement and the repartitioning of elements after refinement.

In short, we will discuss two algorithms in this paper: (1) a parallel algorithm for adaptively refining meshes, and (2) an algorithm for partitioning these *perturbed* meshes. Our underlying goal for these algorithms and implementations is that their run time for p processors be small relative to the solution time for the PDE on p processors, *independent of* p.

2 Parallel Adaptive Refinement

There are several variants of the serial bisection refinement algorithm. We consider the version given by Rivara [10]. This bisection algorithm bisects triangles across the largest edge (dividing the largest angle) with division of noncompatible edges after the triangle has already been bisected once. This algorithm has been shown to yield triangulations whose smallest angle is bounded by at worst one-half the smallest angle in the inital mesh [11]. The bisection refinement algorithm is given in Figure 2.

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

Figure 2: 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 [10].

Our parallel 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 $F = \{(t_a, t_b) \mid e_{i,j} \in t_a, t_b\}$.

We assume that the vertices are partitioned into disjoint subsets, $V = \bigcup_{i=1}^{p} V_i$, such that processor i owns V_i. 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 $T = \bigcup_{i=1}^{p} T_i$ into disjoint subsets. Each processor, i, stores the set of triangles $\bar{T}_i = T_i \cup adj(T_i) \cup T(V_i)$ and the set of vertices $\bar{V}_i = V(\bar{T}_i)$.

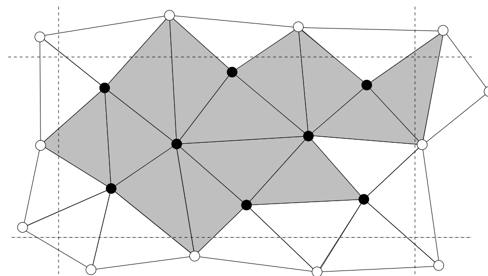

Figure 3: An illustration of $(\bar{V}_i, \bar{T}_i)$ on processor i

Given $(\bar{V}_i, \bar{T}_i)$, processor i has all the information necessary to evaluate all finite elements that have vertices in V_i, assemble complete rows/columns of a sparse matrix associated with each vertex in V_i, and perform the parallel refinement algorithm (yet to be specified) on the triangles in T_i. We give an illustration of these sets for processor i in Figure 3 where the subpartition of the domain for processor i is defined by the orthogonal dashed lines. V_i is the set of filled vertices, $\bar{V}_i$ is the set of unfilled and filled vertices, T_i is the set of shaded triangles, and $\bar{T}_i$ is the set of unshaded and shaded triangles. Note that $\bar{T}_i$ can contain a triangle with no vertices in V_i (for example, the triangle in the far upper right of Figure 3).

Also note that $\bar{V}_i$ can contain a vertex not contained in $T_i \cup adj(T_i)$ (for example, the vertex in the far left lower corner of Figure 3).

The serial bisection algorithm can run into two synchronization problems. First, if processor i refines triangle t_a and processor j refines an adjacent triangle t_b, it is possible that each processor could create a vertex at the same position. An example of such a collision is shown in Figure 4, where processors P_1 and P_2 are trying to refine adjacent triangles by bisecting the same edge. Second, to correctly perform operations such as sparse matrix assembly, vertices and triangles in $(\bar{V}_i - V_i, \bar{T}_i - T_i)$ must be properly updated when they are changed by their owners on other processors. An example of a situation where neighbor information may not be updated properly is given in Figure 5. Here processors P_1 and P_2 are trying to refine adjacent triangles simultaneously. Triangle U_1 may believe that W rather W_1 is its neighbor, and triangle W_1 may believe that U rather U_1 is its neighbor.

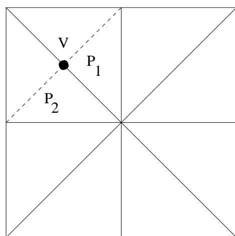

Figure 4: An example of a collision of neighbor information; processors P_1 and P_2 simultaneously create the vertex V.

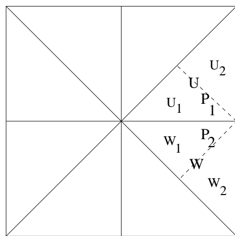

Figure 5: An example of incorrect updating of neighbor information; simultaneous creation of triangles U_1 and W_1 leads to incorrect neighbor information.

We solve these synchronization problems by refining independent sets of triangles on different processors. Independent sets are chosen according to the Monte Carlo rule: $t_a \in I$ if for each of its neighbors, t_b, in D, if (a) t_b not $\in Q_i \cup R_i$, (b) $t_a, t_b \in T_i$, or (c) $\rho(t_a) > \rho(t_b)$, The complete algorithm is given in Figure 6.

Based on local error estimates, a set of triangles, Q, is marked for refinement.
Each triangle, t_a, in Q is assigned a random number $\rho(t_a)$
$R = \emptyset$
While $(Q \cup R) \neq \emptyset$ **do**
  Choose an independent set in D, $I = \bigcup_{j=1}^{p} I_j$, from triangles in $(Q \cup R)$, where $I_j = I \cap T_j$
  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_b, in I_j on processor j, notification of bisection is sent to each processor l for which $((adj(t_b) \cap T_l) \neq \emptyset)$ or $((V(t_b) \cap V_l) \neq \emptyset)$
  Each processor receives notification and updates its $(\bar{V}_j, \bar{T}_j)$ accordingly
  $R = (R - (I \cap R)) \cup$ Any triangles embedded in Q made incompatible
  $Q = (Q - (I \cap Q)) \cup$ All other triangles made incompatible
Endwhile

Figure 6: A practical parallel algorithm for refinement

We also have algorithms for mesh de-refinement (removal of unnecessary triangles and vertices), but we omit these because of space constraints.

We now show that this algorithm has a fast expected run time under the P-RAM computational model. For this analysis we assume that we have as many processors as we have triangles.

Theorem: *The P-RAM version of this algorithm terminates in a finite number of steps and has an expected run time on a P-RAM of* $EO(\frac{\log Q_{max}}{\log\log Q_{max}}) \times L_P$ *where* $Q_{max} = \max_k |Q_k|$ *and* L_P *is the number of levels of propagation.*

Proof: We sketch the proof of this theorem as follows, a complete version is given in [8]. From [10], we have that L_P is finite. We know that the graph D_k is a bounded degree graph; in fact, any node (triangle) has at most three neighbors. Given that this is a bounded

degree graph, we use the algorithms and theorems in [7] to find $EO(\frac{\log n}{\log\log n})$ independent sets in a graph in time proportional to the number of independent sets, where n is the number of vertices in the graph. □

3 Mesh (Re-)Partitioning

In this section we present a new partitioning heuristic that uses geometric information to partition the vertex set into equal sets. The goal of this heuristic is to obtain partitions with good aspect ratios. First, we review the goals of a partitioning heuristic.

Let Π be a partitioning of the vertex set $V = \bigcup_{i=1}^{p} V_i$. Recall that there exists an edge (V_i, V_j) in the quotient graph $Q = G/\Pi$ if and only if there exists an edge (u, v) in G with $u \in V_1$ and $v \in V_2$. We would like to determine a partition, Π, with the following properties:

1. for load balancing, each partition should be nearly equal in size;
2. the cardinality of the set of cross edges, $CE = \{e_{i,j} \mid v_i \in V_k, v_j \in V_l, k \neq l\}$, is minimized to reduce the amount of data that must be communicated; and
3. the number of edges in Q is minimized to reduce the number of messages that must be sent, where $Q = (V_Q, E_Q)$, V_Q is the set of processors, and $E_v = \{(i,j) \mid \exists e_{k,l}, v_k \in V_i, v_l \in V_j\}$.

Typically, we expect to have an initial partitioning problem to solve and then many repartitioning problems that arise as the mesh is adaptively (de-)refined. Both problems have been studied, and many interesting methods have been proposed. The partitioning problem in our context has four important features: (1) the geometric location of every vertex is available, (2) the perturbations to the meshes caused by refinement are often small and localized, (3) vertices should be moved from processor to processor as little as possible, and (4) large amounts of time cannot be spent partitioning the meshes because we do not wish to dominate the execution time with mesh algorithms.

We propose an inexpensive method that utilizes geometric information and allows for existing partitions to be perturbed when a small perturbation to the mesh occurs. This method is a variation of the orthogonal recursive bisection (ORB) algorithm [2]. The ORB algorithm makes an initial cut to divide the vertices into two sets of equal size. Orthogonal cuts are then made recursively in the new subdomains until the vertices are equally distributed among the processors. Although this algorithm obtains good load balancing, it can result in less than optimal communication requirements. Long, thin partitions may be created that have a large number of edges crossing the partition boundaries. In addition, these long, thin partitions may cause Q to be very dense.

To address this problem, we have developed a modification of ORB which we call unbalanced recursive bisection (URB). Instead of dividing the vertices into equal sets, we choose the cut that minimizes partition aspect ratio and divides the vertices into $\frac{nk}{p}$ and $\frac{n(p-k)}{p}$ sized groups, where $n = |V|$, p is the total number of processors, and $k \in \{1, 2, ..., p-1\}$. This algorithm leads to an equal distribution of grid points with better partition aspect ratios than the ORB algorithm. In fact, we conjecture that the aspect ratio of the partition generated by the URB heuristic is largely independent of p, the number of processors, for the meshes arising from refinement algorithms that yield graded meshes. It has been shown that for the ORB heuristic, partitions can be adjacent to $O(\sqrt{p})$ other partitions [2].

To repartition a perturbed mesh without massive movement of vertices, we simply perturb the cuts in the existing partition to rebalance the sizes of V_i. If no such perturbation exists or if the perturbation would result in poor aspect ratios, then the mesh is repartitioned from scratch. As we will show in the results section, if the partition can be perturbed, the number of vertices that must move from one partition to another is small; otherwise most vertices will be moved (as in most partitioning algorithms).

One point worth noting is that methods, such as ORB and URB, that yield convex polygonal partitions make the scalable mapping of vertices to partitions trivial given the geometric location of a vertex. For methods that do not yield convex polygons, the mapping of vertices to partitions in a scalable manner becomes problematic.

Finally, we note that currently we do not handle the assignment problem in a sophisticated manner. We mainly rely on our target architectures to have wormhole routing that reduces the effect of distance between communicating nodes. We plan to utilize a more sophisticated technique such as that given in [5].

4 Experimental Results

We have implemented these algorithms in a library of routines that is called by an application program. The software uses Chameleon [4] to achieve portability across several architectures, including the Intel DELTA which is our focus here. We have tested our algorithms using three different PDEs (Poisson's equation, linear elasticity equations, and the nonlinear Ginzburg-Landau equations) on a variety of geometries. In this paper, we consider results obtained for the first two problems; results for the latter problem are presented in [3].

The first problem is given by

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = f(x,y) \text{ on } S \quad (1)$$

$$u = 0 \text{ on boundary} \quad (2)$$

on a square domain where $f(x, y)$ is a Gaussian charge distribution that forces refinement around a point (S_x, S_y). To test the implementation we move the point (S_x, S_y) several times and find a new solution/mesh from the old solution/mesh. This movement requires mesh refinement around the new position and de-refinement around the old position while the rest of the mesh remains nearly constant. To solve the linear systems arising from this problem, we use the parallel conjugate gradient method preconditioned by an incomplete factorization [6].

The second problem, planar linear elasticity, is given by the equations (forces are not included in these equations)

$$\begin{aligned} \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} &= \frac{1+\nu}{2}\left(\frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 v}{\partial x \partial y}\right) \quad (3) \\ \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} &= \frac{1+\nu}{2}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 u}{\partial x \partial y}\right) . \end{aligned}$$

These equations are solved on an annulus with a constant load on one side and the opposite side fixed. We use linear basis functions in our finite element formulation. We selectively refine the mesh according to the element energy norm until the local error estimate at each triangle is acceptable. The linear systems arising from the problem are solved with the same code we used for the Poisson problem.

The maximum sizes of the adaptive meshes generated for the two problem sets are given in Table 1. The column labeled $|V|_{\max}$ shows the maximum number of vertices obtained in the sequence of adaptive meshes; the column labeled $|T|_{\max}$ gives the maximum number of triangles. The final column, labeled Triangle Ratio gives the maximum sized area ratio between two triangles in the mesh.

Table 1: The problem set

Problem Name	$\|V\|_{\max}$	$\|T\|_{\max}$	Triangle Ratio
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

Problem Name	$\|V\|_{\max}$	$\|T\|_{\max}$	Triangle Ratio
ELASTIC1	1,460	2,767	156
ELASTIC2	2,798	5,382	419
ELASTIC3	5,534	10,766	512
ELASTIC4	10,736	21,043	1,677
ELASTIC5	21,329	42,049	2,048
ELASTIC6	41,936	82,997	4,096
ELASTIC7	83,349	165,468	6,443
ELASTIC8	165,253	328,736	16,384
ELASTIC9	329,201	655,919	12,886

In Table 2 we show the number of times each mesh was refined during the solution process; the number of processors used is given in the column labeled P. In addition, we show the number of iterations through the loop in the algorithm in Figure 6. We observe that, as expected, it takes more mesh refinement steps to construct the larger meshes. We also see that the number of loop iterations needed is a slowly growing function of the number of processors. This result indicates that we can, in general, expect scalable performance. Such a result is not surprising given the run-time results of theorem in the preceding section.

Note that a good partitioning of the vertices for each of these problems is necessary for our refinement algorithm to perform efficiently. In Table 3 we give the average number of subpartitions that are adjacent to a given subpartition (average degree of the quotient graph, Q). This measure gives some sense of the number of processors each processors overlaps triangles with and must exchange information with. Also given is the percentage of the total triangle edges that have endpoints on two different processors. This gives

Table 2: The number of refinement stages and average number of loop iteration per stage

Problem Name	P	Num. of Ref. Steps	Avg. Num. Loop Its.
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

Problem Name	P	Num. of Ref. Steps	Avg. Num. Loop Its.
ELASTIC1	1	6	4.00
ELASTIC2	2	8	4.13
ELASTIC3	4	9	4.67
ELASTIC4	8	9	4.56
ELASTIC5	16	10	5.00
ELASTIC6	32	11	4.91
ELASTIC7	64	12	5.67
ELASTIC8	128	15	5.13
ELASTIC9	256	12	5.00

Table 3: Experimental results showing the quality of the final partition obtained with the URB heuristic

Problem Name	P	Avg. Degree of Q	Percent. of Cross Edges
POISSON1	1	0.00	0.00
POISSON2	2	1.00	1.20
POISSON3	4	2.50	1.77
POISSON4	8	3.25	2.79
POISSON5	16	4.63	2.38
POISSON6	32	5.06	2.60
POISSON7	64	5.53	2.70
POISSON8	128	5.64	2.78
POISSON9	256	5.76	2.78

Problem Name	P	Avg. Degree of Q	Percent. of Cross Edges
ELASTIC1	1	0.00	0.00
ELASTIC2	2	1.00	0.24
ELASTIC3	4	2.00	2.02
ELASTIC4	8	3.50	2.52
ELASTIC5	16	4.00	3.56
ELASTIC6	32	4.38	3.99
ELASTIC7	64	4.75	4.26
ELASTIC8	128	5.20	4.27
ELASTIC9	256	5.34	4.35

some sense of the number of triangles each processor has that must be coordinated with some other processor. We see that these values rapidly rise, as one would expect, until approximately 16 processors. After this point, they very slowly increase with the number of processors.

Our experiments were run on up to 256 nodes of the Intel DELTA. The DELTA is a mesh-connected, 16×32 array of Intel i860 microprocessors. Because of constraints on the amount of time available to us on the DELTA, we did not run the 512-processor case. We believe, however, that the results presented convincingly demonstrate the effectiveness of our algorithms. All of our experimental rates and times given in Tables 5 and 6 are computed in seconds. Operations rates indicate the number of bisections and vertex deletions (note that vertex deletions correspond to de-refinement and constitute a small percentage of the total) per second.

To demonstrate the scalability of our algorithm and implementation, we ran both problem sets on the DELTA in such a way that the number of vertices in the final mesh assigned to an individual processor was constant. By nature, each of the test problems is refined in localized regions of the mesh; therefore, we expect that some processors will have more work than others. This is reflected in Table 4, where we give the average number of operations per processors 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 is because we are taking more refinement steps to achieve the same number of vertices per processor in the final mesh. The average maximum number of operations increases because, as we increase the mesh size, more refinement is concentrated in the same size area in which a limited number of processors are working.

Even given these handicaps, we show that the algorithm performs quite well. In Table 5 we concentrate on two different rates of refinement per processor. The first rate is the average number of refinement operations per second per processor. If refinement were occurring uniformly over all the processors, we could expect this to be nearly constant; however, in our test problems, as in most practical problems, this is not

Table 4: The average number of operations per processor per step and the average maximum number of operations on a single processor per step

Problem Name	P	Avg. Num. of Ops.	Avg. Max. Num. Ops.
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

Problem Name	P	Avg. Num. of Ops.	Avg. Max. Num. Ops.
ELASTIC1	1	214	214
ELASTIC2	2	164	166
ELASTIC3	4	149	108
ELASTIC4	8	147	273
ELASTIC5	16	132	276
ELASTIC6	32	118	297
ELASTIC7	64	108	284
ELASTIC8	128	86	263
ELASTIC9	256	107	304

the case. The second, and more interesting rate, is the maximum number of refinement operations per second per processor. We would expect this rate to remain constant, or nearly so, if the algorithm were perfectly scalable. In the POISSON problem set, we see very little degradation. In fact, we should expect some degradation as a result of the increasing number of neighbors each processor must exchange information with as the number of processors increases. This degradation is offset, however, by the rapidly increasing maximum number of operations per processor we see in Table 5. In the ELASTIC problem set we see the reasonable degradation that we expect because we do not have the rapidly increasing maximum number of operations per processor. After reaching 16 processors, the number of processor neighbors and the percentage of cross-edges stop this rapid increase, and we see approximately a 20% degradation in the rate of refinement from 16 to 256 processors.

In the final table of results, Table 6, we 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, we can see that the total refinement time is always less than one percent of the total execution time.

Table 5: The total time (sec) required for the sequence of adaptive meshes, the average rates of element refinement per processor, and the maximum rates of refinement

Problem Name	P	Total Ref. Time	Avg. Rate Ref. per Processor	Max. Rate Ref. per Processor
POISSON1	1	8.2	342	342
POISSON2	2	8.8	306	339
POISSON3	4	11.6	233	300
POISSON4	8	15.3	176	334
POISSON5	16	22.8	118	346
POISSON6	32	31.2	86	344
POISSON7	64	33.8	79	340
POISSON8	128	41.3	65	333
POISSON9	256	47.9	55	332

Problem Name	P	Total Ref. Time	Avg. Rate Ref. per Processor	Max. Rate Ref. per Processor
ELASTIC1	1	2.4	553	553
ELASTIC2	2	2.9	449	453
ELASTIC3	4	4.5	298	417
ELASTIC4	8	5.4	246	458
ELASTIC5	16	6.9	192	400
ELASTIC6	32	8.2	160	399
ELASTIC7	64	9.4	138	362
ELASTIC8	128	11.9	108	331
ELASTIC9	256	11.8	109	310

5 Conclusions

We have presented two new parallel algorithms: an algorithm for the adaptive refinement of meshes, and a partitioning heuristic. We have reviewed a result showing that the refinement algorithm has a provably fast running time under a P-RAM model of computation. In addition, we described an efficient method of implementation for this algorithm on a practical, distributed-memory parallel computer. We have given results for two problems that demonstrate the scalable nature of the refinement algorithm and the good geometric properties of the URB partitioning heuristic.

Table 6: Timing results (sec) for the entire sequence of adaptive meshes for the planar linear elasticity problem

Problem Name	P	Total Ref. Time	Total Matrix Solve Time	Total Time
ELASTIC1	1	2.4	275	278
ELASTIC2	2	2.9	433	437
ELASTIC3	4	4.5	604	612
ELASTIC4	8	5.4	677	688
ELASTIC5	16	6.9	971	989
ELASTIC6	32	8.2	1434	1461
ELASTIC7	64	9.4	2054	2090
ELASTIC8	128	11.9	4391	4458
ELASTIC9	256	11.8	5120	5196

The results given in this paper are for two-dimensional triangular meshes. However, the use of independent sets for parallel synchronization generalizes to the three-dimensional case and can be used to implement 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; we expect to include this functionality in the current parallel implementation.

References

[1] Randolph E. Bank, Andrew H. Sherman, and Alan Weiser. Refinement algorithms and data structures for regular local mesh refinement. In R. Stepleman et al., editor, *Scientific Computing*, pages 3–17. IMACS/North-Holland Publishing Company, Amsterdam, 1983.

[2] Marsha J. Berger and Shahid H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, C-36:570–580, 1987.

[3] Lori A. Freitag, Mark T. Jones, and Paul E. Plassmann. New advances in the modeling of high-temperature superconductors. In *1994 International Simulation Conference – Grand Challenges in Computer Simulation, La Jolla, California, April 11-15*. The Society for Computer Simulation, 1994 (to appear).

[4] William Gropp and Barry Smith. User's Manual for Chameleon Parallel Programming Tools. ANL Report ANL-93/23, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1993.

[5] Steven Warren Hammond. *Mapping Unstructured Grid Computations to Massively Parallel Computers*. Ph.D. thesis, Department of Computer Science, Rensselaer Polytechnic Institute, 1989.

[6] Mark T. Jones and Paul E. Plassmann. BlockSolve v1.0: Scalable library software for the parallel solution of sparse linear systems. ANL Report ANL-92/46, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1992.

[7] Mark T. Jones and Paul E. Plassmann. A parallel graph coloring heuristic. *SIAM Journal on Scientific and Statistical Computing*, 14:654–669, May 1993.

[8] Mark T. Jones and Paul E. Plassmann. Parallel algorithms for adaptive mesh refinement. Preprint MCS-P421-0394, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1994.

[9] William F. Mitchell. A comparison of adaptive refinement techniques for elliptic problems. *ACM Transactions on Mathematical Software*, 15(4):326–347, December 1989.

[10] Maria-Cecilia Rivara. Mesh refinement processes based on the generalized bisection of simplices. *SIAM Journal of Numerical Analysis*, 21(3):604–613, June 1984.

[11] Ivo G. Rosenberg and Frank Stenger. A lower bound on the angles of triangles constructed by bisecting the longest side. *Mathematics of Computation*, 29(130):390–395, April 1975.