THE SCALABILITY OF MESH IMPROVEMENT ALGORITHMS*

LORI F. FREITAG†, MARK T. JONES‡, AND PAUL E. PLASSMANN§

Abstract. In this paper we develop a common framework to explore the scalability of three improvement strategies for unstructured meshes: adaptive refinement, vertex smoothing, and edge flipping. We give a general parallel algorithm for these strategies based on defining, for each algorithm, an elemental operation and a task graph. By choosing the correct task graph, we can ensure the correct parallel execution of the algorithms independent of implementation. Finally, we present experimental results obtained on an IBM SP system and use these results to investigate, in practice, the scaling and relative costs of these algorithms.

Key words. Adaptive mesh refinement, distributed memory computers, edge flipping, mesh improvement, mesh smoothing, parallel algorithms, unstructured mesh computation.

AMS(MOS) subject classifications. Primary 65N50, 65Y05, 68Q20, 68Q22.

1. Introduction.

Unstructured mesh technology has proven successful in reducing the computational and storage requirements of many finite-element and finite-volume applications. The unstructured meshes most commonly used are composed of simplicial elements, triangles in two dimensions and tetrahedra in three dimensions. Unstructured meshes can be constructed to conform to complex domain geometries and can be adapted to more accurately model important features in complex solutions. The construction and adaptation of unstructured meshes require a variety of mesh improvement techniques. These improvement techniques increase the density of vertices in specified areas of the domain and, in addition, ensure that the element shape quality is acceptable.

Even with the use of unstructured mesh technology, many scientific applications require the solution of systems of partial differential equations (PDEs) on meshes with millions of vertices, particularly in three-

---

*The work of the first and third authors is 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 work of the second author is supported by National Science Foundation grants ASC-9501583, CDA-9529459, and ASC-9411394.

†Assistant Computer Scientist, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL.

‡Assistant Professor, The Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, Blacksburg, VA; this paper was written while at the Computer Science Department, The University of Tennessee at Knoxville, Knoxville, TN.

§Assistant Professor, Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA; this paper was written while at the Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL.

dimensional applications. The runtime of these large-scale computations can be significantly reduced through the use of parallel computation. In this environment parallel mesh improvement algorithms are required because (1) the mesh to be improved is typically distributed across the processors, (2) the mesh will not fit within the memory available on a single processor, and (3) the runtime of a parallel algorithm can be much less than a sequential version.

In this paper, we present a unified framework for the construction of parallel algorithms for three mesh improvement schemes: adaptive refinement, edge flipping, and mesh smoothing. This framework provides a common formulation that can ensure correctness and offer a simple means for the running time analysis of these and related algorithms. We show that each of these improvement methods can be implemented as a scalable parallel algorithm. Further, this common framework can allow for the reuse of significant portions of the parallel implementation.

This paper is organized as follows. In §2 we review the three methods of mesh improvement and discuss the specific variants used for the experimental results. In §3 we describe the framework, show how each of the improvement methods can be implemented within this framework, and analyze the resulting algorithms. In §4 we discuss our experimental results from an IBM SP system. Finally, in §5 we summarize this work and discuss possible future research directions.

2. Mesh Improvement Algorithms. The accuracy of discretization schemes, such as finite-element, finite-difference, and finite-volume methods, used in the solution of PDEs typically depends on the number and location of the elements/vertices and the shape of the elements. For example, in the two-dimensional case it is known that the accuracy of the finite-element approximation degrades as the maximum interior angle of an element approaches π [2]; further, the condition number of the resulting matrices grow as $O(\frac{1}{\theta_{\min}})$, where $\theta_{\min}$ is the minimum interior angle of the mesh [11]. The same basic principles apply in three dimensions; however, more complex measures of element quality are required as discussed in [19]. Adaptive refinement of a mesh, using element division, seeks to improve the solution quality by increasing the number of elements and vertices in areas of the domain where the discretization error is large. Mesh smoothing and edge flipping are local optimization techniques that seek to improve the solution quality by improving the shape quality of elements in the mesh. In Figure 1 we put these three techniques in the context of a general algorithm for the adaptive solution of a PDE.

In this section, we briefly review the relevant features of these three methods and, in the next section, give a general framework for their parallelization. First, we require some basic definitions. Let $V = \{v_i \mid i = 1, \ldots, n\}$ be a set of vertices, and let $T = \{t_a \mid a = 1, \ldots, m\}$ be a set of triangular elements. An element consists of a set of $p \geq 3$ vertices, where

$k \leftarrow 0$
Construct an initial mesh, M_0
Improve M_0, using flipping and smoothing, to form M_0'
Solve the PDE on M_0'
Estimate the error on each element
While (the maximum element error is larger than given tolerance) **do**
  Based on error estimates determine a set of elements, S_k, to refine
  Divide the elements in S_k, and any other
    elements necessary to form M_{k+1}
  Improve M_{k+1}, using flipping and smoothing, to form M_{k+1}'
  Solve the PDE on M_{k+1}'
  Estimate the error on each element
  $k \leftarrow k + 1$
Endwhile

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

three of the vertices define the element corners; we allow nonconforming vertices to lie on the element edges. If every element in a mesh has three vertices, we say the mesh is *conforming*. That is, every element in a conforming mesh meets at a vertex, shares an entire edge, or is not adjacent. This definition extends in a straightforward fashion to three-dimensional simplicial meshes. The edge set E is the set of vertex pairs (u, v) such that u and v are adjacent on the boundary of some common element $t \in T$. The dual edge set D is the set of pairs (t, s) such that t and s are adjacent elements in the mesh. We denote the adjacency set of a vertex v or element t, the set sharing an edge in E or D, by $\mathrm{adj}(v)$ or $\mathrm{adj}(t)$, respectively.

2.1. Element Division. The two viable techniques for dividing a triangle, regular refinement and bisection, are shown in Figure 2. Several algorithms for two-dimensional simplicial mesh refinement have been proposed that use one or both of these techniques [3] [23] [24]. All of these methods have the property that the interior angles of all the resulting triangles are bounded away from 0 and π. An algorithm for three dimensional simplicial mesh refinement, based on the selective bisection of tetrahedra, has been proven to result in meshes for which the quality of the resulting tetrahedra is bounded independent of the number of levels of refinement [5] [20].

Generally, it is not sufficient to divide only those triangles that have been selected for refinement; such an approach typically results in a mesh that is not conforming. We see in Figure 4 that it is necessary to *propagate* the refinement to other triangles to force the mesh to be conforming. We give a basic algorithm in Figure 3 that represents the computation in a refinement algorithm; we refer the reader to [16] for an in-depth discussion of propagation as it relates to specific refinement algorithms. We note that

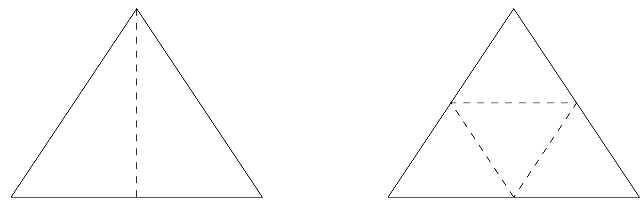

FIG. 2. *A triangle divided by bisection is shown on the left and a triangle divided by regular refinement is shown on the right.*

the length of the propagation is equivalent to the number of outer iterations in Figure 3. The length of propagation has been shown to be finite for these refinement algorithms and is typically $O(1)$ in practice; however, for most of the algorithms the worst-case bound is $O(n)$.

Let S_0 be the set of marked triangles
$i \leftarrow 0$
While $(S_i \neq \emptyset)$ **do**
  $R_i \leftarrow \emptyset$
  Foreach $(t \in S_i)$ **do**
    Divide t
    $R_i \leftarrow R_i \cup$ {triangles made nonconforming by this division}
  Endfor
  $S_{i+1} \leftarrow R_i$
  $i \leftarrow i + 1$
Endwhile

FIG. 3. *A basic refinement algorithm resulting in a conforming mesh.*

2.2. Edge Flipping. Edge flipping can be used to improve mesh quality in several contexts. The fundamental operation, in two dimensions, is to flip an edge shared by two triangles and, in three dimensions, to flip one or more tetrahedral faces [13] [18]. We illustrate such a flip for a triangle pair in Figure 5. An edge is flipped to improve a mesh quality metric; it is a local optimization technique that attempts to improve overall mesh quality. The most commonly used criterion is maximizing the minimum interior angle (maxmin angle). That is, the edge is flipped if the minimum angle in the resulting two triangles is larger than the minimum angle in the original two triangles [12]. This scheme results in a Delaunay triangulation, a triangulation with maximum minimum angle with respect to all other possible triangulations that contain only those vertices [18].[1] Many

[1] While similar criteria can improve the quality of the mesh in three dimensions, there is no guarantee that the resulting mesh will be optimal with respect to all other possible meshes containing only those vertices.

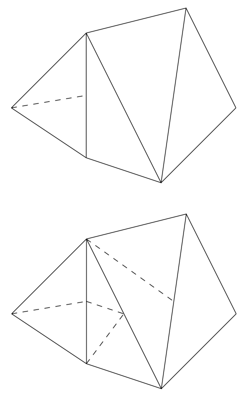

FIG. 4. *The longest edge refinement algorithm is used to create the conforming mesh. At each step of this algorithm, all nonconforming triangles are bisected across their longest edge.*

other metrics exist; for example, some use error estimates of an existing PDE solution on the mesh to evaluate potential flips [6]. However, all are local in these sense that the criteria can be evaluated by using information local to the elements involved in the flip.

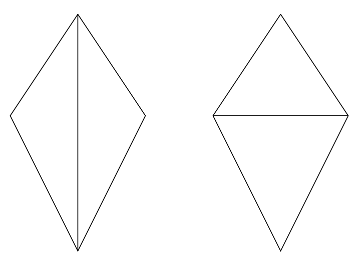

FIG. 5. *The edge shared by the two triangles on the left is flipped to create a triangle pair for which the minimum interior angle has been increased.*

This basic local optimization can be used in several contexts. One could, for example, use it to create the Delaunay triangulation as mentioned above in two dimensions, although methods exist with better worst case complexity for this task. For practical mesh algorithms, one is typically interested in making several passes through the mesh to improve the quality; flipping could be combined with other techniques, including mesh smoothing and mesh refinement, to improve overall mesh quality. We note that when an edge is flipped, the flipping criterion for the adjacent edges is changed. There is, therefore, a propagation inherent in flipping that is similar to that of refinement. We illustrate this propagation in Figure 6. In Figure 7, we give a basic algorithm for edge flipping that takes propagation into account. Note that the length of the propagation is once again equivalent to the number of outer iterations of the loop in Figure 7. The worst case length of the propagation for the maxmin angle criterion is the number of mesh edges.

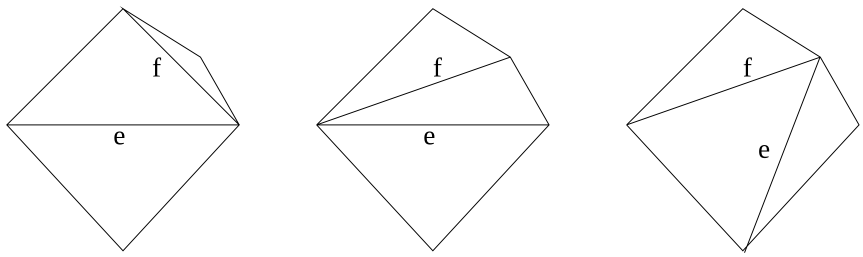

FIG. 6. *In this sequence edges are flipped based on the maxmin angle criterion. In the mesh on the left, edge e should not be flipped, but edge f should be flipped. In the middle mesh, the result of flipping edge f, edge f should not be flipped, but edge e now should be flipped. The resulting mesh on the right is the optimal mesh with respect to the given vertices and the maxmin angle criterion.*

Let S_0 be the set of edges to be considered for flipping
$i \leftarrow 0$
While $(S_i \neq \emptyset)$ **do**
  $R_i \leftarrow \emptyset$
  Foreach $(e \in S_0)$ **do**
    If flipping criteria for e indicate **then**
      Flip e
      $R_i \leftarrow R_i \cup \{\text{edges incident to } e\} \setminus \{e\}$
    Endif
  Endfor
  $S_{i+1} \leftarrow R_i$
  $i \leftarrow i + 1$
Endwhile

FIG. 7. *The flipping algorithm for a two-dimensional mesh.*

2.3. Mesh Smoothing. Unlike edge flipping, mesh smoothing attempts to improve mesh quality by changing vertex locations without affecting the mesh topology. By considering a single vertex at a time, the problem is reduced to solving a series of local optimization problems that involve only the single vertex and the vertices in the elements that contain that vertex. A typical mesh smoothing algorithm, outlined in Figure 8, makes a few passes over the mesh. The number of outer iterations is typically fixed; however, it is not known *a priori* how many sweeps are required to improve the mesh to a given quality level or whether such a quality level can even be achieved.

Let V be the set of vertices considered for smoothing
$i \leftarrow 0$
While $(i < max\_sweeps)$ **do**
    Foreach $(v \in V)$ **do**
        Adjust the geometric location of v based
            on smoothing criterion
    Endfor
    $i \leftarrow i + 1$
Endwhile

FIG. 8. *The smoothing algorithm for max_sweeps passes over all vertices.*

The local optimization problem must be inexpensive to compute, as it is typically solved one or more times for every vertex in the mesh. In Figure 9 we illustrate the local smoothing problem for vertex v, where the elements and vertices for which information is required are shaded. The simplest and least expensive local mesh-smoothing technique is Laplacian smoothing [7] [21]. In this technique, the position of a vertex v is adjusted to be the average of the positions of its neighbors. While the method is computationally inexpensive, it does not guarantee improvement in element quality and can produce an invalid mesh containing elements that are inverted (i.e., having negative volume). Optimization–based smoothing techniques guarantee valid meshes, allow for a variety of mesh quality measures, and, if carefully implemented, can be computationally affordable [1] [4] [8] [10] [26]. In [8] and [10], a local smoothing algorithm is given based on optimization techniques for nondifferentiable functions. This algorithm moves vertices in a manner guaranteed to maintain or improve mesh quality. For example, suppose $\mathbf{x}$ is the position of a vertex v, and let $f_i(\mathbf{x})$, $i = 1, \ldots, m$, be the set of element quality measures affected by a change in $\mathbf{x}$. If the element quality was the minimum sine of the interior angles in a two-dimensional mesh each triangle would require three function values, one for each interior angle of the triangle. Thus, m would be the number of triangles incident to the vertex v multiplied by three.

3. A Parallel Framework for Improvement Algorithms.

In this section, we present a general framework for the parallel mesh improvement algorithms introduced in the preceding section. The framework is based on specifying, for each algorithm, an elemental operation and a task graph with respect to which the parallel algorithm must be synchronized. We present a definition of correct execution for the parallel algorithm and show that the proposed framework satisfies this property. Finally, we note that the general algorithm consists of an inner loop involving local synchronization between processors and an outer loop involving propagation of this local information. We discuss the parallel running times of these loops.

Using the definitions from the preceding section, we now define a num-

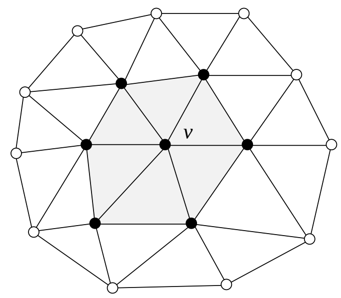

FIG. 9. *The smoothing subproblem for vertex v. The solution of this subproblem optimizes the quality of the shaded elements.*

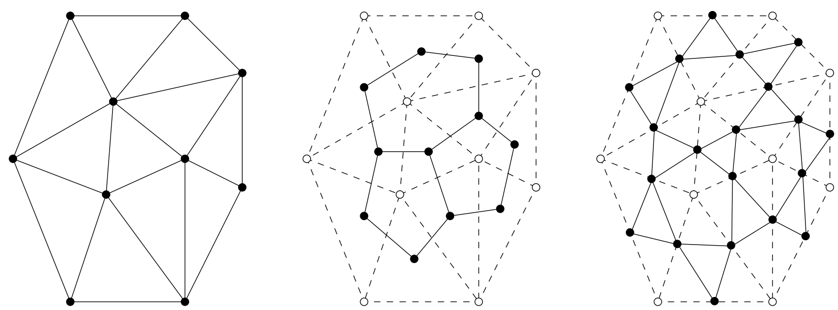

FIG. 10. *Examples of three graphs important for data structure coherency for a particular mesh. The vertex graph G_V is shown on the left, the element graph G_T in the center, and the edge graph G_E on the right.*

ber of mesh-based graph constructions that are important to the analysis of the improvement algorithms. We define the *vertex graph*, G_V, to be (V, E) and the *element graph*, G_T, to be (T, D). A two-dimensional mesh M is defined by the pair (G_V, G_T). We assume that initial meshes form valid triangulations and are conforming. Intermediate meshes, however, may be nonconforming while doing adaptive refinement; hence, we allow for nonconforming elements in the element graph definition.

For a conforming mesh we define the *edge graph* G_E to be the pair (E, F) where F is the set of edge pairs (e, f) that are edges of the same element. We define the *2-neighborhood edge graph*, G_E^2, to be the pair (E, F') where the edge $f \in F'$, $f = (a, c)$ if $f \in F$ or there exists edges (a, b) and (b, c) in F (i.e., there is a path of length 2 or less between a and c in G_E). In Figure 10 we show the three graphs G_V, G_T, and G_E for a mesh example.

3.1. Correct Parallel Execution. The parallel implementation of any of the improvement algorithms requires a distributed data structure. The execution of the subproblems solved in parallel is said to be correct if it is equivalent to some ordering of a sequential algorithm (which we assume to be valid) and maintains the coherency of the distributed data structure. However, it is impossible for the distributed data structure to be consistent at every point in the parallel algorithm. Thus, we introduce the concept of an elemental operation on the distributed data structure, and require only that the data structure be coherent at the completion of the parallel execution of elemental operations.

We define an *elemental operation* on the distributed data structure to be

1. the solution of a well-defined subproblem that may modify data owned by the processor, and
2. the updating of data, which may be owned by other processors, affected by this modification.

Different implementations use different underlying data structures; however, we believe they all must have the following properties to be consistent.

1. The ownership of mesh data (i.e., vertices, edges, and elements) is unique.
2. The coherency of vertex data must be maintained. Following any elemental operation, each vertex v must know its neighbor vertices $adj(v)$. This data must be consistent; that is, vertex $u \in adj(v)$ if and only if the representation of $adj(u)$ contains v on the processor that owns u. By "knowing" a vertex, we mean at least a global index, and in practice its position and other relevant data. This information would be required, for example, to construct the nonzero structure of an assembled stiffness matrix.
3. The coherency of element neighbor data must be maintained. Following any elemental operation, for each element t the representation of $adj(t)$ on the processor owning t must be consistent. Each element must know its neighboring elements to perform basic operations such as computing the element qualities of two adjacent elements to determine if they are improved by an edge flip.

In §3.3 we specify the elemental operation required for each of the improvement algorithms.

3.2. A General Framework for Improvement Algorithms. We base our discussion of elemental operations in the following abstract framework. Let each elemental operation be an element v in a set $\mathcal{V}$ of possible operations.[2] If the result of operation $u \in \mathcal{V}$ affects the input of operation v, we include an edge (u, v) in an edge set $\mathcal{E}$. We assume that these dependencies are reciprocal; thus, if $(u, v) \in \mathcal{E}$ we have $(v, u) \in \mathcal{E}$. We define the operation *task graph* to be the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$.

[2]In the following discussion we identify each operation in a one-to-one correspondence with a geometric object such as an edge or element.

The general improvement algorithm begins by specifying some initial set $\mathcal{S} \subseteq \mathcal{V}$ of operations to be accomplished. As these operations can affect each other, we can accomplish only an independent set of tasks, $\mathcal{I}$, simultaneously. Recall that the set $\mathcal{I}$ is *independent* in $\mathcal{G}$ if for all $u,\ v \in \mathcal{I}$, the edge $(u, v) \notin \mathcal{E}$. Once the tasks specified by $\mathcal{I}$ are accomplished, the distributed data structure is, by definition, consistent. Thus, we can continue by choosing another independent set. We proceed in this way until the set $\mathcal{S}$ is empty.

As a result of the solution of the subproblems specified by $\mathcal{S}$, a remainder set $\mathcal{R}$ of operations is generated that must be carried out subsequently. For example, in the refinement algorithm one sweep through the elements leaves nonconforming edges that must be resolved by subsequent passes through the refinement loop. At the completion of this sweep, the remainder set can be assigned to a new set $\mathcal{S}$ and another sweep begun.

$i \leftarrow 0$
Choose a vertex set $\mathcal{S}$ for operation
$\mathcal{R}_0 \leftarrow \emptyset$
While $(\mathcal{S}_i \neq \emptyset)$ **do** $\{$*outer loop*$\}$
    While $(\mathcal{S}_i \neq \emptyset)$ **do** $\{$*inner loop*$\}$
        Choose an independent set, $\mathcal{I}$, from $\mathcal{S}_i$ in the task graph $\mathcal{G}$
        Perform elemental operations on $\mathcal{I}$ (i.e., solve the
            subproblem for each $v \in \mathcal{I}$ and perform the
            update for $adj(\mathcal{I})$)
        Based on these updates, add select members of
            $adj(\mathcal{I})$ to $\mathcal{R}_i$
    Endwhile
    $\mathcal{S}_{i+1} \leftarrow \mathcal{R}_i$
    $\mathcal{R}_{i+1} \leftarrow \emptyset$
    $i \leftarrow i + 1$
Endwhile

FIG. 11. *General parallel algorithm for mesh improvement*

In Figure 11 we present the general improvement algorithm that correctly performs the elemental operations based on the task graph $\mathcal{G}$. The algorithm has two loops, which we analyze separately. We refer to the outer loop as the *propagation loop*, as it involves the propagation of tasks from nonlocal parts of the mesh.

The number of passes required for the inner loop depends on the graph structure and the way the independent sets are chosen. To assess the scalability of the algorithm, it is helpful to analyze the inner loop in the limit of using as many processors as possible. For this purpose, we use the PRAM computational model.

The PRAM computational model assumes that processors communicate through a common shared memory. The essential aspect of the PRAM model used in our analysis is that a processor can access data computed on another processor and stored in shared memory in constant time. This model underestimates the cost of interprocessor communication, and operations such as global reductions, which involve expensive synchronizations between the processors, are unrealistically cheap. Fortunately, the algorithms considered here run in an asynchronous style that requires only individual processor-to-processor communication and at most a small number of global reductions [14]. Thus, the performance of these algorithms is predicted fairly well by the PRAM model.

For the PRAM analysis, we assume that the number of processors is equal to the number of tasks $|\mathcal{S}|$. Let the maximum time required to execute any of the elemental operations be $t_{\max}$. As these tasks are independent, the time required for one sweep through the inner loop is bounded by $t_{\max}$. Let $\eta_{\mathcal{G}}$ be the number of independent sets required to cover $\mathcal{S}$ in $\mathcal{G}$. We have that the time to execute the inner loop, t_{inner}, under the PRAM model is bounded by

$$t_{\text{inner}} \leq \eta_{\mathcal{G}} t_{\max}. \tag{3.1}$$

Clearly, $\eta_{\mathcal{G}}$ depends on the rule used to choose independent sets and structure of the graph $\mathcal{G}$. We discuss the known bounds after we present specifics of the elemental operations for the improvement algorithms.

3.3. Elemental Operations for Improvement Algorithms. In this subsection we define the elemental operations for refinement, flipping, and vertex smoothing. Based on the data that these elemental operations affect, we can specify the task graph $\mathcal{G}$ for each of these algorithms.

3.3.1. The Elemental Operation for Refinement. Consider the refinement algorithm given in Figure 3 where we use bisection for element subdivision. We refer to this refinement approach as Rivara's bisection algorithm [24]. Thus, the tasks $\mathcal{S}$ are the bisection of some subset of elements, $T' \subseteq T$, marked for refinement.

Unfortunately, we cannot arbitrarily refine elements in parallel and maintain coherency of the distributed data structure. For example, in Figure 12 we show two ways that the structures can become corrupted if adjacent elements are refined simultaneously. On the left side of Figure 12, two elements t and s are simultaneously refined on different processors, resulting in the creation of two copies of the same vertex. In the right figure, two adjacent elements t and s are refined simultaneously. Because the neighbor information cannot be correctly updated, the new element t_1 inherits information from t implying that s is its neighbor; likewise s_1 inherits information claiming that t should be its neighbor.

To solve this potential problem, we illustrate in Figure 13 what is required for the correct execution of the refinement of an element. In the left figure we show the data that is required to perform the bisection: the element data, including the position of the vertices defining the element

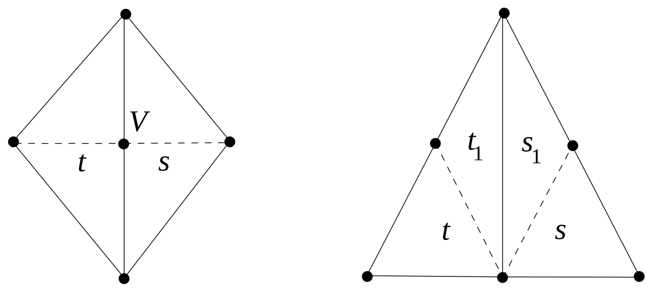

FIG. 12. *Two ways the distributed data structure can be corrupted if adjacent elements are refined simultaneously. On the left, two adjacent elements t and s are simultaneously subdivided on different processors creating two copies of the same vertex. On the right, the parent elements t and s are simultaneously refined on different processors and the children elements t_1 and s_1 inherit incorrect neighbor information.*

corners. We denote the vertex and element data that are required for the operation as filled and shaded, respectively. On the right we show the data that must be updated, namely, the neighboring elements. These elements must update their neighbor lists and add any vertices created along one of their edges. Note that the resulting mesh can be nonconforming; however, Rivara's algorithm guarantees that the overall algorithm will ultimately terminate with a conforming mesh.

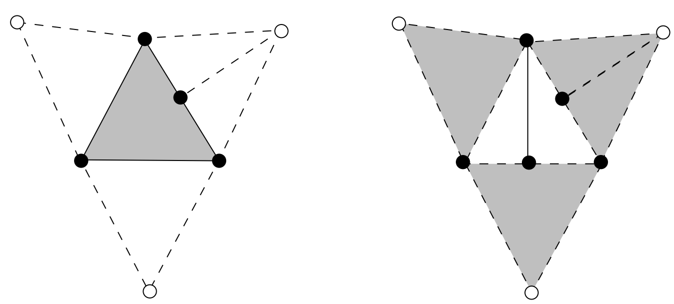

FIG. 13. *The data affected by the refinement elemental operation. On the left we show the data required to bisect an element, the shaded element and its vertices. On the right, the shaded elements and filled vertices show the data that must be updated. These elements have new neighbors, and if a vertex is created, the adjacency structures of some vertices must be updated.*

The elemental operation for refinement is summarized in Figure 14. The **If** statement ensures that each edge is bisected at most once per level of refinement. Thus, any edge can have at most one nonconforming vertex. This approach is a variation upon Rivara's original bisection algorithm which always chooses the longest edge even if edges must be bisected more than once. However, this modification simplifies the implementation without changing the quality bounds [16] and was used in the experimental results presented in the next section.

If element v is child of original element (at this refinement level) **then**
  Split remaining original edge
Else
  Split longest edge
Endif
Update neighbors
Add elements adjacent to split edges (nonconforming elements) to $\mathcal{R}$

FIG. 14. *The elemental operation for element bisection*

We must also specify the task graph $\mathcal{G}$. The updates affect only the elements adjacent to the element being refined. Thus, the correct task graph $\mathcal{G}$ for refinement is the dual graph G_T. An independent set of elements in the dual graph can be refined in parallel by using the elemental refinement operation, and the distributed data structure will remain coherent.

3.3.2. The Elemental Operation for Edge Flipping. As with element refinement, if we allow arbitrary flipping of mesh edges, the distributed data structure will be corrupted. For example, in Figure 15 we show that if edges from the same element are flipped simultaneously the resulting mesh will not be valid.

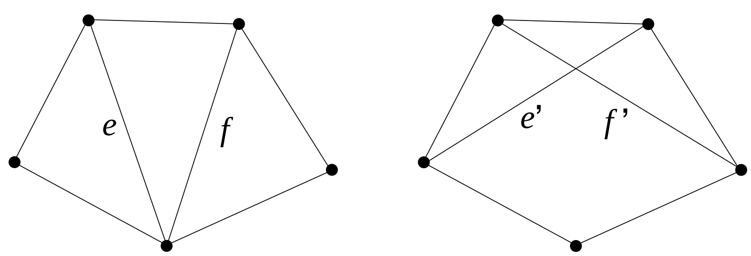

FIG. 15. *An invalid triangulation results when neighboring edges e and f are flipped simultaneously.*

Based on this observation, a first guess might be that the task graph $\mathcal{G}$ for edge flipping is G_E, the edge graph. Operating on an independent set in this graph ensures a valid mesh; however, the element neighbor information can be corrupted. For example, in Figure 16 we show two adjacent elements t_1 and t_2 that have nonadjacent edges flipped. Following this operation, we have two new adjacent elements t_3 and t_4. However, if they are flipped simultaneously, they inherit obsolete neighbor information. In this case t_3 inherits the outdated information that t_2 is its neighbor, rather than being updated correctly with the information that t_4 is its neighbor.

In Figure 17 we show the effect of an elemental flipping operation. On the left we show the data required for the operation: the two elements

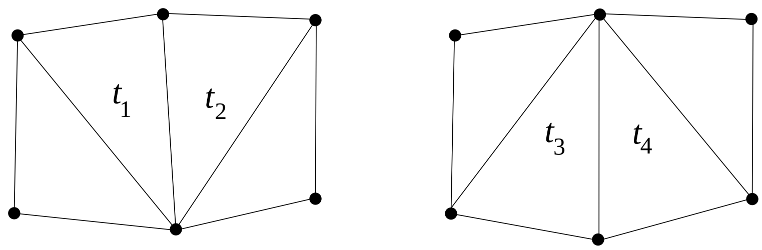

FIG. 16. *Element neighbor coherency is not maintained when edges one edge away from each other are flipped simultaneously. In this example, following the two flips the processor owning element t_3 believes it is adjacent to t_2 instead of element t_4.*

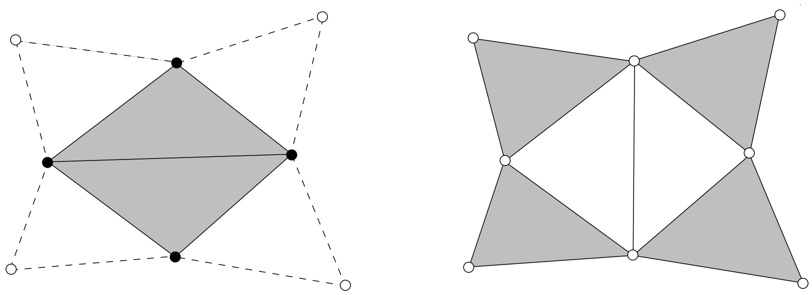

FIG. 17. *The data affected by an elemental flipping operation. The two shaded elements and the positions of the filled vertices as shown on the left figure are required to determine if the edge should be flipped. Because the neighbor information changes for elements adjacent to the two shaded elements in the left figure, the shaded elements show in the right figure must be updated by the elemental operation.*

sharing an edge e, and the positions of the vertices defining those elements. If the edge is flipped, as shown on the right, the data represented by the shaded elements must be updated. These elements need to have their neighbor information updated. In addition, the vertices defining the two initial elements must have their adjacency lists updated.

The 2-neighborhood graph, G_E^2, correctly specifies the edge set that must be updated given the flipping of an edge e in the graph. Thus, we can use this graph to specify the task graph $\mathcal{G}$ in the general framework and ensure the correct parallel execution of the algorithm. The flipping elemental operation can be formulated as shown in Figure 18.

3.3.3. The Elemental Operation for Vertex Smoothing. The elemental operation for vertex smoothing is more straightforward to specify. In Figure 19 we show on the left the data required to compute a smoothing step. We need the positions of the vertices adjacent to the vertex v being smoothed. Given this information, we can compute any local quality measure for the elements containing the vertex v. A change in the position of v will affect a smoothing calculation involving an adjacent vertex. Thus,

If (flipping edge v improves element quality measure) **then**
    Flip v
Endif
Update neighbors of v in G_E^2
If (edge was flipped) **then**
    Add $adj(v)$ in G_E to $\mathcal{R}$
Endif

FIG. 18. *The elemental operation for edge flipping*

these vertices must be updated as shown in the right figure.

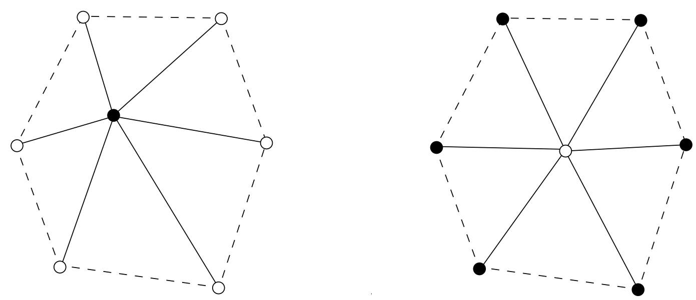

FIG. 19. *The data involved in a smoothing elemental operation. On the left we show the data required to compute a smoothing step, the positions of vertices adjacent to the vertex being smoothed (the filled vertex). On the right we show the data that must be updated, the filled vertices. These vertices require the correct position of the smoothed vertex to correctly compute the quality of elements that they share.*

The correct task graph $\mathcal{G}$ is the vertex graph G_V. Simultaneous smoothing of an independent set of vertices in this graph and the use of the elemental operation ensure that a coherent distributed data structure is maintained [9]. The elemental operation for vertex smoothing is given in Figure 20.

Solve smoothing subproblem centered at vertex v
Update $adj(v)$ with new position of v
If (v is moved) **then**
    Add $adj(v)$ to $\mathcal{R}$
Endif

FIG. 20. *The elemental operation for vertex smoothing*

3.4. Analysis of the Inner Loop Running Time. In the following two subsections we consider the parallel running time of the general mesh improvement algorithm given in Figure 11. We break the analysis into two parts. First, we analyze the running time of the inner loop; in the next subsection we consider the running time of the outer, propagation loop.

A key feature of the task graphs $\mathcal{G}$ that we are considering is that they have bounded degree. The degree of a graph G, $\Delta(G)$, with vertex set V is the maximum of $|adj(v)|$ for $v \in V$. We can summarize the bounds on the graph degrees as follows.

G_V – The initial mesh (before any refinement) has some minimum angle, say $\theta_{\min}$. We assume that this minimum angle is a property of the domain geometry and independent of the mesh size. Rivara's bisection algorithm generates a refined mesh whose minimum angle is no smaller than $\theta_{\min}/2$ [25]. Thus, the maximum number of vertices any vertex can be adjacent to in G_V is bounded by $4\pi/\theta_{\min}$.

G_T – The version of Rivara's algorithm we use guarantees that any edge in the triangulation can have at most one nonconforming vertex. Thus, any element in the mesh can be adjacent to at most six other elements. Therefore, we have that $\Delta(G_T) \leq 6$.

G_E, G_E^2 – Because G_E is defined only for a conforming mesh, we have that each edge is shared by at most two elements. As these elements have at most two other edges we have that $\Delta(G_E) \leq 4$. Likewise, because each of these edges is adjacent to at most two other edges, we have that $\Delta(G_E^2)$ is bounded by 12.

We can use the fact that these task graphs have bounded degree to obtain bounds for the scaling of $\eta_{\mathcal{G}}$ as given in Equation 3.1. In [14] it is shown that a vertex coloring of a bounded degree graph G, with $n = |V|$, can be found by a randomized algorithm in expected time $EO(\log(n)/\log\log(n))$ under the PRAM computational model using $O(n)$ processors. This coloring uses at most $\Delta(G) + 1$ colors. The randomized algorithm is motivated by a Monte Carlo rule for finding independent sets introduced by Luby [22]. The rule works by assigning to each vertex, v, in the graph an independent random number $\rho(v)$ and making the observation that the set, $I = \{v \in V \mid \rho(v) > \rho(u), \forall u \in adj(v)\}$, is an independent set in G. The greedy algorithm presented in [14] shows that every vertex can be colored in $EO(\log(n)/\log\log(n))$ time by coloring the independent set chosen by the Monte Carlo rule, deleting these vertices from the set of vertices remaining to be colored, and reapplying this process until all the vertices are colored.

In [14] it is shown that a distributed-memory implementation of this algorithm is efficient in practice. In a distributed-memory implementation the vertices can be divided into local and global sets. The local vertices are adjacent only to vertices assigned to the same processor and require no communication during the coloring algorithm, whereas the global vertices require interprocessor communication. The algorithm works by first coloring the global vertices and then extending this coloring to a consistent coloring for the local vertices. Another feature is that no communication is required if the pseudorandom numbers used in the Monte Carlo rule are computed based on a seed obtained from the global index of the vertices. In this way a vertex can independently compute the random numbers of nonlocal, adjacent vertices. Using this information, a processor knows which messages it has to wait for before vertices can be colored and where to send messages without any global synchronization. These messages are sent in an asynchronous mode, which minimizes the communication overhead.

If the graph $\mathcal{G}$ in the algorithm presented in Figure 11 does not change in the inner loop, exactly this randomized algorithm can be used to obtain the same bound for $\eta_{\mathcal{G}}$. If the graph changes, the assignment of random numbers to vertices is no longer independent, and the coloring result no longer applies. However, we can modify the algorithm to choose a new assignment of random numbers at the beginning of each pass through the inner loop. These numbers will be independent, and because the graph has bounded degree, a constant fraction of the remaining vertices will be in the independent set at each pass through the inner loop. Hence, the number of passes through the loop is $EO(\log(n))$ [17].

Another alternative is that if a vertex coloring of the graph $\mathcal{G}$ can be maintained, the number of passes through the loop is bounded by the number of colors required. We simply use the colors of the vertices in $\mathcal{S}$, note that the vertices of each color form an independent set, and use these independent sets in the inner loop. Because the number of colors required is bounded by $\Delta(\mathcal{G})+1$, the number of passes through the loop under the PRAM model is $O(1)$.

The approach based on coloring is subtle if the graph $\mathcal{G}$ changes during the elemental operations because the coloring must be updated correctly. We can use the following observation to determine when a coloring of the graph $\mathcal{G}$ can be maintained. Suppose in a pass through the inner loop the edge (u, v) is added to the graph $\mathcal{G}$. As this edge may violate the coloring, one of the vertices, say u, must be in the independent set $\mathcal{I}$ so that a new color can be chosen for u consistent with its neighbors. This approach works for the refinement algorithm where new elements (vertices in $\mathcal{G}$) are added based on elements in the independent set; the colors of these new elements can assigned to be consistent with their neighbors in the dual graph [16].

However, if $u, v \notin \mathcal{I}$, the coloring cannot be extended because more than one processor may need to recolor these vertices simultaneously. This conflict is the problem in the updates for the elemental flipping operation where edges not in the independent set become adjacent in G_E^2. The problem is illustrated in Figure 21.

We summarize these bounds for the scaling of $\eta_{\mathcal{G}}$, the number of iterations through the inner loop in mesh improvement algorithm, as follows.

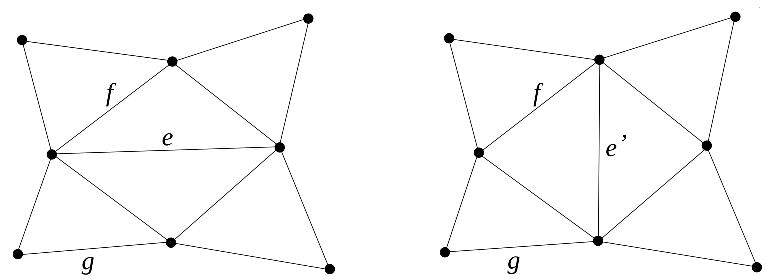

FIG. 21. *Following an elemental flipping operation, vertices not originally adjacent in G_E^2 become adjacent. In this example, when edge e is flipped to edge e', edges f and g that originally were not adjacent in G_E^2 (shown in the left figure) become adjacent (as shown in the right figure).*

1. The vertex colors of $\mathcal{S}$ are used to compute the independent sets — $\eta_{\mathcal{G}}$ is $O(1)$. This approach can be used for vertex smoothing and element refinement where the task graphs are G_V and G_T, respectively. Although G_T changes, a coloring can be maintained, as discussed in [16].
2. The random numbers, $\rho(\mathcal{S})$, are only chosen once and the Monte Carlo rule given above is used to compute independent sets — $\eta_{\mathcal{G}}$ is $EO(\log(n)/\log\log(n))$. The graph $\mathcal{G}$ must be unchanged by the elemental operations for the theoretical bound to apply. This is the case for G_V and the smoothing elemental operation. However, this approach works well in practice even when the task graph changes, since it requires no global synchronization. This approach is used for the experimental results presented in the next section.
3. Choose random numbers, $\rho(\mathcal{S})$, at the beginning of each pass through the loop and use the Monte Carlo rule to compute the independent sets — $\eta_{\mathcal{G}}$ is $EO(\log(n)$. This approach can be used with G_E and the edge flipping elemental operation.

In summary, for the task graph and elemental operations required by each mesh improvement algorithm, a method for choosing the independent sets can be found such that the theoretical bound on $\eta_{\mathcal{G}}$ scales well with problem size. In the next subsection we examine the scaling of the outer loop for these algorithms.

3.5. Propagation. As discussed briefly in §2, local actions accomplished by elemental operations can propagate through the mesh. For the mesh improvement algorithms we consider, worst-case examples demonstrate that this propagation can be $O(n)$, the size of the mesh. Although these worst case examples seem ominous, in practice the propagation length typically is $O(1)$.

In Figure 22 we show a mesh where bisection of the shaded element would result in the bisection of every other element but one in the mesh.

If this were a problem, one could modify the bisection algorithm by introducing a temporary *green edge* [3] to make the mesh conforming, as shown in the figure on the right in Figure 22. This green edge would be removed prior to the next level of refinement and the nonconforming element marked for refinement. This modified algorithm maintains a similar angle bound to Rivara's algorithm and has a worst-case propagation length of $O(1)$.

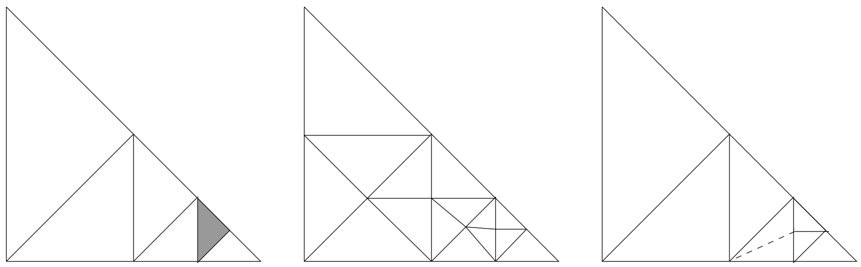

FIG. 22. *Examples showing worst case propagation for bisection. On the right we show that the propagation can be halted with the use of a green edge that is removed at the next refinement level.*

In Figure 23 we demonstrate that the edge flipping algorithm can propagate throughout the entire mesh. In the figure we show that if the Delaunay in-circle criterion is used for flipping, an edge could be required to flip to join two vertices that have $O(n)$ edges between them. Fortunately, a large mesh that is obtained by adaptive refinement will have relatively well-shaped elements. The flipping algorithm applied to these meshes results in propagation that is limited. In practice, $O(1)$ passes through the outer loop of the general algorithm usually suffice to achieve a high-quality mesh.

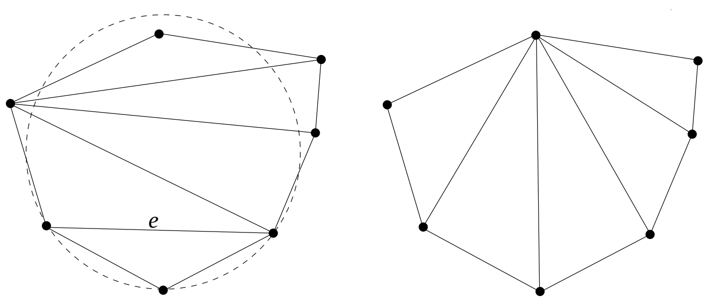

FIG. 23. *In this example every interior edge of the graph must be flipped before edge e can be flipped.*

Likewise, one can produce examples that require the smoothing algorithm to traverse the outer loop $O(n)$ times to affect a poorly placed vertex. Fortunately, empirical results indicate that a small, constant number of sweeps through all the elements suffice to produce a mesh whose quality is not improved significantly by additional smoothing passes [10].

4. Experimental Results. In this section we present experimental results that demonstrate the parallel performance of the mesh improvement algorithms. These results are based on the adaptive solution of two finite-element applications for a sequence of problem sizes. For these two problem sets, we examine the performance of the mesh improvement algorithms from several perspectives: the scalability of algorithms, the cost of the algorithms relative to other tasks required for the PDE solution, and their effect on mesh element quality. All test cases presented in this section are run on an IBM SP system with SP3 thin nodes and a TB3 switch.

The first problem set is a scalar Poisson problem with a Gaussian point charge source on a circular domain (PCHARGE). The second problem set is a linear elasticity problem on a rectangular domain with a circular hole (ELASTIC). The two problem sets are discretized using linear finite elements. For each problem instance, we begin with a initial coarse grid and selectively refine and optimize the mesh using the algorithm given in Figure 1. The number of triangles required to adequately approximate the solution is controlled by element error indicators.

To improve the overall performance of the iterative solver, we use several optimizations. We perform a partial reassembly of the linear system in the areas where mesh refinement or flipping changed the local connectivity. The solutions are interpolated between refinement levels and used as an initial guess for the iterative solver at the new refinement level. The assembled linear systems are solved using the Incomplete Cholesky Conjugate Gradient implementation in the `BlockSolve95` library [15]. More detail on the test problems and solution techniques can be found in [16].

To demonstrate the scalability of the mesh improvement techniques, we adjust the tolerance of the element error indicators so that the number of elemental operations performed on each processor is roughly constant as the number of processors increases. For the refinement and flipping operations the number of vertices on each processor is roughly 21000 for PCHARGE and 7300 for ELASTIC. The smoothing results were run at an earlier time, and the number of vertices per processor in that case was roughly 10000 for PCHARGE and 4500 for ELASTIC. The elemental operation rates are shown in Figure 24 where the rate is defined to be the maximum number of operations on a processor divided by the maximum time required by a processor to perform those operations. The average rate of the mesh improvement algorithms are slowly decreasing functions of the number of processors.

We now compare the relative performance of the mesh improvement algorithms with each other and with the linear system solver. For these results we use the termination criterion given in Figure 1 for the solution of an adaptive PDE; that is, we terminate the solution process when the local error estimate is satisfactory on each element. For each iteration of the

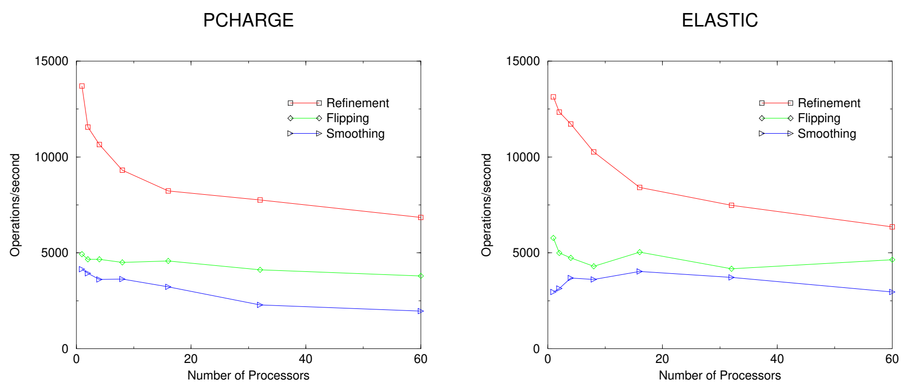

FIG. 24. *The elemental operation rates for the refinement, flipping, and smoothing algorithms as a function of the number of processors on the SP system for the PCHARGE and ELASTIC test problems*

adaptive solution, we permit only one level of refinement and perform one sweep each of edge flipping and vertex smoothing. The smoothing technique used for these results is a combined Laplacian/optimization-based smoothing technique using the minimum sine of element angle as the quality measure [10]. The results presented in the remainder of this section are run on 32 processors of the SP.

In Figure 25 we show the total number of elements, the number of triangles refined and flipped, and the number of vertices smoothed for each iteration of the adaptive solution technique for both application problems. The number of refinement iterations required to reduce the local error indicator to a satisfactory level is 20 for the PCHARGE problem and 13 for the ELASTIC problem. For the first few iterations of both problems, the number of elements that require refinement is a large percentage of the total number of elements in the mesh. Following the initial growth phase, most elements in the mesh have satisfactory error levels and the number of elements remains roughly constant. The Delaunay criterion used for flipping tends to be only violated in elements that have been refined; thus, the number of elements flipped is comparable to the number of elements refined. All of the interior vertices of the mesh are smoothed once each iteration, and the number of elemental smoothing operations is therefore roughly half the total number of elements in the mesh.

The maximum time required by a processor (over the 32 processors) to perform the elemental operations is shown in Figure 26. These times include the cost of checking each pertinent mesh entity (elements for refinement, edges for flipping, and vertices for smoothing) to determine if performing the elemental operation is beneficial or necessary. This cost is considerably less for refinement than for flipping or smoothing because the refinement criterion depends only on information contained in the element.

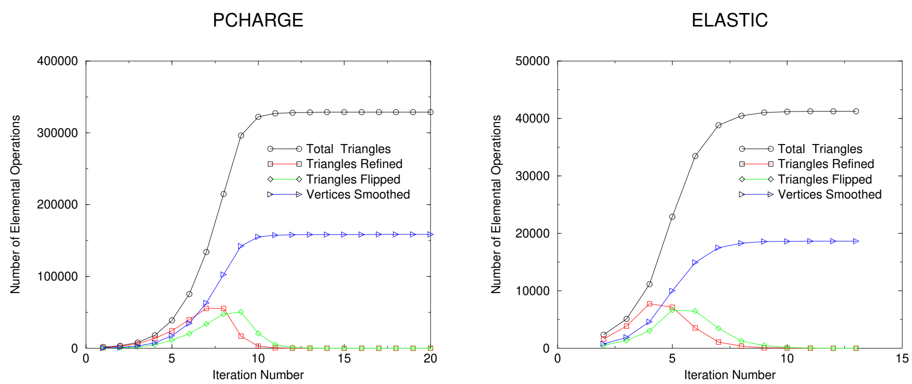

FIG. 25. *The number of triangles and elemental operations for refinement, flipping, and smoothing for the PCHARGE and ELASTIC test problem run on 32 nodes of the SP system*

Therefore, this check can be performed independently by each processor for the elements it owns without communication. In contrast, the determination of whether or not to flip an edge or smooth a vertex is dependent on adjacent element or vertex information. These checks are performed on independent sets and require communication. Therefore the maximum times shown in Figure 26 are functions of the number of elemental operations, the size of the mesh, and the number of processors. Thus, even though there are few elemental operations performed in the final iterations, the maximum time is still significant, particularly in the case of flipping. The smoothing and refinement times are less than a second for even the largest number of elemental operations on a processor for the PCHARGE problem and less than .15 seconds for the ELASTIC problem. The flipping cost is roughly twice that of smoothing and refinement.

We also show the cost of a *single* PCG iteration which consists of a sparse matrix by vector multiply, a forward and back solve, as well as the usual vector operations for CG in Figure 26. We note that although the cost of the improvement algorithms exceeds the cost of a single PCG iteration, the solution of the linear system typically requires hundreds of iterations. In Table 1 we show the time in seconds required by each mesh optimization algorithm and the time required to solve linear system. We also show the percentage of the linear system solution time taken by the combined mesh optimization times. To show the variation in these times as the adaptive solution proceeds we choose three different iterations; iteration two, the iteration which required the maximum amount of time for refinement and flipping, and the final iteration. For the PCHARGE problem, there is one unknown per vertex and the optimization algorithms require approximately 25% of the time required to solve the linear system for later iterations. The relative cost of the optimization algorithms is considerably less for

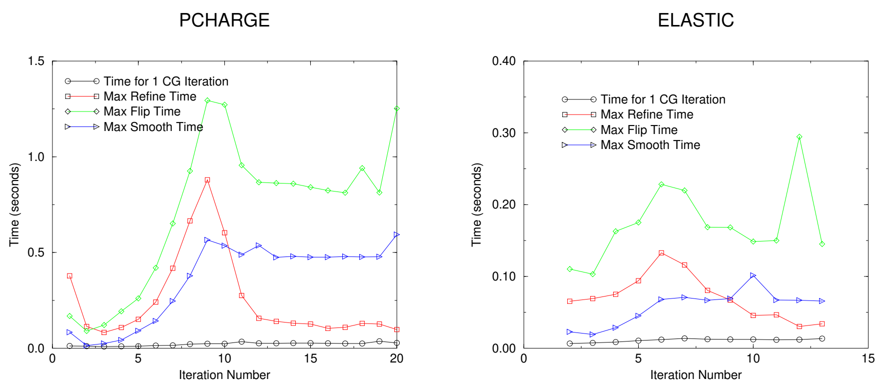

FIG. 26. *The maximum time required by a processor to perform the elemental operations at each iteration and the time required for one PCG iteration for the PCHARGE and ELASTIC problems run on 32 nodes of the SP system*

the ELASTIC problem because there are two unknowns per vertex which means the linear system is twice the size of the number of vertices.

TABLE 1
Time in seconds to solve the linear system and to perform refinement, flipping, and smoothing for selected iterations of the adaptive solution algorithm. Also shown is the percentage of time required by all the mesh optimization operations relative to the linear system solution time (at the refinement level given by the iteration number).

Iteration	PCG Iters.	Solution Time	Refine Time	Flip Time	Smooth Time	Percentage
PCHARGE						
2	49	.5344	.1136	.0907	.0144	40.9
9	507	12.35	.8797	1.293	.5653	22.1
20	238	6.714	.0982	1.251	.5934	28.9
ELASTIC						
2	123	1.578	.0655	.1104	.0228	12.59
6	591	7.087	.1327	.2281	.0681	6.05
13	336	4.584	.0340	.1450	.0657	5.33

Finally, we show the average rates for the elemental operations in Figure 27. Here we define the average rate to be the total number of elemental operations divided by the sum of processor times required to perform those operations. Clearly, the peak average rates for all three of the mesh optimization techniques corresponds to iterations in which a large number of elemental operations were performed and all the processors were busy. In later iterations, very few of the elements are refined or flipped, so most processors are idle, and the average rate is correspondingly slow. Fortunately, as shown in Figure 26 and Table 1 the time required by these algorithms is small in this stage in the solution procedure and the low average rate does not significantly affect overall solution time.

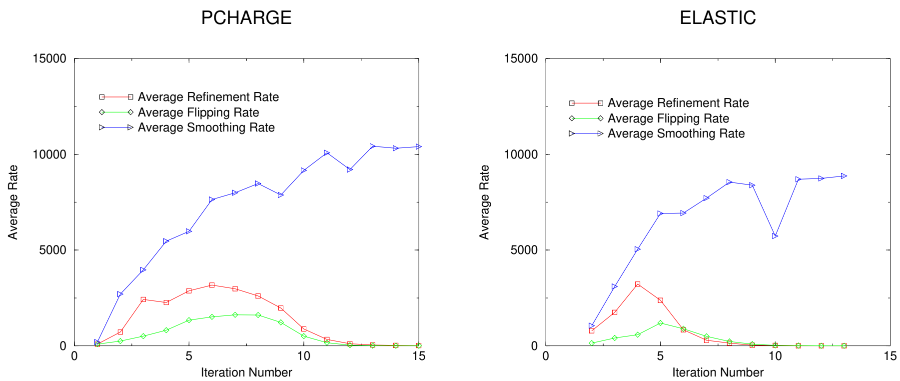

FIG. 27. *The average rates obtained for the for the improvement algorithms for each iteration for the PCHARGE and ELASTIC problems run on 32 nodes of the SP system.*

To illustrate the effect of the mesh improvement algorithms on mesh quality we show two meshes generated for elasticity problem in Figure 28. The mesh on the left shows a mesh generated using the bisection-based refinement algorithms only. The global minimum angle in this mesh is 11.3^o and the average minimum element angle is 35.7^o. The initial edges from the coarse mesh are still clearly evident after many levels of refinement. By contrast, the mesh on the right was obtained by flipping and smoothing the mesh after each refinement step. The bisection lines are no longer evident and the elements in the mesh are less distorted. The global minimum angle in this mesh is 30.4^o and the average minimum element angle is 49.4^o. For the mesh on the right, we also show the effect of each improvement technique on the minimum angle for each iteration in Figure 29. The initial angle in the coarse mesh is 8.650^o. The bisection-based refinement technique tends to have a slightly detrimental effect on the minimum angle, but is well within the theoretical bound [25]. Both the flipping and smoothing techniques improve the minimum angle in the mesh every iteration. Smoothing is somewhat more effective than flipping and improves the minimum angle in the mesh to roughly 30^o every iteration.

5. Concluding Remarks. We have presented a general framework for the analysis and implementation of unstructured mesh improvement algorithms. The approach is based on the identification of an elemental operation for each improvement algorithm. The elemental operation acts on the mesh through an update operation, by specifying the mesh data affected by this update we can identify a task graph that can used to ensure the correct parallel execution of the improvement algorithm.

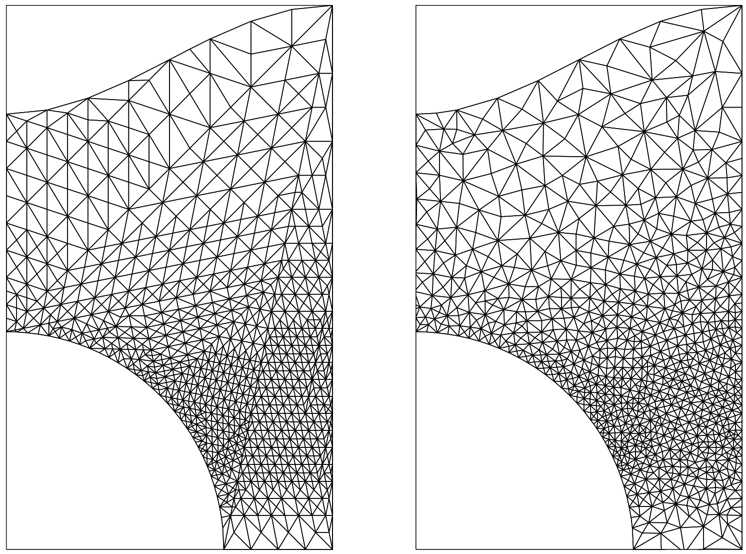

FIG. 28. *Typical meshes for the two-dimensional elasticity problem. The mesh on the left shows a mesh generated using only bisection-based refinement techniques, and the mesh on the right shows the results of interleaving smoothing and flipping with refinement.*

We considered three important improvement algorithms: adaptive refinement by element bisection, edge flipping, and mesh vertex smoothing. An analysis of the general algorithm was presented that separately considers the inner loop and outer loop. Under the PRAM computational model, the running time of the inner loop is shown to be constant or at worst a slowly growing function of the mesh size. Worst case examples show that propagation of the improvement algorithms (the number of passes through the outer loop) could be a problem for a parallel implementation. However, empirical results indicate that the propagation length is independent of the mesh size for practical problems.

Experimental results obtained on the IBM SP system show that an implementation of these algorithms is scalable for two finite-element problem sets solved on two-dimensional triangular meshes. The algorithms are effective in maintaining a high-quality mesh for these problems. In addition, the cost of the algorithms relative to other tasks required in the solution of the PDEs, such as a sparse matrix-vector product, is shown to be small.

Acknowledgment. We thank the referee for detailed and helpful comments.

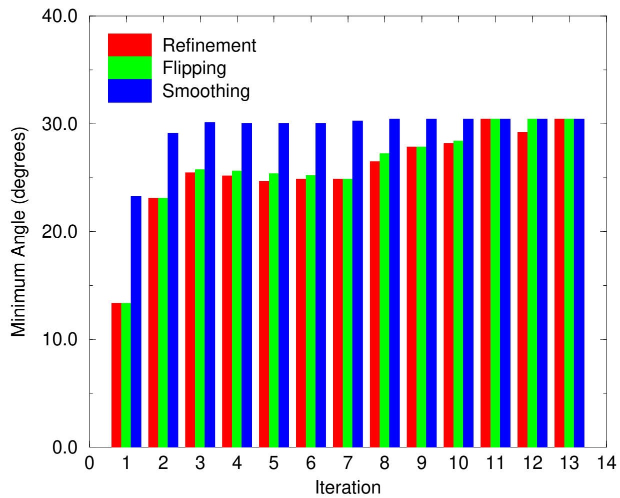

FIG. 29. *The minimum angle in the mesh after refinement, flipping, and smoothing for each iteration of the solution technique*

REFERENCES

[1] N. AMENTA, M. BERN, AND D. EPPSTEIN, *Optimal point placement for mesh smoothing*, in 8th ACM-SIAM Symp. on Discrete Algorithms, New Orleans, to appear.

[2] 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.

[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] R. E. BANK AND R. K. SMITH, *Mesh smoothing using a posteriori error estimates*, SIAM Journal on Numerical Analysis, 34 (1997), pp. 979–997.

[5] E. BÄNSCH, *Local mesh refinement in 2 and 3 dimensions*, Impact of Computing in Science and Engineering, 3 (1991), pp. 181–191.

[6] N. DYN, D. LEVIN, AND S. RIPPA, *Data dependent triangulations for piecewise linear interpolation*, IMA J. Numer. Anal., 10 (1990), pp. 137–154.

[7] D. A. FIELD, *Laplacian smoothing and Delaunay triangulations*, Communications and Applied Numerical Methods, 4 (1988), pp. 709–712.

[8] L. A. FREITAG, M. T. JONES, AND P. E. PLASSMANN, *An efficient parallel algorithm for mesh smoothing*, in Proceedings of the Fourth International Meshing Roundtable, Sandia National Laboratories, 1995, pp. 47–58.

[9] ———, *A parallel algorithm for mesh smoothing*, Preprint ANL/MCS-P668-0697, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1997.

[10] L. A. FREITAG AND C. F. OLLIVIER-GOOCH, *Tetrahedral mesh improvement using face swapping and smoothing*, International Journal for Numerical Methods in Engineering, (to appear).

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

[12] P. Green and R. Sibson, *Computing Dirichelet tesselations in the plane*, The Computer Journal, 21 (1977), pp. 168–173.

[13] B. Joe, *Construction of three-dimensional improved-quality triangulations using local transformations*, SIAM Journal on Scientific Computing, 16 (1995), pp. 1292–1307.

[14] M. T. Jones and P. E. Plassmann, *A parallel graph coloring heuristic*, SIAM Journal on Scientific Computing, 14 (1993), pp. 654–669.

[15] ———, *BlockSolve95 users manual: Scalable library software for the parallel solution of sparse linear systems*, ANL Report ANL-95/48, Argonne National Laboratory, Argonne, Ill., Dec. 1995.

[16] ———, *Parallel algorithms for adaptive mesh refinement*, SIAM Journal on Scientific Computing, 18 (1997), pp. 686–708.

[17] ———, *A parallel mesh improvement algorithm*, Preprint ANL/MCS-P675-0797, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1997.

[18] C. L. Lawson, *Software for C^1 surface interpolation*, in Mathematical Software III, J. R. Rice, ed., Academic Press, New York, 1977, pp. 161–194.

[19] A. Liu and B. Joe, *Relationship between tetrahedron shape measures*, BIT, 34 (1994), pp. 268–287.

[20] ———, *Quality local refinement of tetrahedral meshes based on bisection*, SIAM Journal on Scientific Computing, 16 (1995), pp. 1269–1291.

[21] S. H. Lo, *A new mesh generation scheme for arbitrary planar domains*, International Journal for Numerical Methods in Engineering, 21 (1985), pp. 1403–1426.

[22] M. Luby, *A simple parallel algorithm for the maximal independent set problem*, SIAM Journal on Computing, 4 (1986), pp. 1036–1053.

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

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

[25] 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.

[26] M. Shephard and M. Georges, *Automatic three-dimensional mesh generation by the finite octree technique*, International Journal for Numerical Methods in Engineering, 32 (1991), pp. 709–749.