A Parallel Algorithm For Mesh Smoothing

Article *in* SIAM Journal on Scientific Computing · September 1997

DOI: 10.1137/S1064827597323208 · Source: CiteSeer

3 authors, including:

Mark Jones
Virginia Tech (Virginia Polytechnic Institute and State University)
119 PUBLICATIONS **2,648** CITATIONS

SEE PROFILE

Paul E. Plassmann
Virginia Tech (Virginia Polytechnic Institute and State University)
105 PUBLICATIONS **2,173** CITATIONS

SEE PROFILE

A PARALLEL ALGORITHM FOR MESH SMOOTHING

LORI FREITAG*, MARK JONES†, AND PAUL PLASSMANN‡

Abstract. Maintaining good mesh quality during the generation and refinement of unstructured meshes in finite-element applications is an important aspect in obtaining accurate discretizations and well-conditioned linear systems. In this article, we present a mesh-smoothing algorithm based on nonsmooth optimization techniques and a scalable implementation of this algorithm. We prove that the parallel algorithm has a provably fast runtime bound and executes correctly for a PRAM computational model. We extend the PRAM algorithm to distributed memory computers and report results for two- and three-dimensional simplicial meshes that demonstrate the efficiency and scalability of this approach for a number of different test cases. We also examine the effect of different architectures on the parallel algorithm and present results for the IBM SP supercomputer and an ATM-connected network of SPARC Ultras.

Key words. Parallel Computing, Mesh Smoothing, Unstructured Meshes, Parallel Algorithms, Finite Elements

1. Introduction. Unstructured meshes have proven to be an essential tool in the numerical solution of large-scale scientific and engineering applications on complex computational domains. A problem with such meshes is that the shape of the elements in the mesh can vary significantly, and this variation can affect the accuracy of the numerical solution. For example, for two-dimensional triangulations classical finite element theory has shown that if the element angles approach the limits of 0° and 180°, the discretization error or the condition number of the element matrices can be adversely affected [3, 12].

Such poorly shaped elements are frequently produced by automatic mesh generation tools, particularly near domain boundaries. In addition, adaptive refinement techniques used during the solution of a problem tend to produce more highly distorted elements than were contained in the initial mesh, particularly when the adaptation occurs along curved boundaries [18].

To obtain high-quality meshes, often one must repair or improve the meshes before or during the solution process. This improvement should be based on an element quality measure appropriate for the particular problem being solved. Two mesh improvement techniques that have proven successful on sequential computers are face (edge) swapping and mesh smoothing [2, 6, 7, 8, 15, 16, 22]. However, sequential mesh optimization methods are not appropriate for applications using distributed-memory computers because (1) the mesh is usually distributed across the processors, (2) the mesh may not fit within the memory available to a single processor, and (3) a parallel algorithm can significantly reduce runtime compared with a sequential version. For such applications, parallel algorithms for mesh improvement techniques are required, and in this paper we present an efficient and robust parallel algorithm for mesh smoothing.

We have organized the paper as follows. In Section 2, we briefly review various local mesh smoothing techniques, including Laplacian smoothing and a number of optimization-based approaches. The parallel algorithm and theoretical results for correct execution and the parallel runtime bound are discussed in Section 3. In Section 4, we present numerical results obtained on the IBM SP and an ATM-connected network of SPARC Ultras that demonstrate the scalability of our algorithm.

2. Local Mesh-Smoothing Algorithms. Mesh-smoothing algorithms strive to improve the mesh quality by adjusting the vertex locations without changing the mesh topology. Local smoothing algorithms adjust the position of a single grid point in the mesh by using only the information at

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

†Assistant Professor, Computer Science Department, The University of Tennessee at Knoxville, Knoxville, TN.

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

incident vertices rather than global information in the mesh. A typical vertex, v, and its adjacent set, $adj(v)$, are shown in Figure 2.1. The vertices in the adjacent set are shown as solid circles in the figure. As the vertex v is moved, only the quality of the elements incident on v, shown as shaded triangles in the figure, are changed. Vertices not adjacent to v, shown as unfilled circles, and the quality of elements that contain these vertices are not affected by a change in the location of v. One or more sweeps through the mesh can be performed to improve the overall mesh quality. Thus, it is critical that each individual adjustment be inexpensive to compute.

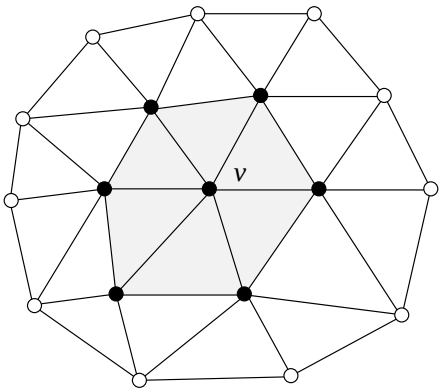

FIG. 2.1. *A vertex v and the elements whose quality is affected by a change in its position. The neighbors of v are shown as solid circles. Only the quality of the shaded elements is affected by changing the position of vertex v.*

To be more specific, we can represent any local smoothing technique as a function, *smooth*(), that given the location $\mathbf{x}_v$ of a vertex v, and its neighbors' locations, $\mathbf{x}_{adj(v)}$, returns a new location, $\hat{\mathbf{x}}_v$, for v.[1] Thus, the sequential form of any local mesh smoothing algorithm is given by the simple loop in Figure 2.2, where V is the set of vertices in the mesh to be smoothed. The positions of

Choose an ordering $V_1, ..., V_n$
For $i = 1, ..., n$ **do**
    $\hat{\mathbf{x}}_v = smooth(\mathbf{x}_v, \mathbf{x}_{adj(v)})$
Enddo

FIG. 2.2. *The local smoothing algorithm for sequential implementation*

the vertices after a sweep is not unique and is determined by the ordering in which the vertices are smoothed. This aspect of local mesh smoothing techniques will be discussed in more detail in Section 2.4.

The action of the function *smooth* is determined by the particular local algorithm chosen, and in this section we briefly review several previously proposed techniques.

2.1. Laplacian Smoothing. Perhaps the most commonly used local mesh-smoothing technique is Laplacian smoothing [9, 20]. This approach replaces the position of a vertex v by the average of its neighbors' positions. The method is computationally inexpensive, but it does not guarantee improvement in element quality. In fact, the method can produce an invalid mesh containing elements that are inverted or have negative volume. An example showing how Laplacian

[1] The smoothing function might require information in addition to neighbor vertex position. For example, for nonisotropic problems the function may require the derivatives of an approximate solution at v and $adj(v)$, or other specific information about the elements that contain these vertices. However, this information is still local and can be included within this framework.

smoothing can lead to an invalid mesh is shown in Figure 2.3.

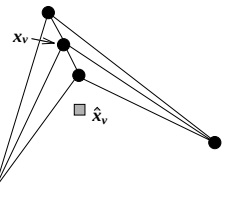

FIG. 2.3. *A set of elements for which Laplacian smoothing of the center vertex v results in an invalid triangulation. The shaded square marks the average of the positions of the vertices adjacent to v.*

A variant of Laplacian smoothing that guarantees a valid or improved mesh allows the vertex v to move only if (1) the local submesh still contains valid elements or (2) some measure of mesh quality is improved. We note that evaluating these rules significantly increases the cost of the Laplacian smoothing technique [10].

2.2. Optimization-based Smoothing. Optimization-based smoothing techniques offer an alternative to Laplacian smoothing that can be inexpensive, can guarantee valid elements in the final mesh, and are effective for a wide variety of mesh quality measures. Several such techniques have been proposed recently, and we briefly review those methods now. The methods differ primarily in the optimization procedure used or the quantity that is optimized.

Bank [4] describes a smoothing procedure for two-dimensional triangular meshes that uses the element shape quality measure given by

$$q(t) = \frac{4\sqrt{3}A}{\sum_{i=1}^{3} l_i^2},$$

where A is the area of the triangular element and l_i is the length of edge i. The maximum value for $q(t)$ corresponds to an equilateral triangle. Each local submesh is improved by using a line search procedure. The search direction is determined by the line connecting the current position of v to the position that results in the worst element becoming equilateral. The line search terminates when at least one other element's shape quality value equals that of the improving element. One variant of this technique attempts to directly compute the new location by using the two worst elements in the local submesh. In this case the line search procedure is used only in the cases for which the new position results in a third element, different from the original two worst elements, with the smallest shape measure.

Shephard and Georges describe a similar approach for tetrahedral meshes [23]. The shape function for each element incident on v is computed by using the formula

$$q(t) = \kappa \frac{V^4}{\left(\sum_{i=1}^{4} A_i^2\right)^3},$$

where V is the volume of the element and A_i is the area of face i. The parameter κ is chosen so that $q(t)$ has a maximum of one corresponding to an equilateral tetrahedron. A line search similar to that done by Bank is performed, where the search direction is determined by the line connecting the current position of v to the position that improves the worst element in the local submesh to equilateral. The line search subproblem is done by using the Golden Section procedure and terminates when the worst element is improved beyond a specified limit.

Freitag et al. [10, 11] propose a method for two- and three-dimensional meshes based on the steepest descent optimization technique for nonsmooth functions. The goal of the optimization

approach is to determine the position that maximizes the composite function

$$\phi(\mathbf{x}) = \min_{1 \le i \le l} f_i(\mathbf{x}), \tag{2.1}$$

where the functions f_i are based on various measures of mesh quality such as max/min angles and/or element aspect ratios and l is the number of functions defined on the local submesh. For example, in two-dimensional triangular meshes, maximizing the minimum angle of a local submesh containing m elements would require $l = 3m - 2$ function evaluations. For most quality measures of interest, the functions are continuous and differentiable. If the derivatives of the composite function $\phi(\mathbf{x})$ are discontinuous, the discontinuity occurs when there is a change in the set of functions that obtain the minimum value. The search direction at each step is computed by solving a quadratic programming problem that gives the direction of steepest descent from all possible convex linear combinations of the gradients in the active set. The line search subproblem is solved by predicting the points at which the set of active functions will change based on the first-order Taylor series approximations of the $f_i(\mathbf{x})$.

Amenta et al. show that the optimization techniques used in [10, 11] are equivalent to the generalized linear programming technique and has an expected linear solution time [1]. The convex level set criterion for solution uniqueness of generalized linear programs can be applied to these smoothing techniques, and they determine the convexity of the level sets for a number of standard mesh quality measures in both two and three dimensions.

All the techniques mentioned previously optimize the mesh according to element geometry. Bank and Smith [5] propose two smoothing techniques to minimize the error for finite element solutions computed with triangular elements with linear basis functions. Both methods use a damped Newton's method to minimize (1) interpolation error or (2) a posteriori error estimates for an elliptic partial differential equation. The quantity minimized in these cases requires the computation of approximate second derivatives for the solution on each element as well as the shape function $q(t)$ for triangular elements mentioned previously.

2.3. Combined Laplacian and Optimization-based Smoothing. Both Shephard and Georges [23] and Freitag and Ollivier-Gooch [10] present experimental results that demonstrate the effectiveness of combining a variant of Laplacian smoothing with their respective optimization-based procedures. The variant of Laplacian smoothing used by Shephard and Georges allows the vertex to move to the centroid of the incident vertices only if the worst element maintains a shape measure $q(t)$ above a fixed limit. Otherwise, the line connecting the centroid and the initial position is bisected, and the bisection point is used as the target position. Freitag and Ollivier-Gooch accept the Laplacian step whenever the local submesh is improved. In both cases, the Laplacian smoothing step is followed by optimization-based smoothing for only the worst elements. Experiments in [10] showed that using optimization-based smoothing when the minimum angle (dihedral angle in 3D) was less than 30 degrees in two dimensions and 15 degrees in three dimensions significantly improve the meshes at a small computational cost. These results also showed that more than three sweeps of the mesh offer minimal improvements for the meshes tested.

2.4. Nonuniqueness of Smoothed Vertex Location. As mentioned earlier, the locations of the vertices in the mesh after a pass of smoothing are not unique but are determined by the ordering in which the vertices are smoothed. An example of this nonuniqueness is shown in Figure 2.4 for a simple two-dimensional mesh. The original mesh is shown on the left, where v and q are the vertices to be smoothed and the position of each vertex is given. In the top series of meshes, the vertex q is relocated by using optimization-based smoothing as described in [11] followed by adjustment of the vertex v as shown by the highlighted submeshes in the middle and rightmost meshes. In the bottom series of meshes, the vertices are smoothed in reversed order, and the resulting final meshes are considerably different. For each of these final meshes, the resulting minimum, maximum, and

average angles for the two orderings are presented in Table 2.1. The higher-quality mesh is obtained by moving the vertex q before moving the vertex v.

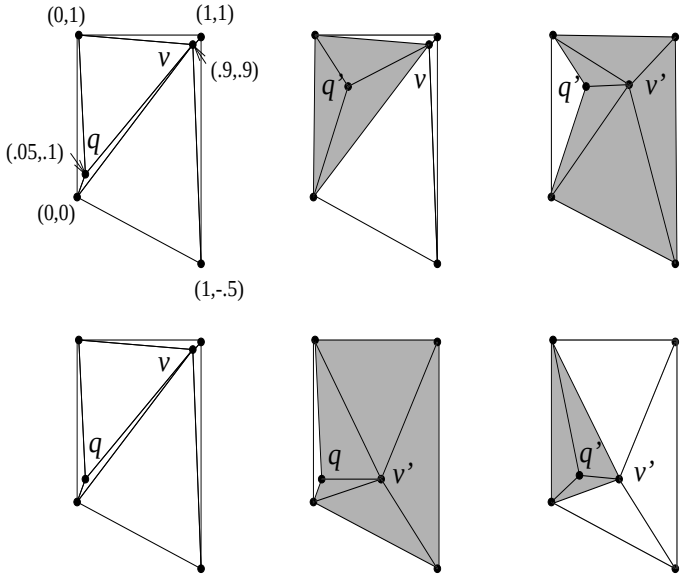

FIG. 2.4. *The order in which vertices are smoothed can significantly affect the final mesh quality. These series of meshes show the intermediate and final meshes when the vertex q is smoothed followed by the vertex v (top) and vice versa (bottom).*

TABLE 2.1

Minimum, maximum, and average angles for the the mesh shown in Figure 2.4 for a single pass of optimization-based smoothing with two different orderings of vertices

Ordering	Min. Angle	Max. Angle	Avg. Angle
Original Mesh	1.736°	159.829°	19.005°
v then q	10.445°	146.429°	23.801°
q then v	19.038°	134.764°	25.534°

In general, vertices incident on poor-quality elements are the most likely to significantly change location during the smoothing process. These large changes can adversely affect the quality of neighboring submeshes, but the effects can be mitigated by subsequent adjustment of the neighboring vertices. Therefore, an ordering of vertices that would tend to be more effective than a random ordering would be to smooth the vertices incident on the elements with the lowest quality first.

3. A Parallel Mesh-Smoothing Algorithm. In this section we present a framework for the correct parallel implementation for any of the local mesh-smoothing algorithms presented in the preceding section. The parallel smoothing algorithm is formulated within the context of the graph of the mesh which we define as follows. Let $V = \{v_i | i = 1, ..., n\}$ be the set of vertices in the mesh and $T = \{t_a | a = 1, ..., m\}$ be the set of mesh elements, either triangles or tetrahedra. Let $G = (V, E)$ be the graph associated with the mesh, where $E = \{e_{i,j} = (v_i, v_j) | v_i, v_j \in t_a\}$.

We first consider the problem of coordinating information about the mesh between processors to ensure that the mesh remains valid during smoothing. An invalid mesh can be created by smoothing two adjacent vertices simultaneously on different processors. Consider the triangulation shown in the first mesh in Figure 3.1 in which the vertices q and v are to be smoothed and are owned by different

processors. The new locations of the vertices after simultaneously being smoothed are indicated in the following mesh by v' and q'. These positions are determined assuming the locations of q and v are fixed to those given in the first mesh. The shaded region in the second mesh shows the inverted triangle that was created by the new locations v' and q'.

We define correct execution of the parallel algorithm as follows. Let the quality of an initial, valid mesh T_0 be q_0. The parallel algorithm has executed correctly if the smoothed mesh, T_1, is valid and the quality q_1 is greater than or equal to q_0. Note that we do not require that the quality of a mesh smoothed in parallel equal the quality of the same mesh smoothed in serial, because a different vertex ordering may be used.

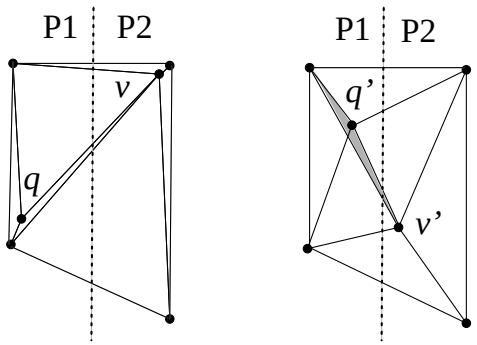

FIG. 3.1. *An example of an invalid mesh created when adjacent vertices on different processors are smoothed simultaneously. The inverted triangle is indicated in the shaded region.*

Because elements not incident on v are not affected by a change in location of vertex v, we can ensure the correct execution of the parallel algorithm by preventing two vertices that are adjacent in the mesh, but on different processors, from being simultaneously smoothed. We define an *independent* set of vertices to be a subset of the mesh vertices, I, such that $v_i \in I \Rightarrow v_i \notin adj(I)$. The approach for the parallel smoothing algorithm is to (1) select an independent set of mesh vertices, (2) smooth these vertices in parallel, and (3) notify their neighbors of their new position so that the procedure can be repeated with a new independent set. This approach avoids synchronization problems between processors. We first formulate the algorithm using a Parallel Random Access Machine (PRAM) computational model for which we can prove algorithm correctness and give a parallel runtime bound. We then formulate a practical variant for distributed memory architectures.

3.1. The PRAM Computational Model. For the PRAM computational model, we assume 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.

Using this model, we assume that we have as many processors as we have vertices to be smoothed and that vertex v_i is assigned to processor p_i. The parallel algorithm that gives the correct implementation of mesh smoothing is given in Figure 3.2.

The minimum number of steps required for correct execution of the parallel PRAM algorithm is given by Lemma 1.

LEMMA 1 *The number of steps required to guarantee correct execution of the smoothing algorithm is at least $|\sigma_{opt}|$, where σ_{opt} is the coloring of $G = (V, E)$ such that $|\sigma_{opt}|$ is minimal among all colorings of G.*

Proof. In the parallel smoothing algorithm, a set of vertices, I, is smoothed at each time step. If for any two vertices in I, v_j and v_k, e_{jk} exists, then two neighboring vertices will be smoothed

$k = 0$
Let $\mathcal{S}_0$ be the initial set of vertices marked for smoothing
While $\mathcal{S}_k \neq \emptyset$
    Choose an independent set I from $\mathcal{S}_k$
    For each $v \in I$ **do**
        $\hat{\mathbf{x}}_v = smooth(\mathbf{x}_v, \mathbf{x}_{adj(v)})$
    Enddo
    $\mathcal{S}_{k+1} = \mathcal{S}_k \backslash I$
    $k = k + 1$
Endwhile

FIG. 3.2. *The PRAM parallel smoothing algorithm.*

simultaneously; as shown earlier, this may result in an invalid mesh or a mesh with lesser quality than that of the initial mesh. Guaranteeing correct execution requires that I be an independent set.

The algorithm requires that a disjoint sequence of such independent sets, $I_1, I_2, ..., I_m$, be found such that the $\cup_j I_j = V$; thus the parallel smoothing algorithm requires m steps. Such a sequence of independent sets is an m-coloring of G. By definition, m must be at least $|\sigma_{opt}|$. □

Determining this optimal coloring for a general graph is known to be an NP-hard problem [13], but effective heuristics for efficiently choosing the independent sets in parallel have been developed and implemented. We now describe two such heuristic approaches: (1) a vertex coloring method, and (2) a randomization scheme. The coloring method assumes that we have a coloring of the vertices, σ, that is not necessarily optimal, but is a labeling such that $\sigma(v) \neq \sigma(u)$ if $u \in adj(v)$. Clearly, vertices of the same color constitute an independent set and can be used for this purpose in the parallel algorithm. If the maximum degree of the graph is Δ, then the number of colors found by these coloring heuristics is bounded above by $\Delta + 1$. The second approach is based on the assignment of a distinct random number, $\rho(v)$, to each vertex. At each step in the algorithm, we choose an independent set I from $\mathcal{S}$ according to the rule given in [17] based on [21]: $v \in I$ if $\rho(v) > \rho(u)$ for $u \in adj(v)$ and $u \in \mathcal{S}$.

The coloring approach yields a running time bound independent of the size of the graph being smoothed; however, the efficient parallel computation of this coloring requires the use of the randomized algorithm [17]. Therefore, the coloring approach is cost effective only if it is used enough times to amortize the initial expense of computing the coloring or is maintained for some other purpose. Because we typically use a small number of smoothing passes, the randomized approach is used in the experimental results presented in the next section. In addition, the randomized approach is more memory efficient because the color of each vertex, $\sigma(v)$, must be stored, whereas the random numbers, $\rho(v)$, can be computed when needed. For practical implementation, we use a pseudo-random number generator to determine $\rho(v)$ based solely on the global number of the vertex.

To evaluate the parallel runtime of the PRAM computational model, we assume that the mesh has been generated for the finite element or finite volume solution of a physical model. The graph of these meshes is local, and the edges connect vertices that are physically close to each other. In general, the maximum degree of any vertex in such a mesh is bounded independent of the size of the system. Given the local nature of the graph, and the assumption that each vertex is assigned a unique independent random number $\rho(v)$, we have that the expected number of independent sets generated by the while loop in Figure 3.2 is bounded by

$$EO(\log n / \log \log n), \tag{3.1}$$

where n is the number of vertices in the system. This bound is a consequence of Corollary 3.5 in [17]. The maximum time required to smooth a vertex, t_{max}, is also bounded because $t_{max} = \mathcal{O}(\text{degree}(v))$,

and we have the following expected runtime bound.

LEMMA 2 *The algorithm in Figure 3.2 has an expected runtime under the PRAM computational model of EO*($\frac{\log \mathcal{S}_0}{\log \log \mathcal{S}_0}$) $\times t_{max}$, *where* $\mathcal{S}_0$ *is the number of vertices initially marked for smoothing.*

Proof. Under the assumptions of the PRAM computational model, the running time of the parallel smoothing algorithm is proportional to the number of synchronized steps multiplied by the maximum time required to smooth a local submesh at step k. The upper bound on this time is given by the maximum time t_{max} to smooth any local submesh. For this algorithm, the number of synchronization steps is equal to the number of independent sets chosen, and from (3.1) the expected number of these is EO($\frac{\log \mathcal{S}_0}{\log \log \mathcal{S}_0}$). □

3.2. Practical Implementation on Distributed Memory Computers. For practical implementation on a distributed memory computer, we assume that the number of vertices is far greater than the number of processors, and we modify the PRAM algorithm accordingly. We assume that vertices are partitioned into disjoint subsets V_j and distributed across the processors so that processor j owns V_j. Based on the partitioning of V, the elements of the mesh are also distributed to the processors of the parallel computer.

Given that each processor owns a set of vertices rather than just one, as was the case in the PRAM model, we choose the independent sets according to a slightly different rule from that used in Figure 3.2. The independent set I from $\mathcal{S}$ is chosen according to the rule: $v_i \in I$ if for each incident vertex v_j, we have that $v_j \notin \mathcal{S}$, $v_j, v_i \in V_p$, or $\rho(v_i) > \rho(v_j)$. This modified rule allows two vertices that are owned by the same processor to be smoothed in the same step.

Because the vertex locations are distributed across many processors that do not share a common memory, we must add a communication step to the algorithm given in Figure 3.2. This communication is asynchronous, requiring no global synchronization.[2] After each independent set is smoothed, we communicate the new vertex locations to processors containing vertices in $adj(I)$ before smoothing the next independent set of vertices. We now show that this additional step ensures that the practical algorithm avoids the synchronization problems mentioned at the beginning of the section and that incident vertex information is correct at each step in the algorithm.

LEMMA 3 *Vertex information is correctly updated during the execution of the parallel smoothing algorithm.*

Proof. The proof is by induction. We assume that the initial incident vertex location is correct and that the incident vertex location is correct following step $k-1$. If the position of vertex v_i is adjusted at step k, by the properties of the independent set none of its incident vertices v_j are being adjusted. Thus, following step k of the parallel smoothing algorithm the incident vertices can be notified of the repositioning of vertex v_i and given the new location. □

We note that finding I requires no processor communication because each processor stores incident vertex information. Communication of the random numbers is not necessary if the seed given the pseudo-random number generator to determine $\rho(v_i)$ is based solely on the global numbering i. Thus, the only communication required in the practical algorithm is the notification of new vertex positions to processors containing nonlocal incident vertices and the global reduction required to check whether S_k is empty.

4. Experimental Results. To illustrate the performance of the parallel smoothing algorithm in both two and three dimensions, we consider two finite-element applications: (1) a scalar Poisson problem with a Gaussian point charge source on a circular domain (PCHARGE), and (2) a linear elasticity problem (ELASTIC). The upper right quadrant of the domain for the two-dimensional elasticity problem is shown in Figure 4.1. The three-dimensional test cases are both solved on a regularly shaped, six-sided solid. The meshes for these problems are generated from a coarse mesh

[2] Global synchronization is expensive on practical distributed memory architectures

by adaptive refinement, where elements are refined by Rivara's bisection algorithm. The refinement indicator function is based on local energy norm estimates. The parallel adaptive refinement algorithm and the test problems are described in more detail in [19]. The meshes are partitioned by using the unbalanced recursive bisection (URB) algorithm, which strives to minimize the number of processor neighbors and ensure that vertices are equally distributed [19].

For each case we compare two different smoothing approaches: one using the optimization-based smoothing approach (Optimization-based) and one using a combined Laplacian/optimization technique (Combined) [10]. For the combined approach, we use Laplacian smoothing as a first step and accept the new grid point position whenever the quality of the incident elements is improved. If the quality of the incident elements exceeds a user-defined threshold (30° in 2D and 15° in 3D [10]), the algorithm terminates; otherwise, optimization-based smoothing is performed in an attempt to further improve the mesh. The quality measure used in all cases is to maximize the minimum sine of the angles (dihedral angles in 3D) which eliminates extremal angles near 0° and 180°. Of the measures considered in [10] (max/min angle and max/min cosine), this measure produced the highest quality meshes at about the same computational cost. For all test cases considered in this paper, we perform two smoothing sweeps over the mesh grid points. Vertices are maintained in a queue and are processed in order.

To illustrate the qualitative effect of mesh smoothing, we present in Figure 4.1 results for the optimization-based approach described in [11] for the two-dimensional elasticity problem. The mesh on the left shows the initial mesh after a series of refinement steps. The global minimum angle in this mesh is 11.3° and the average minimum element angle is 35.7°. 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 smoothing the grid point locations 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 21.7° and the average minimum element angle is 41.1°.

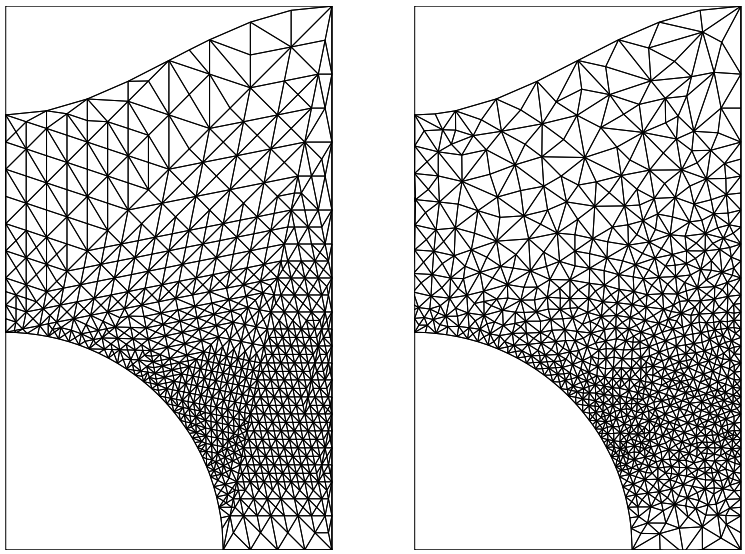

FIG. 4.1. *Typical smoothing results for the optimization-based approach on the two-dimensional elasticity problem. The mesh on the left shows refinement without smoothing, and the mesh on the right shows the results of interleaving smoothing with refinement.*

The experiments described in this section are designed to examine the scalability of the parallel

smoothing algorithm. Therefore, for each problem we have adjusted the element error tolerances so that the number of vertices per processor remains roughly constant as the number of processors is varied. To show the scalability of both the two- and three-dimensional algorithms, we ran all four test cases on 1–64 processors of an IBM SP system with SP3 thin nodes and a TB3 switch. To examine the effect of different architectures on the algorithm, we also ran the two-dimensional test cases on a network of 12 SPARC Ultras connected via an ATM network. Message passing was accomplished by using the MPICH implementation of MPI, in particular, the p4 device on the SPARC Ultra ATM network and the MPL device on the IBM SP [14].

TABLE 4.1
Smoothing results for the 2D problems for the IBM SP

Number of Procs.	Max. Number Local Vtx	Total Number Vtx	**Optimization-based** Max. Smooth Time (sec)	**Optimization-based** Vtx Smoothed per Second	**Combined** Max. Smooth Time (sec)	**Combined** Vtx Smoothed per Second
PCHARGE2D						
1	10335	10335	7.73	1336.9	2.49	4150
2	10151	20301	8.08	1256.3	2.58	3934
4	10371	41481	8.26	1255.5	2.87	3614
8	10100	80783	7.72	1308.2	2.78	3633
16	10598	169553	9.13	1160.8	3.28	3231
32	10167	325214	10.2	996.7	4.44	2290
48	10384	498379	12.5	830.7	5.71	1818
64	10845	693861	11.58	936.5	5.51	1968
ELASTIC2D						
1	4206	4206	2.84	1480	1.25	2962
2	4656	9310	3.10	1501	1.48	3145
4	4236	16942	2.66	1592	1.15	3683
8	4482	35850	2.67	1678	1.24	3614
16	4759	76118	3.51	1356	1.18	4033
32	4504	144067	2.98	1511	1.21	3722
48	4198	201392	4.10	1023	1.60	2623
64	4256	272125	3.46	1230	1.44	2955

In Table 4.1 we give the experimental results for both the optimization-based and the combined smoothing techniques for the two-dimensional test cases on the IBM SP. For each of the different numbers of processors used, we show the maximum number of vertices assigned to a processor and the total number of vertices in the final mesh. The maximum smoothing time is the longest time taken by a processor to perform two smoothing passes through all the mesh vertices. The vertices smoothed per second is the average rate per processor that vertices are smoothed; if the smoothing algorithm scaled perfectly, these numbers would remain constant.

As expected, the combined approach obtains a much higher average rate of smoothing for both applications because the more computationally expensive optimization procedure is performed for only a subset of the mesh vertices. The average smoothing rates of the two applications are different because the amount of work required to smooth the two meshes is different. For the point charge problem, the average vertex smoothing rate slowly decreases as the number of processors increases for both smoothing techniques. For the elasticity problem, the quality of the meshes varies significantly as the number of processors change, resulting in a nonmonotonic change in the smoothing rate for the combined approach. For example, on one processor 16.5 percent of the vertices require optimization-based smoothing, whereas on four processor only 10 percent require optimization-based smoothing.

The number of vertices assigned to each processor is roughly equal, thereby implying that the variation in the smoothing rate is due to primarily to two factors: (1) an increasingly unbalanced load caused by the varying computational cost required to smooth each local submesh; and (2) increased communication costs and implementation overhead associated with the parallel smoothing algorithm. Let $\mathcal{T}_i$ be the time required to compute the new locations of the vertices owned by processor P_i, and let $\mathcal{O}_i$ be the time associated with communication costs and implementation overhead on processor P_i. The time $\mathcal{T}_i$ should be thought of as the time required to smooth the vertices once the local subproblems have been constructed and does not include any overhead associated with determining the adjacency set of the vertex. To quantify these effects on the average smoothing rate, we define the following:

- *Work Load Imbalance Ratio*–the maximum time required to compute the new locations of the vertices on a processor divided by the average time:

$$\mathcal{I} = \frac{\text{Max}_i\ \mathcal{T}_i}{\sum_{i=1}^{P} \mathcal{T}_i / P}.$$

- *Efficiency*–the maximum amount of time required to compute the new locations of the vertices on a processor divided by the maximum time including overhead costs:

$$\mathcal{E} = \frac{\text{Max}_i\ \mathcal{T}_i}{\text{Max}_i\ (\mathcal{T}_i + \mathcal{O}_i)} \times 100.$$

  We note that the implementation overhead costs $\mathcal{O}_i$ include such computations as setting up the adjacency information for the local submeshes and determining independent sets. Thus, even for the sequential case, there is overhead associated with global computations, and the efficiency should be thought of as a percentage of the time solving the local smoothing problems. Therefore, a good parallel implementation will have nearly constant efficiency, indicating that little additional overhead is associated with parallelism.

For these quantities, a value of $\mathcal{I} = 1.0$ implies that the processors are perfectly balanced, and a value of $\mathcal{E} = 100\%$ implies that no overhead costs are associated with the sequential or parallel algorithm.

The work load imbalance ratios and parallel efficiencies corresponding to the test cases in Table 4.1 are given in Table 4.2. As the number of processors increases, the work load stays roughly balanced for 1–8 processors and then becomes increasingly unbalanced. This is especially true for the combined approach where the work load imbalance ratio increases to 1.7 on 64 processors for both test cases. The larger imbalance associated with the combined approach results from the fact that some processors are required to do more optimization-based smoothing than others. The parallel efficiency calculation takes this imbalance into account, and the efficiencies for the optimization-based and combined approaches, $\mathcal{E}_O$ and $\mathcal{E}_C$, remain roughly constant with respect to P. We conclude that the parallel algorithms scale well despite the increasing imbalance in work load. In general, the efficiency of the optimization-based approach is higher than that of the combined approach because the higher computational cost of each smoothing step better amortizes the overhead costs. The numbers are not monotonic because of the varying meshes and corresponding work loads for different numbers of processors.

Performance of the parallel smoothing algorithm could be improved by repartitioning the mesh to account for the imbalance in the work load. However, this approach is not practical in most applications for which smoothing is only a small portion of the overall solution process. It would not be computationally efficient to repartition the mesh just for mesh smoothing. The efficiency results show that the parallel algorithm is performing well even though the partitioning is determined for other aspects of the solution process.

TABLE 4.2

Work load imbalance and parallel efficiency of the parallel smoothing algorithm for the two-dimensional test cases on the IBM SP

Number of Procs.	Optimization-based				Combined			
	Max. Total Time	Max. Smooth Time	$\mathcal{I}_O$	$\mathcal{E}_O\%$	Max. Total Time	Max. Smooth Time	$\mathcal{I}_C$	$\mathcal{E}_C\%$
PCHARGE2D								
1	7.73	7.33	—	94.8	2.49	2.0	—	81.1
2	8.08	7.65	1.1	94.7	2.58	2.1	1.1	81.3
4	8.26	7.8	1.1	94.4	2.87	2.4	1.2	83.6
8	7.72	7.3	1.1	94.5	2.78	2.3	1.1	82.7
16	9.13	8.6	1.2	94.2	3.28	2.5	1.2	76.2
32	10.2	9.8	1.3	96.0	4.44	3.9	1.6	87.8
48	12.5	12	1.4	96.0	5.71	5.0	1.7	87.5
64	11.6	11	1.4	94.9	5.51	4.8	1.7	87.1
ELASTIC2D								
1	2.84	2.69	—	94.7	1.42	1.25	—	88.0
2	3.10	2.9	1.0	93.5	1.48	1.3	1.0	87.8
4	2.66	2.5	1.0	93.9	1.15	.94	1.0	81.7
8	2.67	2.4	1.0	89.8	1.24	.98	1.2	79.0
16	3.51	3.2	1.2	91.1	1.18	.94	1.3	79.6
32	2.98	2.6	1.1	87.2	1.21	.96	1.3	79.3
48	4.10	3.7	1.5	90.2	1.60	1.4	1.8	87.5
64	3.46	3.2	1.4	92.4	1.44	1.2	1.7	83.3

In Table 4.3, we give the number of vertices and average vertex smoothing rates for both smoothing techniques applied to the three-dimensional application problems. The cost of smoothing in three-dimensions is roughly ten times the two-dimensional cost. This increase in cost results from a roughly fivefold increase in the number of function evaluations required for each vertex due to the higher vertex degree. In addition, each function evaluation is approximately twice as expensive in 3D as in 2D. The same trends that are evident in the two-dimensional test cases are apparent in the three-dimensional test cases. In particular, the combined approach is roughly two to three times faster than the optimization-based approach. The average smoothing rates slowly decrease as a function of the number of processors. The work load imbalance and efficiency results are given in Table 4.4. Again we see that the combined approach tends to produce a more imbalanced load as the number of processors increases and that the optimization-based smoothing approach is more efficient than the combined approach because of the higher computational cost. For optimization-based smoothing the efficiency is a slowly decreasing function of the number of processors for all the test cases considered here. In contrast, the efficiency results for the combined approach are slightly more variable because of differing ratios of optimization-based smoothing to Laplacian smoothing. The roughly constant efficiencies demonstrate that the algorithm scales well despite the imbalance in the work load.

In Figure 4.2, we graphically summarize the results for the two- and three-dimensional test cases on the IBM SP and show the average rate of vertices smoothed and the efficiency for each test set and smoothing technique.

We now show that the parallel algorithm achieves roughly the same results whether run in parallel or sequentially for the two-dimensional elasticity problem. In Table 4.5 we show test case results for a single mesh containing 76118 vertices and an initial minimum angle of 5.90° on 1–32 processors. Both smoothing techniques improved the minimum angle to roughly 13°. The column labeled

TABLE 4.3
Smoothing results for the 3D problems for the IBM SP

Number of Procs.	Max. Number Local Vtx	Total Number Vtx	**Optimization-based** Max. Smooth Time (sec)	**Optimization-based** Vtx Smoothed per Second	**Combined** Max. Smooth Time (sec)	**Combined** Vtx Smoothed per Second
PCHARGE3D						
1	5889	5889	36.01	163.5	13.28	443.4
2	5953	11905	39.61	150.2	15.83	376.1
4	5935	23701	42.05	141.1	18.65	318.2
8	6433	51369	52.91	121.5	21.48	299.5
16	5564	88864	47.20	117.8	20.17	275.8
32	6442	205625	59.10	109.0	25.15	256.1
48	6377	305414	72.81	87.58	25.32	251.9
64	6367	406454	79.48	80.11	26.02	244.7
ELASTIC3D						
1	4472	4472	25.46	175.6	11.63	384.0
2	4032	8056	23.98	168.1	10.93	368.9
4	4863	19403	32.65	148.9	14.48	335.8
8	4821	38497	35.25	136.7	15.42	312.6
16	4152	66191	32.42	128.0	13.72	302.6
32	4104	130994	36.68	111.9	17.34	236.8
48	4121	196332	36.26	113.6	17.16	240.1
64	4431	282251	39.54	112.1	18.41	240.7

Time/Call gives the maximum average time to smooth each local submesh across the processors. This time is constant for both techniques on 1–8 processors. The numbers slightly increase on 16 and 32 processors because of an increase in work on one of the processors. This work increase is clearly reflected for the combined approach by the maximum percentage of cells that require optimization on a processor. This percentage increases from approximately 10 percent on 1–8 processors to 14.21 and 20.79 percent on 16 and 32 processors, respectively.

Finally, we show that that the parallel smoothing algorithm is scalable for the two-dimensional application problems on a switched ATM-connected network of SPARC Ultras. In Table 4.6 we show the number of vertices and average smoothing rates for 1–12 processors. The average rate results are more sporadic for the ATM network than they were for the IBM SP, but the same general trends are evident. In particular, the parallel smoothing algorithm effectively handles the higher the message startup latencies and lower bandwidth on the ATM network and delivers scalable performance.

5. Conclusions. In this paper we have presented a parallel algorithm for a class of local mesh smoothing techniques. Numerical experiments were performed for two of the algorithms mentioned in Section 2, and the parallel framework presented here is suitable for use with all of those techniques. Theoretical results show that the parallel algorithm has a provably fast parallel runtime bound under a PRAM computational model. We presented a variant of the PRAM algorithm implemented on distributed memory computers, and proved its correctness. Numerical experiments were performed for an optimization-based smoothing technique and a combined Laplacian/optimization-based technique on two very different distributed memory architectures. These results showed that the parallel smoothing algorithm scales very well despite the variance in processor work load associated with smoothing their individual submeshes.

TABLE 4.4

Work load imbalance ratios and efficiency the optimization-based and combined smoothing techniques for the three-dimensional test cases on the IBM SP

Number of Processors	Optimization-based				Combined			
	Max. Total Time	Max. Smooth Time	$\mathcal{I}_{\mathcal{O}}$	$\mathcal{E}_{\mathcal{O}}\%$	Max. Total Time	Max. Smooth Time	$\mathcal{I}_{\mathcal{C}}$	$\mathcal{E}_{\mathcal{C}}\%$
PCHARGE3D								
1	36.01	33.7	—	93.5	13.28	10.6	—	79.8
2	39.61	37	1.0	93.4	15.83	13	1.1	82.1
4	42.05	38	1.1	90.3	18.65	16	1.3	85.7
8	52.91	47	1.1	88.8	21.48	17	1.1	79.1
16	47.20	41	1.1	86.7	20.17	16	1.2	79.3
32	59.10	52	1.1	88.0	25.15	20	1.3	79.5
48	72.81	69	1.5	94.7	25.32	20	1.3	78.8
64	79.48	75	1.6	94.3	26.02	20	1.3	76.3
ELASTIC3D								
1	25.46	23.9	—	93.8	11.63	9.82	—	84.4
2	23.98	22	1.2	91.7	10.93	9.1	1.1	83.2
4	32.65	30	1.1	91.8	14.48	11	1.1	75.9
8	35.25	30	1.0	85.1	15.42	12	1.2	77.8
16	32.42	28	1.1	86.3	13.72	11	1.3	80.1
32	36.68	30	1.1	81.7	17.34	14	1.6	80.7
48	36.26	30	1.1	82.7	17.16	13	1.5	75.8
64	39.54	34	1.1	86.0	18.41	13	1.5	70.6

TABLE 4.5

Mesh quality and smoothing information for the parallel algorithms on the IBM SP

Number of Processors	Pre-Min Angle	Optimization		Combined		
		Post-Min Angle	Time/ Call (ms)	Post-Min Angle	Time/ Call (ms)	Max. Percent Optimized
ELASTIC2D						
1	5.90°	13.24°	.28	13.11°	.10	9.89
2	5.90°	13.24°	.28	13.11°	.10	10.06
4	5.90°	13.24°	.28	13.11°	.10	10.13
8	5.90°	13.24°	.28	13.11°	.10	10.63
16	5.90°	13.24°	.30	13.12°	.12	14.21
32	5.90°	13.24°	.32	13.51°	.14	20.79

Acknowledgments. 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.

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] E. AMEZUA, M. V. HORMAZA, A. HERNANDEZ, AND M. B. G. AJURIA, *A method of the improvement of 3D solid*

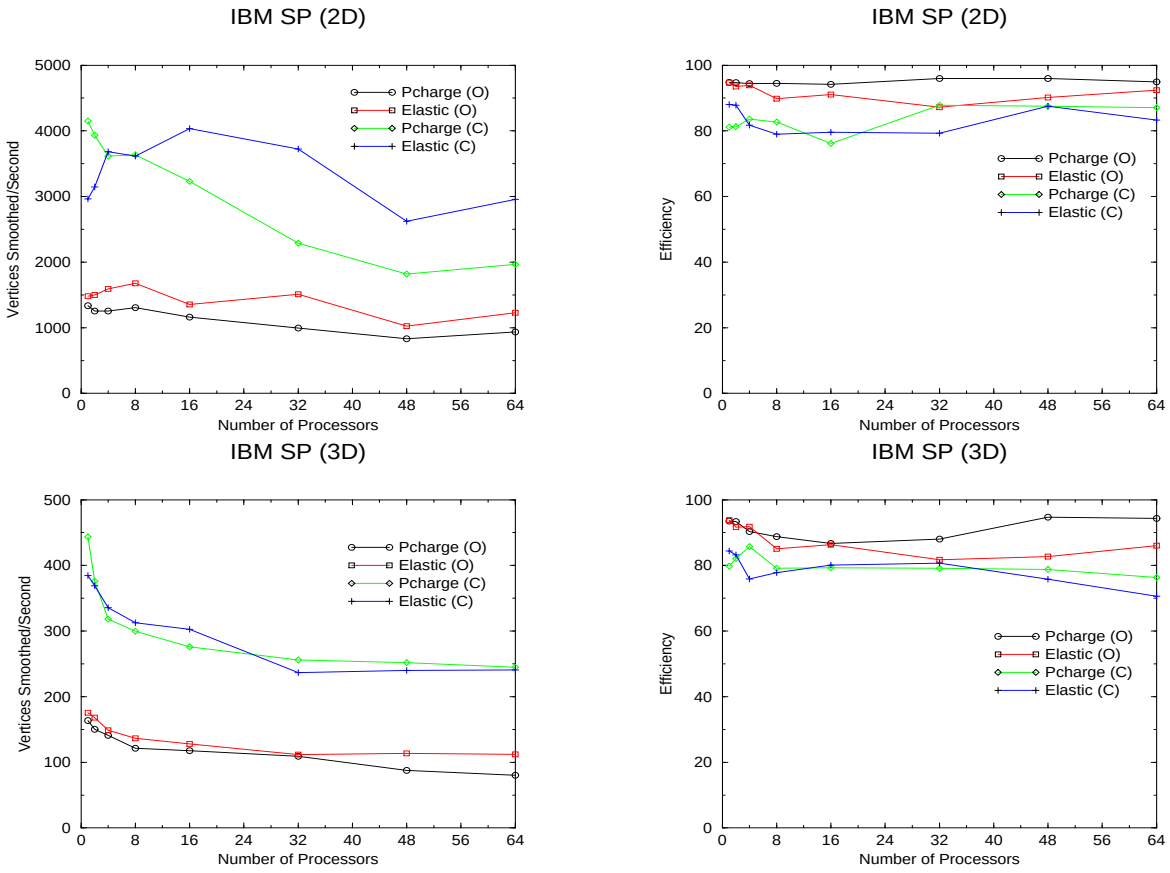

FIG. 4.2. *Plots comparing the average smoothing rates and efficiency of the test problems in both two and three dimensions as a function of the number of processors for the combined approach (C) and the optimization-based approach (O) the IBM SP*

finite-element meshes, Advances in Engineering Software, 22 (1995), pp. 45–53.

[3] I. BABUSKA AND A. AZIZ, *On the angle condition in the finite element method*, SIAM Journal on Numerical Analysis, 13 (1976), pp. 214–226.

[4] R. BANK, *PLTMG: A Software Package for Solving Ellipitc Parital Differential Equations, Users' Guide 7.0*, vol. 15 of Frontiers in Applied Mathematics, SIAM, Philadelphia, 1994.

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

[6] S. CANANN, M. STEPHENSON, AND T. BLACKER, *Optismoothing: An optimization-driven approach to mesh smoothing*, Finite Elements in Analysis and Design, 13 (1993), pp. 185–190.

[7] E. B. DE L'ISLE AND P.-L. GEORGE, *Optimization of tetrahedral meshes*, in Modeling, Mesh Generation, and Adaptive Numerical Methods for Partial Differential Equations, I. Babushka, W. D. Henshaw, J. E. Oliger, J. E. Flaherty, J. E. Hopcroft, and T. Tezduyar, eds., Springer-Verlag, 1995, pp. 97–127.

[8] H. EDELSBRUNNER AND N. SHAH, *Incremental topological flipping works for regular triangulations*, in Proceedings of the 8th ACM Symposium on Computational Geometry, 1992, pp. 43–52.

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

[10] L. FREITAG AND C. OLLIVIER-GOOCH, *A comparison of tetrahedral mesh improvement techniques*, in Proceedings of the Fifth International Meshing Roundtable, Sandia National Laboratories, 1996, pp. 87–100.

[11] 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.

[12] I. FRIED, *Condition of finite element matrices generated from nonuniform meshes*, AIAA Journal, 10 (1972),

TABLE 4.6
Smoothing results for the 2D problems for the ATM connected SPARC Ultras

Number of Procs.	Max. Number Local Vtx	Total Number Vtx	Optimization-based		Combined	
			Max. Smooth Time (sec)	Vtx Smoothed per Second	Max. Smooth Time (sec)	Vtx Smoothed per Second
PCHARGE2D						
1	11024	11024	8.53	1291.7	4.13	2670.7
2	10983	21966	9.54	1151.5	4.60	2388.5
4	10444	41777	10.30	1013.7	8.77	1190.3
6	11029	66175	13.47	818.7	7.16	1541.1
8	10450	83597	12.48	837.3	8.16	1280.7
10	10181	101813	12.85	792.2	10.63	957.8
12	10154	121850	14.21	714.6	8.60	1180.3
ELASTIC2D						
1	4246	4246	3.36	1265.3	1.78	2386.7
2	5726	11451	4.27	1340.2	2.24	2554.5
4	4070	16278	2.92	1393.5	1.40	2898.2
6	4600	27603	4.33	1062.1	2.38	1930.2
8	4258	34066	3.11	1367.9	1.47	2901.5
10	4467	44668	3.37	1326.9	1.67	2668.3
12	4815	57780	4.34	1110.2	1.93	2492.2

pp. 219–221.

[13] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability*, W. H. Freeman, New York, 1979.

[14] W. GROPP, E. LUSK, AND A. SKJELLUM, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, Cambridge, MA, 1994.

[15] B. JOE, *Three-dimensional triangulations from local transformations*, SIAM Journal on Scientific and Statistical Computing, 10 (1989), pp. 718–741.

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

[17] M. T. JONES AND P. E. PLASSMANN, *A parallel graph coloring heuristic*, SIAM Journal on Scientific Computing, 14 (1993), pp. 654–669.

[18] ———, *Adaptive refinement of unstructured finite-element meshes*, Journal of Finite Elements in Analysis and Design, 25 (1997), pp. 41–60.

[19] ———, *Parallel algorithms for adaptive mesh refinement*, SIAM Journal on Scientific Computing, 8 (1997), pp. 686–708.

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

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

[22] V. N. PARTHASARATHY AND S. KODIYALAM, *A constrained optimization approach to finite element mesh smoothing*, Journal of Finite Elements in Analysis and Design, 9 (1991), pp. 309–320.

[23] 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.