A Parallel Algorithm for Mesh Smoothing*

Lori Freitag† Mark Jones‡ 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 report mesh improvement results for two-dimensional simplicial meshes that demonstrate the effectiveness of this approach for a number of different test cases. We also show the scalability of the parallel algorithm on the IBM SP supercomputer and an ATM-connected network of SPARC Ultras.

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, with 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 negatively affected [2, 7].

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, especially when the adaptation occurs along curved boundaries [9]

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. However, sequential mesh optimization methods are not appropriate for applications using parallel 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 be much more efficient than a sequential version. For such

*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, Computer Science Department, The University of Tennessee at Knoxville, Knoxville, TN.

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

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 previous work on mesh smoothing, present a formulation of the problem as a local, nonsmooth optimization problem, and summarize our approach. The parallel algorithm and theoretical results for correct execution and the parallel run-time 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 incident vertices rather than global information in the mesh. One or more sweeps through the mesh are performed to improve the overall mesh quality. Thus, it is critical that each individual adjustment be inexpensive to compute.

Perhaps the most commonly used local mesh-smoothing technique is Laplacian smoothing [4, 11]. This approach replaces the position of a vertex v by the average of the positions of its neighbors. The method is computationally inexpensive, but it does not provide any mechanisms that 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 smoothing can lead to an invalid mesh is shown in Figure 1.

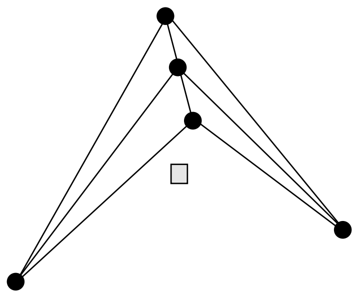

FIG. 1. *A set of elements for which Laplacian smoothing of the center vertex v results in an invalid triangulation. The shaded square marks the average of the positions of the vertices adjacent to v.*

Optimization–based smoothing techniques offer an alternative to Laplacian smoothing that can be inexpensive, can guarantee valid elements in the final mesh, and can be effective for a wide variety of mesh quality measures. In [6] and [5], we proposed an efficient, local smoothing algorithm based on nonsmooth optimization techniques that move the grid points in a manner guaranteed to maintain or improve mesh quality. Suppose $\mathbf{x}$ is the position of the vertex v, and let $f_i(\mathbf{x})$, $i = 1, \ldots, n$, be the set of element quality measures affected by a change in $\mathbf{x}$. For example, if we consider maximizing the minimum sine of dihedral angles in a three-dimensional mesh, each tetrahedron would have six function values, one for each edge of the tetrahedron. Thus, n would be the number of tetrahedra containing the vertex v multiplied by six.

The goal of the optimization approach is to determine the position $\mathbf{x}^*$ that maximizes the composite function

$$\phi(\mathbf{x}) \;=\; \min_{1\leq i\leq n} \; f_i(\mathbf{x}). \tag{1}$$

For most quality measures of interest, the functions $f_i(\mathbf{x})$ are differentiable. However, the composite function $\phi(\mathbf{x})$ has discontinuous derivatives wherever a change occurs in the *active set* (i.e., the set of functions that obtain the minimum value).

We solve this nonsmooth optimization problem using an analogue of the steepest descent method for smooth functions. The search direction $\mathbf{s}$ 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 at $\mathbf{x}$. The line search subproblem along $\mathbf{s}$ 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 $f_i(\mathbf{x})$. Standard step acceptance and termination criteria are used to ensure a robust implementation.

Experimental results demonstrating the effectiveness of the optimization-based method compared with Laplacian smoothing for two and three-dimensional simplicial meshes are given in [5, 6]. The most effective and efficient smoothing approach presented in those articles combined a variant of Laplacian smoothing with the optimization-based algorithm. In this technique, Laplacian smoothing is used as the first step and is accepted if the local mesh is improved. The active value in the local mesh after this step is compared with a user-defined threshold value. If the active value exceeds the threshold value the algorithm terminates; otherwise optimization-based smoothing is performed. Experiments showed that threshold values of 30 degrees in two dimensions and 15 degrees in three dimensions yield significant improvements to the meshes at a small computational cost. These results also showed that more than three sweeps of the mesh offer minimal improvements.

We note that similar optimization-based smoothing methods have been proposed by [1, 3, 12] for a variety of optimization procedures and mesh quality measures.

3 A Parallel Mesh-Smoothing Algorithm

In this section we present a framework for the correct implementation of local mesh-smoothing algorithms on a parallel computer. To begin, consider the partitioning of a mesh across a parallel machine. We assume that the vertices of the mesh, V, are partitioned into disjoint subsets $\{V_1, \ldots, V_p\}$, where p is the number of processors and where processor j is responsible for the vertices V_j. Based on this partitioning of V, we also assume that the elements of the mesh are distributed onto the processors of the parallel computer.

We first consider the problem of coordinating information about the mesh between processors to ensure that the mesh is kept consistent during smoothing. As mentioned above, local techniques use information only at adjacent vertices in the mesh. In Figure 2 we show a typical vertex v and its adjacent set, $adj(v)$. The vertices in the adjacent set are shown as solid circles in the figure. Only the quality of the elements containing v (shaded in the figure) are changed as the vertex v is moved. 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. Thus we can ensure that mesh is kept consistent by preventing two vertices u and v that are adjacent in the mesh, but on different processors from being simultaneously smoothed.

A set of mesh vertices that are not adjacent to each other is said to be *independent*. The basic approach for the parallel smoothing algorithm is to select an independent set, I, of mesh vertices, smooth these vertices in parallel, and then notify their neighbors, $adj(I)$, of their new position so that the procedure can be repeated with a new independent set.

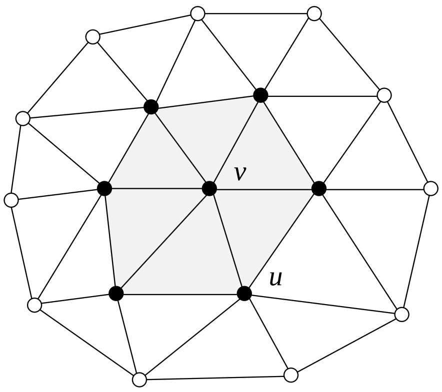

FIG. 2. *A vertex v and the elements whose quality is affected by a change in its position. The neighbors of v, $adj(v)$, are shown as solid circles. Only the quality of the shaded elements is affected by changing the position of vertex v.*

This approach avoids any synchronization problems between processors.

To be more specific, we can represent any of the local smoothing heuristics presented in the proceeding section as a function *smooth*() that, given the location of a vertex v and its neighbors $adj(v)$, returns a new location for v.[1] We express the action of this function as

$$new\_loc = smooth(v, adj(v)) \tag{2}$$

where *new_loc* is the proposed new location of vertex v computed by the smoothing function.

In Figure 3, we present an outline of the parallel algorithm that gives the correct implementation of mesh smoothing. We assume that each processor stores information about the vertices and their neighbors (i.e., processor i maintains current information about the location of V_i and $adj(V_i)$). If we assume, for the moment, that no communication is required to compute the independent sets, the only communication required is to update the neighbor information on different processors. These updates can be bundled into long messages to overcome the message start-up cost and achieve an efficient implementation. We now consider two methods for computing the independent sets; both of these methods achieve a scalable running time for bounded-degree meshes. A proof of the the correctness of the algorithm is given in [6].

We discuss two approaches for efficiently choosing the independent sets in parallel: (1) a vertex coloring method, and (2) a randomization scheme. The coloring method assumes that we have a coloring, σ, of the vertices, that 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. 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 [8]: $v \in I$ if $\rho(v) > \rho(u)$ for $u \in adj(v)$ and $u \in \mathcal{S}$.

Both of these approaches have a provably fast running time under the PRAM computational model for bounded-degree graphs (suppose the vertex degree is bounded

[1] Note that the smoothing function might require additional local information. 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.

Determine $\mathcal{S}$, the set of mesh vertices marked for smoothing
While $\mathcal{S} \neq \emptyset$
  Choose an independent set I from $\mathcal{S}$
  For each $v \in I$ **do**
    $new\_loc(v) = smooth(v, adj(v))$
  Enddo
  Update vertex information for $adj(I)$ on different processors
  $\mathcal{S} \leftarrow \mathcal{S} \backslash I$
Endwhile

FIG. 3. *The parallel smoothing algorithm.*

by Δ). Meshes useful in finite-element calculations fall naturally into this class, since their spatial embedding allows for only local connections between vertices. Under this assumption, any greedy coloring heuristic requires no more than $\Delta + 1$ colors. Thus, the number of passes through the while loop in the smoothing algorithm is bounded by this constant. Using the randomized approach, one can show that the expected number of passes through the loop is bounded by $EO(\frac{\log|\mathcal{S}|}{\log\ \log|\mathcal{S}|})$ [8].

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 [8]. 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, such as mesh refinement [10]). Since 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 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.

We note that using the randomized approach does not require interprocessor communication. The seed given the pseudo-random number generator to determine $\rho(v)$ is based solely on its global number. Thus, the only communication required in the parallel smoothing algorithm is updating vertex positions on processors containing nonlocal neighboring vertices.

4 Experimental Results

To illustrate the performance of the parallel smoothing algorithm we consider two finite-element applications: (1) a scalar Poisson problem with a Gaussian point charge source, and (2) a linear elasticity problem. The meshes for these problems are generated 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 [10].

We compare two different smoothing approaches: one using the optimization-based smoothing only to one using the more targeted combined approach. For the combined approach, we first compute the result given by Laplacian smoothing and check the resulting quality of the adjacent elements. If the quality of these elements is better than some threshold (in this case, no element angles are smaller than 30^o) we do not compute the optimization-based step.

The experiments described below 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. We feel that it is also important to examine the effect of running the algorithm on different architectures and communication networks. Thus, the problems have been run both on a 128-node IBM SP system with a high-performance switch and on a network of 10 SPARC Ultras connected via an ATM network. Message passing is 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.

Table 1

Optimization-based smoothing results for the 2D Poisson problem for the IBM SP.

Number of Processors	Max. Number of Local Vertices	Total Number of Vertices	Max. Smoothing Time (seconds)	Vertices Smoothed Per Second	Imbalance Ratio
1	11063	11063	28.29	391.06	1.0
2	10816	21631	30.17	358.50	1.1
4	10371	41481	29.10	356.39	1.1
8	10387	83083	31.34	331.42	1.2
16	10598	169553	33.64	315.05	1.3
32	11505	368077	46.13	249.40	1.5
64	11849	757973	52.04	227.69	1.5

Table 2

Optimization-based smoothing results for the 2D linear elasticity problem for the IBM SP.

Number of Processors	Max. Number of Local Vertices	Total Number of Vertices	Max. Smoothing Time (seconds)	Vertices Smoothed per Second	Imbalance Ratio
1	4246	4246	10.11	419.98	1.0
2	4404	8807	10.59	415.86	1.0
4	4070	16278	9.26	439.52	1.0
8	4259	34066	9.79	435.03	1.1
16	4414	70602	10.69	412.90	1.1
32	4238	135533	12.63	335.55	1.5

In Tables 1 and 2 we summarize the experimental results for optimization-based smoothing for the Poisson problem and the linear elasticity problem 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 goal was to have a sequence of problems for which the number of vertices per processor was roughly constant; however, these numbers do fluctuate somewhat. The maximum smoothing time is the longest time taken by a processor during one smoothing pass 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. The final number, the imbalance ratio, is the ratio of the maximum time spent smoothing (not including the communication time) on a processor to the average time; if the time spent smoothing was perfectly balanced, this number would be 1.0.

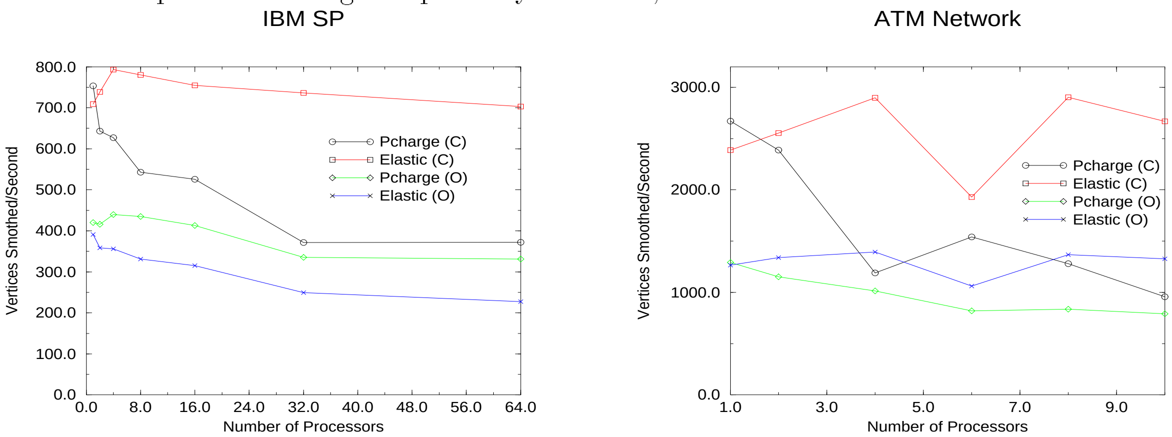

FIG. 4. *Plots comparing the average smoothing rates on processor as a function of the number of processors for the combined Laplacian and optimization-based approach (C) and the solely optimization-based approach (O) for both the IBM SP and the ATM-connected SPARC Ultra Network.*

In Figure 4 we compare the processor smoothing rate on the SP and the ATM-connected SPARC Ultra network for the two problem sets using the two different smoothing strategies. For the optimization strategy (the curves labeled O) all vertices are smoothed using the optimization-based approach. As one would expect, the combined approach can be much more efficient as is evident from the higher vertex smoothing rates shown in the figure.

An interesting effect of the combined approach is that the smoothing computation becomes more intensive as the problem size increases. The reason is that a higher percentage of elements in the mesh require the optimization-based smoothing causing the smoothing costs to increase. Thus, the average number of vertices smoothed per second decreases and approaches the rate achieved by the optimization only approach.

TABLE 3

Smoothing results for the 2D scalar Poisson problem for the ATM connected SPARC Ultras. We compare the optimization-based approach with a combined approach that uses Laplacian smoothing for high-quality elements.

			Optimization-based		**Combined**	
Number of Processors	Max. Number of Local Vertices	Total Number of Vertices	Max. Smoothing Time (seconds)	Vertices Smoothed per Second	Max. Smoothing Time (seconds)	Vertices Smoothed per Second
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

TABLE 4

Smoothing results for the 2D linear elasticity problem for the ATM-connected SPARC Ultras. We compare the optimization-based approach with a combined approach that uses Laplacian smoothing for high-quality elements.

Number of Processors	Max. Number of Local Vertices	Total Number of Vertices	**Optimization-based** Max. Smoothing Time (seconds)	**Optimization-based** Vertices Smoothed per Second	**Combined** Max. Smoothing Time (seconds)	**Combined** Vertices Smoothed per Second
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

In Tables 3 and 4 we show the results for the scalar Poisson problem and the linear elasticity problem on the ATM-connected SPARC Ultra network. For the fully optimized case, the performance of the smoothing algorithm scales very well. For the combined approach, the elasticity examples scale well; however, with the Poisson problem, we again note that the processor workload shows imbalance.

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. Babuska and A. Aziz. On the angle condition in the finite element method. *SIAM Journal on Numerical Analysis*, 13:214–226, 1976.

[3] Randolph E. Bank and R. Kent Smith. Mesh smoothing using a posteriori error estimates. *SIAM Journal on Numerical Analysis*, to appear.

[4] David A Field. Laplacian smoothing and Delaunay triangulations. *Communications and Applied Numerical Methods*, 4:709–712, 1988.

[5] Lori Freitag and Carl Ollivier-Gooch. A comparison of tetrahedral mesh improvement techniques. In *Proceedings of the Fifth International Meshing Roundtable*. Sandia National Laboratories, 1996.

[6] Lori A. Freitag, Mark T. Jones, and Paul E. Plassmann. An efficient parallel algorithm for mesh smoothing. In *Proceedings of the Fourth International Meshing Roundtable*, pages 47–58. Sandia National Laboratories, 1995.

[7] I. Fried. Condition of finite element matrices generated from nonuniform meshes. *AIAA Journal*, 10:219–221, 1972.

[8] Mark T. Jones and Paul E. Plassmann. A parallel graph coloring heuristic. *SIAM Journal on Scientific Computing*, 14(3):654–669, 1993.

[9] Mark T. Jones and Paul E. Plassmann. Adaptive refinement of unstructured finite-element meshes. *Journal of Finite Elements in Analysis and Design*, to appear.

[10] Mark T. Jones and Paul E. Plassmann. Parallel algorithms for adaptive mesh refinement. *SIAM Journal on Scientific Computing*, to appear.

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

[12] Mark Shephard and Marcel Georges. Automatic three-dimensional mesh generation by the finite octree technique. *International Journal for Numerical Methods in Engineering*, 32:709–749, 1991.