An Efficient Parallel Algorithm for Mesh Smoothing

Lori Freitag*, Mark Jones†, and Paul Plassmann*

***Mathematics and Computer Science Division**
Argonne National Laboratory

†Computer Science Department
The University of Tennessee

Abstract

Automatic mesh generation and adaptive refinement methods have proven to be very successful tools for the efficient solution of complex finite element applications. A problem with these methods is that they can produce poorly shaped elements; such elements are undesirable because they introduce numerical difficulties in the solution process. However, the shape of the elements can be improved through the determination of new geometric locations for mesh vertices by using a mesh smoothing algorithm. In this paper we present a new parallel algorithm for mesh smoothing that has a fast parallel runtime both in theory and in practice. We present an efficient implementation of the algorithm that uses non-smooth optimization techniques to find the new location of each vertex. Finally, we present experimental results obtained on the IBM SP system demonstrating the efficiency of this approach.

INTRODUCTION

The finite element method has proven to be an effective tool in the numerical solution of many scientific and engineering applications. An implementation of the method depends on a spatial decomposition of the computational domain into a union of simple geometric elements and a corresponding computational mesh. The task of generating the mesh can be one of the most time-consuming aspects of the entire solution process. If the geometry is complex, automatic mesh generation tools can be used to facilitate the decomposition [10], [12], [13]. Unfortunately, meshes generated in this way can contain poorly shaped or distorted elements which result in numerical difficulties during the solution process. For example, it has been shown that as element angles become too large the discretization error in the finite element solution is increased [2] and as angles become too small the condition number of the element matrix is increased [5]. Thus, for meshes containing distorted elements, the numerical solution is more difficult to compute and the numerical approximation is less accurate.

One approach used to correct these problems is to adjust the mesh point locations in such a way that element distortion is reduced and the overall quality of the mesh is improved. Several mesh smoothing methods and algorithms have been proposed to perform this adjustment on sequential computers [1], [3], [4], [11]. However, there are many large-scale applications, for example the "grand challenge" problems, that require the additional memory capacity and computational power of a massively parallel machine for their solution. In this case, we must assume that the entire mesh mesh cannot be maintained on a single processor and, therefore, new algorithms are required to perform tasks such as smoothing in parallel. To meet this need, we present an efficient parallel mesh smoothing algorithm that guarantees an improved mesh and has a provably fast parallel running time.

The design philosophy of the parallel algorithm follows three basic tenets: (1) the algorithm must be as effective as the best sequential algorithm; (2) the algorithm must have a provably fast parallel running time bound; and (3) the software implementation of the algorithm must be efficient and portable. To meet the first criterion, we have designed a local smoothing algorithm based on nonsmooth optimization techniques gauranteed to maintain or improve mesh quality. The second criterion is addressed by using techniques based on graph coloring to provide a fast mechanism for determining independent sets of vertices that can be manipulated simultaneously on different processors. To meet the final criterion, we use dynamic memory management, the MPI [6] message passing standard, and a portable makefile environment to allow installation across multiple computing platforms. To date, we have successfully installed and used our software on several architectures including an IBM SP supercomputer and networks of Sun, RS6000, and SGI workstations.

The parallel mesh smoothing software discussed in this paper is integratable with software developed for other tasks in unstructured mesh computation. The Scalable Unstructured Mesh Algorithms and Applications (SUMAA3d) project includes tools for parallel mesh generation, adaptive mesh refinement, dynamic mesh partitioning, and the solution of linear systems. The goal of SUMAA3d is to integrate all aspects of the parallel finite element solution process. To date, we have already made significant progress in developing parallel, portable software that meets the design criteria given above in the areas of linear system solution [9], domain partitioning and dynamic repartitioning, and two-dimensional adaptive mesh refinement [8].

The remainder of this paper is organized as follows. In Section 2, we present a mesh smoothing algorithm that formulates the problem as a local, nonsmooth optimization problem. The parallel algorithm and theoretical results for correct execution and the parallel run time bound are presented in Section 3. We then present numerical results obtained on the IBM SP for two test cases demonstrating the mesh improvement obtained using this algorithm and the scalability of the parallel algorithm.

AN OPTIMIZATION APPROACH TO MESH SMOOTHING

A number of approaches have been used to improve the quality of finite element meshes by smoothing. These methods can be broadly classified as either local or global smoothing techniques. A local method adjusts the geometric position of one vertex at a time to obtain improvement in a neighborhood around that vertex. Some number of sweeps over the adjustable vertices are performed to achieve overall improvement in the mesh. It is critical that each individual adjustment be computationally inexpensive as the mesh may contain hundreds of thousands or more grid points. In contrast, a global technique improves the mesh by simultaneously adjusting all unconstrained vertices. This approach involves solving an optimization problem as large as the number of mesh points to be moved and, consequently, this method is computationally expensive.

Laplacian smoothing [4], [10] is the most commonly used local technique. In this method grid points are iteratively moved to the arithmetic center of the polygon determined by the adjacent vertices. This method is computationally inexpensive, but it does not guarantee improvement in the element quality. In fact, it is possible to produce an invalid mesh containing elements that are inverted or have negative area.

One class of algorithms that avoids the creation of invalid elements uses optimization techniques to determine the new locations of mesh vertices [3], [11], [12]. Both local and global optimization based smoothing techniques offer the advantage of guaranteed mesh improvement and validity. However, this guarantee comes at a much higher computational cost. In fact, it has been found that optimization based smoothing algorithms can consume up to fifty percent of total mesh generation costs [11]. In this paper, we propose a low-cost local optimization technique that guarantees improved mesh quality. This approach has the additional advantage that it can be efficiently used as the core of a parallel smoothing algorithm.

Optimization techniques use function and gradient evaluations to find the minimum (or maximum) value that the function obtains in the solution space. Thus, formulating a mesh smoothing algorithm using these techniques requires that the mesh quality to be optimized be expressed as an analytic function of grid point position. There are several mesh quality indicators that can be expressed in this way including element angle size and element aspect ratio [1]. Any of these indicators can be used within the framework of our proposed technique. In this paper, we are interested in improving distorted elements and eliminating small angles and, therefore, choose to maximize the minimum angle in the mesh.

To facilitate the discussion of the optimization based smoothing algorithm, we now introduce some useful notation that describes various mesh entities. Let $\mathcal{T}$ be a triangulation (or tetrahedralization) of a computational domain with vertex set V and edge set E, where each vertex $v_i \in V$ is located at point $\mathbf{x}_i$ in the domain. An edge exists in the mesh between vertex v_i and vertex v_j if an element contains both vertices. A typical two-dimensional triangular element in the mesh with vertices v_i, edge lengths l_i, and angles θ_i is illustrated in Figure 1. The angles θ_i are expressed

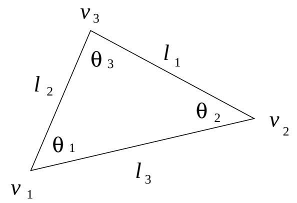

Figure 1: A typical two-dimensional finite element

analytically by using the lengths of the element edges and the law of cosines

$$cos\theta_i = \frac{l_i^2 - l_j^2 - l_k^2}{2l_j l_k} \quad i = 1,2,3 \quad j = 2,3,1 \quad k = 3,1,2 \tag{2.1}$$

where $l_i = \sqrt{(x_j - x_k)^2 + (y_j - y_k)^2}$ and l_j and l_k are analogous. The derivatives of Equation (2.1) are easily calculated, and we now have the function and gradient information necessary for the formulation of the optimization techniques.

In addition to analytic expressions for the function and gradient, we must determine the solution space in which the function is to be maximized. We are interested formulations that are local in nature, and we note that the adjustment of vertex v_i directly affects the angles in incident elements only. Considering elements that are not incident to vertex v_i to be a part of the solution space is unnecessary for this smoothing procedure. Thus, the local solution space we consider is contained by the union of elements that are incident to v_i, and we denote this submesh, T_i. A typical example of a submesh in a two-dimensional triangular mesh is illustrated in Figure 4. We also note that the vertex v_i must remain inside a convex region K in the interior of T_i for the areas of the elements in T_i to remain positive. This feasible region is the interior convex hull of the local submesh and is illustrated by the shaded area in Figure 4.

As the location of vertex v_i changes in K, the minimum angle in the corresponding submesh T_i is given by the composite function

$$\phi(\mathbf{x}) = \min_{j \in S_i} \theta_j(\mathbf{x}), \tag{2.2}$$

where S_i is the set of angles in T_i. We illustrate the character of this function by showing a one-dimensional slice through a typical function ϕ in Figure 2. Note that each $\theta_j(\mathbf{x})$ is a smooth, continuously differentiable function and that multiple angles can obtain the minimum value. Hence, the composite function $\phi(\mathbf{x})$ is nonsmooth; it is nondifferentiable at any point where the set of angles that achieve a minimum value, the active set $\mathcal{A}$, changes.

Since $\phi(\mathbf{x})$ is nonsmooth, the problem of maximizing the minimum angle in the submesh must be formulated as a nonsmooth optimization (NSO) problem

$$\max_{\mathbf{x} \in K} \phi(\mathbf{x}) \ . \tag{2.3}$$

The solution, a local maximizer of (2.3), exists at the point $\mathbf{x}^*$ if all of the directional derivatives of ϕ at $\mathbf{x}^*$ are nonnegative. This criterion can also be expressed in terms

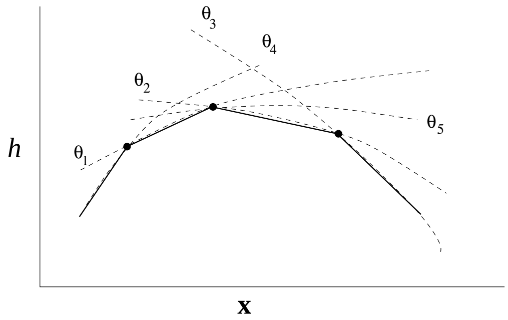

Figure 2: A one dimensional slice through the nonsmooth function $\phi(x)$

of the Jacobian matrix of the active set $\mathcal{A}$, $J_{\mathcal{A}}$ as

$$J_{\mathcal{A}}\lambda = 0 \tag{2.4}$$

where $\lambda > 0$. We note that a local minimizer on the feasible region K is unique because each of the θ_j functions is monotonic along a fixed direction.

To solve the nonsmooth optimization problem (2.3), we use an analogue of the steepest descent method for smooth functions. The search direction $\mathbf{s}_i$ at each step is chosen to be the projected gradient onto the active set of the steepest descent direction. The line search subproblem is solved by finding the linear approximation of each $\theta_i(\mathbf{x})$ given by the first order Taylor series approximation. Since this information is available for each θ_i, we can predict the points at which the active set $\mathcal{A}$ will change. These points are given by the intersection of each linear approximation with the projection the current active function in the search direction. The distance to the nearest intersection point from the current location gives the initial step length, α. The initial step is accepted if the actual improvement of ϕ is within ten percent of the estimated improvement, otherwise α is halved recursively until a step is accepted or α falls below some minimum step length tolerance. Note that a step is accepted if the subsequent step results in a smaller increase in the minimum angle. The optimization process is terminated if one of the following conditions apply: (1) condition (2.4) is satisfied at a minimum $\mathbf{x}^*$; (2) if α falls below the minimum step length with no improvement obtained; (3) if we exceed the maximum number of iterations; or (4) if the achieved improvement of any given step is less than some user-defined tolerance. The complete algorithm is given in Figure 3.

To illustrate the action of the algorithm, the series of meshes in Figure 4 show the progression of the optimization algorithm for a typical two-dimensional submesh T_i. The initial active angle is indicated with an asterisk in the first mesh and has a value of 4.70 degrees. The interior convex hull is calculated and shown in the second mesh as the shaded gray area. Recall that this represents the set of feasible points K for the optimization algorithm. A centroid smoothing step is used to compute a new initial step, $\mathbf{x}_0$. This step is accepted if it increases ϕ otherwise the original $\mathbf{x}_0$ is retained as the starting point for the optimization algorithm. For this example, the step is accepted and the results are shown in the third submesh where the minimum angle measure is increased to 19.21 degrees. The optimization algorithm is then used to compute the final location of the free vertex as shown in the fourth submesh. The

$i = 0$
Compute ϕ_0 and $\mathcal{A}_0$
If ($\mathbf{x}_0 = \mathbf{x}^*$), STOP
If (A new initial guess is to be calculated)
    Compute $\hat{\mathbf{x}}$, $\hat{\phi}$, and $\hat{\mathcal{A}}$
    If ($\hat{\mathcal{A}} > \mathcal{A}_0$)
        $i = i + 1$
        $\mathbf{x_1} = \hat{\mathbf{x}}$, $\phi_1 = \hat{\phi}$, $\mathcal{A}_1 = \hat{\mathcal{A}}$
        If $\mathbf{x}_1 = \mathbf{x}^*$, STOP
    Endif
Endif

While (($\mathbf{x}_i \neq \mathbf{x}^*$) and ($\alpha >$ MIN_STEP) and ($i <$ MAX_ITER) and
        ($|\mathcal{A}_i - \mathcal{A}_{i-1}| >$ MIN_IMPROVEMENT))
    Compute gradient $\mathbf{g}_i$
    Compute search direction $\mathbf{s}_i$
    Compute α
    While (STEP_NOT_ACCEPTED) and ($\alpha >$ MIN_STEP)
        Compute $\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha \mathbf{s}_i$
        Compute ϕ_{i+1} and $\mathcal{A}_{i+1}$
        Test for step acceptance
        $\alpha = \alpha/2$
    Endwhile
    $i = i + 1$
Endwhile

Figure 3: The optimization algorithm.

final submesh contains an equilibrium point with three active values and a minimum angle of 21.87 degrees. Overall, convergence is obtained in two optimization iterations.

THE PARALLEL MESH SMOOTHING ALGORITHM

We now present a parallel framework that correctly implements the smoothing algorithm on a distributed memory computer. A critical aspect of the correct implementation of the parallel algorithm is the synchronization necessary to avoid corrupted neighbor information. It is clear that two adjacent vertices cannot be smoothed simultaneously since the optimal position for v_i in T_i depends critically on the positions of the incident vertices. This synchronization problem can be avoided if vertices that are not adjacent to each other are adjusted in parallel. Such a set of vertices is called an independent set and is denoted I. Once the vertices in I are adjusted and incident

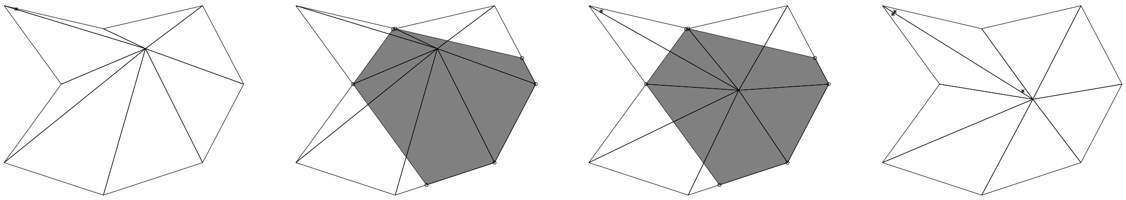

Figure 4: This figure shows the initial submesh consisting of the elements incident to vertex v_i, the feasible region for the optimization problem, the results of a centroid smoothing step, and the final solution obtained by the optimization algorithm.

vertices are notified of the new vertex location, another independent subset of V can be chosen for adjustment.

The PRAM Computational Model

We first formulate the parallel smoothing algorithm within the context of a Parallel Random Access Machine (PRAM) computational model. Recall that the PRAM computational model assumes that processors communicate through a common shared memory. The essential aspect of the PRAM model that we use in our analysis is that a processor can access data computed on another processor and stored in shared memory in constant time.

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. In addition, we can assume that messages sent between processors to keep track of incident vertex information can be done in constant time. The parallel algorithm that gives the correct implementation of mesh smoothing is given in Figure 5. The independent sets are chosen efficiently in parallel by assigning a distinct random number, $\rho(v_i)$, to each vertex v_i. At step k of the algorithm, there is some subset, $\mathcal{S}_k$ of V that remains to be smoothed. We choose an independent set I from $\mathcal{S}_k$ according to the rule given in [7]: $v_i \in I$ if for each incident vertex v_j, we have that $v_j \notin \mathcal{S}_k$ or $\rho(v_i) > \rho(v_j)$.

We now show that this algorithm avoids the synchronization problem discussed above and has a fast parallel run time bound.

LEMMA 1 *Vertex information in the dual graph is correctly updated during the execution of the 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 a 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. $\square$

To evaluate the parallel runtime of the PRAM computational model, we first note that most practical problems arising in finite element analysis are generated from local physical models. Thus, the vertices in the mesh have bounded degree independent of the size of the system. Given the local nature of the graph and the assumption

$k = 0$
Let $\mathcal{S}_0$ be the initial set of vertices marked for smoothing
Each vertex $v_i \in \mathcal{S}_0$ is assigned a random number, $\rho(v_i)$
While $\mathcal{S}_k \neq \emptyset$
    Choose an independent set I from $\mathcal{S}_k$
    Simultaneously reposition each of the vertices in I using the
        optimization algorithm given in Figure 3
    Each processor owning a relocated v_i updates this information
        on processors owning incident vertices
    $\mathcal{S}_{k+1} = \mathcal{S}_k \backslash (I \cap \mathcal{S}_k)$
    $k = k + 1$
Endwhile

Figure 5: The PRAM parallel smoothing algorithm.

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 5 is bounded by

$$EO(\log n / \log \log n) \tag{3.5}$$

where n is the number of vertices in the system. This bound is a consequence of Corollary 3.5 in [7].

We note that the smoothing algorithm is designed to ensure a finite execution time for each submesh. If t_{max} is the maximum time required to smooth a submesh, we have the following expected run time bound.

LEMMA 3 *The algorithm in Figure 5 has an expected run time 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 to the number of synchronized steps multiplied by the maximum time required to smooth a submesh at step k. The upper bound on this time is given by the maximum time t_{max} to smooth any submesh. For this algorithm, the number of synchronization steps is equal to the number of independent sets chosen and from (3.5) the expected number of these is EO$(\frac{\log \mathcal{S}_0}{\log\log \mathcal{S}_0})$. $\Box$

A Distributed Memory Implementation

For practical implementation on a distributed memory computer, we assume that the number of vertices is far greater than the number of processors and modify the PRAM algorithm accordingly. The vertices of the triangulation $\mathcal{T}$ 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 $\mathcal{T}$ 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 P-RAM model, we choose the independent sets according to a slightly different rule than used in Figure 5. The independent set I from $\mathcal{S}$ is now chosen according to the following 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 $\rho(v_i) > \rho(v_j)$. Hence, two vertices that are owned by the same processor can be smoothed in the same step. We note that finding I requires no processor communication, since 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 i. Thus, the only communication required in this 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. Because the practical algorithm has the same synchronization scheme presented in Figure 5, incident vertex information is correct at each step in the algorithm.

Note that the parallel framework for the mesh smoothing algorithm is not restricted to the optimization algorithm given in Section 2. The concept of independent sets may be used successfully with a number of different local smoothing techniques. For instance, the same synchronization points are required for the correct execution of laplacian smoothing techniques.

RESULTS

We now present experimental results demonstrating the effectiveness and scalability of the parallel mesh smoothing algorithm. The meshes used in these experiments are generated during the finite element solution of the linear elasticity equations on a two-dimensional rectangular domain with a hole. A coarse triangular mesh for this domain is generated from an analytic description of the boundary. Succeeding meshes are then generated by recursively bisecting the triangles across their longest edge. The parallel solution of this application is achieved using components of the SUMAA3d software package for adaptive mesh refinement, mesh partitioning, and the solution of linear systems. The test cases are run on a network of Sun workstations and on the IBM SP1 supercomputer.

The effectiveness of the optimization algorithm given in Section 2 for mesh smoothing is illustrated in Figure 6 which shows the upper right quadrant of the computational domain. The mesh on the left is generated by recursively refining the initial mesh with no adjustment in grid point location after each refinement step. The global minimum angle in this mesh is 11.3 degrees and the average minimum element angle is 35.7 degrees. In addition, bisection lines from the coarse mesh are still clearly evident after many levels of refinement. By contrast, the mesh on the right was obtained by adjusting grid point location 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 degrees and the average minimum element angle is 41.1 degrees.

Five smoothing sweeps through the set of nonboundary vertices were used after each refinement step to obtain the smoothed mesh in Figure 6, and we now analyze the

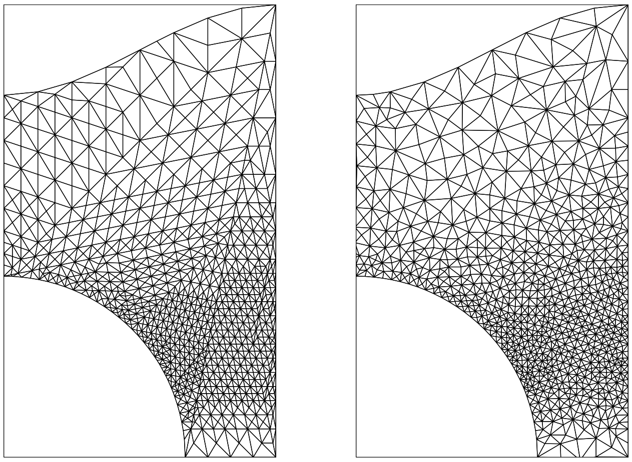

Figure 6: The upper right quadrant of a rectangular region with a circular hole. The figure on the left is the mesh resulting from bisection refinement without smoothing. The figure on the right is the same mesh with smoothing incorporated.

benefits of using an increasing number of sweeps. The mesh we use for these results has an initial minimum angle of 11.31 degrees and 1437 nonboundary vertices that can be adjusted. The experiment is performed on a Sparc 10 scientific workstation, and in Table 1 we show the results for several measures of algorithm performance for one to five smoothing passes. We compare these results with those obtained by using Laplacian smoothing on the same finite element mesh. After five smoothing sweeps, the minimum angle is increased to 20.79 degrees using the optimization algorithm and to 13.13 degrees using Laplacian smoothing. In both cases, the average improvement in the minimum angle of each submesh T_i decreases dramatically after one or two iterations. The optimization algorithm increases the minimum angle in approximately 65% of the local submeshes as compared to 14% when using Laplacian smoothing.

To demonstrate the scalability of the parallel mesh smoothing algorithm, we generate a sequence of problems in which the final mesh in each problem is roughly twice as large as in the preceding problem. If we also assign twice as many processors to each successive problem, then the number of vertices per processor will remain roughly constant over the entire problem sequence. The problem sequence is run on 4 to 64 processors of the IBM SP computer, and the timing results for two smoothing sweeps over the final mesh are shown in Table 2. The results in the final column show that the algorithm is scalable up to 64 processors with slight decreases in the number of vertices smoothed per second in the case of 32 and 64 processors. We note that because the time required to smooth each submesh is not constant, it is possible to see slight increases in the number of vertices smoothed per second as is evident for 4, 8, and 16 processors.

Technique	Pass	Average Angle Improvement	Number Equilibrium Points Found	Number Not Improved	Time (seconds)
Optimization	1	2.23	642	625	7.0
	2	1.30	774	403	6.7
	3	.92	787	439	6.4
	4	.68	690	544	5.9
	5	.53	709	576	6.5
Laplacian	1	1.15	21	1223	1.5
	2	.49	15	1207	1.6
	3	.20	16	1275	1.6
	4	.15	18	1284	1.6
	5	.07	17	1345	1.6

Table 1: Results for mesh smoothing contrasting the optimization based approach with Laplacian smoothing.

ACKNOWLEDGEMENTS

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

References

[1] E. Amezua, M. V. Hormaza, A. Hernandez, and M. B. G. Ajuria. A method of the improvement of 3d solid finite-element meshes. *Advances in Engineering Software*, 22:45–53, 1995.

Number of Processors	Total Number of Vertices	Max Local Number of Vertices	Max Smoothing Time (seconds)	Vertices Smoothed Per Second
4	9478	2371	7.36	322
8	17234	2155	6.53	330
16	36673	2293	6.68	343
32	77504	2423	8.45	286
64	144811	2267	9.59	236

Table 2: Results demonstrating the scalability of the mesh smoothing algorithm on a sequence of problems.

[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] Scott Canann, Michael Stephenson, and Ted Blacker. Optismoothing: An optimization-driven approach to mesh smoothing. *Finite Elements in Analysis and Design*, 13:185–190, 1993.

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

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

[6] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, Cambridge, MA, 1994.

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

[8] Mark T. Jones and Paul E. Plassmann. Computational results for parallel unstructured mesh computations. *International Journal of Computing Systems in Engineering*, 5:297–309, 1994.

[9] Mark T. Jones and Paul E. Plassmann. Scalable iterative solution of sparse linear systems. *Parallel Computing*, 20:753–773, 1994.

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

[11] V. N. Parthasarathy and Srinivas Kodiyalam. A constrained optimization approach to finite element mesh smoothing. *Finite Elements in Analysis and Design*, 9:309–320, 1991.

[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.

[13] Nigel Weatherill. Grid generation by the Delaunay triangulation. Lecture series, von Karmen Insititute for Fluid Dynamics, January 1994.