Engineering with Computers (2014) 30:503–515
DOI 10.1007/s00366-014-0362-1

ORIGINAL PAPER

A parallel log-barrier method for mesh quality improvement and untangling

Shankar P. Sastry · Suzanne M. Shontz

Received: 21 July 2013 / Accepted: 9 April 2014 / Published online: 13 May 2014

Abstract The development of parallel algorithms for mesh generation, untangling, and quality improvement is of high importance due to the need for large meshes with millions to billions of elements and the availability of supercomputers with hundreds to thousands of cores. There have been prior efforts in the development of parallel algorithms for mesh generation and local mesh quality improvement in which only one vertex is moved at a time. But for global mesh untangling and for global mesh quality improvement, where all vertices are simultaneously moved, parallel algorithms have not yet been developed. In our earlier work, we developed a serial global mesh optimization algorithm and used it to perform mesh untangling and mesh quality improvement. Our algorithm moved the vertices simultaneously to optimize a log-barrier objective function that was designed to untangle meshes as well as to improve the quality of the worst quality mesh elements. In this paper, we extend our work and develop a parallel log-barrier mesh untangling and mesh quality improvement algorithm for distributed-memory machines. We have used the algorithm with an edge coloring-based algorithm for synchronizing unstructured communication among the processes executing the log-barrier mesh optimization algorithm. The main contribution of this paper is a generic scheme for global mesh optimization, whereby the gradient of the objective function with respect to the position of some of the vertices is communicated among all processes in every iteration. The algorithm was implemented using the OpenMPI 2.0 parallel programming constructs and shows greater strong scaling efficiency compared to an existing parallel mesh quality improvement technique.

Keywords Parallel mesh quality improvement and untangling · Mesh quality improvement · Mesh untangling · Global algorithm · Log-barrier method · Edge coloring-based parallel transfer

S. P. Sastry (✉)
University of Utah, Salt Lake City, UT 84112, USA
e-mail: sastry@sci.utah.edu

S. M. Shontz
Mississippi State University, Mississippi State, MS, USA
e-mail: sshontz@math.msstate.edu

1 Introduction

Scientific simulation codes are typically run on meshes with millions to billions of mesh elements (e.g., [1–3]). Meshes with billions of elements are becoming more and more common due to the advent of supercomputers and shared-memory machines. Meshing algorithms should take advantage of parallelism to efficiently handle such meshes. Parallel meshing algorithms can be run in a distributed manner on a parallel cluster to be effective both in terms of time and memory. Alternatively, they can be performed on a shared-memory machine; however, memory contention of the shared resources is a challenge that must be overcome in the latter case. Alternatively, hybrid algorithms can be designed that use OpenMP and MPI for intra- and inter-core parallelism, respectively.

There are numerous parallel mesh generation algorithms in existence [4]. We focus on those used to generate triangular and tetrahedral meshes in this paper. In regard to the generation of such meshes, several parallel Delaunay mesh generation algorithms have been developed (e.g., [5–11]). In contrast, only a few parallel advancing front methods [12–15] have been developed. Parallel edge subdivision methods [15–20] have also been designed for

generation of triangular and tetrahedral meshes. The algorithm that currently generates the largest meshes is an exascale mesh generator (which generates meshes with up to 10^{18} elements) and is due to Chrisochoides et al. [21].

For large meshes, it is also important that other mesh operations (e.g., smoothing and untangling which we focus on in this article), be performed in parallel. There are only four algorithms that have been developed for parallel mesh smoothing. Freitag, Jones, and Plassmann developed a parallel mesh optimization technique which employed a parallel nonsmooth optimization technique which smoothed independent sets of vertices simultaneously [22]. Their parallel mesh smoothing algorithm was designed for a parallel random access machine (PRAM) model. This technique performs local vertex movement in which each vertex is moved at a time. In order to void conflicting updates to vertex positions, a vertex-coloring scheme has been used in these algorithms. The positions of vertices of a single color are optimized first, and their new positions are communicated before the positions of vertices of other colors are optimized. The unstructured communication of vertex positions is carried out through a root process.

More recently, Jiao et al. developed a parallel feature-preserving mesh smoothing algorithm for preservation of features, such as corners and creases, in surface meshes. Most recently, Gorman and collaborators developed a parallel hybrid OpenMP/MPI anisotropic mesh smoothing algorithm [23]. To date, only one parallel mesh untangling technique has been developed [24]. This algorithm has been developed for shared-memory processors.

Since there are very few parallel mesh smoothing algorithms and only one parallel mesh untangling algorithm, we also review serial algorithms developed for these purposes. The vast majority of the mesh smoothing algorithms employ optimization techniques to improve the quality of the mesh by adjusting the positions of the vertices (e.g., [25–32]). Other authors have developed physics-based approaches to mesh smoothing. Approaches have been developed based on a torsion-spring system [33, 34] and an electrical system [35]. A force-based mesh smoothing method based on graphs was developed by Djidjev [36]. The vast majority of these methods perform average mesh quality improvement. However, there are a few methods that have been proposed that improve the quality of the worst element in the mesh (instead of the average mesh quality) [31, 32, 37, 38]. For example, Freitag and Plassmann [37] developed an active set method for improving the worst quality element in the mesh; however, the quality metric employed must lead to a convex objective function. In addition, Park and Shontz [38] developed derivative-free mesh optimization algorithms for improving the worst quality element; their algorithms were based on pattern search and multidirectional search methods. The latter optimization methods do not use the gradient of the objective function and hence are slower to converge. In collaboration with Stephen Vavasis, the authors of this paper have developed log barrier mesh quality improvement and untangling techniques for improvement of the worst quality mesh elements [31, 32]. These techniques were more general in that they could perform mesh quality improvement employing any shape-based geometric mesh quality metric. In addition, more improvement was typically seen than with the other methods [31, 32].

Improving the quality of the worst element in the mesh is more challenging in that it involves solving a nonsmooth optimization problem. However, doing so is typically more beneficial from the viewpoint of the associated finite element solver in that it improves its stability, accuracy, convergence, and efficiency [39].

In regard to triangular or tetrahedral mesh untangling, optimization techniques are often used to untangle the mesh and generate valid mesh elements [32, 40–43]. The majority of these optimization-based mesh untangling techniques converge to a local optima of the objective function; however, the method in [40] converges to the global optimum. Agarwal et al. [44] has developed a remeshing procedure for mesh untangling. Bhowmick and Shontz [45] recently designed a graph-based mesh untangling approach. In addition, Remacle et al. [46] and Jiao et al. [47] and his students have proposed mesh untangling schemes for curvilinear meshes and high-order surface triangulations, respectively. Although parallel local mesh untangling algorithms have not been formally studied, Freitag and Plassmann's [37] optimization-based untangling algorithm, for example, can be implemented in parallel in a similar way to their parallel mesh quality improvement technique [22].

Several approaches have recently been developed that combine various aspects of mesh smoothing and untangling [43, 48–50]. A parallel algorithm combining mesh smoothing and untangling has also been developed [24]. Such combined approaches are appealing in that it is sometimes the case that one optimization problem can be solved in lieu of solving two or more optimization problems separately to smooth or untangle the mesh.

In this paper, we describe a parallel log-barrier algorithm for global mesh quality improvement and untangling. We first review the serial log-barrier algorithm [31, 32] and discuss the techniques for its parallel implementation on distributed-memory machines. As we have described before (in [32]), and as we shall see in Sect. 2, the log-barrier method is an efficient global untangling and mesh quality improvement technique in which all the vertices are moved simultaneously. Parallel global mesh

quality improvement methods communicate the gradient of the vertices to neighboring processes. For this purpose, a new use of a coloring technique for synchronization of the data communication is employed to ensure a consistent and efficient execution of our parallel algorithm. We use an edge-based coloring communication synchronization technique in which edges corresponding to a graph of communicating processes (NOT mesh edges) are colored to synchronize the communication. Note that our edge coloring-based technique can be used for local mesh quality improvement or any other parallel algorithm with an unstructured communication requirement. Related node-based versions of such algorithms have been used before to synchronize communication in wireless networks [51]. We carry out numerical experiments to determine the strong scaling efficiency of our algorithm as applied to mesh untangling and mesh quality improvement on a distributed-memory machine with two large meshes. We also carry out a numerical experiment to examine the weak scaling efficiency of our algorithm. The numerical experiments in which we evaluate the strong and weak scaling efficiency of our algorithm and the associated results are discussed in Sect. 3. In Sect. 4, we conclude the paper and provide future research directions.

2 Parallel algorithm for mesh quality improvement and untangling and its implementation

In this section, we describe our parallel algorithm and its implementation in detail. We first recall the mesh untangling and quality improvement algorithm developed in our earlier papers [31, 32] and then describe the challenges and modifications necessary for efficient execution of our parallel algorithm in the subsequent subsections.

2.1 The log-barrier method for mesh untangling and quality improvement

In this paper, the quality of a mesh is improved by a numerical optimization algorithm that dictates the vertex movement in unstructured meshes. In particular, we use the log-barrier method in which the quality of the worst element is improved by maximizing an objective function that uses logarithmic barrier terms. Our previous papers [31, 32] describe the mathematical formulation and serial mesh untangling and mesh quality improvement algorithms in detail. Here, we simply provide the algorithm along with some intuition behind it. The serial algorithm is provided in Algorithm 1 below.

Algorithm 1 The log-barrier method for mesh untangling and quality improvement.

start the iterations with the vector of vertex locations x.
while the quality of mesh is not satisfactory **do**
  for all vertices in the mesh **do**
    if vertex i can be moved **then**
      compute the gradient and the descent vector
    end if
  end for
  {carry out a line search as described below:}
  while the log-barrier objective function $F(\mu, t)$ is not maximized, i.e., the gradient has not vanished **do**
    move all vertices along the descent direction by a distance proportional to the magnitude of the descent vector
    compute the objective function value
    adaptively increase or decrease the distance based on the objective function value
  end while
  update μ and t in the log-barrier objective function such that $\frac{\partial F(\mu,t)}{\partial t} \approx 0$
end while

The main differences between earlier mesh quality improvement methods and the log-barrier method is the use of the log-barrier objective function and a nonsmooth objective function. Earlier methods used some composite functions such as the average value or the root-mean-squared value of the qualities of all the elements in the mesh. Such objective functions improved the average mesh quality, but the quality of the worst element was not guaranteed to be improved. Our method solves a reformulation of the nonsmooth unconstrained optimization problem of improving the quality of the worst element as a smooth constrained optimization problem through the use of the following log-barrier objective function:

$$F(\mu, t) = t + \mu \sum_{i=1}^{m} \log (q_i - t),$$

where q_i is the quality of element i, μ and $t < q_i$ are auxiliary terms, and m is the number of elements in the mesh. Note that the constraints are added to the objective function.

In this paper, we use the smooth aspect ratio quality metric,

$$q_i = \frac{\text{vol}}{\left(\sum_{j=1}^{6} l_j^2\right)^{3/2}},$$

where vol is the volume of the mesh element and l_j, $1 \leq j \leq 6$, are the lengths of the sides of the tetrahedron. We assume that a larger quality q_i for an element i implies that it is of better quality. We lower μ after every iteration and maximize $F(\mu, t)$ so that we ultimately end up maximizing t. We wish to maximize t so that the quality of the elements also improves to some value greater then t. Note that, due

to the log-barrier term, the method has a greater incentive to improve elements with qualities close to t rather than elements that are already of good quality. Thus, our algorithm preferentially improves the quality of poor elements.

In order to untangle meshes, we compute a hybrid quality metric [32] that assumes the signed area or volume of an element as its quality if the element in inverted or assumes the aspect ratio as the quality of an element if it is not inverted. The aspect ratio is usually a function of the signed volume and the lengths of the sides of the element. The hybrid metric can be made smooth through a sigmoid function that provides relative weights for the signed volume and aspect ratio as shown below:

$$Q_i = w_{\text{vol}}\text{vol}_i + w_{\text{qual}}q_i,$$

where Q_i is the hybrid quality, vol_i is the signed volume of the element, q_i is the signed quality of the elements, and w_{vol} and w_{qual} are sigmoid weights. The sigmoid weights are given by

$$w_{\text{vol}} = \frac{1}{1 - e^{\alpha \text{vol}_i}} \text{ and } w_{\text{qual}} = \frac{1}{1 - e^{\beta q_i}},$$

where α and β are scaling factors. If the hybrid quality of all elements is improved from a negative value to a positive value, the mesh is untangled.

We have also shown that the log-barrier method converges to a stationary point by satisfying the Kuhn–Karush–Tucker (KKT) conditions [52]. The KKT conditions for a generic constrained optimization problem are described below. Consider a constrained optimization problem of maximizing $f(x)$, while respecting the k constraints $c_i(x) \leq 0$, $\forall i \in [1, k]$. The Lagrangian is given by

$$L(x, \lambda) = f(x) + \lambda c(x),$$

where λ is a vector of Lagrange multipliers. The active set is given by

$$A(x) = \{i | c_i(x) = 0\}.$$

For a given point x, linear independence constraint qualification (LICQ) holds if the active set gradients $\{\nabla c_i(x) | i \in A(x)\}$ are linearly independent.

Suppose that x^* is a solution to our constrained optimization problem and that LICQ holds at x^*. Then, there is a Lagrange multiplier vector λ^* such that the following conditions (i.e., the KKT conditions) are satisfied at (x^*, λ^*):

- stationarity condition:
  $$\nabla_x L(x^*, \lambda^*) = 0$$
- primal feasibility:
  $$c_i(x^*) \leq 0, \quad \forall i \in [1, k]$$
- dual feasibility:
  $$\lambda_i^* \geq 0, \quad \forall i \in [1, k]$$
- complementarity condition:
  $$\lambda_i^* c_i(x^*) = 0, \quad \forall i \in [1, k].$$

In addition, the satisfaction of the KKT conditions at a point is a first-order necessary condition for the algorithm to converge to a stationary point.

The results from our experiments indicate that the quality improvement by our methods is better then those seen by local mesh quality improvement techniques for some cases. In fact, our technique is able to untangle meshes in less time than existing techniques.

2.2 A parallel algorithm in a distributed-memory environment

In order to develop a parallel algorithm for a distributed-memory system, the following questions must be answered: (a) how should the data be distributed, (b) is the algorithm "embarrassingly" parallel, (c) if not, how can each step of the algorithm be implemented in parallel, and (d) which data need to be communicated from one process to another for correct execution of the algorithm.

In the context of mesh untangling and quality improvement, the data can be distributed through a suitable mesh partitioning technique in which each contiguous part of a mesh is assigned to a process. In particular, each vertex is assigned to a process, and the elements that contain the vertex are also assigned to the process. Note that an element can be assigned to more than one process because it may contain vertices that are assigned to different processes. It must be ensured that only one of these processes computes the quality of the element when the log-barrier objective function is computed during mesh quality evaluation. Also, "ghost" vertices, which are assigned to one of the processes and are also neighbors of vertices in some other process, are also present in all the relevant processes.

Clearly, the entire algorithm is not embarrassingly parallel, but some of the steps in the algorithm can be easily implemented in parallel, whereas other steps require synchronization. It is possible for each process to compute the gradient of the objective function of its own vertices, but each process also needs the gradient of the neighboring vertices during the line search. The gradients can be communicated among the processes at the beginning of each iteration. If a nonlinear conjugate gradient algorithm is being employed to compute the descent direction, the norm of the gradient and the descent direction in the previous iteration are required. Such reduction operations (finding the sum, finding the minimum/maximum element in a vector, etc.) are easily supported in several parallel

programming constructs. All the processes can independently move the vertices during the line search step. Reduction operations are again used to compute the new log-barrier objective function value and appropriate decisions are taken to increase or decrease the step size during this step. Since the same deterministic technique is used by all processes to adaptively change the step size, this can also occur in parallel. After the line search, μ is updated by multiplying it with a constant factor, and a new t is computed. The computation of t is carried out using the bisection method to determine t such that

$$\frac{\partial F(\mu, t)}{\partial t} \approx 0.$$

This step also requires reduction operations which we describe below.

2.3 Edge coloring-based synchronized, unstructured communication

Reduction operations are usually carried out in a structured manner, where a value is communicated from every process to some other process, and the required "reduced" value is communicated back to every process through a series of steps optimized for the network architecture. There is also a need for unstructured communication in which each process transfers the gradient of its boundary nodes to the corresponding neighboring processes in our algorithm. Some processes may communicate with just one other process, whereas some processes may communicate with five or more processes. We must ensure that every process is aware of the processes with which it needs to communicate and to ensure that the order of communication does not result in a deadlock.

We propose to use a greedy, edge coloring-based algorithm to synchronize the unstructured communication. Based on the vertex connectivity, it is easy to compute a graph of communicating processes. Such coloring algorithms have been used to synchronize communication in wireless networks [51]. A detailed analysis of such algorithms can also be found in [51, 53]. In our implementation, we use an edge coloring algorithm. Prior implementations have used node coloring algorithms. A node in the graph corresponds to a process, and an edge represents a communication requirement. We employ a greedy algorithm to color the edges such that no two edges incident on a node have the same color. We carry out a breadth first search (BFS) of the graph and choose an independent set of edges and prioritize the corresponding communication. The BFS is repeated until all the edges are accounted for. Since the communication takes place among independent edges, deadlocks do not occur, and the communications happen in parallel. Note that the number of processes are very low compared to the size of the mesh, and the coloring has to be computed only once. Thus, a serial implementation of the algorithm will suffice for our purposes.

2.4 Distributed data structure for synchronized communication

In our implementation, serial steps at the beginning of the algorithm involve (a) reading a mesh and its vertex partition (after another algorithm is used to partition the mesh), (b) determining the inter-process communication network and the corresponding edge coloring, (c) determining the vertices whose gradients must be communicated, (d) constructing a data structure that facilitates a deadlock-free synchronized communication, and (e) distributing the data structure to all the processes. Steps (a)–(c), and (e) are all straightforward to implement. Below, we discuss the construction of a data structure that is used to determine the vertex gradients which are communicated to other processes and the order of the communication.

The vertex connectivity information is used to determine the list of vertices whose gradients must be communicated to other processes. For each process, a separate array with the indices of the vertices is used to denote the list. Similarly, vertices whose gradient has to be obtained from other processes is also listed using separate arrays for each process. Based on the edge coloring, an ordered list of processes is determined for each process. For a particular process, the gradient commutation should take place with other processes in that order. Since the edge coloring determines the priority of the edge communication, a local list of processes for each process that respects the same priority does not result in a deadlock. The serial steps of the algorithm are described in Algorithm 2 below.

Algorithm 2 The serial steps in the parallel implementation of the log-barrier method.

1: read the mesh vertex and element information
2: read the vertex partitioning information (obtained from Metis, for instance)
3: distribute each partition (including ghost vertices) in separate arrays for each process
4: find the network of communicating processes and color the graph edges
5: **for** all partition i **do**
6: find the list of vertices whose gradient have to be communicated to/from partition j
7: store the list in listTo_j and listFrom_j, respectively
8: store the edge color-based order of the processes with which partition i communicates in my_order
9: send the information above to process i (lines 1-4 in Algorithm 3)
10: **end for**
11: proceed to Algorithm 3 for the parallel mesh untangling and quality improvement algorithm.

2.5 MPI-based parallel implementation

Our detailed parallel mesh optimization algorithm based on the discussion above is presented in Algorithm 3. Every process executes the algorithm until convergence. We have used message passing interface (MPI) constructs in our C++ implementation of the parallel algorithm. Specifically, we used MPI_Allreduce() for our reduction operations and MPI_Send() and MPI_Recv() for the gradient communication. In Algorithm 3,"use reduction" is specified in parentheses for those steps for which the reduction operation is necessary. In the pseudocode, lines 1–4 obtain the data from the root process. Lines 5–11 and lines 15–23 are identical to the serial algorithm except for the use of the reduction operator whenever necessary. In lines 12–15, the computed gradient in the previous steps is communicated to neighboring processes in an orderly manner. All the steps are executed in parallel by all the processes.

Algorithm 3 A parallel log-barrier method for mesh untangling and quality improvement.

1: obtain the list of vertices and elements (for this line and the lines below, from Algorithm 2)
2: obtain the list of vertices, listTo_i, whose gradients should be communicated to process i
3: obtain the list of vertices, listFrom_i, whose gradients should be obtained from process i
4: obtain the order of the list of processes, my_order, with which my process communicates
5: start the iterations with a vector of vertex locations x.
6: **while** the quality of mesh is not satisfactory **do**
7: **for** all vertices in the mesh in my process **do**
8: **if** vertex i can be moved and it belongs to my process **then**
9: compute the gradient and the descent vector
10: **end if**
11: **end for**
12: **for** all the processes with which my process is communicating **do**
13: j = first/next process in my_order
14: send and receive gradients for vertices in listTo_j and listFrom_j, respectively
15: **end for**
 {carry out a line search as described below:}
16: **while** the log-barrier objective function $F(\mu, t)$ is not maximized, i.e., the gradient has not vanished **do**
17: move all vertices along its descent direction by a distance proportional to the magnitude of the descent vector
18: compute the objective function value (use reduction)
19: adaptively increase or decrease the distance based on the objective function value
20: **end while**
21: update the μ and t in the log-barrier objective function such that $\frac{\partial F(\mu,t)}{\partial t} \approx 0$ (use reduction)
22: **end while**

3 Numerical experiments

In this section, we describe the experimental setup and report on the strong and weak scaling efficiency of our algorithm. We compare the strong and weak scaling efficiency to that of Mesquite's [54] implementation of a parallel, MPI-based local mesh quality improvement algorithm. Mesquite's implementation is based on Freitag et al.'s algorithm [22]. Both the log-barrier method and the local mesh quality improvement method compute the gradient of the objective function with respect to the positions of the vertices and carry out a line search to optimize an objective function. Although our objective function is designed to untangle meshes, the types of computations being performed in both algorithms are identical. Thus, the comparison of the strong and weak scaling efficiency of the two algorithms is appropriate.

3.1 Setting up of the experiments

We implemented our parallel algorithm in C++ using MPI constructs. The same mesh and mesh partitioning were used by both the local mesh quality improvement technique and by our global log-barrier technique.

The algorithm used in Mesquite has been developed for local smoothing, i.e., when one vertex is optimized at a time. Thus, a vertex coloring algorithm is used to determine an independent set of vertices to be moved in parallel. Also the new positions of vertices are communicated to all processes through the main process. Thus, the communication is a serial process, i.e., every process communicates the information to the main process, and the main process sends the information to all the other processes that need the information. Our parallel algorithm is developed for global smoothing, i.e., all the vertices are moved together. We use an edge coloring algorithm to identify an independent set of communicating processes. In every iteration, the gradient of vertex positions with respect to each axis is communicated to the neighboring partitions. This communication happens in parallel.

3.1.1 Generation and partitioning of meshes

We generated meshes containing 371,013 nodes and 1,867,366 elements on the support domain and 1,515,275 nodes and 8,911,929 elements on the flange domain (Fig. 1). Both domains were obtained from Inria's surface mesh database [55] for mechanical objects. Tetgen [56] was used to generate the meshes, and Metis [57] was used to partition them. The objective of the Metis partitioner was to lower the number of edge cuts as well as to lower the maximum degree of partition connectivity so that the

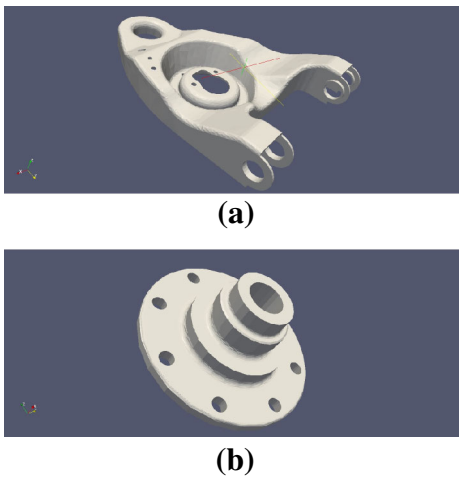

Fig. 1 The two domains on which we constructed large meshes to examine the strong and weak scaling efficiency of our parallel algorithm. The domains were obtained from the Inria database [55]. **a** Support domain, **b** flange domain

number of MPI send and receive operations employed in every iteration is minimized. Metis ensures that the partitioning is well balanced, i.e., the number of vertices assigned to each partition does not vary by more than 3 % between any two partitions.

3.1.2 Parallel architecture

An Intel Xeon CPU E-7-4870 cluster was used to execute our algorithm on the meshes above for a fixed number of iterations. The cluster contains 80 cores each with a clock speed of 2.40 GHz and 750 GB of RAM, and it runs the OpenSUSE 12.2 (x86_64) operating system. GCC 4.7 and OpenMPI 2.0 were used to compile our code. Note that Mesquite [54] also uses the same compilers.

3.2 Results

For our experiments, we report the time taken to execute the code excluding the time taken for input/output operations and the time taken to distribute the mesh among the processes. There are three main parts of the code which contribute to the running time: (a) reading the mesh, (b) distributing the mesh partitions, and (c) running the parallel algorithm. We denote the time taken for (a) and (b) as initialization time. For a single-process execution, the time taken for (b) is not applicable.

We carried out numerical experiments for both meshes on 1, 2, 4, 8, 16, 32, and 64 cores. For the support mesh, we carried out 40 iterations of mesh quality improvement, and for the flange mesh, we carried out ten iterations of mesh optimization. The number of iterations were chosen so that sufficient computation cycles were present in the execution to eliminate the effect of other factors that may affect the running time of the code. In addition, convergence was obtained for this many iterations. Note that the effectiveness of our algorithm has been discussed in detail in our previous two papers [31, 32]. In particular, we proved that our mesh optimization algorithm satisfies the Karush–Kuhn–Tucker (i.e., KKT) conditions for constrained optimization and hence converges to a stationary point [31, 32]. Our optimization method explicitly checks to be sure that it moves the mesh vertices in a direction of ascent (i.e., in order to maximize the objective function). In addition, our mesh quality improvement technique can be used with any smoothly varying mesh quality metric [31, 32]. Our mesh untangling technique can be used with any smoothly varying metric for which the gradient of the objective function points towards the ideal element and the magnitude of the gradient is proportional to the distance from the ideal element [31]. Thus, we will focus mainly on the strong scaling efficiency of our parallel algorithm in this paper. We define the strong scaling efficiency of our algorithm as follows:

$$\frac{T_1}{(p \times T_p)} \times 100\,\%,$$

where p is the number of processes, T_p is the time taken to complete the execution of code for p processes, and T_1 is the time taken by a single process to complete the execution of the code.

A weak scaling efficiency analysis is useful when the number of floating point operations for an algorithm has a direct correlation with the size of the problem. For our problem, the number of floating point operations is not directly related to the size of the problem. Specifically, during the line search to determine the step length by which the vertices must be moved, it is not possible to determine the number of required function evaluations in advance. The number of function evaluations depends on the problem itself in addition to its size. For mesh quality improvement, to carry out a reasonable weak scaling efficiency analysis, the number of function evaluations during the line search has to be kept constant. We carry out weak scaling efficiency tests in our paper and briefly describe the results at the end of this section; however, weak scaling efficiency is not the main focus of this paper. We define the weak scaling efficiency of our algorithm as follows:

$$\frac{T_1}{T_p} \times 100\,\%,$$

where T_p is the time taken to complete the execution of code for p processes and T_1 is the time taken by a single process to complete the execution of the code. Note that the problem size should be proportional to the number of processes to compute the weak scaling efficiency.

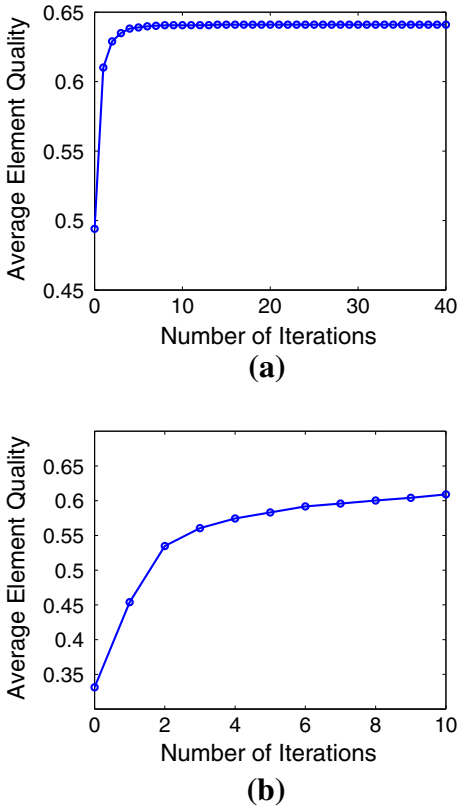

Fig. 2 The average quality vs. number of iterations plot of the parallel local mesh quality improvement algorithm in [58]. **a** Support mesh, **b** flange mesh

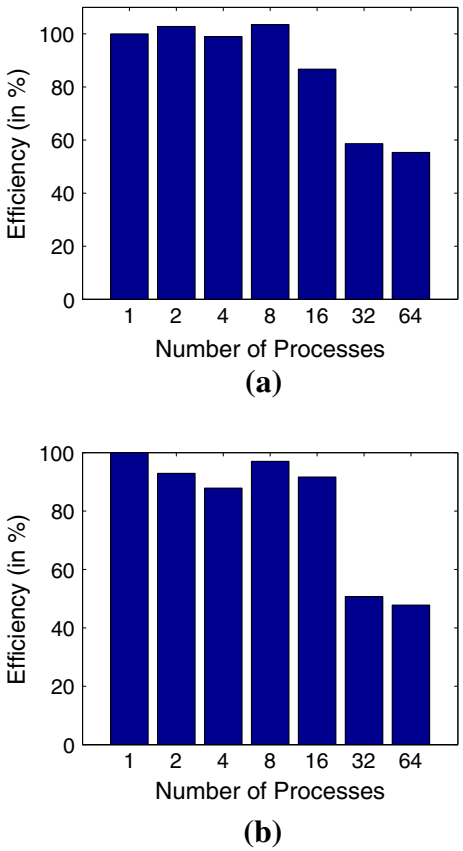

Fig. 3 The strong scaling efficiency of the parallel local mesh quality improvement algorithm in [58] for the two meshes for 1–64 processes. **a** Support mesh, **b** flange mesh

To examine the strong scaling efficiency of our algorithm, we compare it against that of the parallel local mesh quality improvement algorithm [22] implemented in Mesquite [54]. The quality versus iteration plot of this experiment is provided in Fig. 2. We see that the mesh quality has converged. The strong scaling efficiency results of the experiments are provided in Fig. 3. In Fig. 3a, the timing results for the support mesh are shown for all the parallel executions. The objective was to improve the root-mean-square quality of the mesh elements. The mesh quality was improved from 0.49 to 0.64, where the quality was normalized to be 1 for an equilateral element and 0 for a degenerate element. For a single-core execution, the time taken was 23 min and 48 s. The strong scaling efficiency is close to 100 % until about 16 cores, but soon drops off to 50 % for 32 and 64 cores. This is probably due to the serialized communication technique used in their implementation. For the flange mesh, the results are shown in Fig. 3b. The results follow the same trend. The time taken for a single-core execution in this case is 30 min and 47 s, and the root-mean-square mesh quality was improved from 0.33 to 0.61. Note that Mesquite expects the input mesh is already partitioned and split into several files that each process independently reads.

Table 1 provides the time taken by all the processes to read the file containing its own partition. The time mostly reduces with the increase in number of partitions because the mesh has already been partitioned, and the connectivity and neighborhood information is provided as part of the input. Thus, each process has to read smaller files with the increase in the number of partitions. Figure 6 provides the time taken by both local and global techniques for support and flange meshes. The figures excludes the initialization and file I/O time, and the data from the table have been used in Figs. 3 and 5.

For the log-barrier mesh quality improvement and untangling method, we first randomly perturbed the boundary nodes of the mesh by a small amount so that some of the mesh elements were inverted. Our algorithm was then used to untangle the resulting mesh. The worst element quality versus iteration plots are provided in Fig. 4. The initial worst element quality is negative because the mesh is inverted. The hybrid quality metric assumes the signed volume of the element as the quality of the element. As soon as the mesh is untangled, the shape-based aspect ratio is assume to be the quality of the element. Hence, there appears a discontinuity in the plot. The

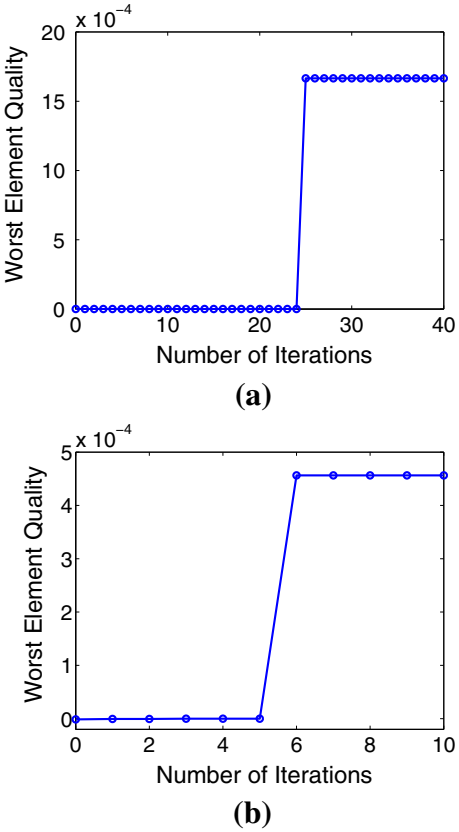

Fig. 4 The worst element quality vs. iterations plot for the parallel log-barrier algorithm. The data points that appear very close to zero are negative qualities associated with the signed volume of the inverted elements. As soon as the mesh is untangled, the quality improves drastically, as the shape-based aspect ratio quality is improved. **a** Support mesh, **b** flange mesh

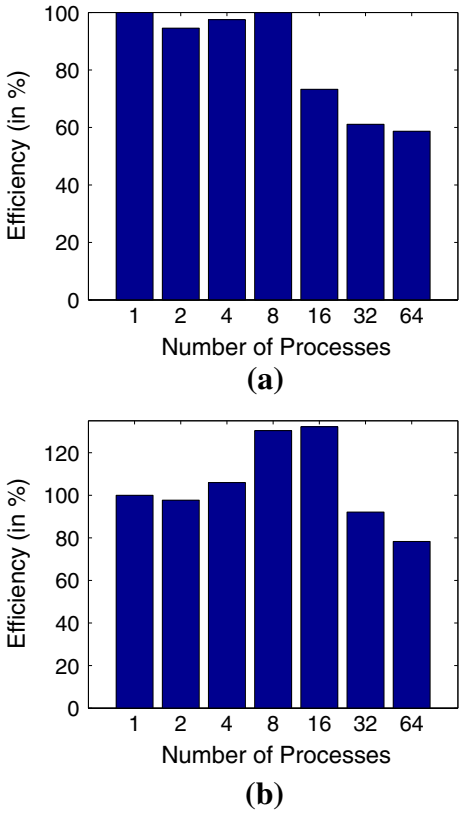

Fig. 5 The strong scaling efficiency of the parallel log-barrier mesh quality improvement and untangling algorithm for the two meshes for 1–64 processes. **a** Support mesh, **b** flange mesh

worst element quality of the mesh converges, but other poor quality elements in the mesh are still being improved during those iterations. The strong scaling efficiency results for our experiments for the log-barrier method are provided in Fig. 5. In Fig. 5a, the timing results for the support mesh are shown for all the parallel executions. It took 25 iterations to successfully untangle the mesh. Forty iterations were carried out in total for the purpose of evaluation of our code. For a single-core execution, the time taken was 24 min and 46 s. The strong scaling efficiency for the parallel executions is very close to 100 % until eight cores. The strong scaling efficiency starts to slowly drop after eight cores to about 60 % for 64 cores.

The strong scaling efficiency results for the flange mesh are provided in Fig. 5b. It took six iterations for our algorithm to successfully untangle the mesh. Ten iterations of untangling and mesh quality improvement were carried out in this experiment for the purpose of evaluation. This is a larger mesh, and better strong scaling efficiency is observed. This is because the volume of gradient communication is proportional to the number of mesh vertices that border two partitions. Typically, it grows slower than the total number of mesh vertices. For this case, a single-core execution took 42 min and 58 s. For 2, 4, 8, and 16 cores, the strong scaling efficiency is >100 %. This is due to availability of additional resources, such as the cache memory space, with the greater number of cores. As the number of partitions increases, the partition size gets smaller. Therefore, it is more likely that the partition fits in the cache memory. As a result, the memory access time reduces, and sometimes, the strong scaling efficiency is >100 %. For more cores, the strong scaling efficiency drops gradually to about 80 % for 64 cores. Thus, for larger problems, the strong scaling efficiency is better for a large number of cores.

Table 1 provides the initialization time that includes the time taken to read the file containing the entire mesh, the time taken to read the file containing the vertex partitioning information, the time taken to distribute the mesh among all the processes, and the time taken to construct the data structure that facilitates the synchronized unstructured communication. For a single-process execution, only the time taken to read the mesh is applicable. Since this is a

Table 1 The time taken (in seconds) for the initialization steps for the mesh quality improvement algorithms

Vertex movement	Domain	Number of processes						
		1	2	4	8	16	32	64
Local	Support	7.75	6.46	3.81	3.17	3.46	4.48	4.83
	Flange	39.23	32.53	13.80	5.81	6.13	5.60	5.90
Global	Support	2.90	6.98	7.12	6.95	7.65	12.26	15.93
	Flange	14.15	34.17	35.43	30.57	32.93	44.44	56.23

For the local vertex movement, the mesh has already been partitioned, and each partition was written into separate files. The table provides the time taken by all processes to read its own partition. For global vertex movement, the root process reads the file containing the mesh and the vertex partitioning and then distributes the mesh to other processes. The times also include the time taken for constructing the data structure that facilitates the synchronized unstructured communication

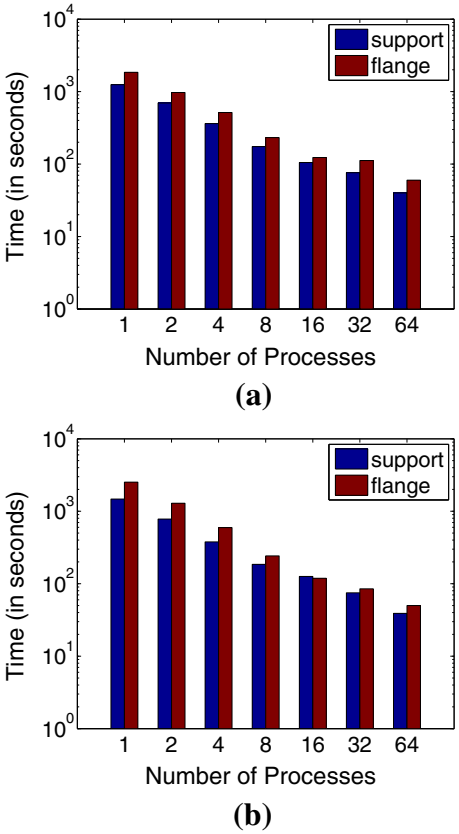

Fig. 6 The time taken (in seconds) for **a** parallel local mesh quality improvement [58] and **b** parallel log-barrier mesh quality improvement and untangling algorithms (excluding the I/O and initialization time). For the support mesh, 14 iteration were executed, and for the flange mesh, ten iterations were executed. These data was used for the bar plots in Figs. 3 and 5. Note that the y-axis is logarithmic

single-process execution, the time taken to read the entire mesh is the same for any number processes used to execute the algorithm.

As opposed to the local vertex movement case, the time taken increases with the number of partitions because the mesh has to be divided among the processes, and the connectivity and neighborhood information has to be explicitly evaluated. In fact, we use the same code to partition a mesh and write the partitioned mesh files for Mesquite's parallel local smoothing algorithm. Therefore, the timing results provided in Table 1 must be interpreted accordingly. For the local vertex movement case, the timing results are for cases where the mesh is already partitioned, and for the global vertex movement case, the timing results are for a case where the mesh is partitioned from a single file and then distributed among various processes. Note that we had partitioned the mesh separately using Metis [57] and stored the results in a separate file. We use the separate file in our implementation to distribute

Table 2 The number of vertices for which the gradient information was transferred to neighboring partitions for both the support and flange mesh for various number of partitions

Domain	Number of processes					
	2	4	8	16	32	64
Support	2,308	5,162	8,818	14,241	22,804	34,984
Flange	15,395	27,031	48,797	74,183	115,888	169,042

Table 3 Results from the numerical experiment to determine the weak scaling efficiency of our algorithm

Mesh size		Number of processors	Time of execution
# of vertices	# of elements		
196,288	836,443	1	200
390,748	1,991,752	2	419
788,300	4,450,220	4	807
1,592,272	9,408,742	8	661
3,184,281	19,329,705	16	816
6,380,652	39,371,494	32	789
12,583,782	78,536,484	64	738

The meshes were generated on the Flange domain. Ideally, the time taken should be constant as the size of the mesh is proportional to the number of processors. In our experiments, we find the time taken is around 800 s when more than four processors are used

the mesh among the processes. The timing for Metis is not included because Mesquite does not include it. Both local and global mesh quality improvement algorithms can use either form of input described above.

As the number of mesh partitions increases, the volume of gradient communication increases, as there are more vertices on the partition boundary. The number of neighboring partitions for a given partition is likely to stabilize to a constant value as the number of partitions increases. As a result, the effect of network latency is likely to become constant as the number of partitions increases. Since the volume of data to be communicated increases, the strong scaling efficiency of our parallel algorithm drops with the increase in number of processes used for the execution. Table 2 provides the number of vertices whose gradients were communicated to neighboring partitions when the meshes were divided into various number of submeshes. It can be clearly seen that the number of vertices increases with the number of partitions.

We also carried out a set of numerical experiments for examining the weak scaling efficiency of our algorithm. We generated meshes on the Flange domain of various sizes that were proportional to the number of processors on which the mesh quality improvement algorithm was executed. Table 3 provides the mesh sizes and the number of processors used to improve their quality. We executed our global mesh quality improvement and untangling algorithm for 25 iterations. The table also provides the results for our experiments. Ideally, the time for execution should remain a constant as the number of processors increases proportionally to the size of the mesh. In our experiment, the time for execution increases as we move from one processor to four processors and then oscillates up to 64 processors. The reason for the increase in the time is possibly due to additional communication requirements as we increase the number of processors and the necessary synchronization operations. Since the same problem is not being solved, oscillation in the time taken to complete the execution is seen in our experiments. The results are comparable to the results reported in [58]. Since the problems we solve are unstructured in nature, it is very difficult to obtain a constant running time as the number of processors and the problem size are proportionally increased. Oscillation in the reported solution times are hard to model in unstructured problems because of factors such as the volume of communication, load balancing, and uncertainty in the number of floating point operations required to solve the problem. Also, the communication latency and bandwidth between any two processors vary as a function of their physical proximity and the network topology on the chip.

4 Conclusions and future work

We have proposed a parallel mesh optimization algorithm based on a log barrier technique and implemented in MPI and C++. As shown in [31, 32], our algorithm satisfies the KKT conditions and converges to a stationary point of the objective function. Our method explicitly checks to see that it moves the mesh vertices in a direction of ascent (i.e., so that the objective function is maximized). Also, our mesh quality improvement algorithm can be used with any smoothly varying mesh quality metric [31, 32]. Any smoothly varying metric for which the gradient of the objective function points towards the ideal element and the magnitude of the gradient is proportional to the distance from the ideal element can be used with our mesh untangling algorithm [31]. All of these properties also hold for our parallel mesh optimization algorithm.

In this paper, we demonstrated the effectiveness of our global algorithm on mesh untangling and mesh quality improvement of 3D tetrahedral volume meshes. In particular, our results demonstrate the strong scaling efficiency of the parallel implementation of our log-barrier mesh untangling and mesh quality improvement algorithm. We have compared the strong scaling efficiency of our algorithm with that of the parallel local mesh quality improvement algorithm by Freitag et al. [22] and have observed an increase in the performance. The strong scaling efficiency of our algorithm can be mainly attributed to the edge coloring-based synchronized parallel communication technique we employed which was not present in earlier work. The algorithm in [22] relied on a serial communication strategy where all the data was first sent to the root node and then distributed to the respective nodes.

We found that our parallel mesh optimization algorithm achieves about 60 % strong scaling efficiency for 64 processes for smaller meshes and about 80 % strong scaling efficiency for larger meshes. We also found that the strong scaling efficiency is more then 100 % for a small number of cores for large meshes. We also found that our algorithm has reasonable weak scaling efficiency beyond four processors, i.e., the time of execution oscillates around 800 s when the size of mesh is proportional to the number of processors used to execute the algorithm. Thus, we expect that the parallel algorithm will scale well for meshes stemming from real-world applications that are employed in large-scale scientific computation codes on machines with hundreds of cores.

For future work, we plan to further reduce the runtime of our parallel mesh smoothing and untangling algorithms by improving the partition of the constraints similar to the approach in [59]. We will also investigate parallel Newton-based and primal-dual Newton-based approaches, which

may result in faster convergence. We also plan to study other edge coloring techniques to synchronize unstructured communication seen in scientific computing applications. In order to reduce the additional time due to network latency, vertex gradients may be communicated through intermediate processes when the volume of communication is small. Finally, we plan to use our parallel mesh smoothing and untangling techniques in parallel simulations involving dynamic meshes arising from applications in medicine and mechanical engineering.

Acknowledgments The authors are indebted to Thap Panitanarak for the use of his partitioned mesh data structure from his MPI implementation of a parallel log-barrier mesh warping algorithm (PLBWARP) in [60]. The work of the first author was supported by the NIH/NIGMS Center for Integrative Biomedical Computing Grant 2P41 RR0112-553-12 and DOE NET DE-EE0004449 Grant. The work of the second author is supported in part by NSF CAREER Grant ACI-1330056 (formerly ACI-1054459). The authors would also like to thank the three anonymous referee for their comments which improved the paper.

References

1. Tautges T, Jain R (2012) Creating geometry and mesh models for nuclear reactor core geometries using a lattice hierarchy-based approach. Eng Comput 28:319–329
2. Aliabadi S, Johnson A, Abedi J, Zellars B (2002) High performance computing of fluid-structure interactions in hydrodynamics applications using unstructured meshes with more than one billion elements. In: Proceedings of the 2002 conference on high performance in computing, lecture notes in computer science, vol 2552. pp 519–533
3. Komatitsch D, Tsuboi S, Ji C, Tromp J (2003) A 14.6 billion degrees of freedom, 5 teraflops, 2.5 terabyte earthquake simulation on the earth simulator. in: Proceedings of the ACM/IEEE SC2003 conference. pp 1–58113-695, 1 March 2003
4. Chrisochoides N (2006) A survey of parallel mesh generation methods. In: Bruaset A, Tveito A (eds) Numerical solution of partial differential equations on parallel computers. Springer, Berlin
5. Nave D, Chrisochoides N, Chew L (2004) Guaranteed-quality parallel Delaunay refinement for restricted polyhedral domains. Comput Geom Theor Appl 28:191–215
6. Galtier J, George P (1997) Prepartioning as a way to mesh subdomains in parallel. in: Proceedings of the ASME/ASCE/SES summer meeting, special symposium on trends in unstructured mesh generation. pp 107–122
7. Linardakis L, Chrisochoides N (2006) Delaunay decoupling method for parallel guarantee quality planar mesh refinement. SIAM J Sci Comput 27:1394–1423
8. Chew L, Chrisochoides N, Sukup F (1997) Parallel constrained Delaunay meshing. In: Proceedings of the ASME/ASCE/SES summer meeting, special symposium on trends in unstructured mesh generation. pp 89–96
9. Chernikov A, Chrisochoides N (2004) Parallel guaranteed quality planar Delaunay mesh generation by concurrent point insertion. In: Proceedings of the 14th fall workshop on computational geometry. pp 55–56
10. Chernikov A, Chrisochoides N (2004) Practical and efficient point insertion scheduling method for parallel guaranteed quality Delaunay refinement. In: Proceedings of the 18th annual international conference on supercomputing. ACM Press, pp 48–57
11. Chrisochoides N, Antonopoulos C, Blagojevic F, Chernikov A, Nikolopoulos D (2009) A multigrain Delaunay mesh generation method for multicore SMT-based architectures. J Parallel Distrb Comput
12. Löhner R, Cebral J (1999) Parallel advancing front grid generation. In: Proceedings of the 8th international meshing roundtable. pp 67–74
13. Löhner R, Camberos J, Marsha M (1990) Unstructured scientific compuation on scalable multiprocessors. In: Hehrotra P, Saltz J (eds) Parallel unstructured grid generation. MIT Press, Cambridge, pp 31–64
14. Löhner R (2013) A 2nd generation parallel advancing front grid generator. In: Proceedings of the 21st international meshing roundtable. pp 457–474
15. De Cougny H, Shephard M (1999) Parallel refinement and corasening of tetrahedral meshes. Int J Meth Eng 46:1101–1125
16. Castanos J, Savage J (1999) PARED: a framework for the adaptive solution of PDEs. in: Proceedings of the 8th IEEE symposium on high performance, distributed computing
17. Oliker L, Biswas R, Gabow H (2000) Parallel tetrahedral mesh adaptation with dynamic load balancing. Parallel Comput 26:1583–1608
18. Rivara M, Pizarro D, Chrisochoides N (2004) Parallel refinement of tetrahedral edges using terminal-edge bisection algorithm. in: Proceedings of the 13th international meshing roundtable
19. Williams R (1991) Adaptive parallel meshes with complex geometry. In: Numerical grid generation in computational fluid dynamics and related fields
20. Rivara M, Carlderon C, Pizaro D, Fedorov A, Chrisochoides N (2005) Parallel decoupled terminal-edge bisection algorithm for 3D meshes. Eng Comput
21. Chrisochoides N, Chernikov A, Fedorov A, Kot A, Linardakis L, Foteinos P (2009) Towards exascale parallel Delaunay mesh generation. In: Proceedings of the 18th international meshing roundtable. pp 319–336
22. Freitag L, Jones M, Plassmann P (1999) A parallel algorithm for mesh smoothing. SIAM J Sci Comput 20(6):2023–2040
23. Gorman G, Southern J, Farrell P, Piggott M, Rokos G, Kelly P (2012) Hybrid OpenMP/MPI anisotropic mesh smoothing. In: Proceedings of the 2012 international conference on computational science, ICCS 2012, procedia computer science, vol 9. pp 1513–1522
24. Benítez D, Rodríguez E, Escobar J, Montenegro R (2013) Performance evaluation of a parallel algorithm for simultaneous untangling and smoothing of tetrahedral meshes. In: Proceedings of the 22nd international meshing roundtable. Springer International Publishing, pp 579–598
25. Canann S, Stephenson M, Blacker T (1993) Optismoothing: an optimization-driven approach to mesh smoothing. Finite Elem Anal Des 13:185–190
26. Parthasarathy V, Kodiyalam S (1991) A constrained optimization approach to finite element mesh smoothing. Finite Elem Anal Des 9:309–320
27. Shephard M, Georges M (1991) Automatic three-dimensional mesh generation by the finite octree technique. Int J Numer Meth Eng 32:709–749
28. Bank R, Smith R (1997) Mesh smoothing using a posteriori error estimates. SIAM J Numer Anal 34:979–997
29. Staten M, Canann S, Tristano J (1998) An approach to combined Laplacian and optimization-based mesh smoothing for triangular, quadrilateral, and quad-dominant meshes. in: Proceedings of the 7th international meshing roundtable. Sandia National Laboratories, pp 479–494
30. Knupp P (1999) Achieving finite element mesh quality via optimization of the Jacobian matrix norm and associated

quantities. Part 1—a framework for surface mesh optimization, technical report SAND 99–0110J, Sandia National Laboratories
31. Sastry S, Shontz S, Vavasis S (2011) A log-barrier method for mesh quality improvement. In: Proceedings of the 20th international meshing roundtable. pp 329–346
32. Sastry S, Shontz S, Vavasis S (2012) A log-barrier method for mesh quality improvement and untangling. Eng Comput 1–15
33. Xu H, Newman T (2006) An angle-based optimization approach for 2D finite element mesh smoothing. Finite Elem Anal Des 42:1150–1164
34. Zhou T, Shimada K (2000) An angle-based approach to two-dimensional mesh smoothing. In: Proceedings of the 9th international meshing roundtable. Sandia National Laboratories, pp 373–384
35. Mezentsev A (2004) A generalized graph-theoretic mesh optimization model. in: Proceedings of the 13th international meshing roundtable. Sandia National Laboratories, pp 255–264
36. Djidjev H (2000) Force-directed methods for smoothing unstructured triangular and tetrahedral meshes. in: Proceedings of the 9th international meshing roundtable. Sandia National Laboratories, pp 395–406
37. Freitag L, Plassmann P (2000) Local optimization-based simplicial mesh untangling and improvement. Int J Numer Meth Eng 49:109–125
38. Park J, Shontz S (2010) Two derivative-free optimization algorithms for mesh quality improvement. In: Proceedings of the 2010 international conference on computational science, vol 1. pp 387–396
39. Shewchuk J (2002) What is a good linear element? Interpolation, conditioning, and quality measures. In: Proceedings of the 11th international meshing roundtable. pp 115–126
40. Knupp P (2001) Hexahedral and tetrahedral mesh untangling. Eng Comput 17:261–268
41. Freitag L, Plassmann P (2000) Local optimization-based simplicial mesh untangling and improvement. Int J Numer Meth Eng 49:109–125
42. Freitag L, Plassmann P (2001) Local optimization-based untangling algorithms for quadrilateral meshes. In: Proceedings of the 10th international meshing roundtable. Sandia National Laboratories, pp 397–406
43. Vachal P, Garimella R, Shashkov M (2004) Untangling of 2D meshes in ALE simulations. J Comput Phys 196:627–644
44. Agarwal P, Sadri B, Yu H (2008) Untangling triangulations through local explorations. In: Proceedings of the 2008 symposium on computational geometry (SoCG 2008)
45. Bhowmick S, Shontz S (2010) Towards high-quality, untangled meshes via a force-directed graph embedding approach. In: Proceedings of the 2010 international conference on computational science, procedia computer science, vol 1. pp 357–366
46. Toulorge T, Geuzaine C, Remacle J, Lambrechts J (2013) Robust untangling of curvilinear meshes. J Comput Phys 254:8–26
47. Clark B, Ray N, Jiao X (2013) Surface mesh optimization, adaption, and untangling with high-order accuracy. In: Proceedings of the 21st international meshing roundtable. pp 385–402
48. Wilson T, Sarrate J, Roca X, Montenegro R, Escobar J (2012) Untangling and smoothing of quadrilateral and hexahedral meshes. in: Topping B (ed) Proceedings of the 8th international conference on engineering computational technology
49. Kim J, Panitanarak T, Shontz S (2013) A multiobjective mesh optimization framework for mesh quality improvement and untangling. Int J Numer Meth Eng 94:20–42
50. Garanzha V, Kudriavtseva L (2011) Gradient projection based optimization methods for untangling and optimization of 3D meshes in implicit domains. In: Proceedings of the II international conference on optimization and applications (OPTIMA 2011)
51. Parthasarathy S, Gandhi R (2004) Distributed algorithms for coloring and domination in wireless adhoc networks. In: Lodaya K, Mahajan, M (eds) FSTTCS 2004: foundations of software technology and theoretical computer science, vol 3328 of lecture notes in computer science. Springer, Berlin Heidelberg, pp 447–459
52. Nocedal J, Wright SJ (2006) Numerical optimization, 2nd edn. Springer, New York
53. Durand D, Jain R, Tseytlin D (1994) Distributed scheduling algorithms to improve the performance of parallel data transfers. SIGARCH Comput Archit News 22(4):35–40
54. Brewer M, Freitag Diachin L, Knupp P, Leurent T, Melander D (2003) The Mesquite mesh quality improvement toolkit. In: Proceedings of the twelfth international meshing roundtable. Sandia National Laboratories, pp 239–250
55. Inria Mesh Database. http://www-roc.inria.fr/gamma/gamma/download/download.php
56. Si H (2007) TetGen: a quality tetrahedral mesh generator and three-dimensional Delaunay triangulator
57. Karypis G, Kumar V (2009) MeTis: unstructured graph partitioning and sparse matrix ordering system, version 4.0. http://www.cs.umn.edu/~metis
58. Freitag L, Jones M, Plassmann P (1995) An efficient parallel algorithm for mesh smoothing. In: Proceedings of the 4th international meshing roundtable. pp 1–18
59. Xu Y, Chen Y (2008) A framework for parallel nonlinear optimization by partitioning localized constraints. in: Proceedings of the international symposium on parallel architectures, algorithms, and programming (PAAP-08)
60. Panitanarak T, Shontz S (2014) A parallel log-barrier based mesh warping algorithm for distributed memory machines (in preparation)