Noname manuscript No.
(will be inserted by the editor)

A Parallel Log Barrier-Based Mesh Warping Algorithm for Distributed Memory Machines

Thap Panitanarak · Suzanne M. Shontz

Received: date / Accepted: date

Abstract Parallel dynamic meshes are essential for computational simulations of large-scale scientific applications involving motion. To address this need, we propose parallel LBWARP, a parallel log barrier-based tetrahedral mesh warping algorithm for distributed memory machines. Our algorithm is a general-purpose, geometric mesh warping algorithm that parallelizes the sequential LBWARP algorithm proposed by Shontz and Vavasis. The first step of the algorithm involves computation of a set of local weights for each interior node which describe the relative distances of the node to each of its neighbors. The weight computation step is the most time consuming in the parallel algorithm. Based on our choice of the mesh partition and the corresponding distribution of data and assignment of tasks to processors, communication among processors is avoided in an embarrassingly parallel computation of the weights. Once this representation of the initial mesh is determined, a target deformation of the boundary is applied, also in an embarrassingly parallel manner. Finally, new coordinates of the interior nodes are obtained by solving a system of linear equations with multiple right-hand sides that is based on the weights and boundary deformation. This linear system can be solved using one of three parallel sparse linear solvers, i.e., the distributed block BiCG, block GMRES, or LU algorithm, all of which support the solution of

T. Panitanarak
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, Pennsylvania 16802
E-mail: txp214@cse.psu.edu

S. M. Shontz
Department of Electrical Engineering and Computer Science
Bioengineering Graduate Program
Information and Telecommunication Technology Center
University of Kansas
Lawrence, Kansas 66045
E-mail: shontz@ku.edu

linear systems with multiple right-hand side vectors. Our numerical results demonstrate good efficiency and strong scalability of parallel LBWARP on up to 64 processors, as the experiments show close to linear speedup in all cases. Weak scalability is also demonstrated. The performance of the parallel sparse linear solvers is dependent on factors such as the mesh size, the amount of available memory, and the number of processors. For example, the distributed LU algorithm gives better performance on small meshes, whereas the distributed block BiCG and distributed block GMRES algorithms yield better performance when the amount of available memory is limited. Finally, we demonstrate the parallel LBWARP performance for a sequence of mesh deformations which can significantly reduce the runtime of the overall algorithm. When applied to k deformations, parallel LBWARP reuses the weight matrix, that was computed during the first deformation, when the distributed LU linear solver is employed. This gives close to k-time performance for sufficiently many deformations.

Keywords parallel computing · mesh warping · mesh deformation · log-barrier method · tetrahedral meshes · sparse linear solvers · multiple right-hand side problem

1 Introduction

Meshes are used to discretize the geometry as required by geometry-based analysis techniques, such as methods for the numerical solution of partial differential equations (PDEs). Such techniques are widely used in many computational simulations in engineering, science, and medicine. For example, meshes have been used to enable accurate prediction of the performance, reliability, and safety of solid propellant rockets [21]. In addition, meshes have been used to enable computational simulations of heart function aiding physicians in their understanding of the heart's anatomy and physiology [53].

There are many applications, such as these, which involve a deformation of the geometric domain as a function of time. For such applications, the mesh must be updated at each timestep in response to the deforming domain boundary in order for the mesh to remain a valid approximation to the geometry. Several mesh warping techniques have been developed recently which update the mesh by mapping it from the source onto the target domain to enable the tracking of the deformation. For example, several PDE-based approaches for solving the mesh-updating problem have been developed [17,31,64,65,73]. Other research has focused on the development of approaches based on elasticity [63,66,74,76] or optimization [37]. These approaches assume that the boundary map from the source domain to the target domain is specified at a particular timestep. Several mesh warping methods [10,18,19,39,61] combine node movement with other techniques which alter the mesh topology in order to generate a high-quality mesh on the target domain. Biomedical engineering and medicine is one area in which mesh warping techniques have recently become very popular [8,9,11,42,51,52,57,68–70].

As computational simulations become more complex and are often multiphysics and multiscale in nature, it is important that meshes be generated and manipulated in parallel on either parallel clusters or multicore machines. The SciDAC Interoperable Technologies for Advanced Petascale Simulations (ITAPS) Center [3] is one example of a large project that addresses the needs of petascale mesh simulations. Furthermore, several parallel mesh generation techniques have been developed (see [23] for a survey); recent techniques have been developed for parallel Delaunay mesh generation (e.g., [7,22,24,28,48]), parallel advancing front mesh generation (e.g., [25,43–45]), and parallel edge subdivision mesh generation (e.g., [20,25,50,54,55,77]).

There are several areas of active research involving parallel post processing of meshes. For example, parallel mesh quality improvement and untangling algorithms have been developed which employ numerical optimization methods to untangle the mesh and improve its quality by repositioning the nodes [14, 58]. Parallel remeshing and mesh adaptation methods have also been proposed which alter the mesh topology in order to improve its quality; often times this is done in response to a change in the PDE solution [30,38,46].

However, in regards to parallel mesh warping, only a few algorithms have been developed for use in dynamic computational simulations. Parallel algorithms have been developed which combine mesh warping with topological operations [59,60,62]. Other parallel mesh warping algorithms have been designed for use in computational fluid dynamics (CFD) applications [29,75]. More recent research on parallel mesh warping algorithms includes meshless techniques developed in [26,47] but still focuses on CFD applications.

In this paper, we propose parallel LBWARP, a parallel log barrier-based mesh warping algorithm for distributed memory systems. Parallel LBWARP is a parallel formulation of the general-purpose, geometric mesh warping algorithm named LBWARP which was proposed by Shontz and Vavasis [64]. Even though LBWARP is computationally intensive when a single deformation is applied, it is rather efficient when multiple deformations are performed. In this case, the computational complexity and also the overall run time of the algorithm decreases significantly. We discuss this advantage of parallel LBWARP in more detail in the paper. The remainder of the paper is organized as follows. In Sections 2 and 3, we provide overviews of the sequential LBWARP algorithm and sparse linear solvers used by the LBWARP algorithm, respectively. Then, we describe our parallelization of the sequential LBWARP method and introduce parallel LBWARP in Section 4. In Section 5, an analysis of the run time of the parallel LBWARP algorithm is discussed. We describe several numerical experiments which were designed to test the performance of our parallel LBWARP method on 3D domains and the resulting run times, speedup, and strong and weak scalability results in Section 6. In Sections 7, we demonstrate an application of parallel LBWARP on heart motion problems. In Section 8, we summarize our work and discuss some future research possibilities related to extensions of our parallel algorithm.

2 An Overview of LBWARP

LBWARP is a log barrier-based mesh warping algorithm which was proposed by Shontz and Vavasis [64]. The algorithm consists of three main steps. The first step is to generate a set of local weights (or inverse distances) describing the relationship of each interior node to its neighbors using a log barrier technique. These sets of weights are computed using an interior point method from nonlinear programming. Next, the boundary nodes are deformed by applying a transformation given by the user. Lastly, a system of linear equations is constructed from the sets of weights in the first step and the new positions of the transformed boundary nodes from the second step. This linear system is then solved for the final positions of the interior nodes.

To compute the weights for interior nodes of a 3D mesh, a local optimization problem is solved for the coordinates of each interior node using a log barrier technique. The optimization problem is as follows

$$\begin{aligned} & \max_{w_{ij}, j \in N_i} \sum_{j \in N_i} \log(w_{ij}) \\ \text{subject to} \quad & w_{ij} > 0, \\ & \sum_{j \in N_i} w_{ij} = 1, \\ & x_i = \sum_{j \in N_i} w_{ij} x_j, \\ & y_i = \sum_{j \in N_i} w_{ij} y_j, \\ \text{and} \quad & z_i = \sum_{j \in N_i} w_{ij} z_j, \end{aligned} \tag{1}$$

where w_{ij} is the weight of node j acting on node i, N_i is the set of neighbors of node i, (i.e., $j \in N_i$ if and only if node j is connected to node i), and x_i, y_i, and z_i are the xyz-coordinates of node i. Since the objective function along with the constraints forms a strictly convex optimization problem, there exists a unique solution which can be found using an interior point method provided an initial feasible point exists [64]. In this paper, we solve equation (1) using the projected Newton method [15] with an initial feasible point in the interior of the domain as described in [64].

Consider the convex combinations of x_i, y_i, and z_i in (1), by defining an $m \times n$ weight matrix A where $A(i,j) = -w_{ij}$ and $A(i,i) = 1$. Assuming the interior nodes are numbered first, and the boundary nodes are numbered last, the matrix A can be written as $[A_I \; A_B]$. Similarly, let x, y, and z be vectors of the x-, y-, and z-coordinates of all of the mesh nodes, respectively. Thus, we have $x = [x_I \; x_B]^T$, $y = [y_I \; y_B]^T$, and $z = [z_I \; z_B]^T$. For m interior nodes and n total nodes, A_I is an $m \times m$ matrix specifying how each node is connected to each of its interior neighbors, and A_B is an $m \times (n-m)$ matrix specifying how each interior node is connected to each of its boundary neighbors. Each of the subvectors x_I, y_I, and z_I contain coordinates of the m interior nodes, and each of the subvectors x_B, y_B, and z_B contain coordinates of the $n - m$ boundary nodes. Hence, we can write the resulting linear system in (1) as follows

$$A_I[x_I\, y_I\, z_I] = \quad -A_B[x_B\, y_B\, z_B]. \tag{2}$$

Once the representation of the initial mesh has been determined, a user-specified boundary deformation can be applied as follows:

$$[x_B\, y_B\, z_B] \rightarrow \quad [\hat{x_B}\, \hat{y_B}\, \hat{z_B}]. \tag{3}$$

Final positions of the interior nodes in the deformed mesh, i.e., $\hat{x_I}, \hat{y_I}$, and $\hat{z_I}$ are then determined by solving the following linear system based on equation (2) and the updates in equation (3);

$$A_I[\hat{x_I}\, \hat{y_I}\, \hat{z_I}] = \quad -A_B[\hat{x_B}\, \hat{y_B}\, \hat{z_B}]. \tag{4}$$

See [64] for more details.

Although the LBWARP algorithm is initially computationally intensive due to the construction of the weight matrix, once the weight matrix has been computed, the LBWARP algorithm can reuse this matrix in additional mesh deformations provided an LU factorization method is used to solve the linear system. In other words, only the boundary deformation and the linear solution steps are performed for additional deformations. The complexity is then typically $O(m^2)$ for additional deformations but depends on the number of nonzero elements in A_I, i.e., $nnz(A_I)$ per deformation [16].

3 Sparse Linear Solvers with Multiple Right-Hand Sides

The choice of linear solver which is employed to solve (4) affects the runtime of the warping algorithm. Certainly, the sparsity and structural symmetry of A_I should be taken into account when selecting a linear solver.

Moreover, each deformation requires three linear solves, i.e., one for each of the x-, y- and z-coordinates (for 3D tetrahedral meshes). Each linear solve employs the same left-hand side matrix but a different right-hand side (RHS) vector. Thus, a single linear solver which takes into account the above properties and is able to address the multiple right-hand side problem should be employed.

Both direct and iterative methods can be used to solve (4), and both categories of methods have their advantages and disadvantages. Direct solvers directly support multiple RHS vectors, but their use can increase the number of nonzero elements in the matrix during row elimination. Thus, matrix reordering is required to minimize any nonzero fill-in. On the other hand, iterative solvers do not require reordering but instead need to be modified to support multiple RHS.

Thus, in this paper, we consider the use of three different parallel sparse linear solvers for the solution of (4) in parallel LBWARP. Next, we give an overview of the serial versions of these methods.

Block BiCG (BiConjugate Gradient) [49] is a modified version of the biconjugate gradient (BiCG) method [27] to support multiple RHS. Block BiCG is essentially identical to the standard BiCG method with the only difference being that operations are performed with multivectors instead of single vectors. Thus, both methods are based on the conjugate gradient (CG) method [32] with an extension to provide a capability for solving nonsymmetric linear systems $Ax = b$. As in CG, the method uses search directions,

$$d^{(i+1)} = r^{(i+1)} + \beta_i d^{(i)}$$

to update the residuals, $r^{(i+1)}$, and solution approximations, $x^{(i+1)}$, such that

$$r^{(i+1)} = r^{(i)} - \alpha_i A d^{(i)} \quad \text{and} \quad x^{(i+1)} = x^{(i)} + \alpha_i d^{(i)},$$
$$\text{where} \quad \alpha_i = {r^{(i)}}^T r^{(i)} / {d^{(i)}}^T A d^{(i)} \quad \text{and} \quad \beta_i = {r^{(i+1)}}^T r^{(i+1)} / {r^{(i)}}^T r^{(i)}.$$

To handle nonsymmetric systems, instead of using only one orthogonal sequence of residuals and conjugate directions as in CG, BiCG uses two mutually orthogonal sequences (or two sequences of residuals and conjugate directions). In addition to computing $d^{(i)}$ and $r^{(i)}$, $\tilde{d}^{(i)}$ and $\tilde{r}^{(i)}$ are also computed similarly by replacing A with A^T. Moreover, the computations of α_i and β_i are replaced by

$$\alpha_i = {\tilde{r}^{(i)}}^T r^{(i)} / {\tilde{d}^{(i)}}^T A d^{(i)} \quad \text{and} \quad \beta_i = {\tilde{r}^{(i+1)}}^T r^{(i+1)} / {\tilde{r}^{(i)}}^T r^{(i)}.$$

Thus, the cost per iteration of BiCG is approximately twice that of CG. The cost per iteration of BiCG is approximately the cost of computing two inner products, five scalar-vector multiplications and additions, and two matrix-vector products or $O(12m + 2m^2)$.

To handle multiple RHS, BiCG can be modified to solve all RHS at once or to solve the system $AX = B$ where X and B are $m \times c$ matrices and c is the number of right-hand side vectors. To update the approximations to the solution, the following formulas are now used

$$D^{(i+1)} = R^{(i+1)} + D^{(i)}\mathrm{B}^{(i)}, \qquad R^{(i+1)} = R^{(i)} - AD^{(i)}\mathrm{A}^{(i)},$$
$$\tilde{D}^{(i+1)} = \tilde{R}^{(i+1)} + \tilde{D}^{(i)}\mathrm{B}^{(i)}, \qquad \tilde{R}^{(i+1)} = \tilde{R}^{(i)} - A\tilde{D}^{(i)}\mathrm{A}^{(i)},$$

and

$$X^{(i+1)} = X^{(i)} + D^{(i)}\mathrm{A}^{(i)}$$

where

$$\mathrm{A}^{(i)} = ({\tilde{D}^{(i)}}^T A D^{(i)})^{-1} {\tilde{R}^{(i)}}^T R^{(i)} \quad \text{and} \quad \mathrm{B}^{(i)} = ({\tilde{R}^{(i)}}^T R^{(i)})^{-1} {\tilde{R}^{(i+1)}}^T R^{(i+1)}.$$

Note that D and R are now $m \times c$ matrices, while A and B are $c \times c$ matrices. As we can see, the complexity of the algorithm has increased, as it requires more matrix operations. Furthermore, the convergence and stability of the algorithm are also affected. Thus, additional efforts to maintain these properties are needed such as deflation, i.e., removing some right-hand sides from the process (see [49, 71]). Since operations on vectors are now performed using matrices, for c right-hand sides, the complexity of block BiCG is $O(12cm+2cm^2)$.

Block GMRES (Generalized minimal residual) [72] is a modified version of GMRES [56] to support multiple RHS using a similar approach as the block BiCG method. GMRES itself is an extension of the minimal residual (MINRES) method which can only be used to solve symmetric systems. Similar to MINRES, it approximates the solution by generating a sequence of orthogonal vectors with minimal residual. However, without symmetry, all previously generated vectors are needed and must be retained to construct the approximations as follows:

$$x^{(i)} = x^{(0)} + y_1 v^{(1)} + \cdots + y_i v^{(i)},$$
$$\text{where } y_k \text{ minimizes } \|b - Ax^{(i)}\| \text{ and } v^{(i+1)} = w^{(i)}/\|w^{(i)}\|.$$

For each i, $w^{(i)}$ is initialized with $Av^{(i)}$ and explicitly updated by $w^{(i)} = w^{(i)} - (w^{(i)}, v^{(k)})v^{(k)}$ for $k = 1, \ldots, i$. By defining $r^{(0)} = b - Ax^{(0)}$ and the $(k+1) \times k$ upper Hessenburg matrix H_k from the Arnoldi's relation $AV_k = V_{k+1}H_k$ where V_k is the orthonormal basis of the Krylov subspace $K_k(A, r^{(0)})$ built by the Arnoldi procedure, we can compute $y_k = \|r^{(0)}\| H_k^{-1} e_1$. The cost of the the k^{th} iteration of GMRES (without restarting) is approximately the total time of $k+1$ inner products, $k+1$ scalar-vector multiplications and additions, and one matrix-vector product or $O(3(k+1)m + m^2)$. Consequently, the extension to block GMRES results in $O(3(k+1)cm + cm^2)$ complexity.

***LU* Decomposition** is a factorization of a matrix A into LU [1], where L and U are $m \times m$ unit lower triangular and upper triangular matrices, respectively. This factorization can be used to indirectly solve the linear system $Ax = b$ with the equivalent system $LUx = b$ which can be solved by performing two triangular solves as follows:

$$Ax = b \Leftrightarrow LUx = L(Ux) = Ly = b.$$

More specifically, forward substitution is performed to solve for y from $Ly = b$, and then backward substitution is performed to solve for x from $Ux = y$.

The LU decomposition can be directly applied to any matrix including sparse matrices. However, a row operation during the decomposition can result in the generation of additional nonzero elements which were previously zero (called nonzero fill-in). This increases memory usage and the number of

[1] The factor LU exists only if A is nonsingular. In the event an element on the diagonal of A is zero or nearly zero, partial pivoting is required.

computations in the algorithm. In this paper, we use nested dissection, a fill-reducing ordering, in order to minimize nonzero fill-in which occurs during the LU decomposition.

4 Parallel LBWARP

In this section, we describe our parallel formation of the LBWARP algorithm; the resulting method is referred to as parallel LBWARP. The parallel LBWARP method contains three steps, i.e., weight generation, boundary deformation, and linear solution.

4.1 Parallelization of Weight Generation Step

As described in the previous section, the first step of the LBWARP algorithm is to use optimization to generate a set of local weights which specifies how a given interior node is represented as a convex combination of its neighbors. To parallelize this step, it is important that the interior nodes are equally distributed among the processors to balance the workload. Assume that p processors are used for solving m optimization problems corresponding to the m interior nodes. A subset of $\lceil m/p \rceil$ consecutive interior nodes are assigned to each processor for simplicity to identify the ownership of the distributed interior nodes. As the computation during the weight generation step is node-based, we represent a mesh as a graph such that each graph node and edge represent the corresponding mesh node and edge, respectively, and the connectivity among nodes is preserved.

Before the actual computation of the weight matrix begins, the neighbors of each subset of the interior nodes are pre-computed and sent to the corresponding processors. There will be redundant copies of nodes among processors, i.e., one node can be assigned to more than one processor, and the amount of memory required by each processor varies since the number of neighbors for each subset of the interior nodes is different. Even though mesh partitioning can be used to balance the distribution, it also introduces additional complexity, and can decrease the overall performance.

Fig. 1 demonstrates our partitioning approach. First, interior nodes are determined as shown (Fig. 1(a)). Then, they are equally distributed to the processors (Fig. 1(b)). After that, the neighbors of each subset of interior nodes are computed and sent to the corresponding processors (Fig. 1(c)). Once all processors receive their subsets of interior nodes and the subset's neighboring nodes, they compute their local weights independently in an embarrassingly parallel manner using the projected Newton method without communication during the weight computations.

During the weight generation step, the $m \times n$ matrix $A = [A_I \;\; A_B]$ is constructed by formulating and solving m local optimization problems. More specifically, solving a single optimization problem for the i^{th} interior node given by (1) yields the i^{th} row of A. With the parallel approach described above, since each processor acquires a subset of approximately $\lceil m/p \rceil$ consecutively numbered interior nodes and all of their neighbors, approximately $\lceil m/p \rceil$ consecutive rows of the matrix A can be generated simultaneously on p processors without communication. After the processors finish generating the local weights for their assigned interior nodes, each processor owns non-overlapping, consecutive rows of the weight matrix A, (i.e., processor zero generates rows one through $\lceil m/p \rceil$, processor two generates rows $\lceil m/p \rceil + 1$ through $2\lceil m/p \rceil$, and so on).

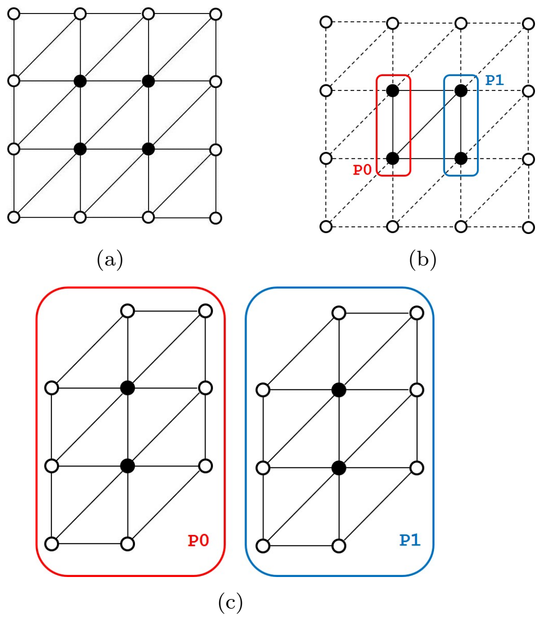


Fig. 1 (a) The original mesh before partitioning; the black and white nodes represent the interior and boundary nodes, respectively. (b) For computational load balance during the weight gneration step, first, only interior nodes are partitioned and sent to processors. (c) Then, neighbors of each subset of interior nodes are computed and sent to the corresponding processors.

4.2 Parallelization of Boundary Deformation Step

The next step is to apply the deformation of the boundary nodes in parallel. This step involves computing the right-hand side of equation (4). Since part of the matrix A_B is generated and owned locally by each processor during the weight generation step, the boundary deformation step can also be performed in an embarrassingly parallel manner by sending each processor the coordinates of the boundary nodes, i.e., $\hat{x}_B$, $\hat{y}_B$, and $\hat{z}_B$. Although there is redundancy in all of the processors owning the entire set of boundary nodes, this allows the processors to simultaneously compute their respective parts of the right-hand side vector without communication. This is equivalent to computing a portion of the right-hand side vector locally on each processor.

4.3 Parallelization of Linear Solution Step

The last step is to solve the three linear systems in equation (4) for the final positions of the interior nodes in the deformed mesh, i.e., $\hat{x}_I, \hat{y}_I$, and $\hat{z}_I$. The parallel linear solver which is used in the linear solution step with the distributed matrix A_I should take advantage of its sparsity and also support multiple right-hand side vectors.

To this end, we consider parallel versions of block BiCG, block GMRES, and LU decomposition for distributed memory machines for use in the linear solution step of parallel LBWARP. We refer to them as DistBlBiCG, DistBlGMRES, and DistLU, respectively. We chose these three algorithms based on our preliminary experiments with the parallel block GMRES and SuperLU_DIST algorithms [40] implemented in the Amesos2 and Belos packages [13] for the Trilinos project [4]. Our preliminary results with the Trilinos package demonstrated good speedups when solving linear systems based on multiple RHS solvers on matrices generated from the weight generation step. By implementing our own linear solvers, we can control a consistency among the solvers as we can implement them under the same environments such as data structures and compilers.

Distributed block BiCG and distributed block GMRES: Our implementations of DistBlBiCG and DistBlGMRES are based on [12,49,71,72]. The $m \times m$ sparse matrix is distributed based on a row-wise distribution among the p processors. Thus, each processor owns a non-overlapping $\lceil m/p \rceil$ consecutive rows of the original matrix. A RHS vector corresponding to the matrix is also distributed in a similar manner. Thus, matrices that normally cannot be fit in memory on a single processor can be processed with this approach. However, there is a trade-off in terms of processing time since some matrix-matrix and/or matrix-vector operations typically require off-processor information in the form of message-passing communication. The two main parallel routines in the main loop of both solvers are matrix-vector multiplication and vector dot product. For parallel matrix-vector multiplication, this can be done by distributing the vectors (d or v for DistBlBiCG or DistBlGMRES, respectively) to all processors according to the rows of matrix that each processor owns. Then, each processor multiplies the received vector with its own rows and the result vector is stored. For parallel dot product, each processor computes the partial result of the inner product from the rows which it owns. After that, these partial results are summed globally using MPI_Reduce to obtain the inner product.

We performed a set of preliminary experiments in order to determine whether or not it was necessary to perform either reordering on A_I or preconditioning when solving the linear system in equation (4). We experimented with the use of the nested dissection (ND) reordering and application of an ILU(0) preconditioner on block GMRES using the Belos package. Typical results from our preliminary experiments can be seen in Fig. 2. Although applying the preconditioner increases the convergence rate (Fig. 2(a)), it only reduces the linear solution time when up to 16 processors are used. For a larger number of processors, (i.e., 32 processors or more), the overhead of computing the preconditioner becomes more visible, as the size of our deforming mesh problem is not large enough, (i.e., our preliminary experiments are for a deforming mesh with approximately 6M nodes) for this strategy to pay off (Fig. 2(b)). (For this problem, the linear solution step takes less than 10% of the overall warping time, which is around 20 seconds.) Given the relatively low condition number of the weight matrix, (i.e., 37 for this problem), the linear systems can be solved without use of a preconditioner and can still obtain a good convergence rate. In such cases, it should be clear that is it the most beneficial to employ block GMRES without either reordering or preconditioning. Thus, we do not use either reordering or preconditioning when solving the linear systems in equation (4) with the DistBlBiCG and DistBlGMRES solvers.

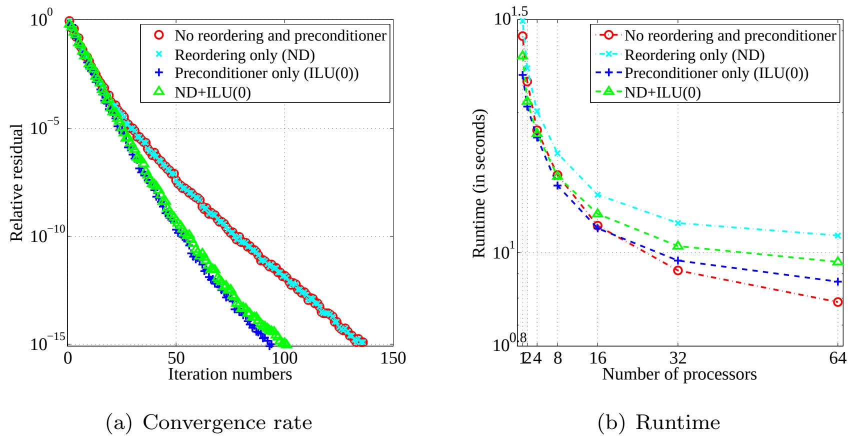


Fig. 2 The effect of the nested dissection reordering and ILU(0) preconditioner on the performance of the block GMRES solver.

Distributed LU: Like other sparse LU factorizations, our DistLU algorithm consists of reordering to reduce fill-in, symbolic factorization, numerical factorization, and triangular solves. While partial pivoting is essential for general matrices, the factorization of A_I does not require partial pivoting since A_I is weakly diagonally dominant. Without partial pivoting, some communication

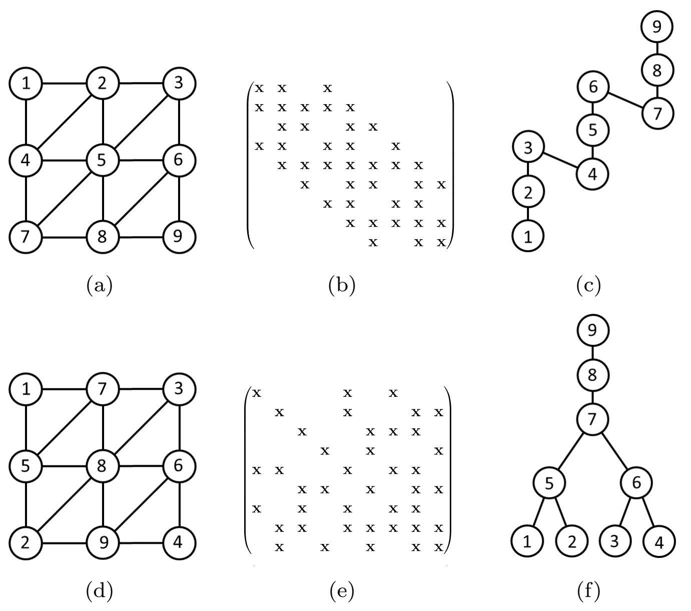
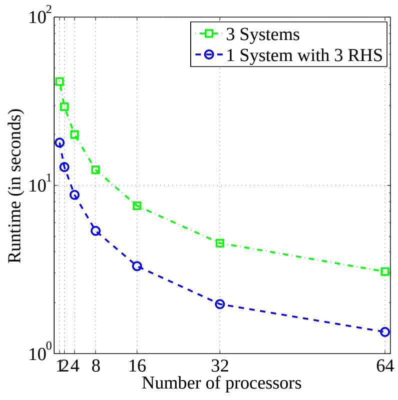
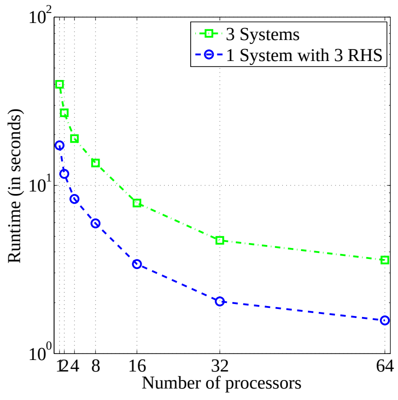


Fig. 3 (a) A mesh with natural ordering, (b) its adjacency matrix, and (c) its elimination tree. The same mesh after applying (d) nested dissection reordering, and (e) its corresponding adjacency matrix and (f) elimination tree. Note that '**x**' indicates a nonzero element in the matrix.

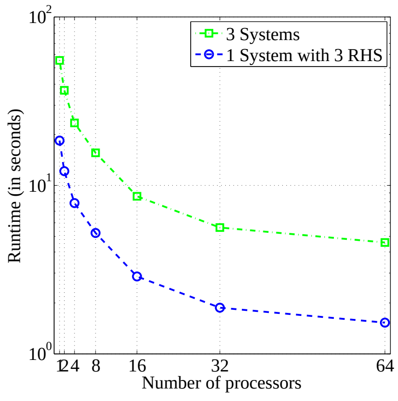
and computation can be avoided. Also, based on the symmetric structure of A_I, the symbolic factorization (which is used to determine the fill-in in the L and U factors) is easier to compute than for unsymmetric matrices. More specifically, our parallel sparse LU factorization is similar to SuperLU_DIST [40] introduced by Li *et al.* but is simpler in terms of the amount of computation performed and the complexity. We apply the ND reordering [5] which reduces fill-in for the sparse matrix A_I using MeTiS [36]. The algorithm finds an elimination ordering of the matrix using a divide and conquer approach. The new elimination ordering can be used to exploit the matrix columns that can be updated simultaneously, (i.e., they are independent). The result of the nested dissection algorithm also yields an elimination graph or task graph of the matrix to use as an elimination order of all of the columns. Fig. 3 illustrates meshes with their natural ordering and after applying a nested dissection reordering along with their adjacency matrices and elimination trees. A mesh with natural ordering (Fig. 3(a)), its adjacency matrix and elimination tree are shown in Figs. 3(b) and 3(c), respectively. The elimination ordering is sequential and dependent on the previous nodes. Thus, to be able to perform elimination of the k^{th} column of the matrix, all columns from one up to $(k-1)$ need to be eliminated first. Figure 3(d) shows the same mesh as above after nested dissection reordering is applied. Similarly, its adjacency matrix and elimination tree are shown in Figs. 3(e) and 3(f), respectively. Now, since nodes 1-4 are pairwise independent, columns 1-4 of the matrix can be eliminated simultaneously. Similarly, columns 5-6 can also be eliminated at the same time.



(a) DistBlBiCG



(b) DistBlGMRES



(c) DistLU

Fig. 4 A comparison of the runtimes for the three parallel sparse linear solvers when solving with and without multiple RHS vectors: (a) DistBlBiCG, (b) DistBlGMRES, and (c) DistLU. (Note: A log scale is used for the vertical axis.)

Benefits of parallel linear solvers with multiple RHS support: A comparison of the runtimes for the three parallel sparse linear solvers, i.e., the DistBlBiCG, DistGMRES, and DistLU algorithms, with and without multiple RHS vector support, is shown in Fig. 4. Recall that solving equation (4) for the final positions of the interior nodes in the deformed mesh requires three linear systems to be solved, i.e., one linear system per nodal coordinate. For systems without multiple RHS vector support, the linear systems are solved independently. Whereas, the nodal coordinates are solved for simultaneously when linear solvers with multiple RHS vector support are employed. Simultaneous linear solves result in reduced runtime because the overhead during the initialization and some additional operations can be combined and reused in order to avoid redundant computation.

Algorithm 1 Parallel LBWARP algorithm

```
1:  // Generate neighbor lists
2:  for all $p$ processors in parallel do
3:      generate neighbor lists of all local interior nodes
4:      request coordinates of non-local neighbors using MPI_Alltoallv
5:  end for
6:  synchronization using MPI_Barrier
7:  // Step 1: Generate a weight matrix, $A = [A_I \; A_B]$
8:  for all $p$ processors in parallel do
9:      for each local node $v$ do
10:         solve the optimization at $v$ for the weight matrix row $v$
11:     end for
12: end for
13: synchronization using MPI_Barrier
14: // Step 2: Compute the right-hand side vectors in equations (4)
15: for all $p$ processors in parallel do
16:     compute $b_x = -A_B\hat{x}_B$, $b_y = -A_B\hat{y}_B$, $b_z = -A_B\hat{z}_B$
17: end for
18: synchronization using MPI_Barrier
19: // Step 3: Solve the linear systems (4) for $\hat{x}_I$, $\hat{y}_I$, and $\hat{z}_I$
20: solve $A_I\hat{x}_I = b_x$, $A_I\hat{y}_I = b_y$, $A_I\hat{z}_I = b_z$ using
21:     either DistBlBiCG, DistBlGMRES or DistLU
```

4.4 Parallel LBWARP Algorithm

The complete parallel LBWARP algorithm is shown in Algorithm 1. Assume that interior nodes are distributed in a row-wise distribution among all processors. Thus, each processor owns non-overlapping $\lceil m/p \rceil$ interior nodes including their x, y, and z coordinates, and the completed target boundary coordinates, $\hat{x_B}$, $\hat{y_B}$, and $\hat{z_B}$. The first step of the algorithm (lines 1-6) is to generate neighbor lists which will be used in the next step. It involves **MPI_Alltoallv** communication to exchange the coordinates of non-local neighbors. After that, each processor can compute its local weights independently without any communication as shown in lines 7-14. Since each interior node corresponds to one row in the weight matrix, a set of local interior nodes at processor p results in the partial weight matrix rows, A. The rest of the algorithm focuses on construction and solution of the system (2) for $\hat{x_I}$, $\hat{y_I}$, and $\hat{z_I}$ using one of the parallel sparse linear solvers, i.e., DistBlBiCG, DistBlGMRES, or DistLU.

Lines 15-19 show the boundary deformation step. Each processor has its own copy of $\hat{x}_B$, $\hat{y}_B$, and $\hat{z}_B$. Thus, the processors simultaneously compute their respective parts of the right-hand side vectors using equation (4). Finally, the linear systems shown in equation (4) can be solved for $\hat{x}_I$, $\hat{y}_I$, and $\hat{z}_I$ using one of the three parallel sparse linear solvers, DistBlBiCG, DistBlGMRES, or DistLU.

5 Parallel Analysis

As described in Section 4, parallel LBWARP consists of three main steps, i.e., the weight generation, the boundary deformation, and linear solution the same as does LBWARP. We now discuss the performance gain in each these steps of parallel LBWARP.

Assume that the maximum time to solve a single optimization problem in (1) to determine the weights for interior node i is t_{op}. Thus, LBWARP takes at most mt_{op} total time to compute the sets of weights for the m interior nodes.

In the case of parallel LBWARP, for analysis purposes, assume optimal load balancing across all available p processors and that each processor is assigned to work on approximately $\lceil m/p \rceil$ distinct interior nodes. Hence, each processor takes at most $\lceil m/p \rceil t_{op}$ time to compute its own set of local weights. Since all processors can work simultaneously without any communication among them during this step, the total time to compute all weights is still $\lceil m/p \rceil t_{op}$. There is some additional computation and communication to generate neighbor lists for the interior nodes which can be viewed as a conversion from the original mesh representation to a graph representation. It is easy to see that if the nodes belong to the same element, they are neighbors of one another. Assume that there are n_e elements in the mesh. For LBWARP, to find all possible neighbor relationships of the interior nodes, all mesh elements have to be visited. Thus, the total time for the sequential conversion is $n_e t_{con}$ where t_{con} is the maximum time required to inspect a single mesh element. For parallel LBWARP, if $\lceil n_e/p \rceil$ elements are distributed among all p processors, each processor can visit its elements in $\lceil n_e/p \rceil t_c$ time. However, a local neighbor list generated by each processor is not yet complete since each interior node belongs to multiple mesh elements and those elements may be distributed to different processors. To obtain the complete neighbor list for a given processor, each processor needs to gather neighbor lists from other processors which requires at most $(\lceil m/p \rceil pd)t_{com}$ time where d is the maximum degree of a node and t_{com} is the time to send a single message. Thus, the total time required in the weight generation step is bounded above by $mt_{op} + n_e t_{con}$ for LBWARP and $\lceil m/p \rceil t_{op} + \lceil n_e/p \rceil t_{con} + (\lceil m/p \rceil pd)t_{com}$ for parallel LBWARP. In general, the time used for distribution and gathering is very small compared to the time for solving the optimization problems used to generate the weights.

Fortunately, since there is no further communication after all local neighbor lists have been generated in the weight generation step, this step of parallel LBWARP is very scalable in terms of both strong and weak scaling.

The boundary deformation step involves simple parallel computation of the right-hand side vectors as shown in equation (4).

Since rows of A_B have already been distributed among the processors, the only work that needs to be done is to distribute the vectors $\hat{x}_B$, $\hat{y}_B$ and $\hat{z}_B$ to the processors so that the processors can simultaneously compute portions of the right-hand side vectors based on the part of A_B that each one owns. Assuming the time to compute the right-hand side vectors sequentially is t_c, the approximate run time in parallel is $\lceil m/p \rceil t_c$.

In the linear solution step, the time required to solve the system depends on the algorithm used for this step. For iterative solvers, similar to the sequential versions, the complexity is based on the number of iterations required for convergence (and the time per iteration) which is approximately $O(m^2/p)$ per iteration with some communication overhead. For the distributed LU, the complexity is based on $nnz(A_I)$ and the structure of A_I, which is approximately $M = \sum_{k-1}^{n} c_k r_k$, where c_k and r_k are the numbers of off-diagonal elements in each column of block column k and each row of block row k, respectively. Thus, the parallel runtime is approximately $O((nnz(A_I)+M)/p+H(p))$, where $H(p)$ is communication overhead of p processors from broadcasting messages.

6 Numerical Experiments

In order to test the performance of our parallel LBWARP algorithm, we perform several numerical experiments on 3D tetrahedral meshes. The implementation of parallel LBWARP is in C/C++ using the message-passing interface (OpenMPI version 1.7.3). We use the dense and sparse vector and matrix routines in the Eigen library [35] where vector or matrix operations are required. All of our experiments were run on the CyberStar cluster available for our use at The Pennsylvania State University [1]. More specifically, 192 Dell PowerEdge R610 servers were used. Each server provides 2 quad-core Intel Nehalem processors running at 2.66 GHz and 24 GB of RAM. In all of our experiments, we use only one core per server node to maximize the memory capacity. For both DistBlBiCG and DistBlGMRES, the relative convergence tolerance and the maximum numbers of iterations are set to 10^{-5} and 10^{3}, respectively. The initial guess vector for both algortihms is set to zero. The GMRES restart iteration is 30.

The 3D domains that we use in our experiments, i.e., Menger sponge and Luer connector, are shown in Figs. 5(a) and (d), respectively. Meshes on these domains were generated using TetGen [67] and have approximately 6M and 9M nodes, respectively. The deformed boundaries of the Menger sponge are shown in Figs. 5(b) and (c), and the deformed boundary of the Luer connector domains is shown in Fig. 5(e). These boundaries are used in the boundary deformation step of the mesh warping process. For the Menger sponge mesh,

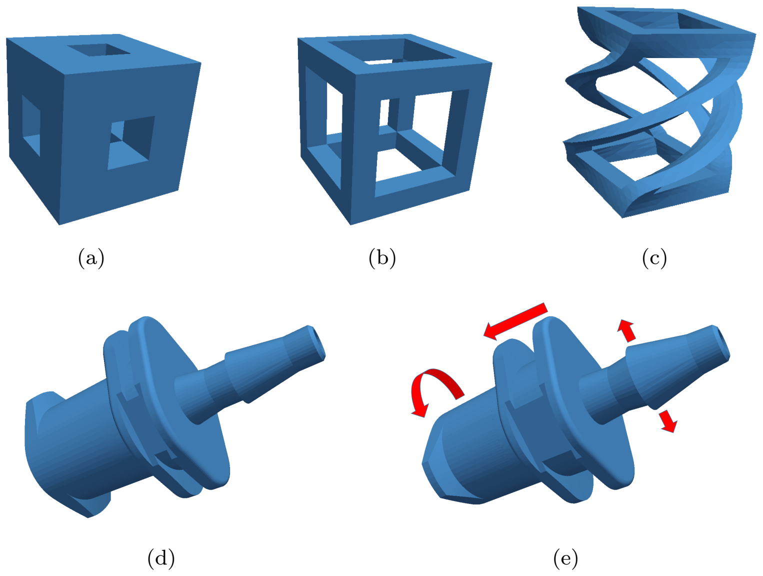


Fig. 5 (a) The Menger sponge domain and its two (b and c) deforming boundaries, and (d) the Luer connector domain and its (e) deforming boundary.

the first deformation (Fig. 5(b)) was generated by increasing the size of all square holes by 50%. Mesh elements were compressed in one dimension and stretched in the other two dimensions. The latter deformation is much more pronounced near the hole. The second deformation (Fig. 5(c)) was generated by applying the first deformation, and then counter-clockwise twisting the model by 90 degrees while increasing the height of the model by 30%. In this case, mesh elements were extremely compressed and twisted, as we can see in the figure. For the Luer connector mesh, the deformation was generated by increasing the size of the small tube on the top, extending the gap between the two middle plates, and rotating the lowest plate by 90 degrees. With these deformations, the mesh elements around the top are affected by two-dimensional expansion. The mesh elements around the middle of the model are stretched in one dimension. Finally, the mesh elements around the bottom are both compressed and distorted. Statistics for both meshes, such as the numbers of nodes and tetrahedral elements in the meshes and the mesh quality (as measured by the mean ratio (MR) mesh quality metric[2]) before and after the mesh deformation process are shown in Tables 1 and 2, respectively. Fig. 6

[2] The MR mesh quality metric η is given by

$$\eta = \frac{12(3v)^{2/3}}{\sum_{0\le i<j\le 3} l_{ij}^2}, \tag{5}$$

where v and l_{ij} denote the volume and various edge lengths of the tetrahedron, respectively [41].

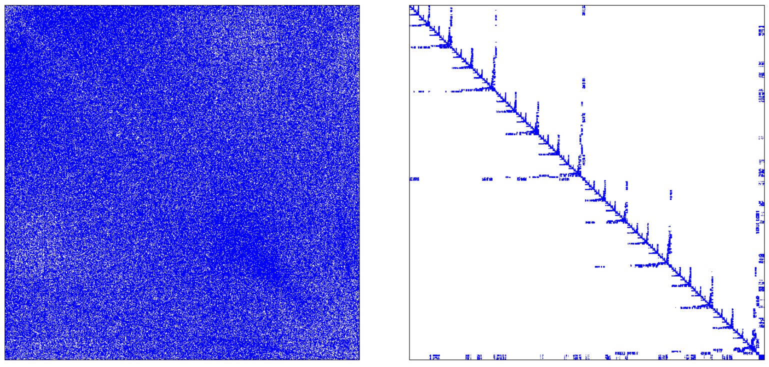
shows the spy plot of A_I for a coarse mesh of the initial Menger sponge model with (a) natural ordering and (b) nested dissection ordering, respectively. Note that the coarse mesh is used only for the purpose of visualizing the spy plot of A_I. As we can see from the figure, the mesh yields a very sparse matrix, A_I. A spy plot of A_I for the initial Luer connector model shows a similar result.

Table 1 The sizes of the Menger sponge and Luer connector meshes.

Mesh	# nodes	# elements
Menger Sponge	6,025,426	37,723,148
Luer Connector	10,523,992	59,291,516

Table 2 The mean ratio (MR) mesh quality of the Menger sponge and Luer connector meshes.

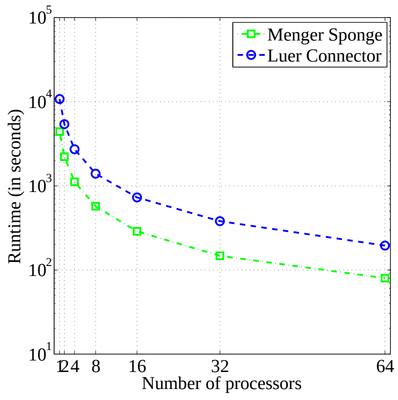
Mesh	Initial Mesh Quality (MR)			Final Mesh Quality (MR)		
	Min.	Avg.	Max.	Min	Avg.	Max.
Menger Sponge	1.0000	0.7842	0.2196	1.0000	0.3894	0.0177
(twisted)	-	-	-	1.0000	0.2059	0.0102
Luer Connector	1.0000	0.7180	0.1926	1.0000	0.2877	0.0159



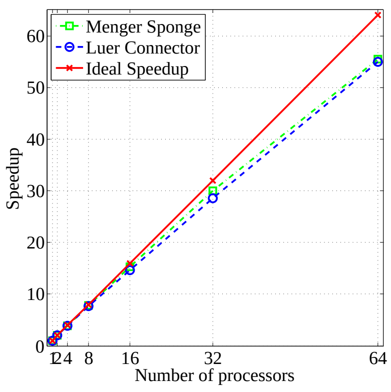
(a) Menger sponge: Natural ordering (b) Menger sponge: Nested Dissection ordering

Fig. 6 The spy plots of A_I for a coarse mesh on the initial Menger sponge model with (a) natural ordering and (b) nested dissection reordering, respectively.

Figs. 7(a) and (b) show the total runtime and speedup of the parallel LBWARP algorithm using DistLU for the linear solution step on the Menger sponge and Luer connector meshes running on different numbers of processors,



(a) Total runtime



(b) Speedup

Fig. 7 (a) The total runtime and (b) speedup of parallel LBWARP using DistLU on the Menger sponge and Luer connector meshes. (Note a log scale is used for the vertical axis.)

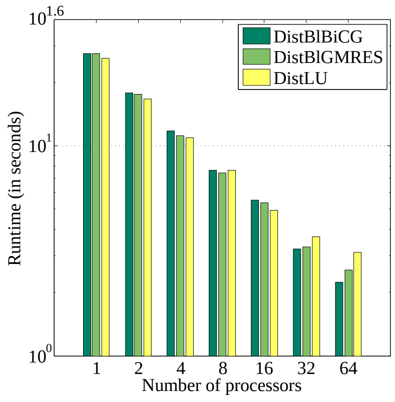
Table 3 Breakdown of the runtime (in seconds) for parallel LBWARP: neighbor computation, weight generation, boundary deformation, and linear solution steps using DistLU on the Menger sponge mesh.

# procs.	Neighbor Computation	Weight Generation	Boundary Deformation	Linear Solution
1	48.74	4311.76	7.43	26.08
2	32.95	2169.22	5.27	16.72
4	21.23	1097.54	3.55	10.98
8	16.12	539.98	2.74	7.68
16	11.59	274.56	1.77	4.95
32	7.56	142.92	1.24	3.69
64	3.78	76.59	0.86	3.12

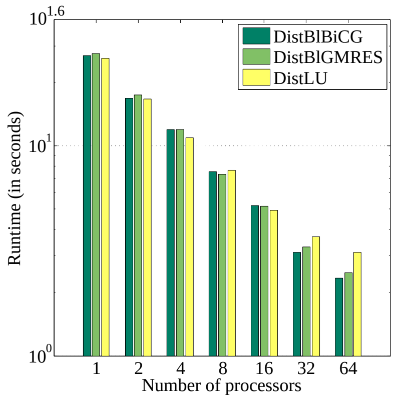
Table 4 Breakdown of the runtime (in seconds) for parallel LBWARP: neighbor computation, weight generation, boundary deformation, and linear solution steps using DistLU on the Luer connector mesh.

# procs.	Neighbor Computation	Weight Generation	Boundary Deformation	Linear Solution
1	102.41	10,593.24	13.98	35.93
2	65.20	5,365.32	11.02	23.29
4	44.52	2,679.88	7.51	16.06
8	32.74	1,342.59	5.95	11.32
16	20.21	704.73	4.32	8.63
32	14.82	342.67	3.64	6.64
64	6.18	186.99	2.37	5.75

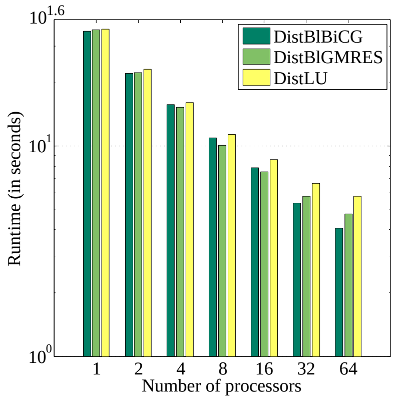
respectively. (Note a log scale is used for the vertical axis.) For the Menger



(a) Menger sponge



(b) Twisted Menger sponge



(c) Luer connector

Fig. 8 The runtime (in seconds) for the parallel sparse linear solvers for the (a) Menger sponge, (b) Luer Connector, and (c) twisted Menger sponge meshes. (Note a log scale is used for the vertical axis.)

Table 5 Weak scaling results for parallel LBWARP using DistBlBiCG on the Luer connector mesh.

# procs.	# nodes	# elements	Time (s.)
1	203,182	1,032,988	132.64
2	392,588	2,124,529	151.12
4	875,476	3,829,734	168.15
8	1,478,923	6,842,864	171.63
16	2,445,328	14,522,967	175.23
32	5,241,554	30,942,386	180.74
64	10,523,992	59,291,516	182.31

sponge mesh, although there are two different deformations, the total runtime for both deformations is the same when using DistLU in the linear solution step, as they generate and solve the same weight matrix. The experiments on the two meshes gave very similar results in terms of runtime and speedup. They both achieve speedup very close to the ideal speedup for a small number of processors. This slightly decreases as the number of processors increases. The runtime for the weight generation step dominates the overall time as shown in Tables 3 and 4 resulting in good strong scaling of the algorithm since this step is the most effective one to parallelize. We observe some slight performance deterioration due to the overhead in the pre-processing step, i.e., computation of the neighbor lists. However, the overall scalability of the algorithm is still good up to 64 processors. We have not extended our experiments to more than 64 processors due to limited processor accessibility on the Cyberstar cluster.

We compare the performance of the three linear solvers, i.e., the DistBlBiCG, DistBlGMRES, and DistLU solvers, as shown in Fig. 8. The figure shows the runtime of the linear solution step of the parallel LBWARP algorithm for the (a) Menger sponge, (b) twisted Menger sponge, and (c) Luer connector meshes. The DistLU solver gives good performance on a smaller number of processors and on smaller meshes, (i.e., with 6M nodes for the Menger sponge mesh), with up to 8 processors, as we can see from Fig. 8(a) and Fig. 8(b). However, for more than 8 processors, the iterative solvers are more scalable, and yield a lower runtime. With larger meshes, (i.e., with 9M nodes for the Luer connector mesh), both iterative solves perform better than the direct solver in all cases (see Fig. 8(c)). Moreover, the lower complexity of the DistBlBiCG solver results in the best scalability on the largest number of processors in the experiments.

The weak scaling efficiency, which is a measure of speedup based on an assumed fixed problem size per processor, of parallel LBWARP is shown in Tab. 5. The results are obtained from the parallel LBWARP algorithm with the DistBlBiCG solver on various sizes of Luer connector meshes. In the table, the timing results for the parallel LBWARP algorithm on up to 64 processors are given. The ideal ratio for successive speedups should be constant in this case. However, in our case, there are some factors that affect the results. For instance, the ratio of the number of nodes to the number of elements does not exactly double as the number of processors is doubled, as the mesh is unstructured. Such weak scaling results are typical for parallel unstructured mesh computations.

7 Multiple Mesh Deformations in a Heartbeat Simulation

One of the main advantages of parallel LBWARP is the re-usability of the weight matrix for additional deformations. Although the overall runtime for parallel LBWARP is dominated by the weight generation step, the weight matrix is generated only once and can be used for a series of deformations. This greatly reduces the computational complexity of the algorithm. Moreover, we can also apply linear solvers that support multiple RHS problems to further reduce the overall runtime. This can be done by computing the RHS vectors (from the boundary deformation step) for all deformations and solving the relevant linear systems simultaneously. Note this can only be done, however, in cases where all of the boundary deformations are known at once. In this section, we demonstrate the performance of parallel LBWARP on multiple mesh deformations with weight matrix re-utilization in a heartbeat simulation.

Heartbeat simulations have been developed in [6,33,34]. Such simulations (and their corresponding visualizations) may aid clinicians in medical diagonsis/treatment and may also be used for education. In addition, simulations may aid in obtaining a deeper understanding of a particular biological phenomenon of the heart and its ventricular systems. For example, beating heart meshes can be used to simulate the bioelectricity, biomechanics, and calcium dynamics of the human heart. For this application, we focus on applying multiple deformations to the initial heart mesh which are representative of actual heart motion. Note our heart motion simulation is symbolic and does not correspond to motion obtained from experimental data.

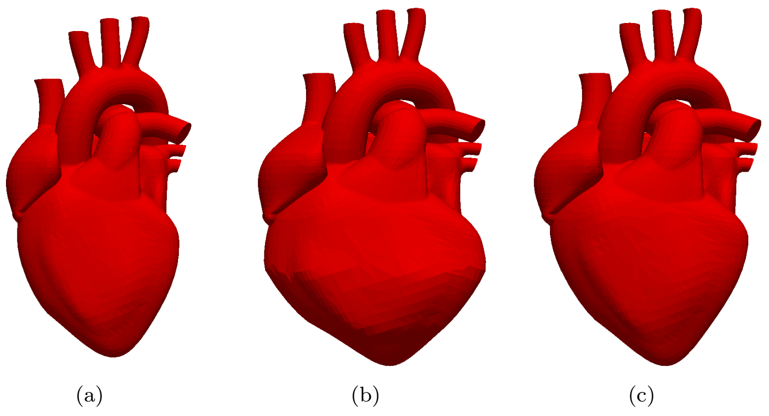
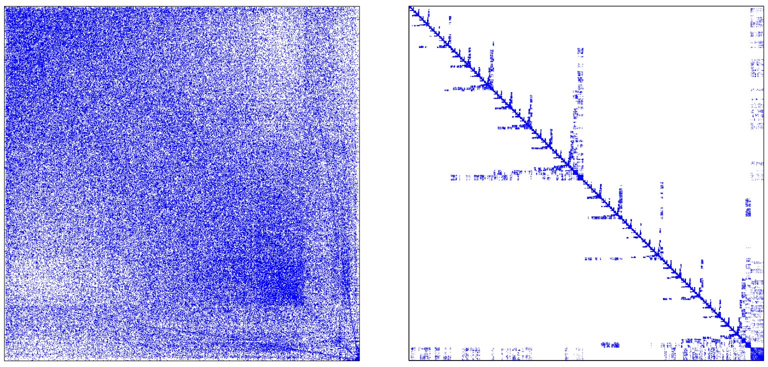


Fig. 9 Simulation of the heartbeat cycle. The initial motion of the heart is shown in (a), whereas (b) and (c) show sample deformations of the heart at two different timesteps within the cycle.

The initial heart domain was obtained from a model in GrabCAD [2], a community database of CAD models. The initial volume heart mesh has approximately 5M nodes and 30M tetrahedral elements and was generated using TetGen. The deformed boundaries are deformations of the surface meshes based on the initial volume heart mesh. After warping the initial mesh to the first deformed boundary, we consequently perform a series of deformations of the original mesh to other target boundaries. We demonstrate the use of multiple deformations with parallel LBWARP. We experiment by computing the deformations from the initial mesh (Fig. 9(a)) to five different deformations. Figs. 9(b) and (c) shows the sample motions of these five deformations. Spy



(a) Natural ordering (b) ND reordering

Fig. 10 Spy plots of A_I for the initial heart mesh with (a) natural ordering and (b) ND reordering.

plots of A_I for the initial heart mesh with (a) natural ordering and (b) nested dissection reordering are shown in Fig. 10. The mesh quality is shown in Table 6 as measured by the MR mesh quality metric. The average MR remains fairly constant throughout the mesh deformation process; only the maximum MR increases throughout the deformation process. This was also observed for the cardiology application in [64]. Note the noticeable decrease in mesh quality is from large deformations of the initial mesh to the target geometric domains. Smaller deformation steps could be taken (similar to those by the methods in [52,65]) if less change in the mesh quality is needed per deformation step. In our case, we are interested in how well the algorithm performs with large deformations.

Table 6 The mean ratio (MR) mesh quality of the heart meshes.

# Deformation	Mesh Quality (MR)		
	Min.	Avg.	Max.
0	1	0.6484	0.1745
1	1	0.2746	0.0248
2	1	0.2643	0.0220
3	1	0.3051	0.0207
4	1	0.2786	0.0202

Fig. 11 shows the total runtime of the parallel LBWARP algorithm for four deformations of the heart mesh using the DistLU algorithm as the linear solver. Reusing A_I can significantly reduce the total runtime of the algorithm. For k deformations, when A_I is reused, the algorithm is close is close to k-times faster than without reuse for sufficiently large k. The algorithm with and without multiple RHS support for the linear solution step does not show

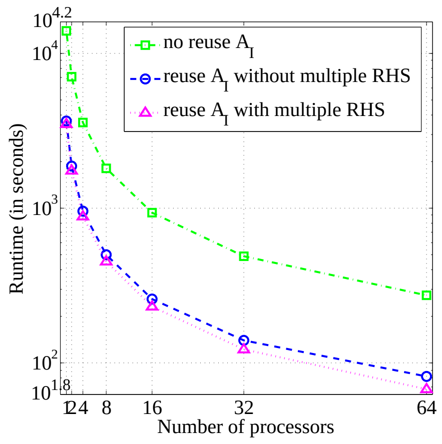
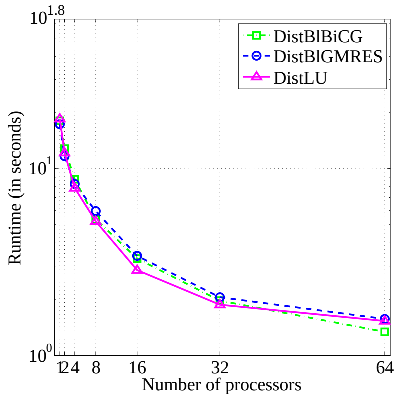
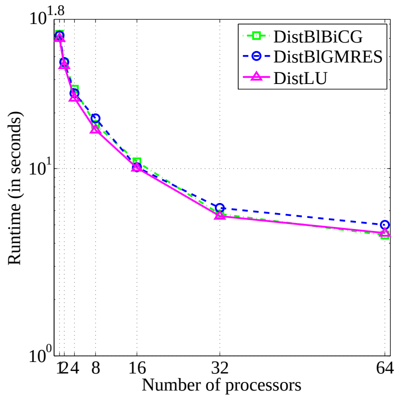


Fig. 11 The total runtime for four deformations of the heart mesh using DistLU as a solver.

much difference in runtime. (It is around 5% faster for four deformations on 64 processors. This is because the linear solution step takes less than three percent of the total time.) Although the advantage of employing multiple RHS support is much less than the advantage of reusing A_I, the combination of both approaches is rather advantageous when DistLU is used on large problems.



(a)



(b)

Fig. 12 The runtime for the linear solver step for (a) one deformation and (b) four deformations.

We also compare the performance of the three parallel sparse linear solvers, i.e., DistBlBiCG, DistBlGMRES and DistLU, on the linear solution step of parallel LBWARP on the heart meshes as shown in Fig. 12. Fig 12(a) gives a runtime comparison between the three algorithms for one deformation (i.e., by solving the linear systems with three RHS). The DistLU algorithm gives the best performance over both the DistBlBiCG and DistBlGMRES algorithms for lower numbers of processors, (i.e., up to 32 processors). However, the complexity of the algorithm does not scale well, and, thus, the DistBlBiCG and DistBlGMRES algorithms give better performance for larger numbers of processors, (i.e., more than 32 processors). Since DistBlBiCG has the lowest complexity and also scales well when increasing the numbers of processors, it shows the best performance on 64 processors. With 64 processors, the performance of DistBlBiBG is approximately 17% and 13% faster than that of DistBlGMRES and DistLU, respectively.

Despite the higher complexity, the direct solver has the advantage of reusing the L and U factors and reduces further the overall runtime compared with the parallel block iterative solvers when solving multiple RHS problems on small numbers of processors. Fig 12(b) shows the runtimes of the three parallel sparse linear solvers for four deformations (or by solving the linear system with 12 RHS vectors). The advantage of reusing the L and U factors for multiple linear solves on up to 32 processors can be seen in this figure. Once the L and U factors of the matrix have been computed, they can be used to (triangular) solve with additional RHS vectors in much less time. However, due to the high complexity of the sparse LU algortihm, it has the worst scalability. For large numbers of processors, we can see that the DistLU algorithm shows worse performance than the DistBlBiCG algorithm which has lower complexity and better strong scalability. With 64 processors, the performance of DistBlBiBG is approximately 13% and 2% faster than those of DistBlGMRES and DistLU, respectively.

8 Conclusions and Future Research

We have proposed a parallel formulation of the LBWARP algorithm in [64] for warping tetrahedral meshes on distributed memory machines. The algorithm generates the p distributed neighbor lists from the input mesh in which all interior nodes are numbered first and the boundary nodes are numbered last and sends each distributed neighbor list to a processor (assuming there are at least p processors available). Once the processors receive their neighbor lists, they perform the local weight generation for nodes in their neighbor list in parallel and without any communication. After that, the mesh boundary is deformed in parallel. Parallel LBWARP distributes the entries of the deformed boundary to the corresponding processors based on the rows of the weight matrix that the processors have generated. Finally, the linear system, which is based on the weight matrix and the boundary deformation, is solved for the final coordinates of the interior nodes in the deformed mesh using one of three parallel sparse linear solvers, i.e., the distributed block BiCG, distributed block GMRES, and distributed LU algorithms. These solvers support multiple right-hand side vectors which reduces the overall runtime of the parallel LBWARP algorithm since it otherwise requires solution of a sparse linear system with three right-hand side vectors for each deformation of a tetrahedral mesh.

Our experimental results show good strong scalability and speedup on several 3D tetrahedral meshes as a result of efficiently parallelizing the most time consuming step in the algorithm, i.e., the weight generation step. This step takes approximately 80-90% of the algorithm's overall runtime. However, we implement it in an embarrassingly parallel manner in order to avoid all communication during this step. Weak scaling results typical of those for unstructured meshes are also demonstrated. In regards to the performance of the linear solvers, the DistBlBiCG and DistBlGMRES algorithms generally perform well on a large number of processors, (i.e., $p > 32$). On the other hand, the DistLU algorithm performs better on a small number of processors, (i.e., $p \leq 32$) and large numbers of deformations due to the reuse of the L and U factors. Our experiments only use up to 64 processors due to the limited computing system. We expect the two iterative algorithms to outperform the DistLU algorithm when using more than 64 processors or when warping larger meshes (e.g., meshes with more than ten million nodes), as they have lower algorithm complexity and memory requirements.

We applied parallel LBWARP to a heartbeat simulation and demonstrated its performance on a sequence of mesh deformations. For multiple mesh deformations, once the weight matrix has been computed, the parallel LBWARP algorithm can reuse this matrix to determine the interior nodes for the other mesh deformations. That is, only the boundary deformation and linear solution steps are needed which further reduces the algorithm complexity and runtime. With the use of parallel sparse linear solvers that support multiple right-hand side vectors, the overall runtime can be reduced even further.

Possibilities for future research include extension of parallel LBWARP to other mesh element types such as hexahedral elements on 3D domains. Another possible avenue for research is the implementation of a parallel hybrid OpenMP/MPI LBWARP algorithm which can utilize both intra- and internode parallelism, as shared memory architectures are becoming increasingly more common. In regards to the parallel sparse linear solvers, our current implementations of the DistBlBiCG and DistBlGMRES algorithms apply rowwise partitioning and distribution of the weight matrix. It is possible to further improve their performance by applying a block matrix partition and distribution. Determination of ways to reuse a portion of the computations performed by these parallel iterative methods when multiple deformations are applied is also of interest.

Acknowledgements The work of the first author was funded by the Royal Thai Government scholarship. The work of the second author was supported in part by NSF grants CNS-0720749 and NSF CAREER Award ACI-1500487 (formerly ACI-1330054 and ACI-1054459). This work was also supported in part through instrumentation funded by the National Science Foundation through grant ACI0821527.

References

1. CyberSTAR: A scalable terascale advanced resource for discovery through computing. `http://www.ics.psu.edu/infrast/index.html`
2. GrabCAD. `https://grabcad.com`
3. Interoperable technologies for advanced petascale simulations (ITAPS) center. `http://www.itaps.org/`
4. Trilinos Project. `http://trilinos.org/`
5. Alan, G.: Nested dissection of a regular finite element mesh. SIAM Journal on Numerical Analysis **10**, 345–363 (1973)
6. Amano, A., Kanda, K., Shibayama, T., Kamei, Y., Matsuda, T.: Model generation interface for simulation of left ventricular motion. Electronics and Communications in Japan (Part II: Electronics) **90**(12), 87–98 (2007)
7. Antonopoulos, C., Ding, X., Chernikov, A., Blagojevic, F., Nikolopoulos, D., Chrisochoides, N.: Multigrain parallel Delaunay mesh generation. In: Proceedings of the 19th Annual International Conference on Supercomputing, pp. 367–376. ACM Press (2005)
8. Aycock, K., Campbell, R., Manning, K., Sastry, S., Shontz, S., Lynch, F., Craven, B.: A computational method for predicting inferior vena cava filter performance on a patient-specific basis. Journal of Biomechanical Engineering **136**, 081,003 (2014)
9. Bah, M., Nair, P., Browne, M.: Mesh morphing for finite element analysis of implant positioning in cementless total hip replacements. Medical Engineering and Physics **31**, 1235–1243 (2009)
10. Baker, T.: Mesh movement and metamorphosis. Engineering with Computers **18**(3), 188–198 (2002)
11. Baldwin, M., Langenderfer, J., Rullkoetter, P., Laz, P.: Development of subject-specific and statistical shape models of the knee using an efficient segmentation and mesh-morphing approach. Computer Methods and Programs in Biomedicine **97**, 232–240 (2010)
12. Barrett, R., Berry, M.W., Chan, T.F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., Van der Vorst, H.: Templates for the solution of linear systems: building blocks for iterative methods, vol. 43. SIAM (1994)
13. Bavier, E., Hoemmen, M., Rajamanickam, S., Thornquist, H.: Amesos2 and Belos: Direct and iterative solvers for large sparse linear systems. Scientific Programming **20**(3), 241–255 (2012)
14. Benítez, D., Rodríguez, E., Escobar, J.M., Montenegro, R.: Performance evaluation of a parallel algorithm for simultaneous untangling and smoothing of tetrahedral meshes. In: Proceedings of the 22nd International Meshing Roundtable, pp. 579–598. Springer (2014)
15. Bertsekas, D.P.: Projected Newton methods for optimization problems with simple constraints. SIAM Journal on Control and Optimization **20**(2), 221–246 (1982)
16. Botsch, M., Bommes, D., Kobbelt, L.: Efficient linear system solvers for mesh processing. In: Proceedings of the 11th IMA International Conference on the Mathematics of Surfaces, pp. 62–83 (2005)
17. Branets, L., Carey, G.: A local cell quality metric and variational grid smoothing algorithm. Engineering with Computers **21**, 19–28 (2005)
18. Cardoze, D., Cunha, A., Miller, G., Phillips, T., Walkington, N.: A Bézier-based approach to unstructured moving meshes. In: Proceedings of the 20th ACM Symposium on Computational Geometry, pp. 71–80 (2004)
19. Cardoze, D., Miller, G., Olah, M., Phillips, T.: A Bézier-based moving mesh framework for simulation with elastic membranes. In: Proceedings of the 13th International Meshing Roundtable, pp. 71–80. Sandia National Laboratories (2004)
20. Castanos, J., Savage, J.: Pared: A framework for the adaptive solution of PDEs. In: Proceedings of the 8th IEEE Symposium on High Performance Distributed Computing, pp. 133–140 (1999)
21. Center for Simulation of Advanced Rockets. `http://www.csar.illinois.edu/about/index.html`
22. Chernikov, A., Chrisochoides, N.: Parallel guaranteed quality Delaunay uniform mesh refinement. SIAM Journal on Scientific Computing **28**, 1907–1926 (2006)

23. Chrisochoides, N.: A survey of parallel mesh generation methods. Tech. Rep. SC-2005-09, Brown University (2005)
24. Chrisochoides, N., Chernikov, A., Fedorov, A., Kot, A., Linardakis, L., Foteinos, P.: Towards exascale parallel Delaunay mesh generation. In: Proceedings of the 18^{th} International Meshing Roundtable, pp. 319–336 (2009)
25. De Cougny, H., Shephard, M.: Parallel refinement and coarsening of tetrahedral meshes. International Journal for Numerical Methods in Engineering **46**(7), 1101–1125 (1999)
26. Estruch, O., Lehmkuhl, O., Borrell, R., Segarra, C.P., Oliva, A.: A parallel radial basis function interpolation method for unstructured dynamic meshes. Computers and Fluids **80**, 44–54 (2013). Selected contributions of the 23^{rd} International Conference on Parallel Fluid Dynamics ParCFD2011
27. Fletcher, R.: Conjugate gradient methods for indefinite systems. In: Numerical analysis, pp. 73–89. Springer (1976)
28. Galtier, J., George, P.: 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 (1997)
29. Gerhold, T., Neumann, J.: The parallel mesh deformation of the DLR TAU-code. In: New results in numerical and experimental fluid mechanics VI, notes on numerical fluid mechanics and multidisciplinary design, vol. 96, pp. 162–169. Springer (2008)
30. Gorman, G.J., Rokos, G., Southern, J., Kelly, P.H.: Thread-parallel anisotropic mesh adaptation. In: New Challenges in Grid Generation and Adaptivity for Scientific Computing, pp. 113–137. Springer (2015)
31. Helenbrook, B.T.: Mesh deformation using the biharmonic operator. International Journal for Numerical Methods in Engineering **56**, 1007–1021 (2003)
32. Hestenes, M.R., Stiefel, E.: Methods of conjugate gradients for solving linear systems. Journal of Research of the National Bureau of Standards Vol **49**(6) (1952)
33. Hunter, P.J., Pullan, A.J., Smaill, B.H.: Modeling total heart function. Annual review of biomedical engineering **5**(1), 147–177 (2003)
34. Ijiri, T., Ashihara, T., Umetani, N., Igarashi, T., Haraguchi, R., Yokota, H., Nakazawa, K.: A kinematic approach for efficient and robust simulation of the cardiac beating motion. PLOS ONE **7**(5), e36,706 (2012)
35. Jacob, B., Guennebaud, G.: Eigen. `http://eigen.tuxfamily.org`
36. Karypis, G., Kumar, V.: A fast and highly quality multilevel scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing **20**, 359–392 (1999)
37. Knupp, P.: Updating meshes on deforming domains: An application of the target-matrix paradigm. Communications in Numerical Methods for Engineering **24**, 467–476 (2007)
38. Lachat, C., Dobrzynski, C., Pellegrini, F.: Parallel mesh adaptation using parallel graph partitioning. In: 5^{th} European Conference on Computational Mechanics, vol. 3, pp. 2612–2623. CIMNE-International Center for Numerical Methods in Engineering (2014)
39. Li, R., Tang, T., Zhang, P.: Moving mesh methods in multiple dimensions based on harmonic maps. Journal of Computational Physics **170**, 562–688 (2001)
40. Li, X.S., Demmel, J.W.: SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. ACM Transactions on Mathematical Software **29**(2), 110–140 (2003)
41. Liu, A., Joe, B.: Relationship between tetrahedron shape measures. BIT Numerical Mathematics **34**(2), 268–287 (1994)
42. Liu, Y., D'Arceuil, H., He, J., Duggan, M., Gonzalez, G., Pryor, J., de Crespigny, A.: A nonlinear mesh-warping technique for correcting brain deformation after stroke. Magnetic Resonance Imaging **24**, 1069–1075 (2006)
43. Löhner, R.: A 2^{nd} generation parallel advancing front grid generator. In: Proceedings of the 21^{st} International Meshing Roundtable, pp. 457–474 (2013)
44. Löhner, R., Camberos, J., Marsha, M.: Unstructured scientific computation on scalable multiprocessors. In: P. Hehrotra, J. Saltz (eds.) Parallel Unstructured Grid Generation, pp. 31–64. MIT Press (1990)
45. Löhner, R., Cebral, J.: Parallel advancing front grid generation. In: Proceedings of the 8^{th} International Meshing Roundtable, pp. 67–74 (1999)
46. Lu, Q., Shephard, M.S., Tendulkar, S., Beall, M.W.: Parallel mesh adaptation for high-order finite element methods with curved element geometry. Engineering with Computers **30**(2), 271–286 (2014)

47. Luke, E., Collins, E., Blades, E.: A fast mesh deformation method using explicit interpolation. Journal of Computational Physics **231**, 586–601 (2012)
48. Nave, D., Chrisochoides, N., Chew, L.: Guaranteed-quality parallel Delaunay refinement for restricted polyhedral domains. In: Computational Geometry: Theory and applications, vol. 28, pp. 191–215 (2004)
49. O'Leary, D.P.: The block conjugate gradient algorithm and related methods. Linear algebra and its applications **29**, 293–322 (1980)
50. Oliker, L., Biswas, R., Gabow, H.: Parallel tetrahedral mesh adaptation with dynamic load balancing. Parallel Computing Journal pp. 1583–1608 (2000)
51. Park, J., Shontz, S., Drapaca, C.: Automatic boundary evolution tracking via a combined level set method and mesh warping technique: Application to hydrocephalus. In: Proc. of the Mesh Processing in Medical Image Analysis 2012 - MICCAI 2012 International Workshop, MeshMed 2012, pp. 122–133 (2012)
52. Park, J., Shontz, S., Drapaca, C.: A combined level set/mesh warping algorithm for tracking brain and cerebrospinal fluid evolution in hydrocephalic patients. Image-Based Geometric Modeling and Mesh Generation, Lecture Notes in Computational Vision and Biomechanics **3**, 107–141 (2013)
53. Peskin, C., McQueen, D.: Heart throb. `http://www.psc.edu/science/Peskin/Peskin.html`
54. Rivara, M., Carlderon, C., Pizaro, D., Fedorov, A., Chrisochoides, N.: Parallel decoupled terminal-edge bisection algorithm for 3D meshes. Engineering with Computers (2005)
55. Rivara, M., Pizarro, D., Chrisochoides, N.: Parallel refinement of tetrahedral edges using terminal-edge bisection algorithm. In: Proceedings of the 13^{th} International Meshing Roundtable (2004)
56. Saad, Y., Schultz, M.H.: GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM Journal on Scientific and Statistical Computing **7**(3), 856–869 (1986)
57. Sastry, S., Kim, J., Shontz, S., Craven, B., Lynch, F., Manning, K., Panitanarak, T.: Patient-specific model generation and simulation for pre-operative surgical guidance for pulmonary embolism treatment. Image-Based Geometric Modeling and Mesh Generation, Lecture Notes in Computational Vision and Biomechanics **3**, 223–249 (2013)
58. Sastry, S.P., Shontz, S.M.: A parallel log-barrier method for mesh quality improvement and untangling. Engineering with Computers **30**(4), 503–515 (2014)
59. Selwood, P., Berzins, M., Dew, P.: 3D parallel mesh adaptivity: Data-structures and algorithms. In: Proceedings of the 8^{th} SIAM Conference on Parallel Processing for Scientific Computing. SIAM (1997)
60. Selwood, P., Verhoeven, N., Nash, J., Berzins, M., Weatherill, N., Dew, P., Morgan, K.: Parallel mesh generation and adaptivity: Partitioning and analysis. In: Proceedings of 1996 Parallel CFD Conference (1996)
61. Seol, E., Shephard, M.: Efficient distributed mesh data structure for parallel automated adaptive analysis. Engineering with Computers **22**, 197–213 (2006)
62. Shephard, M., Flaherty, J., Bottasso, C., de Cougny, H., Özturan, C., Simone, M.: Parallel automated adaptive analysis. Parallel Computing **23**, 1327–1347 (1997)
63. Shontz, S.: Numerical methods for problems with moving meshes. Ph.D. thesis, Cornell University (January 2005)
64. Shontz, S., Vavasis, S.: A mesh warping algorithm based on weighted Laplacian smoothing. In: Proceedings of the 12^{th} International Meshing Roundtable, pp. 147–158 (2003)
65. Shontz, S., Vavasis, S.: Analysis of and workarounds for element reversal for a finite element-based algorithm for warping triangular and tetrahedral meshes. BIT, Numerical Mathematics **50**, 863–884 (2010)
66. Shontz, S., Vavasis, S.: A robust solution procedure for hyperelastic solids with large boundary deformation. Engineering with Computers **28**, 135–147 (2012)
67. Si, H.: Tetgen: A quality tetrahedral mesh generator and three-dimensional Delaunay triangulator. `http://tetgen.berlios.de/`
68. Sigal, I., Hardisty, M., Whyne, C.: Mesh-morphing algorithms for specimen-specific finite element modeling. Journal of Biomechanics **41**, 1381–1389 (2008)
69. Sigal, I., Whyne, C.: Mesh morphing and response surface analysis: Quantifying sensitivity of vertebral mechanical behavior. Annals of Biomedical Engineering **38**, 41–56 (2010)

70. Sigal, I., Yang, H., Roberts, M., Downs, J.: Morphing methods to parameterize specimen-specific finite element model geometries. Journal of Biomechanics **43**, 254–262 (2010)
71. Simoncini, V.: A stabilized QMR version of block BICG. SIAM Journal on Matrix Analysis and Applications **18**(2), 419–434 (1997)
72. Simoncini, V., Gallopoulos, E.: Convergence properties of block GMRES and matrix polynomials. Linear Algebra and its Applications **247**, 97–119 (1996)
73. Stein, K., Tezduyar, T., Benney, R.: Mesh moving techniques for fluid-structure interactions with large displacements. Transactions of the ASME 2003 **70**, 58–63 (2003)
74. Stein, K., Tezduyar, T., Benney, R.: Automatic mesh update with the solid-extension mesh moving technique. Computational Methods in Applied Mechanical Engineering **193**, 2019–2032 (2004)
75. Tsai, H., Wong, A., Cai, J., Zhu, Y., Liu, F.: Unsteady flow calculations with a parallel multiblock moving mesh algorithm. AIAA Journal **39**, 1021–1029 (2001)
76. T.Tezduyar, Behr, M., Mittal, S., Johnson, A.: Computation of unsteady incompressible flows with the finite element methods – Space-time formulations, iterative strategies and massively parallel implementations. In: New Methods in Transient Analysis, vol. PVP-vol.246/AMD-vol.143, pp. 7–24. ASME (1992)
77. Williams, R.: Adaptive parallel meshes with complex geometry. In: Numerical Grid Generation in Computational Fluid Dynamics and Related Fields (1991)