# Parallelization of the Finite Element-based Mesh Warping Algorithm Using Hybrid Parallel Programming

Abir Haque * Suzanne M. Shontz †

**Abstract**

Warping large volume meshes has applications in biomechanics, aerodynamics, image processing, and cardiology. However, warping large, real-world meshes is computationally expensive. Existing parallel implementations of mesh warping algorithms do not take advantage of shared-memory and one-sided communication features available in the MPI-3 standard. We describe our parallelization of the finite element-based mesh warping algorithm for tetrahedral meshes. Our implementation is portable across shared and distributed memory architectures, as it takes advantage of shared memory and one-sided communication to precompute neighbor lists in parallel. We then deform a mesh by solving a Poisson boundary value problem and the resulting linear system, which has multiple right-hand sides, in parallel. Our results demonstrate excellent efficiency and strong scalability on up to 32 cores on a single node. Furthermore, we show a 33.9% increase in speedup with 256 cores distributed uniformly across 64 nodes versus our largest single node speedup while observing sublinear speedups overall.

## 1 Introduction

Mesh warping is important for modeling dynamic problems, such as those found in biomechanics, aerodynamics, image processing, and cardiology applications. Specific use cases include tracking brain movements during disease progression and treatment [19], surface deflections of an aircraft's wing [10], image warping [9], stitching multiple images into one [27], tracking heart rhythms [18], and deforming reference organs to create patient-specific models [14]. Although warping large meshes which stem from real-world applications is computationally expensive, it can benefit from some form of parallelization, whether that be multiple cores, nodes, GPUs, etc. Hence, we propose a parallelization of the FE-based mesh warping algorithm (FEMWARP) proposed by Shontz and Vavasis in [23, 24]. The method deforms a volume mesh by first solving a Poisson boundary value problem (BVP) using a finite element method (FEM) and then solving a linear system with multiple right-hand sides based on the initial mesh and boundary deformation.

Several parallel algorithms related to the FEM and mesh warping show strong scaling on various architectures. Sastry *et al.* developed a shared-memory, OpenMP parallelization of FEMWARP to warp tetrahedral meshes [21]. A distributed-memory, MPI-based parallelization of an inverse distance (IDWARP) method to deform structured hexahedral meshes was developed by Secco *et al.* in [22]. Additionally, a distributed memory, MPI parallelization of an unstructured grid deformation method via domain decomposition was developed by Gerhold and Neumann [6]. Panitanarak and Shontz developed a distributed-memory, MPI parallelization of a log-barrier-based warping algorithm to warp tetrahedral meshes, but utilized one core per node to maximize available memory per rank to achieve strong scalability [18]. A hybrid, MPI/OpenMP parallelization for a radial basis function-based mesh deformation algorithm was developed by Zhao *et al.* and obtained strong scalability [29]. Krysl utilized a shared-memory, pthreads implementation to construct sparse FEM matrices [12]. We propose a novel, hybrid, MPI-only parallelization of FEMWARP (ParFEMWARP)[1] via shared-memory (SHM) for shared-memory intra-node communication and remote memory access (RMA) for one-sided, inter-node communication [2].

In this paper, we describe ParFEMWARP for tetrahedral meshes. Section 2 describes the FEMWARP algorithm. Details regarding hybrid parallel programming in the MPI-3 standard are provided in Section 3. Section 4 describes how we take advantage of shared memory and one-sided communication features available in MPI-3 to precompute neighbor lists in parallel, which is necessary for executing various stages of FEMWARP. The preconditioned block conjugate gradient method, which we use to solve the multiple right-hand sides problem, is outlined in Section 5. A description of how we parallelize the various stages of FEMWARP is given in Section 6. We provide a parallel runtime analysis for our method in Section 7. The results of our numerical experiments are given in Section 8. Conclusions and

*University of Kansas, Lawrence, Kansas. abirhaque@ku.edu
†University of Kansas, Lawrence, Kansas. shontz@ku.edu

[1] https://github.com/AbirHaque/ParFEMWARP

directions for future work are described in Section 9.

## 2 FEMWARP

FEMWARP [23, 24, 19, 21, 26] is a finite element-based mesh warping algorithm. FEMWARP follows a 3-step process to compute the warped mesh.

FEMWARP first quantifies the representation of the initial mesh. To do so, it uses the standard Galerkin FEM to solve the following Poisson BVP on the domain $\Omega$

$$\Delta u = 0 \text{ on } \Omega \tag{2.1}$$

with $u = u_0$ on $\partial\Omega$. Any $u_0$ can be chosen according to [23, 24]. Discretization yields the global stiffness matrix with the following entries

$$A(i,j) = \int_\Omega \nabla\phi_i \cdot \nabla\phi_j \, d\Omega, \tag{2.2}$$

where $\phi_i$ is a piecewise-linear shape function satisfying $\phi_i(v_i) = 1$ and $\phi_i(v_j) = 0$, $\forall j \neq i$. The resulting global stiffness matrix is of the form

$$A = \begin{pmatrix} A_I & A_B \\ A_B^T & X \end{pmatrix}, \tag{2.3}$$

where vertices map to row and column indices. Interior and boundary vertices are indexed as $[1 : I]$ and $[I+1 : I+B]$, respectively. $A_I$ and $A_B$ store each interior vertex's weights between neighboring interior and boundary vertices, respectively. Note that computing $X$ is not necessary when executing FEMWARP, as it is not used. The following multiple right-hand sides problem

$$A_I[x_I, y_I, z_I] = -A_B[x_B, y_B, z_B] \tag{2.4}$$

then results from Equation 2.1, where $[x_I, y_I, z_I]$ and $[x_B, y_B, z_B]$ represent the coordinate lists of the interior and boundary vertices, respectively. Next, we apply a user-supplied boundary deformation:

$$[x_B, y_B, z_B] \rightarrow [\hat{x}_B, \hat{y}_B, \hat{z}_B]. \tag{2.5}$$

Note that the boundary deformation can come from experimental data or from a PDE that prescribes the motion.

Finally, we solve the updated linear system to determine final interior vertex positions:

$$A_I[\hat{x}_I, \hat{y}_I, \hat{z}_I] = -A_B[\hat{x}_B, \hat{y}_B, \hat{z}_B]. \tag{2.6}$$

The connectivity of the vertices in the original mesh is preserved throughout the warping process. Additionally, the construction of $A$ only needs to happen once, and the mesh can be updated multiple times based on each user-supplied boundary deformation.

Computing FEMWARP for large meshes calls for several optimizations to FEMWARP, because operating on a large $A_I$ matrix in serial is infeasible. Such optimizations include storing $A_I$ in a sparse format, precalculating the sparsity of $A_I$, and developing an iterative solver for sparse linear systems containing multiple right-hand sides. It is also necessary to parallelize the previously stated operations across multiple cores on many distributed machines via MPI to further reduce the total runtime. Additionally, we wish to use hybrid parallel programming features available in the MPI-3 standard to reduce intra- and inter-node communication in the sparsity precomputation. Before introducing our novel parallel mesh warping method, we provide a brief overview of advanced features offered by MPI-3 that we use in ParFEMWARP.

## 3 Hybrid Parallel Programming with MPI-3

In modern parallel scientific computing, MPI and OpenMP are commonly paired for inter-node and intra-node communication, respectively, when parallelizing software across a cluster. Such a pairing is an example of hybrid parallel programming, where parallelism at both the node and core level is achieved. The MPI-3 standard offers a more uniform alternative to hybrid parallelism by incorporating SHM as part of the MPI RMA interface, as depicted by Figure 1. Additionally, Hoefler *et al.* show that SHM has significantly less overhead than blocking MPI solutions [8]. Furthermore, Li *et al.* redesigned the Graph500 benchmark to make extensive use of MPI-3 features, such as shared memory and one-sided communication. This benchmark is relevant to our neighbor precomputation strategy, as the benchmark's first kernel involves generating a large graph and compressing edge lists into a sparse format. Li *et al.* compared their implementation with non-blocking MPI, MPI+OpenSHMEM, and MPI+OpenMP implementations and found that their MPI-3 implementation achieved better performance in terms of runtime [15]. We briefly describe the RMA interface and capabilities offered by SHM.

The RMA interface offers a one-sided communication model, which means that any process may move data between itself and a remote process without requiring that process to synchronize. A rank may expose a region of memory for other ranks to interact with via windows. These windows can be created via **MPI_Win_create**. Other ranks may "put" and "get" memory from this exposed memory via **MPI_Put** and **MPI_Get** operations. RMA offers several options for achieving synchronization. We specifically use **MPI_Win_flush_local_all** and **MPI_Win_flush_all** to complete all RMA operations across ranks.

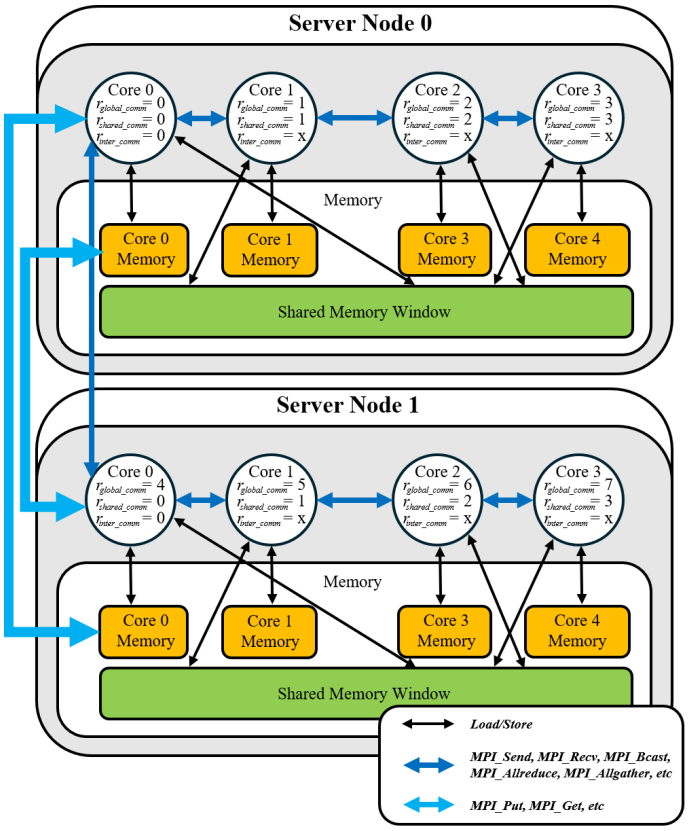



Figure 1: MPI-3 Features Visualized.

While RMA is similar in purpose to non-blocking MPI, the RMA interface offers shared memory windows as of MPI-3. Before creating such windows, the programmer had to split their MPI communicator into groups based on which ranks can share memory with the **MPI_COMM_TYPE_SHARED** split type passed into **MPI_Comm_split_type**. Once a shared memory communicator is established via **MPI_Comm_group**, shared memory windows that are only accessible within a node may be created via **MPI_Win_allocate_shared**. Any data exchanging across shared memory and between ranks is now possible via simple load/store operations.

## 4 Precomputing Global Stiffness Matrix Sparsity

The matrix $A_I$ is stored in the Compressed Sparse Row (CSR) format, because $A_I$ is a large, sparse matrix. It is also symmetric positive definite (SPD). Inserting new non-zero elements into a CSR matrix, thus modifying its sparsity pattern, is well-known to be extremely expensive and highly discouraged [4]. Considering there can be several billion non-zero values within FEM global stiffness matrices for large tetrahedral meshes, it is necessary to precalculate the sparsity pattern for $A_I$. As a result, element-wise updates occur in place of element-wise insertions during global stiffness matrix generation. To calculate $A_I$'s sparsity, we developed a parallel method of precomputing neighbor lists (see Fig. 2) that is portable to shared and distributed memory architectures. Figs. 3 - 8 show an example of how our method would generate neighbor lists for a triangle mesh with 4 cores equally distributed across 2 nodes.

When ParFEMWARP is initialized, only one rank contains a full copy of the tetrahedron list $T$ containing $t$ tetrahedral elements, which is used to generate our neighbor list. Only one rank contains $T$, as $T$ must be extremely large to store duplicates. Instead, we first partition the tetrahedron list $T$ into $p$ subsets $T_1, T_2, ..., T_p$ based on the desired depth of data locality, with each subset $T_i$ being a unique list of tetrahedra $t_i, t_{i+1}, ..., t_j$, where $j - i$ is approximately $\lceil t/p \rceil$. For our implementation, we use **MPI_Win_create** on the buffer representing $T$. Additionally, we want all ranks with shared memory to store their neighbor lists in a single buffer. To achieve this, all ranks with $r_{shared\_comm} = 0$ use **MPI_Win_allocate_shared** to create a shared memory buffer $neighbor\_list_{shared}$,

while other ranks use **MPI_Win_shared_query** to allow each rank to know the address of the shared buffer. Each rank then extracts their subset $T_i$ from $T$ by using **MPI_Get_accumulate** (see Fig. 3).

Then, we construct neighbor lists with distinct integers in ascending order at the lowest level of the hierarchy. The implementation of this is straightforward and does not call for any MPI programming. Each rank builds their list of lists in an exclusive portion of $neighbor\_list_{shared}$ to avoid race conditions (see Fig. 4).

Lastly, we merge neighbor lists up the hierarchy and across the shared memory (see Fig. 5). If multiple nodes are utilized, we merge again across nodes via RMA operations to obtain a global view of the complete neighbor lists (see Figs. 6 - 8). Once the sparsity pattern is determined, we initialize the matrix as zero.

## 5 Preconditioned Block Conjugate Gradient

The preconditioned conjugate gradient (PCG) method is an iterative method for solving large, sparse SPD linear systems [20]. However, suppose one were to solve the multiple right-hand sides problem within FEMWARP with PCG. Then, one would need to solve the linear system with each right-hand side separately (i.e., $A_I\hat{x}_I = -A_B\hat{x}_B$, $A_I\hat{y}_I = -A_B\hat{y}_B$, and $A_I\hat{z}_I = -A_B\hat{z}_B$). Whereas, the preconditioned block conjugate gradient (PBCG) method [17, 3] can be used to solve the multiple right-hand sides problem $A_I[\hat{x}_I, \hat{y}_I, \hat{z}_I] = -A_B[\hat{x}_B, \hat{y}_B, \hat{z}_B]$ directly. The pseudocode for the method is given below in Algorithm 5.1. More information regarding PBCG can be found in [17].

Algorithm 5.1. (PBCG) Precondtioned Block Conjugate Gradient

**Input:** $A$ CSR matrix, $B$ multiple right-hand sides matrix, $\lambda_{min}$ minimum residual
**Output:** CSR matrix $X$ such that $AX = B$
**function** PBCG($A, B, \lambda_{min}$)
  $R = B$
  $P = R$
  $R_{old} = R^T R$
  $R_{new} = R_{old}$
  **while trace**($R_{new}$) $< \lambda_{min}$ **do**
    $K = (P^{-1}AP)^{-1}R_{old}$
    $X = X + PK$
    $R = R - APK$
    $R_{new} = R^T R$
    $G = R_{old}^{-1}R_{new}$
    $R_{new} = R_{old}$
    $P = R + PG$
  **end while**
  **return** X
**end function**

## 6 ParFEMWARP

Our implementation involves the neighbor precomputation step described in Section 4 followed by the 3-step algorithm shown in Section 2. Note that we do not use a mesh partitioner. Computing stiffness matrix contributions based on rank-local tetrahedra is embarrassingly parallel, as Figure 9 depicts. Each rank first computes a local, partial version of $A_I$ and $A_B$ (see Equations 2.2-2.3). Note that entries corresponding to edges not owned by a particular rank will remain as zero during this stage. Next, we sum the local copies of $A_I$ across ranks via **MPI_Allreduce** to account for shared edges across partitions. Similarly, we sum local copies of $A_B$. Next, we apply a user-supplied boundary deformation (see Equation 2.5). For example, this may come from a PDE describing the motion of the deforming domain or experimental data. Performing this boundary update is trivial. Finally, we solve the updated multiple right-hand sides problem (see Equation 2.6) via our parallelization of the PBCG method (see Section 5) with the Jacobi preconditioner. Sparse-dense matrix-matrix multiplications arise when multiplying sparse $A_I$ with dense matrices and are $i \times 3$ (or similarly $b \times 3$ for $A_B$) in PBCG, where $i$ and $b$ are the number of interior and boundary vertices, respectively. Sparse-dense matrix-matrix multiplications are parallelized, where each rank out of $p$ total ranks owns approximately $\lceil n/p \rceil$ rows, i.e., each rank owns approximately the same number of rows via a uniform partition. Dense matrix-matrix multiplications, which arise during inner product computations between two matrices, utilize similar partitioning with block operations.

Algorithm 6.1. (ParFEMWARP) Parallel FEMWARP

**Input:** $[x, y, z]$ node list, $T$ tetrahedron list, **usr_def()** boundary deformation function
**Output:** $[\hat{x}_I, \hat{y}_I, \hat{z}_I]$ updated interior node list
**function** ParFEMWARP($[x, y, z], T,$ **usr_def()**)
  // See Section 4
  sparsity = **parallel_precompute**(T)
  $(A_I, A_B)$.**gen_CSR**(sparsity)
  // See Equations 2.1-2.3
  $A_I, A_B =$ **gen_stiffness_matrix_parallel(T,V)**
  // See Equation 2.5
  $[\hat{x}_B, \hat{y}_B, \hat{z}_B] =$ **usr_def**($[x_B, y_B, z_B]$)
  // See Equation 2.6
  $[\hat{x}_I, \hat{y}_I, \hat{z}_I] =$ **PBCG**($A_I, -A_B[\hat{x}_B, \hat{y}_B, \hat{z}_B]$)
  **return** $[\hat{x}_I, \hat{y}_I, \hat{z}_I]$
**end function**

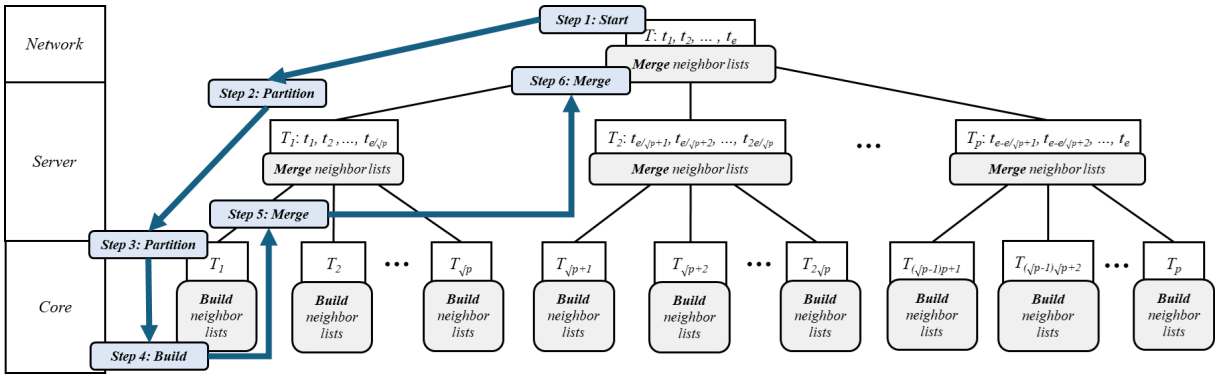



Figure 2: Merging neighbor lists across a cluster. For simplicity, this figure assumes there are $p$ cores within the cluster, and that there are $\sqrt{p}$ servers each containing $\sqrt{p}$ cores per server. Step 1: Input a list of tetrahedra. Steps 2-3: Partition the list of tetrahedra for each server and/or core. Step 4: Build neighbor lists within each core. Step 5: Merge neighbor lists between cores in each node. Step 6: Merge neighbor lists between nodes.

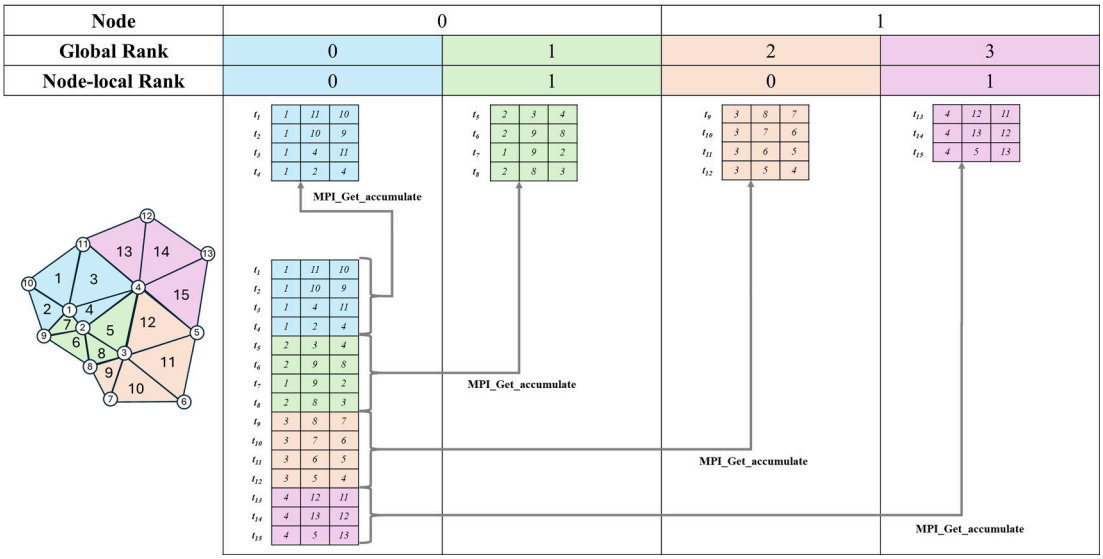



Figure 3: Partition $T$ into $p$ subsets $T_1, T_2, ..., T_p$, where $p = 4$, via RMA (4 cores equally distributed across 2 nodes). Note that HPC servers are denoted as "nodes" and that this example uses triangles instead of tetrahedra for simplicity. Also note that this example only builds neighbor lists for interior vertices ($vertex_1$, ..., $vertex_4$).

## 7 Parallel Runtime Analysis

In this section, we analyze ParFEMWARP to provide upper bounds on runtime for various stages of our parallelization. Let $t$ denote the number of tetrahedral elements, $v$ the number of vertices, $d$ the maximum degree of any given vertex, $n$ the number of nodes, $p_{global}$ the number of processes uniformly distributed across $n$ nodes, $p_{nodal}$ the number of processes within a node, i.e., $p_{nodal} = p_{global}/n$, and $C_x$ the time taken to for event $x$ to happen. Assume it takes $t_t$, $t_v$, $t_d$, and $t_n$ time to process $t$, $v$, $d$, and $n$ items, respectively.

**7.1 Precomputation** Assume all loads/stores across shared memory require negligible time. The first stage of neighbor list precomputation, which is at the core-level, involves iterating through approximately $\lceil t/p_{global} \rceil$ separate tetrahedra, where each tetrahedron is visited only once. This means that each core iterates through $\lceil t/p_{global} \rceil$ tetrahedra. Since all 4 vertices of the tetrahedron are neighbors of one another, we must insert them into rank-local neighbor lists. Assume each rank receives $\lceil t/p_{global} \rceil$ randomly assigned tetrahedra, i.e., rank-local tetrahedra may be in several small patches, rather than one large patch. This is the worst case scenario, and we will only consider this case for

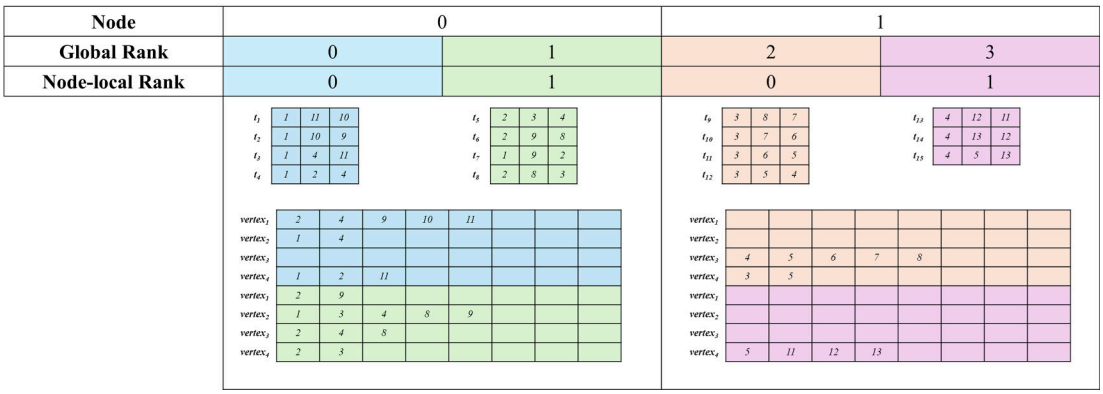



Figure 4: Generate rank-local neighbor lists based on local $T_p$ (4 cores equally distributed across 2 nodes)

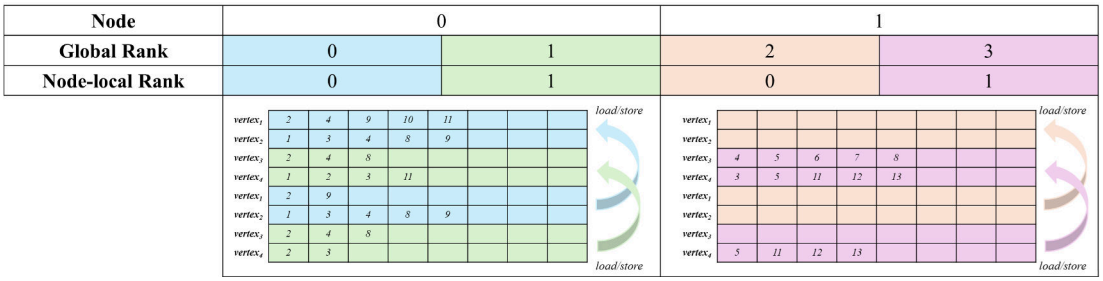



Figure 5: Merge neighbor lists across shared memory (4 cores equally distributed across 2 nodes)

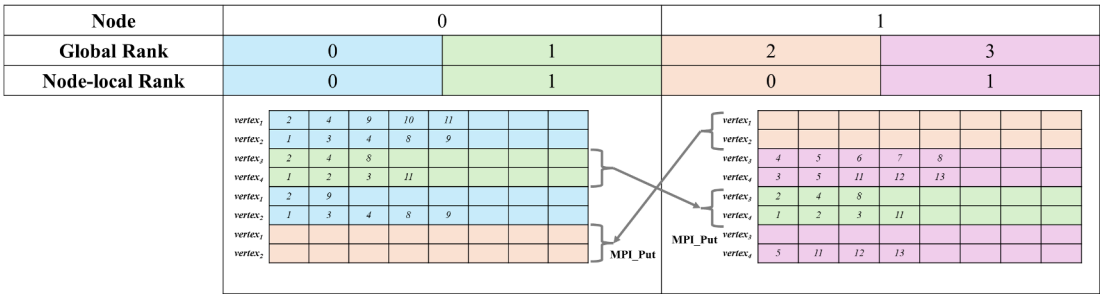



Figure 6: Exchange neighbor lists between nodes to merge via RMA (4 cores equally distributed across 2 nodes)

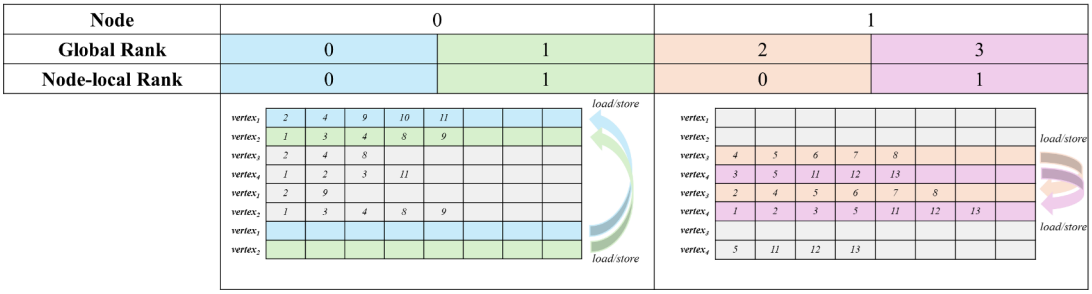



Figure 7: Merge neighbor lists across shared memory (4 cores equally distributed across 2 nodes)

the rest of our analysis. As a result, we would need to insert up to $4\lceil t/p_{global} \rceil$ unique vertices into each rank's neighbor lists. Searching for the index to insert a neighbor into a neighbor list, which is an ordered

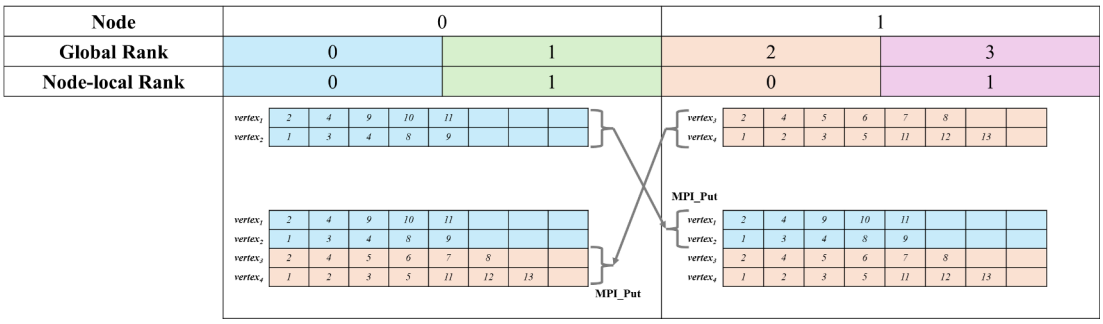



Figure 8: Obtain global view of neighbor lists via one final inter-node RMA exchange (4 cores equally distributed across 2 nodes)

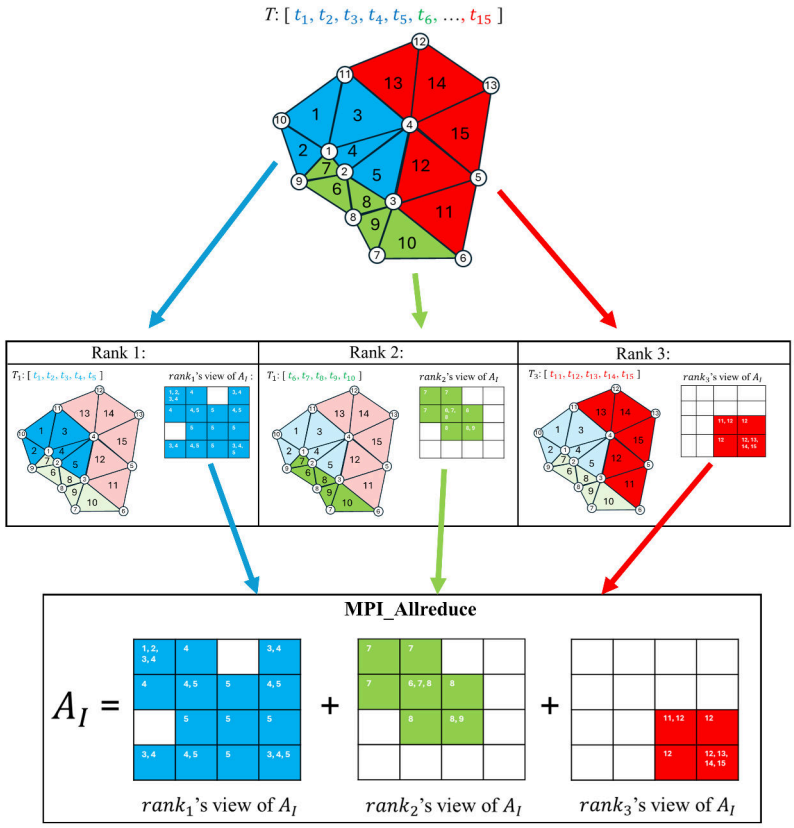



Figure 9: Parallel global stiffness matrix assembly across three ranks (see Section 6)

array of distinct integers, takes $\log_2(t_d)$ time. The cost of insertion is at most $t_d$ time, with the worst case situation being that we move all elements ahead of a new entry by one index. Note that each rank now owns partially complete neighbor lists each containing $v$ lists. As a result, the time taken to compute all rank-local neighbor lists in parallel is bounded above by

$$4\,\lceil t_t/p_{global}\rceil\,(\log_2(t_d)+t_d). \tag{7.7}$$

Next, we must merge neighbor lists within a node across shared memory. Each rank is responsible for updating $\lceil v/p_{nodal}\rceil$ node-local lists by merging rank-local $p_{nodal}\lceil v/p_{nodal}\rceil$ neighbor lists. The upper bound for merging two arrays of ordered distinct integers is $t_d\,(\log_2(t_d)+t_d)$, meaning the upper bound for node-local merging of lists in parallel is $(p_{nodal}\,\lceil t_v/p_{nodal}\rceil)\,t_d\,(\log_2(t_d)+t_d)$. The upper bound for merging neighbor lists in parallel using only shared

memory is

$$p_{nodal}\,\lceil t_v/p_{nodal}\rceil\,t_d\,(\log_2(t_d)+t_d). \tag{7.8}$$

Finally, we must merge neighbor lists across nodes. Each node is responsible for updating $\lceil v/n\rceil$ neighbor lists by merging node-local $n\lceil v/n\rceil$ neighbor lists. Since $(n-1)\lceil v/n\rceil$ of the $n\lceil v/n\rceil$ neighbor lists are on $n-1$ different nodes, each rank is responsible for merging $n-1$ lists that are located on distributed memory, which we exchange via the **MPI_Put** RMA operation. Assume the time for this initial, one-time, inter-node data exchange takes at most $C_{exchange}$ seconds. Each node at most takes $t_n\lceil t_v/t_n\rceil\,t_d\,(\log_2(t_d)+t_d)$ time to perform the actual merging across shared memory (see Eq. 7.8). To complete the global view of the neighbor lists, we conduct one last exchange across nodes, again via **MPI_Put**. Again, we assume this also requires $C_{exchange}$ time. As a result, this inter-node merging step of neighbor lists takes at most

$$t_n\,\lceil t_v/t_n\rceil\,t_d\,(\log_2(t_d)+t_d)+2\,C_{exchange} \tag{7.9}$$

time.

The upper bound for merging neighbor lists in parallel across multiple nodes is essentially the sum of all the steps described in Equations 7.7-7.9, which is

$$\begin{aligned}&4\,\lceil t_t/p_{global}\rceil\,(\log_2(t_d)+t_d)\\&+p_{nodal}\,\lceil t_v/p_{nodal}\rceil\,t_d\,(\log_2(t_d)+t_d)\\&+t_n\,\lceil t_v/t_n\rceil\,t_d\,(\log_2(t_d)+t_d)+2C_{exchange}.\end{aligned} \tag{7.10}$$

We further simplify Equation 7.10 to the following final expression:

$$\begin{aligned}&(\log_2(t_d)+t_d)\,(4\,\lceil t_t/p_{global}\rceil\\&\qquad+p_{nodal}\,\lceil t_v/p_{nodal}\rceil\,d\\&\qquad+t_n\,\lceil t_v/t_n\rceil\,t_d\,)+2C_{exchange}.\end{aligned} \tag{7.11}$$

**7.2 Weight generation** Global stiffness matrix assembly involves computing the local element stiffness matrices of $\lceil t/p_{global}\rceil$ elements per rank in an embarrassingly parallel manner. We assume that the computation of each element's contributions to $A_I$ and $A_B$ requires $C_{weight}$ time. We also assume the one-time **MPI_Allreduce** to account for shared edges across partitions takes $C_{allreduce}$ time. As a result, the upper bound for this step is $\lceil t_t/p_{global}\rceil\,C_{weight}+C_{allreduce}$.

**7.3 Linear Solution** The linear solution step, which utilizes the preconditioned block conjugate gradient method, will be solved in at most $\lceil \dim(A_I)/s\rceil$ iterations, where $s$ is the number of systems to be solved [17]. As a result, O'Leary notes in [17] that this method may require less work than using CG $s$ times. Since we solve for the $x, y, z$ coordinates, we require at most $\lceil \dim(A_I)/3\rceil$ iterations. We assume that each iteration takes $C_{iteration}$. Then the upper bound for runtime will be $\lceil \dim(A_I)/3\rceil C_{iteration}$.

**7.4 Overall Runtime** The upper bound for the runtime of the algorithm for the single node case is the sum of the expressions determined in all of the previous subsections excluding Equation 7.9:

$$\begin{aligned}&(\log_2(t_d)+t_d)\,(4\,\lceil t_t/p_{nodal}\rceil\\&+p_{nodal}\,\lceil t_v/p_{nodal}\rceil\,t_d)\\&+\lceil t_t/p_{nodal}\rceil\,C_{weight}+C_{allreduce}\\&+\lceil \dim(A_I)/3\rceil C_{iteration}.\end{aligned} \tag{7.12}$$

The upper bound for the runtime of the algorithm for the multiple node case (i.e., $p_{global}=p_{nodal}$) is the sum of the expressions determined in all of the previous subsections:

$$\begin{aligned}&(\log_2(t_d)+t_d)\,(4\,\lceil t_t/p_{global}\rceil\\&+p_{nodal}\,\lceil t_v/p_{nodal}\rceil\,t_d\\&+t_n\,\lceil t_v/t_n\rceil\,t_d\,)+2C_{exchange}\\&+\lceil t_t/p_{nodal}\rceil\,C_{weight}+C_{allreduce}\\&+\lceil \dim(A_I)/3\rceil C_{iteration}.\end{aligned} \tag{7.13}$$

For the single node case, we conclude that speedup for precomputing neighbor lists should increase by a factor of $p_{nodal}$ if $t$ is significantly larger than $v$, i.e. a linear speedup, which is the case seen in the meshes used for our experiments in Table 1. For the multiple node case, we conclude that the speedup precomputing neighbor lists should increase by a factor of $n$. Although the speedups should be linear with respect to $n$, we note that $n$ will almost always be less that $p$ unless $p_{nodal}=1$. This means that the speedups for the multi-node case will be less than the single-node case when comparing with the same values for $p$, but will remain linear as $n$ increases.

## 8 Numerical Experiments

We evaluate the performance of ParFEMWARP by running a series of numerical experiments. ParFEMWARP was developed using C++, OpenMPI 4.0.7, and Eigen 3.4.0 [7]. Experiments utilized up to 256 cores on the KU Community Cluster Bigjay partition which contains Intel Xeon Gold 6349 CPUs running at 2.60GHz with FDR Infiniband. The nodes contain a mix of 256GB or 1024GB of RAM per node.

We utilize two meshes for our experiments: 1) hand (Fig. 10) containing approximately 102 million tetrahedra [28], and 2) NASA's High-Lift Common Research

Model (HL-CRM) (Fig. 11) containing approximately 48 million tetrahedra [13]. The hand model was tetrahedrized via Tetgen 1.6.0 [25] and the HL-CRM was unmodified. Further details regarding the meshes utilized for our experiments are provided in Table 1.

| **Mesh** | **Hand** | **HL-CRM** |
|---|---|---|
| **Tetrahedra** | **102,363,432** | **47,791,227** |
| Interior Vertices | 15,548,805 | 7,807,613 |
| Boundary Vertices | 1,894,189 | 281,184 |
| **Total Vertices** | **17,442,994** | **8,088,797** |
| $A_I$ Non-zeros | 233,467,643 | 116,215,667 |
| $A_B$ Non-zeros | 7,003,946 | 1,111,065 |
| **Total Non-zeros** | **206,870,949** | **117,326,742** |

Table 1: Mesh Information

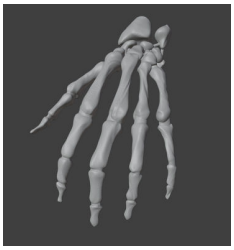


Figure 10: Hand mesh

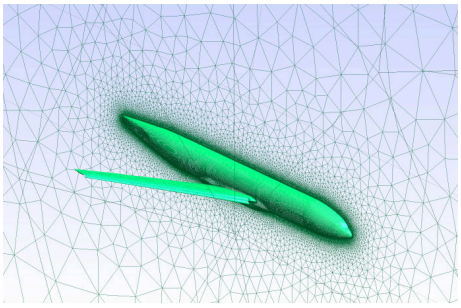


Figure 11: Surface of NASA Common Research Model Mesh

**8.1 Scaling Experiments** To test the scalability and portability of ParFEMWARP across shared and distributed architectures, we ran scaling experiments up to 32 cores on a single node and up to 256 cores across 64 nodes, as Fig. 14 shows. Figs. 16 and 17 show the scaling for different steps of the algorithm, and Tables 2a and 2b report average run-times. Note that our experiments were limited to 4 cores per node in the multi-node experiments, as we were limited by our accessibility to the cluster.

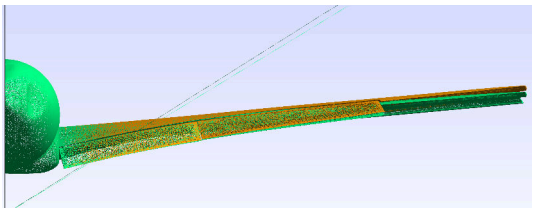


Figure 12: Surface deformation of CRM with the initial mesh **(green)** and the deformed mesh **(orange)** overlaid.

| **ParFEMWARP Average Single Node Runtime Breakdown (Seconds)** | | | | |
|---|---|---|---|---|
| **Cores** | **Neighbor Computation** | **Global Stiffness Generation** | **PBCG** | **Overall** |
| 1 | 252.80 | 15,640.25 | 86,561.65 | 102,464.15 |
| 2 | 83.42 | 7,552.07 | 43,328.45 | 51,132.65 |
| 4 | 49.74 | 3,829.19 | 32,151.30 | 26,561.45 |
| 8 | 32.54 | 1,845.51 | 14,312.55 | 16,422.95 |
| 16 | 21.27 | 927.88 | 9,737.79 | 10,921.80 |
| 32 | 15.60 | 468.23 | 7,533.33 | 8,275.55 |

(a) FEMWARP single node runtimes

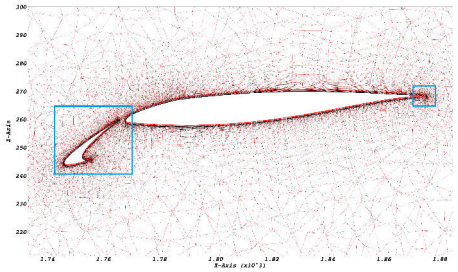
| **ParFEMWARP Average Multi-Node Runtime Breakdown (Seconds)** | | | | |
|---|---|---|---|---|
| **Cores** | **Neighbor Computation** | **Global Stiffness Generation** | **PBCG** | **Overall** |
| 1 | 252.80 | 15,640.25 | 86,561.65 | 102,464.15 |
| 2 | 83.42 | 7,552.07 | 43,328.45 | 51,132.65 |
| 4 | 49.74 | 3,829.19 | 32,151.30 | 26,561.45 |
| 8 | 61.46 | 1,888.53 | 16,330.35 | 18,510.15 |
| 16 | 50.49 | 1,004.39 | 16,050.25 | 17,333.05 |
| 32 | 49.22 | 521.89 | 14,089.15 | 14,882.90 |
| 64 | 35.86 | 262.09 | 9,288.29 | 9,835.51 |
| 128 | 28.33 | 127.88 | 7,371.07 | 7,788.25 |
| 256 | 25.74 | 67.97 | 5,828.62 | 6,181.26 |

(b) FEMWARP multi-node runtimes

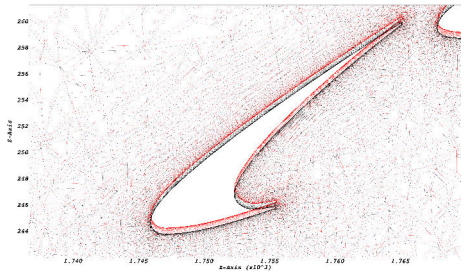
Table 2: FEMWARP runtimes

According to Fig. 17 (a), the performance of the precomputation stage degrades slightly when introducing more than one node. However, performance continues to improve with larger node counts due to inter-node and intra-node merging via RMA and SHM, respectively. Tables 2a and 2b show that the linear solver dominates the runtime.

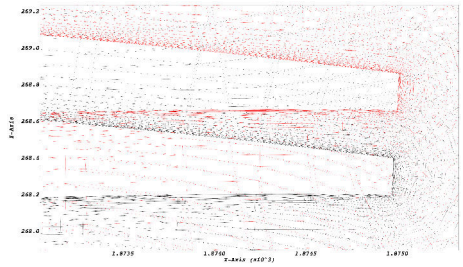
**8.2 Application** We use ParFEMWARP to warp the HL-CRM mesh upon surface deflections of the wing. Such deflections of the wing and how they relate to the rest of the volume has been studied in previous literature [10] [11]. For our user-defined boundary deformation, we calculate deformations of the wing based on formulas for calculating the deflection of an end-loaded cantilevered beam [5]. Since we assess this problem completely geometrically, we set elasticity and inertia to 1 and discard them. Equation 8.14 shows the



(a) Cross section with inset regions marked in **blue**



(b) Left inset region



(c) Right inset region

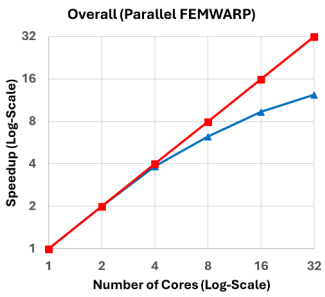
Figure 13: Cross sections of the HL-CRM mesh before and after deformation via ParFEMWARP. **Black** is the initial mesh and **red** is the deformed mesh.

resulting equations used to calculate the deformation for use in Equation 2.5. We perform this deformation in an embarrassingly parallel manner. For deforming the HL-CRM model, we utilize

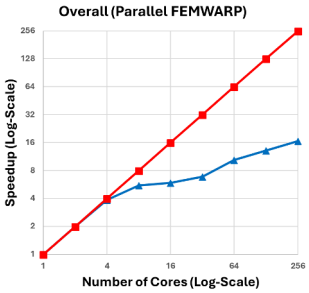
$$\begin{aligned} \hat{z}_B &= z_B + P\frac{y_B^2(3l - y_B)}{6} \\ \hat{y}_B &= y_B \cos(\theta) \\ \hat{x}_B &= x_B \\ \theta &= \frac{Pl^2}{2}, \end{aligned} \tag{8.14}$$

where $l$ is the length of the beam, $\theta$ is the angle of deflection, and $P$ is force, which we set to 0.02 for our experiments.

Fig. 12 shows our boundary deformation on a surface mesh, and Fig. 13 shows the cross sections of the original mesh and the mesh deformed via ParFEMWARP, respectively. The cross sections are both located 4.1% into the the y-axis.




(a) Single-node speedup




(b) Multi-node speedup

Figure 14: ParFEMWARP speedups **(blue)** for versus ideal speedup **(red)** for single and multiple node experiments.

| Mean Ratio Mesh Quality | | | |
|---|---|---|---|
| **State** | **Maximum** | **Average** | **Minimum** |
| Initial | 0.9996000 | 0.2748491 | 0.0002246 |
| Deformed | 0.9995954 | 0.2748399 | 0.0002246 |

Table 3: Mesh quality comparison before and after deformation of HL-CRM

We also provide mesh quality statistics in Table 3 and Figure 15 using the mean ratio metric [16]. This metric ranges from 0 to 1, with 0 representing a degenerate element and 1 representing an ideal element. Also, note that the x-axis in Figure 15 is on a log scale. We see very little change in the mesh quality after the deformation. This is due to our experiment involving a small deformation to avoid tangling the mesh, which is highly anisotropic before and after applying the deformation as indicated by Table 3 and Figure 15. Note that larger deformations that yield valid meshes are possible with ParFEMWARP for isotropic tetrahedral meshes [24]. Regardless, the average mean ratio before and after the deformation remained nearly the same. Additionally, according to Figure 15, a slight degradation in mesh quality was observed. This is typical for mesh warping algorithms for a deformation of this size.

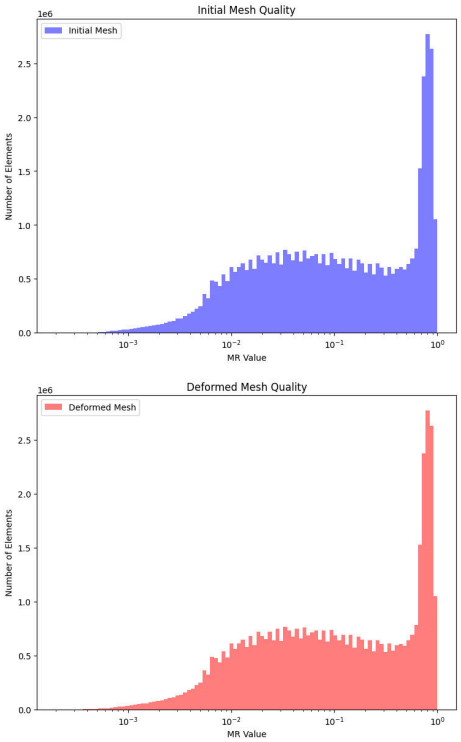



Figure 15: Mesh quality histograms for before and after deformation of HL-CRM in log-scale

## 9 Conclusions and Future Work

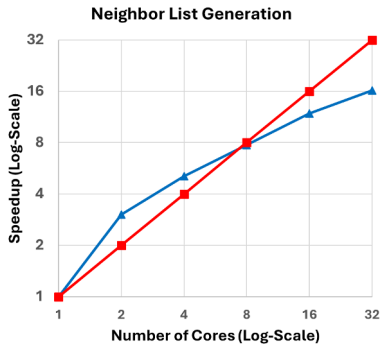
We proposed ParFEMWARP, a parallel version of the FEMWARP algorithm [23, 24], for warping tetrahedral meshes. Our method utilizes shared-memory and one-sided communication features available in MPI-3, and hence the implementation is portable across shared and distributed memory architectures. We utilized hybrid parallelism to precompute global stiffness matrix sparsity. Specifically, we use shared memory and one-sided communication to achieve good performance when calculating neighbor lists. The authors are not aware of any other work that achieves this across a cluster of multi-core systems. We believe the approach we described can benefit other FEM-based applications, such as the one described by Krysl [12]. In that paper, Krysl showed that their sparsity computation method achieves better speedups when solving nonlinear FEM problems, since the global stiffness matrix needs to be reassembled multiple times. Since Krysl's method has only been scaled to a single node, it would be interesting to modify our multi-node strategy to support larger nonlinear problems. Furthermore, the precomputation method may serve useful for simulations that require regenerating global stiffness matrices sporadically. In the context of mesh warping via FEMWARP, one may perform several deformations with a fixed $A_I$ and $A_B$, then remesh when the quality eventually degrades too much, and then perform several more deformations. In addition, we utilized the preconditioned block conjugate gradient method to efficiently solve a linear system with multiple right-hand sides.

ParFEMWARP exhibits excellent near-linear speedup in the single node experiments and sublinear speedups in the multi-node experiments. Our largest multi-node experiment shows a 16.6x speedup, while our largest single node speedup is 12.1x, thus achieving a 33.9% increase between multi-node and single node speedups. Utilizing a multi-node configuration for ParFEMWARP may achieve better scaling when solving larger problems, whereas a single node configuration is suitable for relatively small problems. Furthermore, the cluster resources available to the authors provided medium-scale parallelism, which dictated the number of cores and nodes and the problem size utilized in our experiments.

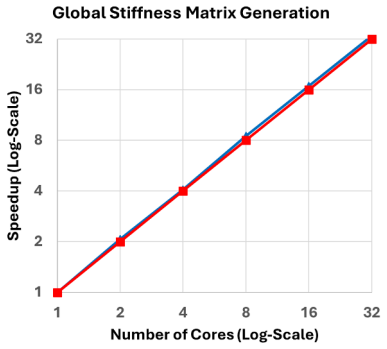
Future work revolves around improving the performance of the linear solution step, as this dominates the runtime. This includes exploring various preconditioners that may yield improved results. Furthermore, carefully utilizing mixed-precision operations may increase the number of flops per iteration while maintaining reasonable accuracy. This has already been observed in mixed-precision PCG [1], so it would be interesting to study how mixed-precision PBCG would perform. Additionally, collective operations in PBCG can be replaced with shared memory and RMA operations to overlap communication with computation. Furthermore, it would be interesting to see whether non-blocking MPI can achieve better speed-ups in place of one-sided communication for inter-node merging of neighbor lists. Finally, developing a multi-GPU parallelization in the weight generation and linear solving steps may further improve the performance.

## Acknowledgments

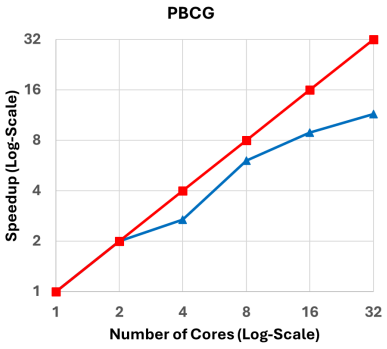
This work was performed at the HPC facilities operated by the Center for Research Computing at the University of Kansas (KU) supported by National Science Foundation grant OAC-2117449. The work of the first author was supported by REU supplements to NSF grants OAC-1808553 and CBET-2245153 in addition to the KU School of Engineering Undergraduate Research Fellows Program. The work of the second author was supported in part by NSF grant OAC-1808553 and NSF grant CBET-2245153.




(a)

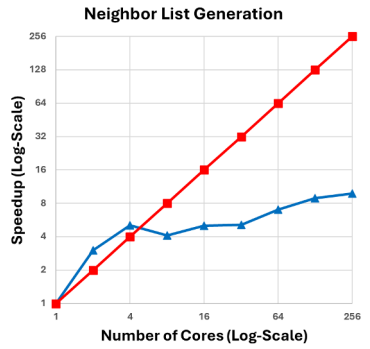



(b)

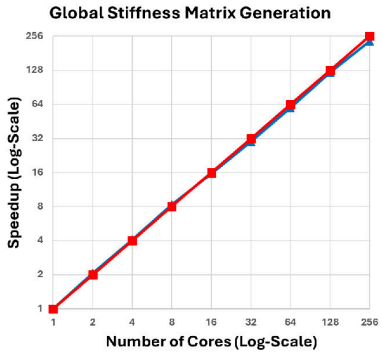



(c)

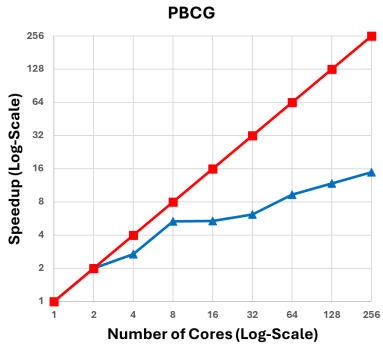
Figure 16: ParFEMWARP speedup **(blue)** breakdown for single node versus ideal speedup **(red)**.




(a)




(b)




(c)

Figure 17: ParFEMWARP speedup **(blue)** breakdown for multiple nodes versus ideal speedup **(red)**.

## References


[1] A. Abdelfattah, H. Anzt, E. G. Boman, E. Carson, T. Cojean, J. Dongarra, A. Fox, M. Gates, N. J. Higham, X. S. Li, J. Loe, P. Luszczek, S. Pranesh, S. Rajamanickam, T. Ribizel, B. F. Smith, K. Swirydowicz, S. Thomas, S. Tomov, Y. M. Tsai, and U. M. Yang, *A survey of numerical linear algebra methods utilizing mixed-precision arithmetic*, The International Journal of High Performance Computing Applications, 35 (2021), pp. 344–369.

[2] M. Brinskiy and M. Lubin, *An introduction to MPI-3 shared memory programming*, Technical Report, Intel, 2017.

[3] R. Cockett, *The block conjugate gradient for multiple right hand sides in a direct current resistivity inversion*. row1.ca/pdf/dc-resistivity-block-cg.pdf, 2012.

[4] T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar, *A survey of direct methods for sparse linear systems*, Acta Numerica, 25 (2016), p. 383–566.

[5] J. Gere and B. Goodno, *Mechanics of Materials*, Cengage Learning, 2009.

[6] T. Gerhold and J. Neumann, *The parallel mesh deformation of the DLR TAU-code*, in New Results in Numerical and Experimental Fluid Mechanics VI, C. Tropea, S. Jakirlic, H.-J. Heinemann, R. Henke, and H. Hönlinger, eds., Berlin, Heidelberg, 2008, Springer Berlin Heidelberg, pp. 162–169.

[7] G. Guennebaud, B. Jacob, et al., *Eigen v3*. eigen.tuxfamily.org, 2010.

[8] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur, *MPI + MPI: A new hybrid approach to parallel programming with MPI plus shared memory*, Computing, 95 (2013), p. 1121–1136.

[9] P. Kaufmann, O. Wang, A. Sorkine-Hornung, O. Sorkine-Hornung, A. Smolic, and M. Gross, *Finite element image warping*, Computer Graphics Forum (Proceedings of EUROGRAPHICS), 32 (2013), pp. 31–39.

[10] G. Kenway, G. Kennedy, and J. R. R. A. Martins, *Aerostructural optimization of the common research model configuration*, in 15th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, 2014, pp. 1–19.

[11] S. Keye and M. Gammon, *Development of deformed CAD geometries of NASA's common research model for the Sixth AIAA CFD Drag Prediction Workshop*, in 34th AIAA Applied Aerodynamics Conference, 2016, pp. 1–7.

[12] P. Krysl, *Parallel assembly of finite element matrices on multicore computers*, Computer Methods in Applied Mechanics and Engineering, 428 (2024), p. 117076.

[13] D. S. Lacy and A. J. Sclafani, *Development of the high lift common research model (HL-CRM): A representative high lift configuration for transonic transports*, in 54th AIAA Aerospace Sciences Meeting, pp. 1–24.

[14] P. Lamata, S. Niederer, D. Nordsletten, D. C. Barber, I. Roy, D. R. Hose, and N. Smith, *An accurate, fast and robust method to generate patient-specific cubic Hermite meshes*, Medical Image Analysis, 15 (2011), pp. 801–813.

[15] M. Li, X. Lu, S. Potluri, K. Hamidouche, J. Jose, K. Tomko, and D. K. Panda, *Scalable Graph500 design with MPI-3 RMA*, in 2014 IEEE International Conference On Cluster Computing (CLUSTER), Los Alamitos, CA, USA, Sept. 2014, IEEE Computer Society, pp. 230–238.

[16] A. Liu and B. Joe, *Relationship between tetrahedron shape measures*, BIT, 34 (1994), p. 268–287.

[17] D. P. O'Leary, *The block conjugate gradient algorithm and related methods*, Linear Algebra and its Applications, 29 (1980), pp. 293–322. Special Volume Dedicated to Alson S. Householder.

[18] T. Panitanarak and S. M. Shontz, *A parallel log barrier-based mesh warping algorithm for distributed memory machines*, Engineering with Computers, 34 (2017), pp. 59–76.

[19] J. Park, S. M. Shontz, and C. S. Drapaca, *Automatic boundary evolution tracking via a combined level set method and mesh warping technique: Application to hydrocephalus*, in Mesh Processing in Medical Image Analysis 2012, J. A. Levine, R. R. Paulsen, and Y. Zhang, eds., Springer Berlin Heidelberg, 2012, pp. 122–133.

[20] Y. Saad, *Iterative Methods for Sparse Linear Systems: Second Edition*, Society for Industrial and Applied Mathematics, 2003.

[21] S. P. Sastry, E. Kultursay, S. M. Shontz, and M. T. Kandemir, *Improved cache utilization and preconditioner efficiency through use of a space-filling curve mesh element- and vertex-reordering technique*, Engineering with Computers, 30 (2014), p. 535–547.

[22] N. Secco, G. K. W. Kenway, P. He, C. A. Mader, and J. R. R. A. Martins, *Efficient mesh generation and deformation for aerodynamic shape optimization*, AIAA Journal, (2021), pp. 1151–1168.

[23] S. M. Shontz, *Numerical Methods for Problems with Moving Meshes*, PhD thesis, Cornell University, January 2005.

[24] S. M. Shontz and S. A. Vavasis, *Analysis of and workarounds for element reversal for a finite element-based algorithm for warping triangular and tetrahedral meshes*, BIT, Numerical Mathematics, 50 (2010), pp. 863–884.

[25] H. Si, *Tetgen, a Delaunay-based quality tetrahedral mesh generator*, ACM Transactions on Mathematical Software, 41 (2015), pp. 1–36.

[26] M. L. Staten, S. J. Owen, S. M. Shontz, A. G. Salinger, and T. S. Coffey, *A comparison of mesh morphing methods for 3D shape optimization*, in Proceedings of the 20th International Meshing Roundtable, W. R. Quadros, ed., Berlin, Heidelberg, 2012, Springer Berlin Heidelberg, pp. 293–311.

[27] W. Tang, F. Jia, and X. Wang, *An improved*

*adaptive triangular mesh-based image warping method*, Frontiers in Neurorobotics, 16 (2023), p. 1042429.

[28] G. Turk and B. Mullins, *Large geometric models archive: Skeleton hand*. sites.cc.gatech.edu/projects/large_models/.

[29] Z. Zhao, R. Ma, L. He, X. Chang, and L. Zhang, *An efficient large-scale mesh deformation method based on MPI/OpenMP hybrid parallel radial basis function interpolation*, Chinese Journal of Aeronautics, 33 (2020), pp. 1392–1404.