SAND2017-0977J

# A distributed-memory hierarchical solver for sparse matrices

Chao Chen[a], Hadi Pouransari[b], Sivasankaran Rajamanickam[c], Erik G. Boman[c], Eric Darve[a,b,*]

[a] *Institute of Computational and Mathematical Engineering, Stanford University, Stanford, USA*
[b] *Department of Mechanical Engineering, Stanford University, Stanford, USA*
[c] *Center for Computing Research, Sandia National Laboratories, Albuquerque, USA*

**Abstract**

We present a parallel hierarchical linear solver for solving large, sparse linear systems on distributed-memory machines. The fully algebraic algorithm exploits the low-rank structure of fill-in blocks during the Gaussian elimination process; thus it is faster and more memory-efficient compared with sparse direct solvers. Our hierarchical solver can be used as a direct solver (high-accuracy setting) or a preconditioner (low-accuracy setting). The parallel implementation is based on a data decomposition such that only local communication is needed for updating boundary information on every processor. Our algorithm is based on dense linear algebra subroutines, which can potentially be accelerated using modern many-core processors. We show various numerical experiments to demonstrate the scalability of our solver and compare it with a state-of-the-art sparse direct solver. The new hierarchical solver achieves an average speedup of 45 in factorizing three two-million sized test problems on 256 cores of a supercomputer.

*Keywords:* Parallel linear solver, Sparse matrix, Hierarchical matrix

*Corresponding author

*Email addresses:* cchen10@stanford.edu (Chao Chen), hadip@stanford.edu (Hadi Pouransari), srajama@sandia.gov (Sivasankaran Rajamanickam), egboman@sandia.gov (Erik G. Boman), darve@stanford.edu (Eric Darve)

Sandia National Laboratories is a multi-mission laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energys National Nuclear Security Administration under contract DE-AC04-94AL85000.

## 1. Introduction

Solving large sparse linear systems is an important building block in many engineering applications. For example, the common approach to solve elliptical partial differential equations (PDE) is to first discretize the equations with local bases such as finite difference or finite element methods, and then solve the subsequent sparse linear systems. Our goal is to develop a robust, algebraic, parallel linear solver for solving large-scale sparse linear systems efficiently on distributed-memory machines.

To solve sparse linear systems more efficiently, several sparse direct solvers based on Gaussian elimination have been developed for sequential [1, 2, 3], shared-memory [4, 5] and distributed-memory computers [6, 7]. They organize computation efficiently (e.g., the multifrontal algorithm [8]), and leverage different elimination orderings, such as the nested dissection ordering [9] and the approximate minimum degree ordering [10] to reduce fill. For a survey of sparse direct solvers, the reader is referred to Davis et al. [11]. However, those state-of-the-art sparse direct solvers still have $O(N^2)$, time complexity in the factorization phase for solving three-dimensional problems, where $N$ is the problem size. Sparse direct methods also introduce a large amount of nonzero entries (fill-in). When used with nested-dissection ordering methods the fill-in is $O(N^{\frac{4}{3}})$ for 3D problems. The quadratic factorization cost and the high memory usage limits the problem sizes where sparse direct methods can be used. However, their robustness make them attractive for harder problems, such as Helmholtz equations.

A completely different approach for solving sparse matrices is to use iterative methods [12]. Unlike sparse direct solvers, iterative methods are more memory efficient because they treat a sparse matrix as a black-box linear operator and do not modify the matrix itself. For example the multigrid method [13, 14] is an effective preconditioner for iterative methods, leading to some of the fastest solvers for many elliptic PDEs. For example, the geometric multigrid method has $O(N)$ time complexity for solving Poisson's equation on regular domains. However, the convergence of iterative methods is not guaranteed when solving general linear systems, and the iteration number may grow dramatically for matrices with large condition numbers. For example, having varying coefficients in a Poisson equation may slow down the convergence of multigrid, in some cases, compared to the constant coefficient

case. Moreover, multigrid may fail to converge for indefinite systems such as Helmholtz (see [15, 16, 17] for examples of multigrid algorithms for indefinite systems). Multigrid methods are difficult to develop for linear systems resulting from the coupling of different PDEs. The convergence of multigrid is largely unproven for hyperbolic or parabolic PDEs. The cost of set up of algebraic multigrid methods depends on sparse matrix-sparse matrix multiplication which is complex to scale [18, 19].

To reduce the memory usage and improve the time complexity of sparse direct solvers, several hierarchical solvers [20, 21, 22, 23, 24] based on the $\mathcal{H}$-matrix theory [25] have been developed, and some parallel algorithms have been introduced in [26, 27, 28, 29]. The key idea is data sparsity, in the form of low-rank approximation of dense submatrix blocks such that the memory footprint is reduced and matrix arithmetic operations become less costly. As was shown recently in [24], $\mathcal{H}$-preconditioners share similarities with incomplete LU (ILU) [30] and multigrid but with some important differences. Similar to ILU, $\mathcal{H}$-preconditioners are based on a factorization of the matrix using sparse triangular factors. However, low-rank approximations are used to design optimal orthogonal transformations of the variables so that a higher accuracy than ILU can be achieved (for similar computational cost). For example, singular values of fill-in blocks are often found to be decreasing geometrically so that high accuracy can be achieved by varying the rank of the approximation. This is in contrast with ILU where tuning the accuracy is difficult and often leads to a rapid increase in computational cost.

Similar to multigrid, $\mathcal{H}$-preconditioners use a series of nested grids, with increasingly coarser resolutions to achieve an $O(N)$ complexity. In algebraic multigrid, the construction of restriction and prolongation operators [31, 32] for general PDE types and matrices is problematic and often requires tuning and manual adjustments. Instead in $\mathcal{H}$-preconditioners, generic algebraic techniques are used for low-rank approximations (e.g., singular value decomposition [33], rank revealing LU [34, 35], rank-revealing QR [36, 37], adaptive cross approximation [38], ULV factorization [39]), from which restriction and prolongation operators can be naturally extracted in a systematic fashion, regardless of the underlying PDE or physical problem.

We investigate the parallelization of a recent hierarchical preconditioner named LoRaSp [24]. LoRaSp is based on $\mathcal{H}^2$-matrices [40, 41], which introduce an additional hierarchical structure to reduce the storage requirements and computational complexity of $\mathcal{H}$-matrices. LoRaSp employs domain partitioning as opposed to the multifrontal nested dissection algorithm that is

used in [20, 21, 23]. LoRaSp exhibits almost linear time complexity (under certain assumptions), and is more memory efficient than sparse direct solvers (when low-rank compression is used). We propose a new parallelization approach to LoRaSp that is based on data decomposition and graph coloring. Various optimizations, including asynchronous communications and execution of computational kernels, are proposed. Specific contributions in this paper include:

1. New derivation of LoRaSp. This leads to a much simplified presentation of the algorithm so that the structure of the calculation and data dependencies are more obvious and easy to reason about.
2. Analysis of the data dependency in the algorithm. Based on this two parallel approaches are proposed.
3. Bulk synchronous parallel algorithm: this is the simplest to implement. It decomposes the calculations in 3 phases (per level) with blocking communications in between.
4. Task-based asynchronous parallel algorithm: an optimal scheduling is proposed based on a critical path analysis. This algorithm executes the phase with highest priority first (located along the critical path), followed by a phase with medium priority, and low priority. Communication is initiated in an asynchronous manner as soon as the data is available so that maximum concurrency is achieved, that is the maximum number of tasks with ready data is generated as soon as possible. The large number of executable tasks allows hiding some of the communication latency, thereby minimizing the idle time due to communication.
5. Coloring algorithm to extract concurrency in the execution. Optimizing the load-balancing in the presence of coloring constraints is discussed.
6. Benchmarks for a variety of problems including: elliptic PDEs, PDEs with varying coefficients, indefinite systems (Helmholtz equation). We separately run sequential and parallel benchmarks.
7. Analysis of the parallel efficiency under different conditions
8. Comparison with a conventional, highly-optimized sparse direct solver of the runtime and memory footprint
9. Breakdown of the running time per kernel and communication phases for various problems

Our parallel algorithm has the following beneficial features:

- Only local communication is needed, since only boundary information is updated on every processor. This is roughly comparable to the communication pattern in multigrid.

- The communication volume is small compared to the amount of computation. It will be shown later that the communication-to-computation ratio is roughly the surface-to-volume ratio for each subdomain. Therefore good scalability can be expected, provided sufficient memory is available on each computer node so that large enough leaf subdomains can be stored.

- The bulk of the work in our parallel algorithm lies in local dense linear algebra operations. Compared to many iterative methods, our algorithm is well suited to modern architectures and provides an opportunity for using modern many-core processors such as graphics processing units (GPU) and Intel Xeon Phi processors.

The remainder of this paper is organized as follows. Section 2 reviews the sequential algorithm in [24] as the foundation for the parallel algorithm. Section 3 discusses the details of our parallel algorithm. Section 4 shows numerical benchmarks.

## 2. Sequential hierarchical solver

Our algorithm is based on the sequential algorithm in [24] to solve a sparse linear system $Ax = b$. For simplicity, we assume that $A$ is a symmetric positive definite (SPD) matrix, although our approach extends to indefinite and nonsymmetric matrices as well.

Similar to the multifrontal method, our algorithm eliminates all degrees of freedom (DOFs), i.e., rows/columns, level by level. The difference is that our algorithm compresses dense fill-in blocks during the elimination process to maintain the sparsity of $A$. Instead of using the nested dissection ordering as usually used in the multifrontal method, our algorithm starts with a clustering of all DOFs.

Suppose $V$ is the set of all DOFs and the clustering is $V = \cup_i \pi_i$, where $\pi_i$ stands for a cluster of DOFs. Our algorithm proceeds by eliminating some portions of DOFs in every cluster. To be more specific, the DOFs in a cluster $\pi_i$ are split into fine DOFs $\pi_i^f$ and coarse DOFs $\pi_i^c$, after the dense fill-in blocks with respect to $\pi_i$ are compressed. The fine DOFs $\pi_i^f$ are then

eliminated; the coarse DOFs $\pi_i^c$ will belong to the next level. After all fine DOFs are eliminated, we have a smaller linear system corresponding to only the coarse DOFs. The same idea is applied recursively until the linear system is small enough to be solved using some other technique, say a conventional sparse Cholesky factorization.

*2.1. Sparsification and Low-rank Elimination*

We first illustrate the key step in our algorithm that exploits the low-rank structure of the dense fill-in blocks. Let's consider a block decomposition of the partially factored matrix $\bar{A}$ at some stage in the algorithm, with only 3 blocks as shown in Eq. (1), where:

- $s$ stands for a cluster of DOFs $\pi_s$ to be eliminated ("self"),
- $n$ stands for the union of all neighbor clusters ("neighbor"), and
- $w$ stands for the union of all other clusters ("well-separated"), i.e.,

$$V \setminus (\pi_s \bigcup_{j \text{ neighbor of } s} \pi_j)$$

$$\bar{A}x = \begin{pmatrix} A_{ss} & A_{sn} & A_{sw} \\ A_{sn}^T & A_{nn} & A_{nw} \\ A_{sw}^T & A_{nw}^T & A_{ww} \end{pmatrix} \begin{pmatrix} x_s \\ x_n \\ x_w \end{pmatrix} = \begin{pmatrix} b_s \\ b_n \\ b_w \end{pmatrix} \tag{1}$$

where $\bar{A}$ is the partially factored matrix after few eliminations on the original matrix $A$. The non-zero entries of $A_{sw}$ and $A_{ws} = A_{sw}^T$ are fill-ins created due to elimination of previous blocks.

Different definitions of the well-separated criterion can be used (which implies a partitioning of the clusters in $n$ and $w$ above). Here, for easier illustration, we simply decide that two clusters are well-separated if they are not connected in the original matrix $A$. This means that well-separated blocks are associated with fill-in.

We make this connection more precise by using terminology from ILU($k$) preconditioners (incomplete LU with number of levels of fill $k$). We introduce a new notation, $A_B$, which is a matrix whose size is the number of clusters, and such that $[A_B]_{IJ} \neq 0$ if and only if there is an $i \in I$ and $j \in J$ such that $a_{ij} \neq 0$. Then, well-separated blocks correspond to entries in $[A_B]^k$ that are not in $A_B$ for some $k >1$. This is often called ILU($k-1$) fill-in. An interesting property of our algorithm is that fill-in entries (not in $A_B$

by definition) correspond to non-zero blocks in $[A_B]^2$ only, that is the fill-in never goes beyond ILU(1).[2]

Now consider the fill-in block $A_{sw}$ and assume that it is approximately low-rank [24]:

$$A_{sw} = UZ + O(\epsilon) \tag{2}$$

where $U$ is an $m \times r$ orthonormal matrix, where $r < m$. We can compute an orthonormal matrix $V$ that is the $A_{ss}^{-1}$-orthogonal complement of $U$ (e.g., using a Gram-Schmidt type orthogonalization process):

$$V^T A_{ss}^{-1} U = 0 \tag{3}$$

where $V$ is $m \times (m - r)$.

We introduce the sparsification matrix, $\mathcal{E}$, which is a block diagonal matrix defined as follows

$$\mathcal{E} = \begin{pmatrix} V^T A_{ss}^{-1} & & \\ U^T A_{ss}^{-1} & & \\ & I & \\ & & I \end{pmatrix} \tag{4}$$

Since the columns of $U$ and $V$ form a basis for $\mathbb{R}^n$, we can apply the following decomposition:

$$A_{ss} x_s = V x_f + U x_c \tag{5}$$

Therefore, we have:

$$x = \begin{pmatrix} x_s \\ x_n \\ x_w \end{pmatrix} = \mathcal{E}^T \begin{pmatrix} x_f \\ x_c \\ x_n \\ x_w \end{pmatrix}$$

[2]Note that the fill-in is defined in terms of clusters or blocks, using $A_B$. This is different from the fill-in in ILU(1) when applied to $A$ directly, that is considering the scalar entries in $A$.

Multiplying Eq. (1) by $\mathcal{E}$ to the left we get

$$\mathcal{E}\bar{A}\mathcal{E}^T \begin{pmatrix} x_f \\ x_c \\ x_n \\ x_w \end{pmatrix} = \mathcal{E}b$$

The sparsified matrix $\mathcal{E}\bar{A}\mathcal{E}^T$ is then

$$\begin{pmatrix} (V^T A_{ss}^{-1} V) & & V^T A_{ss}^{-1} A_{sn} & \\ & (U^T A_{ss}^{-1} U) & U^T A_{ss}^{-1} A_{sn} & (U^T A_{ss}^{-1} U) Z \\ A_{sn}^T A_{ss}^{-1} V & A_{sn}^T A_{ss}^{-1} U & A_{nn} & A_{nw} \\ & Z^T (U^T A_{ss}^{-1} U) & A_{nw}^T & A_{ww} \end{pmatrix} \tag{6}$$

The key result of this operation is that $x_f$ is now disconnected from the $w$ DOFs, that is the corresponding entries in the matrix are zeros. The elimination of $x_f$ will lead to fill-in only among the neighbor DOFs $n$.

The elimination of the $x_f$ DOFs can be formally defined as a multiplication of $\mathcal{E}\bar{A}\mathcal{E}^T$ from left by $\mathcal{G}$ and to right by $\mathcal{G}^T$, where:

$$\mathcal{G} = \begin{pmatrix} I & & & \\ & I & & \\ -A_{sn}^T A_{ss}^{-1} V (V^T A_{ss}^{-1} V)^{-1} & & I & \\ & & & I \end{pmatrix} \tag{7}$$

The elimination of the $x_c$ DOFs is then postponed to the next stage or level in the factorization process. This is similar to the multigrid approach. We start from a fine grid and end up with a coarser grid that has fewer unknowns. This process is repeated until the coarse grid is sufficiently small for a conventional factorization method.

Again, formally, postponing the elimination of the $x_c$ DOFs can be defined as a multiplication by a permutation matrix $P$:

$$P \begin{pmatrix} x_f \\ x_c \\ x_n \\ x_w \end{pmatrix} = \begin{pmatrix} x_f \\ x_n \\ x_w \\ x_c \end{pmatrix}$$

*2.2. Low-rank Approximation Strategies*

If all blocks have full rank and we perform no low-rank approximation, then our algorithm is a direct solver. The key part of the algorithm is to identify blocks that have (approximate) low-rank structure and exploit this. Any low-rank factorization may be used in our algorithm, for example, the singular value decomposition (SVD), the rank-revealing QR (RRQR), the rank-revealing LU (RRLU), the adaptive cross approximation (ACA), the LU factorization with rook pivoting, or the ULV decomposition. In our implementation, we used the SVD.

We investigated two options to select the rank $r$: (a) dynamic $r$ based on the singular values of the fill-in blocks, and computed for a given user-prescribed error tolerance $\epsilon$ (error in the 2-norm), and (b) a fixed value of $r$. The former typically gives better quality preconditioners but may be more expensive, while the latter puts a strict upper limit on the memory usage and the factorization cost.

*2.3. Hierarchical solver algorithm*

We introduced the idea of sparsification for a single block $s$ in $A$. We now focus on the details for the full hierarchical factorization.

A (symmetric) sparse matrix can be represented by a graph, where each vertex represents a row/column and edges between vertices represent nonzero matrix entries. To get the block structure of $A$, we partition the adjacency graph of $A$ (or $A + A^T$ for the unsymmetric case) using a graph partitioning algorithm as found in METIS/ParMETIS [42], Scotch [43], or Zoltan [44].

For a cluster $\pi_i$ of a partitioning $\Pi$, we denote the sequence of sparsification, elimination and permutation by

$$LRE(A, \Pi, \pi_i) = P_i \mathcal{G}_i \mathcal{E}_i A \mathcal{E}_i^T \mathcal{G}_i^T P_i^T$$

where LRE stands for Low-Rank Elimination. After applying LRE to all clusters of $\Pi$, we end up with a set of un-eliminated coarse variables ($x_c$ corresponding to each cluster). We then use a partitioning of the coarse graph, and recursively repeat the low-rank elimination steps. This is explained in Algorithm 1.

Hence, the full process is similar to a Cholesky factorization:

$$GAG^T = I, \text{ where } G = P_k \mathcal{G}_k \mathcal{E}_k \cdots P_1 \mathcal{G}_1 \mathcal{E}_1 \tag{8}$$

where $\mathcal{E}_i$, $\mathcal{G}_i$, and $P_i$ are the sparsification, elimination, and permutation matrices at step $i$, respectively. Since all the factors are sparse triangular matrices, we can solve the linear system through the standard forward and backward substitution process. For a detailed explanation of the algorithm and analysis of the hierarchical solver, we refer the readers to [24].

**Algorithm 1** Hierarchical solver: factorization phase

**procedure** HIERARCHICAL_SOLVER($A$)
  **if** the matrix $A$ is small enough **then**
    Factor $A$ using a conventional method
    **return**
  **end if**
  Partition the graph of $A$ and compute the vertex clusters $\Pi = \cup_{i=1}\pi_i$
    ▷ The number of clusters is adjusted so the cluster sizes are roughly constant
  **for** $\pi_i \in \Pi$ **do**
    $A \leftarrow \text{LRE}(A, \Pi, \pi_i)$
  **end for**
  Extract $A_c$, the matrix associated with the coarse DOFs
    ▷ $A_c$ is the Schur complement resulting from the elimination of all fine nodes at this level
  HIERARCHICAL_SOLVER($A_c$) ▷ Recursive call with smaller matrix $A_c$
**end procedure**

### *2.4. Relationship to LoRaSp*

In the previous sections, we presented a hierarchical solver algorithm. Although the derivation is novel, the algorithm is in fact equivalent to the LoRaSp algorithm [24]. We believe the new formulation is both simpler to understand and could lead to more efficient implementations.

While the previous algorithm used *extended* sparsification (introduce new variables that expands the linear system), the new algorithm uses sparsification in-place. While the previous algorithm used a binary tree with black and red nodes that were later eliminated, the new algorithm avoids all that complexity. In the special case of recursive bisection used to generate a set of nested partitionings (clusterings), it reduces to the classic LoRaSp algorithm, but our new version allows more general partitioning strategies.

## 3. Parallel hierarchical solver

Our algorithm is similar to the multigrid method in that both use multiple levels of grids. In the parallel algorithm, every processor owns a piece of the entire grid and exchanges information with its neighbor processors to update information at the boundary of the grid. With this data decomposition scheme, the communication is always local, and importantly the communication-to-computation ratio can be bounded by the surface-to-volume ratio of each subdomain (owned by a processor). In other words, the amount of communication can be made small compared to the computation for subdomains that are large enough.

Furthermore, our algorithm uses more flops than the multigrid method per grid node (higher arithmetic intensity), so it has the potential to be more scalable than the multigrid method.

### *3.1. Data decomposition*

We present our parallel algorithm for the finest grid; the same algorithm applies for the other levels as well. The fine grid is the quotient graph $G$ of matrix $A$ with respect to a clustering of the row/column indices. The entire graph $G$ is decomposed over all processors: $G = \cup_P G_P$, where processor $P$ owns subgraph $G_P$. We will use the term nodes and clusters interchangeably in the rest of the paper. With this decomposition, nodes on the same processor can be classified into three categories:

1. d1 nodes: boundary nodes, that is nodes that share an edge with a node on another processor
2. d2 nodes, which are the neighbors of d1 nodes (and are not d1)
3. d3 nodes, which are the remaining nodes (not d1 or d2)

Figure 1 shows a simple example, where the entire grid is decomposed in two subdomains with processors $P0$ and $P1$.

We assume the matrix $A$ is distributed by rows among the processes. Each process $P$ owns the submatrix corresponding to the locally owned graph $G_P$ and also stores the edges to neighboring processes. For example, the matrix $A_0$ owned by process 0 in Fig. 1 is

$$A_0 = \begin{pmatrix} A_0^{d3} & A_0^{d3,d2} & & \\ A_0^{d2,d3} & A_0^{d2} & A_0^{d2,d1} & \\ & A_0^{d1,d2} & A_0^{d1} & A_{0,1}^{d1} \end{pmatrix}$$

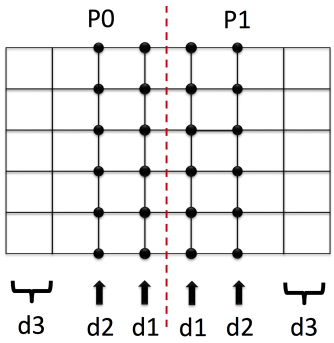



Figure 1: A two-processor example showing d1 nodes, d2 nodes and d3 nodes on each processor. The structured grid represents the quotient graph; this is the graph obtained by grouping original DOFs in matrix $A$ into clusters. The nodes are distributed across two processors $P0$ and $P1$. Boundary nodes on each processor are the d1 nodes, and d2 nodes are their neighbors. The remaining interior nodes are the d3 nodes.

where for simplicity we assumed that d3 nodes were ordered before d2 and d1 nodes. The remote coupling to process 1 is contained within the $A_{0,1}^{d1}$ block, since only d1 nodes have edges to process 1.

In our algorithm, no fill-in is introduced between nodes farther than **distance two** from each other, where the distance is measured as the length of the shortest path in the original quotient graph $G$. If we denote $A_G$ the adjacency matrix of the quotient graph, this corresponds to the sparsity of $A_G^2$ [24]. As a result, a processor can work on its d3 nodes independently from the other processors.

Since the distance between a pair of d2 nodes owned by two different processors is at least three, all processors can work on their d2 nodes in parallel. Because the distance between a d2 node on one processor $P$ and a d1 node owned by one of $P$'s neighbors can be two, a fill-in edge may exist between them. The amount of data to communicate this information is at most proportional to the number of d1 nodes.

Last, d1 nodes on different processors are coupled. We use a coloring scheme such that all processors can work on d1 nodes with the same color in parallel. The details of the coloring scheme are given in Section 3.2. Since the distance between a d1 node on processor $P$ and another d1 node on a neighbor of $P$'s neighbors can be two, a fill-in edge may exist between this pair of d1 nodes, so the two processors may need to communicate. The amount of communication is again proportional to the number of d1 nodes.

With the above observations, we have the parallel algorithm shown in Algorithm 2[3], where $\mathcal{N}_k(P)$ are the distance-k neighboring processor of processor $P$ (for example, $\mathcal{N}_2(P)$ is neighbor of neighbor).

In Algorithm 2, the communications correspond to "right-looking style" communications in a conventional sparse direct solver. There are two types of data to be communicated:

1. When we eliminate $f$ for a d1 node, say in processor $P$, the Schur complement will contain edges that connect nodes on processor $P$ with its neighbors, and also edges that connect nodes on neighboring processors. Only neighboring processors, $\mathcal{N}_1(P)$, receive data.
2. When we sparsify edges, a sparsification of a d2 node leads to communications with $\mathcal{N}_1(P)$ only, while sparsification of a d1 node involves the larger set $\mathcal{N}_2(P)$.

The ratio of communication vs. computation can be controlled by adjusting the subdomain size assigned to a processor. The communication volume is proportional to the number of d1 nodes (note in particular that d2 nodes only exchange data with d1 nodes, never with other d2 nodes), while the amount of computation is proportional to the total number of d1 nodes, d2 nodes and d3 nodes. This communication-to-computation ratio is therefore proportional to the surface-to-volume ratio of the subdomain.

The previous algorithm described a classical bulk synchronous style of programming. We also explored an asynchronous "task-based" (based on an MPI implementation) algorithm. If one considers the critical path in the algorithm, we identify 3 priorities: d1 nodes must be processed first, followed by d2 and d3. This is detailed in Algorithm 3. Essentially, we attempt to process d1 nodes; if none are ready, we attempt to process a d2 node, and finally a d3 node if all else fails. The pattern of communication can be organized as follows.

Processing of d1-node $v$, with color $i$, on processor $P$:

1. Check that all communications for $v$ from all d1 nodes in $\mathcal{N}_2(P)$, with color $< i$, are complete
2. Sparsify distance-2 edges connected to $v$
3. **Send** updated edges (distance-2 from $v$) for all d1 nodes in $\mathcal{N}_2(P)$, and d2 nodes in $\mathcal{N}_1(P)$

[3]all parallel pseudo-code is written in *single program multiple data (SPMD)* style.

**Algorithm 2** Bulk synchronous parallel algorithm

**function** HSolver_Bulk_Synchronous(local vertex clusters $\Pi$, local subgraph $G$, submatrix $A$ for $G$)
  $P$ = processor rank
  Partition $\Pi$ into d1 clusters $\Pi^{(1)}$, d2 clusters $\Pi^{(2)}$, and d3 clusters $\Pi^{(3)}$
  Parallel (distance-2) coloring of the graph of d1 clusters, $\Pi^{(1)}$
  **for** color $i = 1$ to NumColors **do**
    **for** $\pi_j \in \Pi^{(1)}$ with color $i$ **do**
      $A \leftarrow \text{LRE}(A, \Pi, \pi_j)$
    **end for**
    Communicate with $\mathcal{N}_2(P)$ processors ▷ Neighbor of neighbor processors
  **end for**
  **for** $\pi_i \in \Pi^{(2)}$ **do**
    $A \leftarrow \text{LRE}(A, \Pi, \pi_i)$
  **end for**
  Communicate with $\mathcal{N}_1(P)$ processors
  **for** $\pi_i \in \Pi^{(3)}$ **do**
    $A \leftarrow \text{LRE}(A, \Pi, \pi_i)$
  **end for**
  Create the required data for the coarse DOF matrix for rank $P$
  HSolver_Bulk_Synchronous($\Pi^C$, $G^C$, $A^C$)
**end function**

4. Eliminate fine DOFs in $v$ on processor $P$
5. **Send** updated edges (Schur complement) for all d1 nodes in $\mathcal{N}_1(P)$

Processing of d2-node $v$:

1. Check that all communications for $v$ from all d1 nodes in $\mathcal{N}_1(P)$ are complete
2. Sparsify distance-2 edges connected to $v$
3. **Send** updated edges (distance-2 from $v$) for all d1 nodes in $\mathcal{N}_1(P)$ (this data is only required at the next level of the elimination, when moving to the coarser grid)
4. Eliminate fine DOFs in $v$

Processing of d3-nodes can be straightforward eliminations that can all be done in parallel in all processors. It does not require any additional communication. Algorithm 3 presents a simplified version of this.

*3.2. Coloring of d1 nodes*

To coordinate all processors in eliminating d1 nodes in parallel, we assign colors to all d1 nodes so that all processors can process d1 nodes with the same color at the same time. To avoid any conflict, every pair of d1 nodes within distance two must have different colors if they belong to different processors, i.e.,

$$\begin{aligned} &\forall\, v, w \in V \text{ s.t. } w \in \mathcal{N}_2(v) \text{ and } v,\, w \text{ belong} \\ &\text{to different processors, then: } \mathrm{Color}(v) \neq \mathrm{Color}(w) \end{aligned} \tag{9}$$

This is *distance-2* coloring of the boundary graph. For best performance such a coloring should both minimize the number of colors and should also be load-balanced, that is the number of nodes for a given color should be roughly constant across all processors. Even with only the first objective, this problem is NP-hard but fast linear-time greedy heuristics work well in practice for graph coloring. One option is to bring the boundary graph to a single process and compute the coloring sequentially, but this is not scalable in terms of memory. We therefore compute the coloring itself in parallel[45]. Parallel software is available in the Zoltan [44] library.

For a structured grid in 2D or 3D, the maximum number of colors needed is 4. One example showing the node coloring scheme is given in Figure 2, where the graph is divided into four pieces and owned by four processors respectively.

**Algorithm 3** Parallel asynchronous algorithm

**function** HSolver_Asynchronous(local vertex clusters $\Pi$, local sub-graph $G$, submatrix $A$ for $G$)
  $P$ = processor rank
  Partition $\Pi$ into d1 clusters $\Pi^{(1)}$, d2 clusters $\Pi^{(2)}$, and d3 clusters $\Pi^{(3)}$
  Parallel (distance-2) coloring of the graph of d1 clusters, $\Pi^{(1)}$
  color $i = 0$
  **while** $\Pi^{(1)} \cup \Pi^{(2)} \cup \Pi^{(3)}$ is not empty **do**
    **if** $\pi_j \in \Pi^{(1)}$ has color $i$ and all communications (for $\pi_j$) with d1-nodes with colors $< i$ in $\mathcal{N}_2(P)$ are complete **then**
      $A \leftarrow \text{LRE}(A, \Pi, \pi_j)$
      Send data to $\mathcal{N}_2(P)$ processors ▷ Data is needed for colors $> i$
      Pop $\pi_j$ from $\Pi^{(1)}$
    **else if** all communications for $\pi_j \in \Pi^{(2)}$ with d1-nodes in $\mathcal{N}_1(P)$ are complete **then**
      $A \leftarrow \text{LRE}(A, \Pi, \pi_j)$
      Send data to d1-nodes in $\mathcal{N}_1(P)$
      ▷ Data is not needed until this function returns
      Pop $\pi_j$ from $\Pi^{(2)}$
    **else if** there is $\pi_j \in \Pi^{(3)}$ **then**
      $A \leftarrow \text{LRE}(A, \Pi, \pi_j)$
      Pop $\pi_j$ from $\Pi^{(3)}$
    **end if**
    If we are done with color $i$, increment $i$
  **end while**
  Wait for all remaining communications to complete
  Create the required data for the coarse DOF matrix of rank $P$
  HSolver_Asynchronous($\Pi^C$, $G^C$, $A^C$)
**end function**

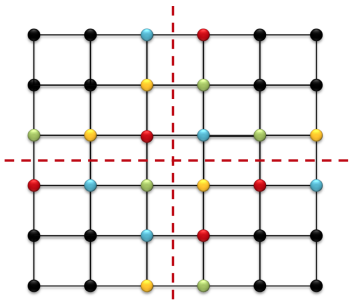


Figure 2: A four-processor example of node coloring for d1 nodes. Nodes that are at a distance less than 2 from each other cannot have the same color, unless they are on the same processor. With this coloring scheme, all four processors are able to process a subset of the d1 nodes concurrently. As seen in this example however, load imbalance is hard to reduce. Each processor has one node for 3 given colors and 2 nodes for last 4th color, leading to 2/1 imbalance during the loop over colors.

## 4. Numerical results

To demonstrate the performance of our parallel solver, we ran both sequential and parallel experiments to solve different types of PDEs with randomly initialized Dirichlet boundary condition. The discretization scheme uses the standard seven-point stencil, and we don't assume symmetry of the matrices to be solved. Our parallel solver is implemented with the *message passing interface* (MPI) [46], and we obtained the sequential results by running our code with one MPI process.

Three types of PDEs are tested. The first one is Poisson's equation:

$$-\Delta u(x) = f(x)$$

on a cube. The second one is variable-coefficient Poisson's equation:

$$-\nabla \cdot (a(x)\nabla u(x)) = f(x)$$

on a cube, where $a(x)$ is a quantized high-contrast random field generated in the following way: (1) initialize $a(x)$ randomly with a uniform distribution; (2) convolve the initial $a(x)$ with Gaussian distribution of deviation $4h$, where $h$ is the stencil spacing; and (3) set $a(x)$ to $10^2$ if it is larger than 0.5, or $10^{-2}$ otherwise. We chose a quantized high-contrast random field because these are problems known to be difficult to solve using iterative methods. Although the

performance worsens with our hierarchical solver, the algorithm still remains very efficient and is faster than some direct sparse solvers.

The third one is the Helmholtz equation:

$$(-\Delta - k^2)u(x) = f(x)$$

where $k$ is the wave number. We fix the number of DOFs per wave length and increase $k$ proportionally to the number of DOFs in each dimension. Helmholtz problems are indefinite and are very challenging for iterative solvers. Multigrid methods for example fail on this type of problems. As the frequency increases, hierarchical solvers will eventually break down because the ranks required to reach a good accuracy become too large for the method to be efficient. However, in the mid-frequency range we still observe very good performance, as illustrated by our numerical results.

In Section 4.1, we show the number of iterations and the total time (factorization + solve) corresponding to choosing different criteria for the low-rank truncation: fixing the rank $\mathcal{K}$ directly, or setting an upper bound for the low-rank truncation error $\epsilon$. In Section 4.2, we show the parallel performance of our solver, including wall-clock time, speedup and parallel efficiency. In Section 4.3, we show the fraction of time spent in the different components with computation and communication time, and analyze the performance bottleneck. In Section 4.4, we compare with SuperLU-Dist [47, 7] for the total time for solving a single right-hand-side, and the memory footprint.

Sequential experiments in Section 4.1 were run on the Vesper a large shared-memory machine at Sandia National Laboratories. It uses AMD Magnycour processors and has 128GB of main memory. All other results were run at NERSC's Edison[4], a Cray XC30 supercomputer. Each node of Edison has two 12-core Intel "Ivy Bridge" processors. Nodes are connected by a Cray Aries high-speed interconnect with Dragonfly topology.

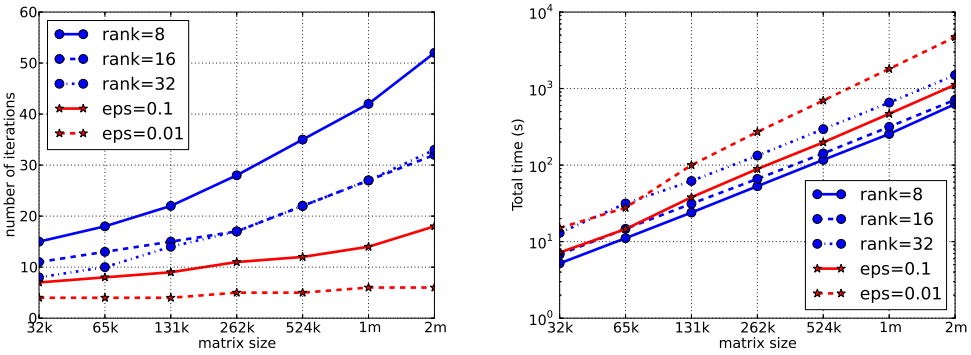
### *4.1. Sequential results*

For sequential results, we ran our code with one MPI process. We present the number of iterations and total CG/GMRES time with respect to using different low-rank truncation criteria: either fixing the rank $\mathcal{K}$ directly or fixing the error $\epsilon$. For Poisson's equation and variable-coefficient Poisson's equation, we use our solver as a preconditioner for CG [48] with a tolerance

[4]http://www.nersc.gov/users/computational-systems/edison/configuration/

of $10^{-12}$; for Helmholtz equation, we use our solver as a preconditioner for GMRES [49] with a lower tolerance of $10^{-3}$ because the Helmholtz equation is often solved with relatively low accuracy in applications such as seismic imaging. The sizes of tests matrices are generated on regular grids ranging from 32 thousand ($32^3$) to 2 million ($128^3$). For the Helmholtz equation, we fixed the resolution to 32 DOFs per wavelength and the frequencies increase from $f = 1$Hz ($32^3$ grid) to $f = 4$Hz ($128^3$ grid).

Results corresponding to three types of PDEs: Poisson's equation, variable-coefficient Poisson's equation and Helmholtz equation are shown in Figure 3, Figure 4 and Figure 5 respectively. From the results, we can see that fixing the truncation error $\epsilon$ usually leads to smaller number of iterations, but the total time can be larger compared to the cases where the rank $\mathcal{K}$ is fixed.



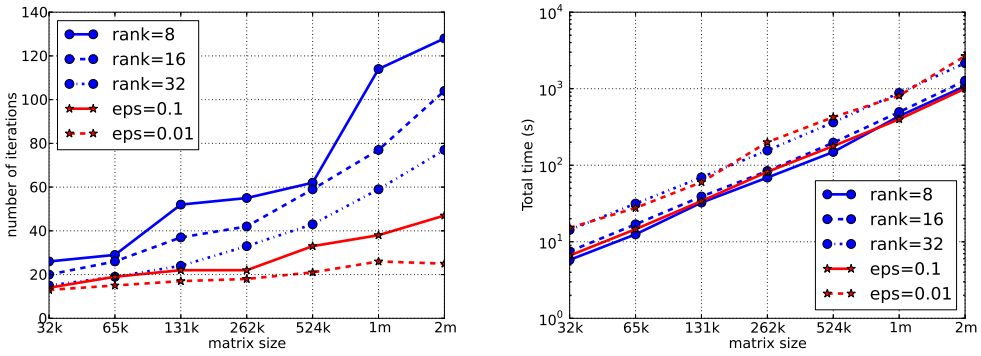

(a) Iteration number (b) Total time

Figure 3: Poisson's equation: number of iterations and total CG time for a sequential run. Different low-rank truncation criteria have been used, including fixing the rank $\mathcal{K} = 8$, 16, 32 and fixing the truncation error $\epsilon = 0.1$, 0.01.

### *4.2. Parallel results*

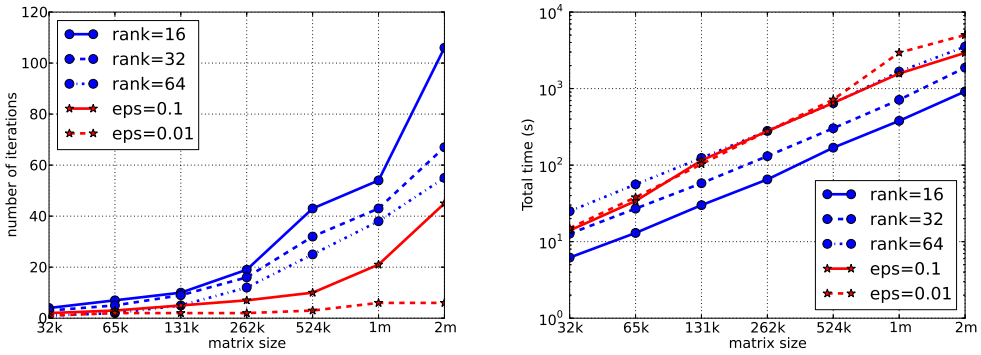
In this section, we analyze the parallel performance of our solver. We present both sequential factorization time on a single processor and corresponding parallel timing results on up to 256 processors (16 processors per node).

Factorization time for Poisson's equation with respect to the low-rank truncation criteria $\mathcal{K} = 8$ are shown in Figure 6. Figure 6(a) shows the sequential and parallel factorization time. Figure 6(b) shows the speedups




(a) Iteration number (b) Total time

Figure 4: Variable coefficient Poisson's equation: number of iterations and total CG time for a sequential run. Different low-rank truncation criteria have been used, including fixing the rank $\mathcal{K} = 8$, 16, 32 and fixing the truncation error $\epsilon = 0.1$, 0.01.




(a) Iteration number (b) Total time

Figure 5: Helmholtz equation: number of iterations and total GMRES time for a sequential run. Different low-rank truncation criteria have been used, including fixing the rank $\mathcal{K} = 16$, 32, 64 and fixing the truncation error $\epsilon = 0.1$, 0.01.

for the factorization phase in our parallel solver. For a fixed problem size, we measure the efficiency of the parallel factorization when increasing the number of processors. Define $T(N, P)$ as the wall-clock time to factorize a matrix of size $N$ on $P$ processors. The parallel speedup $S(N, P)$ is:

$$S(N, P) = \frac{T(N, 1)}{T(N, P)}$$

and the efficiency is:

$$E_s = \frac{S(N, P)}{P} \tag{10}$$

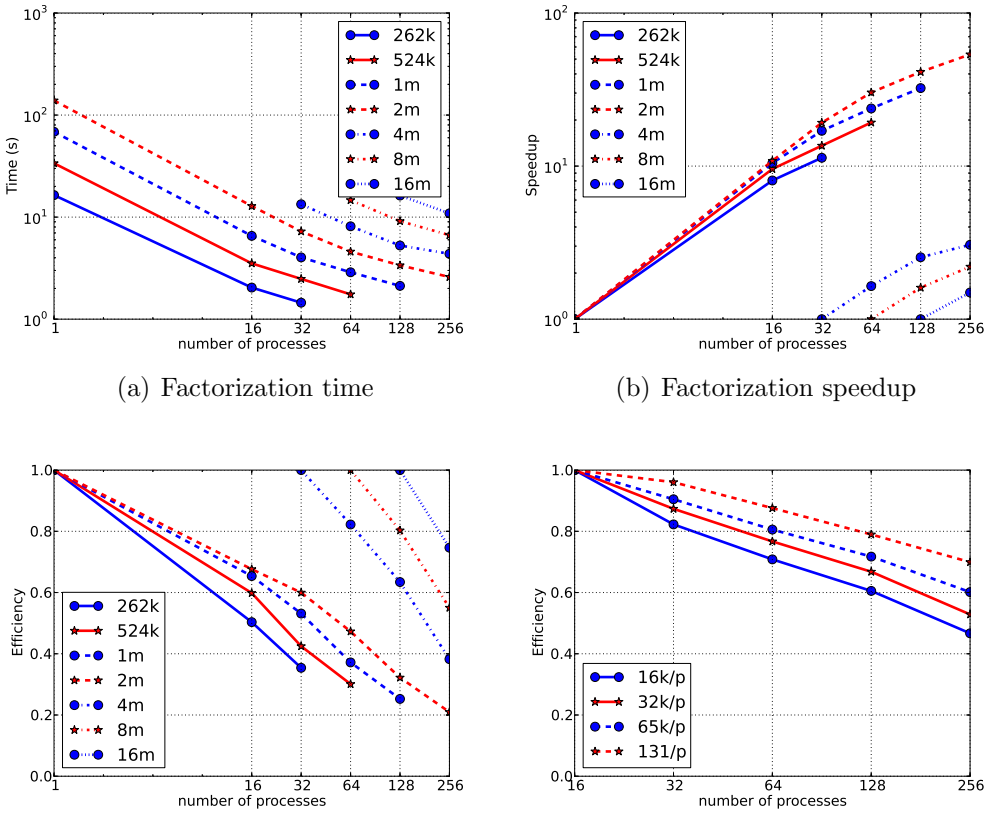
The other way to measure parallel performance is to fix the problem size per processor, i.e., the total problem size increases proportionally as the number of processors increases, i.e., the problem size on every processor is fixed. The efficiency is defined as:

$$E_w = \frac{T(N, 1)}{T(N\,P, P)} \tag{11}$$

We want to clarify that a constant efficiency $E_w$ with a fixed problem size per processor should not be expected, even if our parallel algorithm was implemented perfectly with perfect underlying hardware (no network saturation, fixed diameter). In our case, maintaining an efficiency of 1 implies that the number of processors scales at most as fast as $O(N/\log(N))$, where $N$ is the problem size. For example, the leaf size should scale as $\log(N)$, which is not the case here. In our weak scaling benchmarks, we keep a fixed leaf size, which is implies that the efficiency is in $O(1/\ln N)$ as $N \to \infty$. This type of result is similar to the behavior observed when computing a reduction in parallel using a tree structure.

The factorization time for variable-coefficient Poisson's equation with respect to the low-rank truncation criteria $\mathcal{K} = 16$ is shown in Figure 7. Figure 7(a) shows the sequential and parallel factorization time. Figure 7(b) shows the speedups of factorization in our parallel solver. Two kinds of efficiency corresponding to fixed total problem size and fixed problem size per processor, as defined in Eqns. (10) and (11), are shown.

The factorization time for Helmholtz equation with respect to the low-rank truncation criteria $\mathcal{K} = 32$ is shown in Figure 8. Figure 8(a) shows the sequential and parallel factorization time. Figure 8(b) shows the speedups of factorization in our parallel solver. Two kinds of efficiency corresponding to fixed total problem size and fixed problem size per processor, as defined



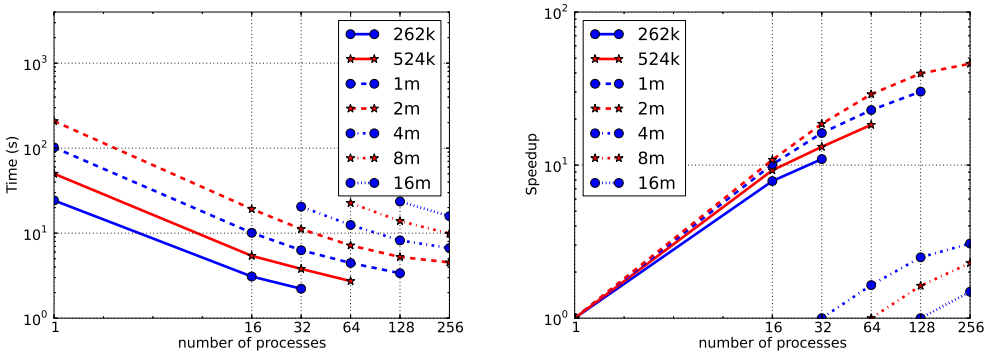

(a) Factorization time

(b) Factorization speedup

(c) Fixed total problem size

(d) Fixed problem size per processor

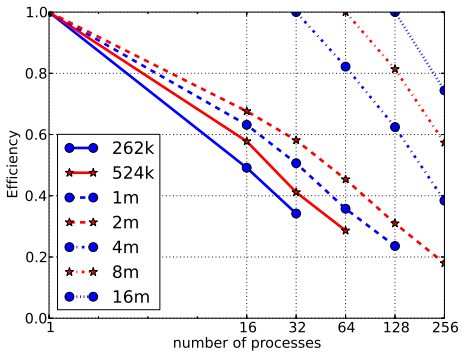
Figure 6: Factorization time for Poisson's equation with respect to the low-rank truncation criteria $\mathcal{K} = 8$. Figure 6(a) shows the factorization time for different problem sizes on different number of processors. We used 16 processors per node. Figure 6(b) shows speedups on multiple processors. Figure 6(c) and Figure 6(d) show two kinds of efficiency corresponding to fixed total problem size and fixed problem size per processor as defined in Eqns. (10) and (11).



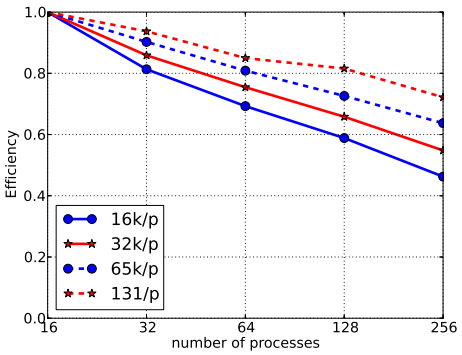

(a) Time




(b) Speedup



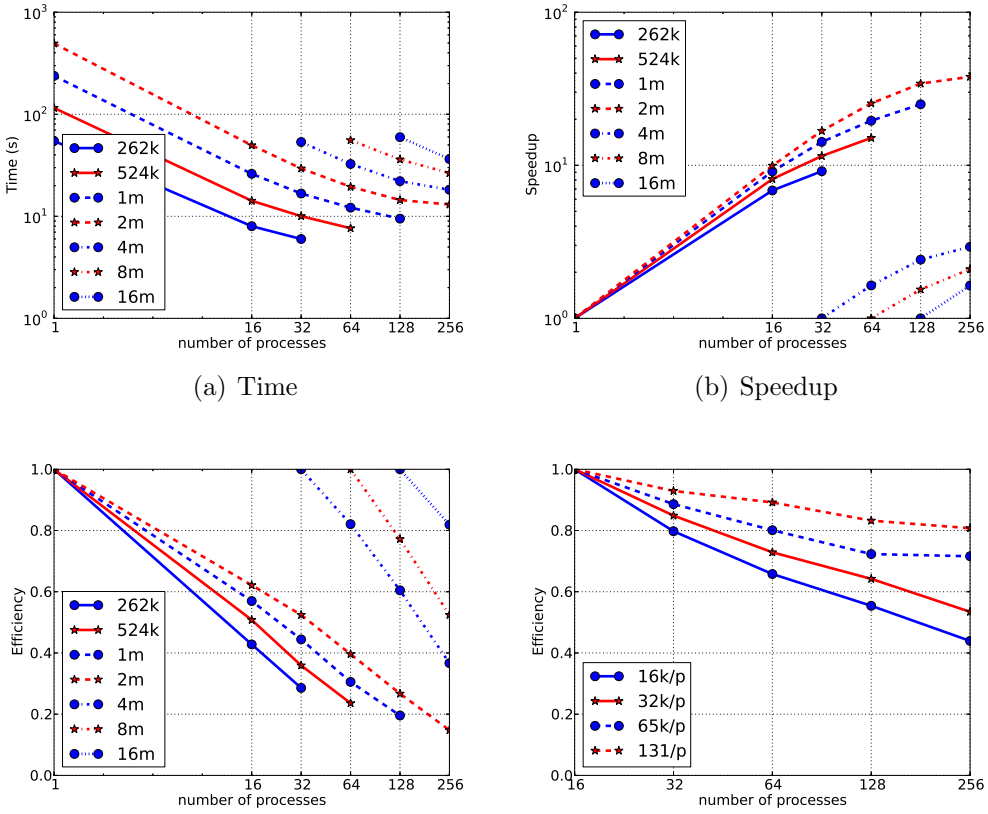

(c) Fixed total problem size




(d) Fixed problem size per processor

Figure 7: Factorization time for variable-coefficient Poisson's equation equation with respect to the low-rank truncation criteria $\mathcal{K} = 16$. Figure 7(a) shows the factorization time for different problem sizes on different number of processors. We used 16 processors per node. Figure 7(b) shows speedups on multiple processors. Figure 7(c) and Figure 7(d) show two kinds of efficiency corresponding to fixed total problem size and fixed problem size per processor as defined in Eqns. (10) and (11).

in Eqns. (10) and (11), are shown. Note that, in all three cases, we stop the scaling experiments for fixed total problem size when number of unkowns are fewer than 8K. The communication costs dominate at that scale.




(a) Time

(b) Speedup

(c) Fixed total problem size

(d) Fixed problem size per processor

Figure 8: Factorization time for Helmholtz equation equation with respect to the low-rank truncation criteria $\mathcal{K} = 32$. Figure 8(a) shows the factorization time for different problem sizes on different number of processors. We used 16 processors per node. Figure 8(b) shows speedups on multiple processors. Figure 8(c) and Figure 8(d) show two kinds of efficiency corresponding to fixed total problem size and fixed problem size per processor as defined in Eqns. (10) and (11).

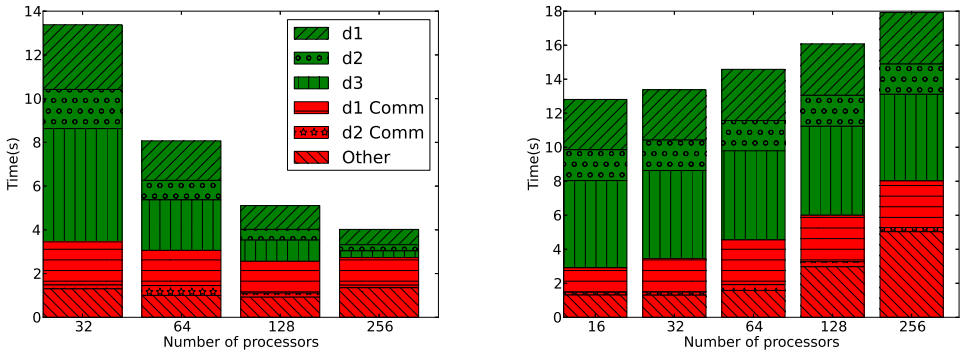
### *4.3. Analysis of the parallel running time*

To further understand the performance and bottleneck of our solver, we show the time fractions of all components in our algorithm: computation

time for d1 nodes, d2 nodes and d3 nodes, communication time for d1 nodes and d2 nodes and others including coloring, graph coarsening and so on.

Results for three PDEs: Poisson's equation ($\mathcal{K} = 8$), variable-coefficient Poisson's equation ($\mathcal{K} = 16$) and Helmholtz equation ($\mathcal{K} = 32$) are shown in Figure 9, Figure 10 and Figure 11 respectively. For a fixed total problem size, the computation time for d1 nodes, d2 nodes and d3 nodes almost halved when the number of processors doubled. But the communication time and time spent in other things remain almost constant. For fixed problem size per processor, the computation time for d1 nodes, d2 nodes and d3 nodes remain almost constant when the number of processors increase. But the communication time and time spent in other things grow slowly.

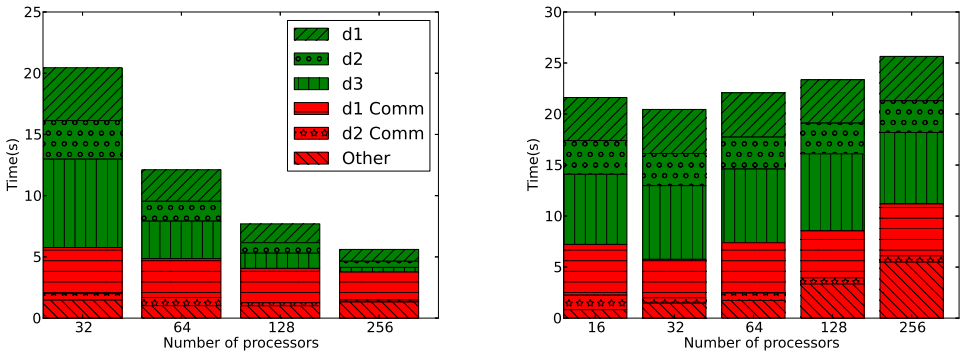
Two bottlenecks exist in our current implementation. The first one is our implementation of communication for d1 nodes and d2 nodes. The second is time spent in calling the coloring subroutine from the Zoltan [44] package. The coloring only needs to be computed once (symbolic phase), so this cost could be amortized when solving a sequence of linear systems. Note that this coloring problem is atypical of coloring a graph as we color an interface graph resulting in high communication costs for the coloring implementation.




(a) Fixed four-million problem size

(b) Fixed 131-thousand problem size per processor

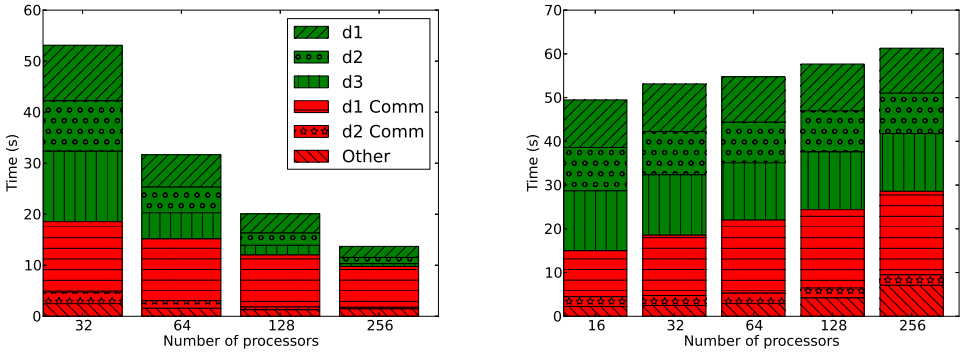
Figure 9: Breakup of parallel factorization time for Poisson's equation ($\mathcal{K} = 8$). "d1", "d2" and "d3" stand for computation. "d1 Comm" and "d2 Comm" stand for communication. "Other" includes the node coloring, and the graph coarsening (a fast step in principle but not implemented in parallel in our code). Node coloring is done using the Zoltan [44] library.




(a) Fixed four-million problem size

(b) Fixed 131-thousand problem size per processor

Figure 10: Breakup of parallel factorization time for variable-coefficient Poisson's equation ($\mathcal{K} = 16$). "d1", "d2" and "d3" stand for computation. "d1 Comm" and "d2 Comm" stand for communication. "Other" includes the node coloring, and the graph coarsening (a fast step in principle but not implemented in parallel in our code). Node coloring is done using the Zoltan [44] library.




(a) Fixed four-million problem size

(b) Fixed 131-thousand problem size per processor

Figure 11: Breakup of parallel factorization time for Helmholtz equation ($\mathcal{K} = 32$). "d1", "d2" and "d3" stand for computation. "d1 Comm" and "d2 Comm" stand for communication. "Other" includes the node coloring, and the graph coarsening (a fast step in principle but not implemented in parallel in our code). Node coloring is done using the Zoltan [44] library.

*4.4. Comparison with SuperLU-Dist*

We show some results comparing our parallel performance to SuperLU-Dist [47, 7], a state-of-the-art parallel sparse direct solver. We show comparisons of the total time for solving a single right-hand-size for three PDEs on 16 processors. The subroutine `pdgssvx()` in SuperLU-Dist was called with the default options. Timing results and corresponding memory footprint are shown in Figure 12. For Poisson's equation and variable-coefficient Poisson's equation, our solver was used as a preconditioner ($\mathcal{K} = 8$, 16 respectively) for CG with a tolerance of $10^{-12}$. For the Helmholtz equation, our solver ($\mathcal{K} = 32$) was used as a preconditioner for GMRES with a tolerance of $10^{-3}$ . (In seismic imaging, the Helmholtz equation is often solved with relatively low accuracy.)

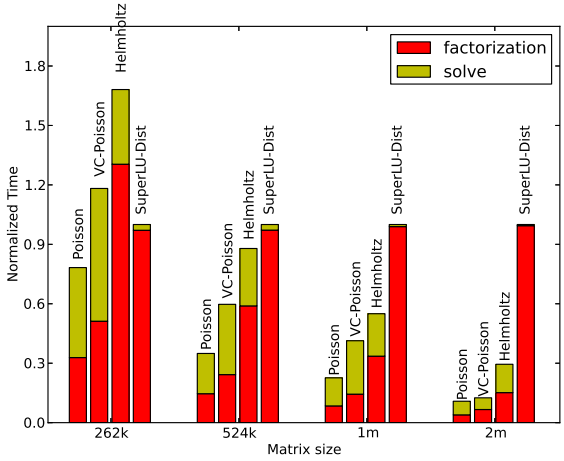
The relative accuracy $\|x - x_{\text{true}}\|/\|x\|$ of the solution from SuperLU-Dist was at the order of $10^{-14}$ for Poisson equation, $10^{-11}$–$10^{-10}$ for variable-coefficient Poisson equation, $10^{-9}$–$10^{-8}$ for Helmholtz equation.

From Figure 12, we can see that for a small problem size of 262k ($128^3$), our solver was a little slower and used more memory than SuperLU-Dist for Helmholtz equation because of using a relatively large rank. But the total time and memory cost of our solver scales linearly as problem size increases. Therefore, our solver becomes faster and more memory-efficient than SuperLU-Dist for solving a two-million ($256^3$) problem.

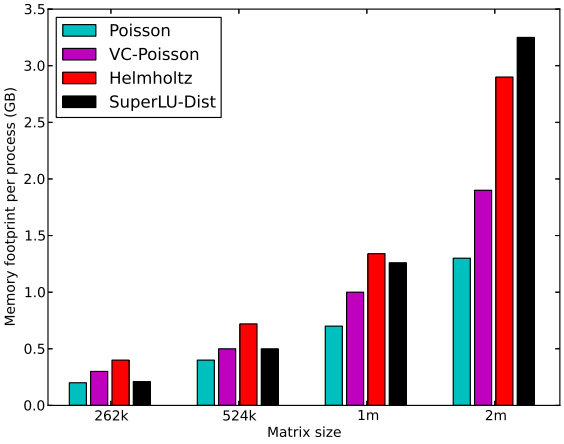
## 5. Conclusions and Future Work

We have presented the first parallel algorithm and implementation of the LoRaSp hierarchical solver. It can be used either as a solver or as a preconditioner, depending on the accuracy in the factorization phase. There are several attractive features for our parallel algorithm: the factorization time, the solve time and the memory usage all scale nearly linearly. When our solver is used as a preconditioner for an iterative solver, the iteration number stays almost constant. This nearly optimal scaling leads to faster running times on large matrices which beats many existing sparse direct solvers.

To solve really large problems that cannot be stored on a single machine, distributed memory parallel computing has to be used. We have shown our solver achieves good speed-ups up to 256 processes (cores). The parallel algorithm requires an exchange of only boundary information between neighboring processors, the amount of which is typically an order of magnitude




(a) Total time




(b) Memory footprint

Figure 12: Comparison with SuperLU-Dist of the total time and memory footprint for solving a single right-hand-side on 16 processors. Every group of time is normalized by the total time of SuperLU-Dist (which was the same for all three test problems: Poisson, variable-coefficient Poisson (VC-Poisson) and Helmholtz). The memory cost of SuperLU-Dist was also the same for all three test problems.

less than the amount of computation to be done. It uses a great deal of dense linear algebra, giving opportunity for an additional layer of parallelism using modern many-core architectures.

We have presented the sequential algorithm and the parallel algorithm for SPD matrices, but they can be extended to general matrices. For example, the sequential algorithm works for unsymmetric matrices [24] and dense matrices [50].

There are several directions for future work:

- Other low-rank approximations than SVD may be more efficient.
- Our graph coloring of the boundary is conservative and other strategies may reduce the number of colors. One could also color the process graph instead of the boundary vertices, which gives more coarse-grained communication but worse load-balance, so it should be studied further.
- One could study various partitioning strategies: geometric, graph, and hypergraph partitioning.
- We plan an MPI+X implementation that exploits thread parallelism on the node for dense linear algebra. We believe such an approach may improve parallel scalability on large number of nodes (cores).
- A more asynchronous MPI implementation of the code would improve performance by removing artificial synchronization points. This would lead to lower load imbalance and less time spent waiting on communications.

## Acknowledgments

Chao Chen, Hadi Pouransari, and Eric Darve were partially supported by the Department of Energy National Nuclear Security Administration under Award Number DE-NA0002373-1; Sivasankaran Rajamanickam and Erik Boman were partially supported by Sandia's LDRD program, the US DOE Office of Science, and the National Nuclear Security Administration's ASC program.

[1] Y. Chen, T. A. Davis, W. W. Hager, S. Rajamanickam, Algorithm 887: CHOLMOD, supernodal sparse cholesky factorization and update/downdate, ACM Transactions on Mathematical Software (TOMS) 35 (2008) 22.

[2] T. A. Davis, Algorithm 832: UMFPACK V4. 3—an unsymmetric-pattern multifrontal method, ACM Transactions on Mathematical Software (TOMS) 30 (2004) 196–199.

[3] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, J. W. H. Liu, A supernodal approach to sparse partial pivoting, SIAM J. Matrix Analysis and Applications 20 (1999) 720–755.

[4] A. Kuzmin, M. Luisier, O. Schenk, Fast methods for computing selected elements of the greens function in massively parallel nanoelectronic device simulations, in: F. Wolf, B. Mohr, D. Mey (Eds.), Euro-Par 2013 Parallel Processing, volume 8097 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2013, pp. 533–544.

[5] J. W. Demmel, J. R. Gilbert, X. S. Li, An asynchronous parallel supernodal algorithm for sparse Gaussian elimination, SIAM J. Matrix Anal. Appl. 20 (1999) 915–952.

[6] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, J. Koster, MUMPS: A general purpose distributed memory sparse solver, in: Applied Parallel Computing. New Paradigms for HPC in Industry and Academia, Springer, 2001, pp. 121–130.

[7] X. S. Li, J. W. Demmel, SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems, ACM Transactions on Mathematical Software (TOMS) 29 (2003) 110–140.

[8] I. S. Duff, A. M. Erisman, J. K. Reid, Direct Methods for Sparse Matrices, Clarendon press Oxford, 1986.

[9] A. George, Nested dissection of a regular finite element mesh, SIAM Journal on Numerical Analysis 10 (1973) 345–363.

[10] P. R. Amestoy, T. A. Davis, I. S. Duff, An approximate minimum degree ordering algorithm, SIAM Journal on Matrix Analysis and Applications 17 (1996) 886–905.

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

[12] Y. Saad, Iterative Methods for Sparse Linear Systems, Siam, 2003.

[13] W. Hackbusch, Multi-grid Methods and Applications, volume 4, Springer Science & Business Media, 2013.

[14] J. Xu, Iterative methods by space decomposition and subspace correction, SIAM review 34 (1992) 581–613.

[15] J. Mandel, Multigrid convergence for nonsymmetric, indefinite variational problems and one smoothing step, Applied Mathematics and Computation 19 (1986) 201–216.

[16] J. H. Bramble, J. E. Pasciak, J. Xu, The analysis of multigrid algorithms for nonsymmetric and indefinite elliptic problems, Mathematics of Computation 51 (1988) 389–414.

[17] J. H. Bramble, D. Y. Kwak, J. E. Pasciak, Uniform convergence of multigrid v-cycle iterations for indefinite and nonsymmetric problems, SIAM journal on numerical analysis 31 (1994) 1746–1763.

[18] P. Lin, M. T. Bettencourt, S. P. Domino, T. C. Fisher, M. Hoemmen, J. J. Hu, E. T. Phipps, A. Prokopenko, S. Rajamanickam, C. Siefert, S. R. Kennon, Towards extreme-scale simulations with second-generation Trilinos, Parallel Processing Letters (2014).

[19] P. Lin, M. Bettencourt, S. Domino, T. Fisher, M. Hoemmen, J. Hu, E. Phipps, A. Prokopenko, S. Rajamanickam, C. Siefert, et al., Towards extreme-scale simulations with next-generation Trilinos: a low Mach fluid application case study, in: Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International, IEEE, pp. 1485–1494.

[20] J. Xia, S. Chandrasekaran, M. Gu, X. S. Li, Superfast multifrontal method for large structured linear systems of equations, SIAM Journal on Matrix Analysis and Applications 31 (2009) 1382–1411.

[21] K. L. Ho, L. Ying, Hierarchical interpolative factorization for elliptic operators: Differential equations, Communications on Pure and Applied Mathematics (2015).

[22] P. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L'Excellent, C. Weisbecker, Improving multifrontal methods by means of block low-rank representations, SIAM Journal on Scientific Computing 37 (2015) A1451–A1474.

[23] A. Aminfar, S. Ambikasaran, E. Darve, A fast block low-rank dense solver with applications to finite-element matrices, Journal of Computational Physics 304 (2016) 170–188.

[24] H. Pouransari, P. Coulier, E. Darve, Fast hierarchical solvers for sparse matrices, arXiv preprint arXiv:1510.07363 (2015).

[25] W. Hackbusch, A sparse matrix arithmetic based on $\mathcal{H}$-matrices. Part i: introduction to $\mathcal{H}$-matrices, Computing 62 (1999) 89–108.

[26] P. Ghysels, X. S. Li, F.-H. Rouet, S. Williams, A. Napov, An efficient multicore implementation of a novel HSS-structured multifrontal solver using randomized sampling, arXiv preprint arXiv:1502.07405 (2015).

[27] S. Wang, X. S. Li, F.-H. Rouet, J. Xia, M. V. De Hoop, A parallel geometric multifrontal solver using hierarchically semiseparable structure, ACM Transactions on Mathematical Software (TOMS) 42 (2016) 21:1–21:21.

[28] Y. Li, L. Ying, Distributed-memory hierarchical interpolative factorization, arXiv preprint arXiv:1607.00346 (2016).

[29] F.-H. Rouet, X. S. Li, P. Ghysels, A. Napov, A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization, arXiv preprint arXiv:1503.05464 (2015).

[30] Y. Saad, ILUT: a dual threshold incomplete LU factorization, Numerical linear algebra with applications 1 (1994) 387–402.

[31] A. Brandt, Algebraic multigrid theory: The symmetric case, Applied mathematics and computation 19 (1986) 23–56.

[32] K. Stüben, A review of algebraic multigrid, Journal of Computational and Applied Mathematics 128 (2001) 281–309.

[33] G. H. Golub, C. Reinsch, Singular value decomposition and least squares solutions, Numerische mathematik 14 (1970) 403–420.

[34] C.-T. Pan, On the existence and computation of rank-revealing LU factorizations, Linear Algebra and its Applications 316 (2000) 199–222.

[35] L. Miranian, M. Gu, Strong rank revealing LU factorizations, Linear algebra and its applications 367 (2003) 1–16.

[36] T. F. Chan, Rank revealing QR factorizations, Linear algebra and its applications 88 (1987) 67–82.

[37] M. Gu, S. C. Eisenstat, Efficient algorithms for computing a strong rank-revealing QR factorization, SIAM Journal on Scientific Computing 17 (1996) 848–869.

[38] K. Zhao, M. N. Vouvakis, J.-F. Lee, The adaptive cross approximation algorithm for accelerated method of moments computations of emc problems, IEEE Transactions on Electromagnetic Compatibility 47 (2005) 763–773.

[39] J. Xia, S. Chandrasekaran, M. Gu, X. S. Li, Fast algorithms for hierarchically semiseparable matrices, Numerical Linear Algebra with Applications 17 (2010) 953–976.

[40] W. Hackbusch, S. Börm, Data-sparse approximation by adaptive $\mathcal{H}^2$-matrices, Computing 69 (2002) 1–35.

[41] W. Hackbusch, $\mathcal{H}^2$-matrices, in: Hierarchical Matrices: Algorithms and Analysis, Springer, 2015, pp. 203–240.

[42] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM Journal on scientific Computing 20 (1998) 359–392.

[43] C. Chevalier, F. Pellegrini, Pt-scotch: A tool for efficient parallel graph ordering, Parallel computing 34 (2008) 318–331.

[44] E. G. Boman, U. V. Catalyurek, C. Chevalier, K. D. Devine, The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and coloring, Scientific Programming 20 (2012) 129–150.

[45] D. Bozdağ, U. Çatalyürek, A. H. Gebremedhin, F. Manne, E. G. Boman, F. Özgüner, Distributed-memory parallel algorithms for distance-2 coloring and related problems in derivative computation, SIAM J. Sci. Comput. 32 (2010) 2418–2446.

[46] W. Gropp, E. Lusk, N. Doss, A. Skjellum, A high-performance, portable implementation of the MPI message passing interface standard, Parallel computing 22 (1996) 789–828.

[47] X. Li, J. Demmel, J. Gilbert, L. Grigori, M. Shao, I. Yamazaki, SuperLU Users' Guide, Technical Report LBNL-44289, Lawrence Berkeley National Laboratory, 1999. `http://crd.lbl.gov/~xiaoye/SuperLU` Last update: August 2011.

[48] M. R. Hestenes, E. Stiefel, Methods of Conjugate Gradients for Solving Linear Systems, volume 49, NBS, 1952.

[49] Y. Saad, M. H. Schultz, GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems, SIAM Journal on scientific and statistical computing 7 (1986) 856–869.

[50] P. Coulier, H. Pouransari, E. Darve, The inverse fast multipole method: Using a fast approximate direct solver as a preconditioner for dense linear systems, arXiv preprint arXiv:1508.01835 (2015).