SIAM J. SCI. COMPUT.
Vol. 39, No. 4, pp. C292–C318

A DISTRIBUTED-MEMORY RANDOMIZED STRUCTURED MULTIFRONTAL METHOD FOR SPARSE DIRECT SOLUTIONS*

ZIXING XIN†, JIANLIN XIA‡, MAARTEN V. DE HOOP§, STEPHEN CAULEY¶, AND VENKATARAMANAN BALAKRISHNAN‖

Abstract. We design a distributed-memory randomized structured multifrontal solver for large sparse matrices. Two layers of hierarchical tree parallelism are used. A sequence of innovative parallel methods are developed for randomized structured frontal matrix operations, structured update matrix computation, skinny extend-add operation, selected entry extraction from structured matrices, etc. Several strategies are proposed to reuse computations and reduce communications. Unlike an earlier parallel structured multifrontal method that still involves large dense intermediate matrices, our parallel solver performs the major operations in terms of skinny matrices and fully structured forms. It thus significantly enhances the efficiency and scalability. Systematic communication cost analysis shows that the numbers of words are reduced by factors of about $O(\sqrt{n}/r)$ in two dimensions and about $O(n^{2/3}/r)$ in three dimensions, where n is the matrix size and r is an off-diagonal numerical rank bound of the intermediate frontal matrices. The efficiency and parallel performance are demonstrated with the solution of some large discretized PDEs in two and three dimensions. Nice scalability and significant savings in the cost and memory can be observed from the weak and strong scaling tests, especially for some 3D problems discretized on unstructured meshes.

Key words. distributed memory, tree parallelism, fast direct solver, randomized multifrontal method, rank structure, skinny matrices

AMS subject classifications. 15A23, 65F05, 65F30, 65Y05, 65Y20

DOI. 10.1137/16M1079221

1. Introduction.

The solution of large sparse linear systems plays a critically important role in modern numerical computations and simulations. Generally, there are two types of sparse solvers, iterative ones and direct ones. Direct solvers are robust and suitable for solving linear systems with multiple right-hand sides but usually take more memory to store the factors that are often much denser. The focus of this work is on a parallel sparse direct solution that uses low-rank approximations to reduce floating-point operations and decrease memory usage.

An important type of direct solvers is the multifrontal method proposed in [9]. Multifrontal methods perform the factorization of a sparse matrix via a sequence of smaller dense factorizations organized following a tree called an elimination or assembly tree. This results in nice data locality and great potential for parallelization. Matrix reordering techniques such as nested dissection [10] are commonly used before the factorization to reduce fill-in.

*Submitted to the journal's Software and High-Performance Computing section June 9, 2016; accepted for publication (in revised form) March 13, 2017; published electronically August 17, 2017.
http://www.siam.org/journals/sisc/39-4/M107922.html

Funding: The work of the second author was supported in part by NSF CAREER Award DMS-1255416.

†Department of Mathematics, Purdue University, West Lafayette, IN 47907 (zxin@purdue.edu).

‡Department of Mathematics and Department of Computer Science, Purdue University, West Lafayette, IN 47907 (xiaj@purdue.edu).

§Department of Computational and Applied Mathematics, Rice University, Houston, TX 77005 (mdehoop@rice.edu).

¶Athinoula A. Martinos Center for Biomedical Imaging, Department of Radiology, Massachusetts General Hospital, Harvard University, Charlestown, MA 02129 (stcauley@nmr.mgh.harvard.edu).

‖School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907 (ragu@ecn.purdue.edu).

In recent years, structured multifrontal methods [33, 30, 27, 2] have been developed to utilize a certain low-rank property of the intermediate dense matrices that arise in the factorization of some discretized PDEs. This low-rank property has been found in many problems, such as the ones arising from the discretization of elliptic PDEs. Hierarchical structured matrices like $\mathcal{H}/\mathcal{H}^2$-matrices [5, 15, 17] and hierarchically semiseparable (HSS) matrices [6, 34] are often used to take advantage of the low-rank property. Other rank structured representations are also used in multifrontal and similar solvers [2, 14].

To enhance the performance and flexibility of the structured matrix operations, some recent work integrates randomization into structured multifrontal methods [31]. Randomized sampling enables the conversion of a large rank-revealing problem into a much smaller one after matrix-vector multiplications [18, 21]. This often greatly accelerates the construction of structured forms and also makes the processing of the data much simpler [24, 35].

In this work, we are interested in the parallelization of the randomized structured multifrontal (RSMF) method in [31] for the factorization of large-scale sparse matrices. The work is a systematic presentation of our developments in the technical reports [36, 37, 38]. The parallelism is essential to speed up the algorithms and make the algorithms available to large problems by the exploitation of more processes and memory. Earlier work in [28] gives a parallel multifrontal solver based on a simplified structured multifrontal method in [30] that involves dense intermediate matrices. Some dense matrices called frontal matrices are approximated by HSS forms and then factorized. The resulting Schur complements called update matrices are still dense and are used to assemble later frontal matrices. The use of dense update matrices is due to the lack of an effective structured data assembly strategy. All these dense matrices tend to be a memory bottleneck if their sizes are large. Moreover, the dense forms make some major operations more costly than necessary, including the structured approximations of the frontal matrices, the computation of the update matrices, and the assembly of the frontal matrices. In addition, the parallel solver in [28] relies on the geometry of the mesh, which is required to be a regular mesh. This limits the applicability of that solver.

Here, we seek to build a systematic framework for parallelizing the RSMF method in [31] using distributed memory. The randomized approach avoids the use of dense frontal and update matrices and also makes the parallelization significantly more convenient and efficient. We also allow more general matrix types. Our main results include the following:

- We design all the mechanisms for a distributed-memory parallel implementation of the RSMF method. A static mapping parallel model is designed to handle two layers of parallelism (one for the assembly tree and another for the structured frontal/update matrices) as well as new parallel structured operations. Novel parallel algorithms for all the major steps are developed, such as the intermediate structured matrix operations, structured submatrix extraction, skinny matrix assembly, and information propagation.
- A sequence of strategies is given to reduce the communication costs, such as the combination of process grids for different tree parallelism, a compact storage of HSS forms, the assembly and passing of data between processes in compact forms, heavy data reuse, the collection of data for BLAS3 operations, and the storage of a small amount of the same data in multiple processes.

- The parallel structured multifrontal method is not restricted to a specific mesh geometry. Graph partitioning techniques and packages are used to efficiently produce nested dissection ordering of a general mesh. The method can be applied to much more general sparse matrices where the low-rank property exists. For problems without the low-rank property, we can also use the method as an effective preconditioner.
- Our nested dissection ordering and symbolic factorization well respect the local geometric connectivity and thus naturally preserve the inherent low-rank property in the intermediate dense matrices.
- As compared with the parallel structured multifrontal method in [28],
  - our parallel method can be applied to sparse matrices corresponding to more general graphs instead of only regular meshes;
  - we do not need to store large dense frontal/update matrices or convert between dense and HSS matrices, and instead, the HSS constructions and factorizations are fully structured;
  - the parallel data assembly is performed through skinny matrix-vector products instead of large dense matrices.

We give a detailed analysis of the communication cost (which is also missing for the method in [28]). The communication cost of our solver includes $O(\sqrt{P}) + O(r\log^3 P)$ messages and $O(\frac{r\sqrt{n}}{\hat{P}}) + O(\frac{r^2\log^3 P}{\sqrt{\hat{P}}})$ words for two-dimensional (2D) problems or $O(\frac{rn^{\frac{2}{3}}}{\hat{P}}) + O(\frac{r^2\log^3 P}{\sqrt{\hat{P}}})$ words for 3D problems, where n is the matrix size, P is the total number of processes, r is the maximum off-diagonal numerical rank of the frontal matrices, and $\hat{P}$ is the minimum size of all the process grids. There is a significant reduction in the communication cost as compared with the solver in [28]. The numbers of words of the new solver are smaller by factors about $O(\frac{\sqrt{n}}{r})$ and $O(\frac{n^{\frac{2}{3}}}{r})$ for two and three dimensions, respectively.

Extensive numerical tests in terms of some discretized PDEs in two and three dimensions are used to show the parallel performance. Both weak and strong scaling tests are done. The nice scalability and significant savings in the cost and memory can be clearly observed, especially for some 3D PDEs discretized on unstructured meshes. For example, for a discretized 3D Helmholtz equation with $n \approx 9.7 \times 10^6$, the new solver takes about 1/4 of the flops of the standard multifrontal solver and about 1/2 of the storage for the factors. With up to 256 processes, when the number of processes doubles, the parallel factorization time reduces by a factor about $1.7 \sim 1.8$. The new solver works well for larger n even if the standard parallel multifrontal solver runs out of memory.

The remaining sections are organized as follows. We discuss nested dissection, parallel symbolic factorization, and graph connectivity preservation in section 2. Section 3 reviews the basic framework of the RSMF method. Section 4 details our parallel randomized structured multifrontal (PRSMF) method. Section 5 shows the analysis of the communication cost. The numerical results are shown in section 6. Finally, we draw our conclusions in section 7.

For convenience, we first explain the basic notation to be used throughout the paper.

- For a matrix A and an index set $\mathbf{I}$, $A|_{\mathbf{I}}$ denotes the submatrix of A formed by the rows with the row index set $\mathbf{I}$.
- For a matrix A and index sets $\mathbf{I}, \mathbf{J}$, $A|_{\mathbf{I}\times\mathbf{J}}$ denotes the submatrix of A with row index set $\mathbf{I}$ and column index set $\mathbf{J}$.

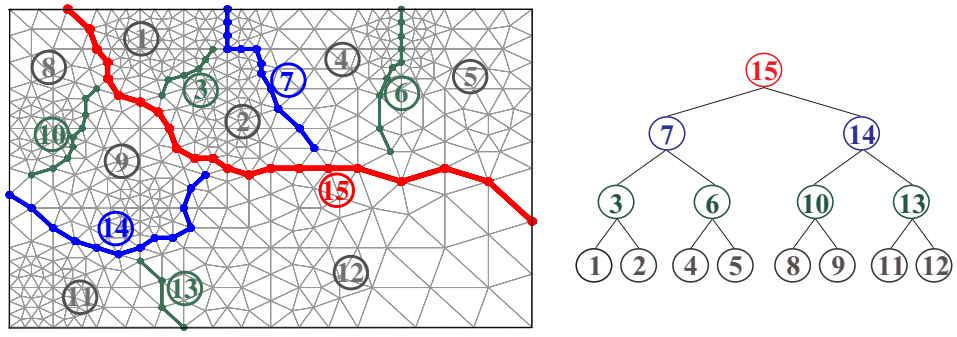
- If T is a postordered binary tree with nodes $i = 1, 2, \ldots$, we use $\mathrm{sib}(i)$ and $\mathrm{par}(i)$ to denote the sibling and parent of a node i, respectively, and use $\mathrm{root}(T)$ to denote the root node of T.
- Bold and calligraphic letters are usually used for symbols related to the multifrontal process.

2. Nested dissection for general graphs, parallel symbolic factorization, and connectivity preservation. For an $n \times n$ sparse matrix A, let $\mathcal{G}(A)$ be its corresponding adjacency graph, which has n vertices and has an edge connecting vertices i and j if $A_{ij} \neq 0$. $\mathcal{G}$ is a directed graph if A is nonsymmetric. In this case, we consider the adjacency graph $\mathcal{G}(A + A^T)$ instead of $\mathcal{G}(A)$ (see, e.g., [3, 20]). In the factorization of A, vertices in the adjacency graph are eliminated, and their neighbors are connected, which yields fill-in in the factors.

To reduce fill-in, reordering techniques are usually applied to $\mathcal{G}$ before the factorization. Nested dissection [10] is one of the most commonly used reordering techniques. The basic idea is to divide a graph recursively into disjoint parts using separators which are small subsets of vertices. Then upper level separators are ordered after lower level ones. Treating each separator as a supernode, we can construct a tree called *assembly tree* $\mathcal{T}$ to organize the later factorization. This provides a natural mechanism of parallelism. See Figure 1. For convenience, the nodes and the corresponding separators are labeled as $\mathbf{i} = 1, 2, \ldots$ in postorder. Denote the set of vertices within separator/node $\mathbf{i}$ by $\mathbf{s_i}$. Suppose the root node is at level $\mathbf{l} = 0$.

We apply nested dissection to $\mathcal{G}$ and, in the mean time, collect relevant connectivity information for the sake of later structured matrix operations. During the later elimination stage (via the multifrontal method which will be reviewed in the next section), it will require the reordering of the vertices within a separator together with those within some neighbor separators. Here, roughly speaking, a separator $\mathbf{j}$ is a *neighbor* of a separator $\mathbf{i}$ if $\mathbf{j}$ is an ancestor of $\mathbf{i}$ and the L factor in the LU factorization of A has nonzero entries in $L|_{\mathbf{s_j} \times \mathbf{s_i}}$. We try to keep nearby vertices close to each other after reordering, so as to preserve the inherent low-rank structures of the intermediate Schur complements [30]. The details are as follows.

- Nested dissection is performed with the package Scotch [25]. A multilevel scheme [19] is applied to determine the vertex separators. The assembly tree $\mathcal{T}$ is built with each nonleaf node corresponding to a separator. The



(i) Nested dissection and separator ordering (ii) Corresponding assembly tree

FIG. 1. *Nested dissection of a mesh and the corresponding assembly tree, where the mesh illustration is based on* Meshpart [13].

Gibbs–Poole–Stockmeyer method [12] is used to order the vertices within each separator so that the bandwidth of the corresponding adjacency submatrix of A is small. Geometrically, this keeps nearby vertices within a separator close to each other under the new ordering and may benefit the low-rank structure of the corresponding frontal matrix. The reason is as follows. For some discretized elliptic PDEs, the frontal matrices are related to (the inverses of) the discretized Green's functions, and the low-rank property of the frontal matrices is related to the geometric separability of the mesh points where the Green's function is evaluated [7, 33]. Well-separated points correspond to blocks with small numerical ranks. Thus intuitively, it is often beneficial to use an appropriate ordering that respects the geometric connectivity.

- In a symbolic factorization, following a bottom-up traversal of the assembly tree $\mathcal{T}$, we collect the neighbor information of each node of $\mathcal{T}$. The neighbor information reflects how earlier eliminations create fill-in. The connectivity of the vertices within a separator $\mathbf{i}$ and its neighbors is also collected. Such connectivity information is accumulated from child nodes to parent nodes and is used to preserve the graph connectivity after reordering.
- The symbolic factorization is performed in parallel. Under a certain parallel level $\mathbf{l}_p$, each process performs a local symbolic factorization of a private subtree and stores the corresponding neighboring information. At level $\mathbf{l}_p$, an all-to-all communication between all processes is triggered to exchange neighboring information of nodes at $\mathbf{l}_p$. For all the nodes above $\mathbf{l}_p$, the symbolic factorization is performed within each processor, so that each process can conveniently access relevant nodes without extra communications.
- Following the nice data locality of the assembly tree, we adopt a subtree-to-subcube static mapping model [11] to preallocate computational tasks to each process. Each process is assigned a portion of the computational tasks associated with a subtree of $\mathcal{T}$ or a subgraph of $\mathcal{G}$. Starting from a certain level, processes form disjoint process groups form process grids to work together. We try to evenly divide the adjacency graph so as to maintain the load balance of processes. See also section 4.1. Static unbalances may still exist as a result of unbalanced partitioning of the graph. Dynamic unbalances may also arise in the factorization stage since the HSS ranks and HSS generator sizes are generally unknown in advance. It will be interesting to investigate the use of dynamic scheduling in our solver. Other static mapping models such as the strict proportional mapping [26] are also used in some sparse direct solvers and may help improve the static balance of our solver. These are not the primary focus of the current work and will be studied in the future.

3. Review of the randomized structured multifrontal method. After nested dissection, the matrix factorization can be performed via the multifrontal method [9, 22], where partial factors are produced along the traversal of the assembly tree $\mathcal{T}$. For simplicity, we use $A_{\mathbf{i},\mathbf{j}}$ to denote $A|_{\mathbf{s}_\mathbf{i}\times\mathbf{s}_\mathbf{j}}$ for nodes $\mathbf{i},\mathbf{j}$ of $\mathcal{T}$ and use $\mathcal{N}_\mathbf{i}$ to denote the set of neighbors of $\mathbf{i}$.

For each leaf $\mathbf{i}$, $\mathcal{N}_\mathbf{i}$ includes all the separators that are connected to $\mathbf{i}$ in $\mathcal{G}$. Form an initial *frontal matrix*

$$\mathcal{F}_\mathbf{i} \equiv \mathcal{F}_\mathbf{i}^0 = \begin{pmatrix} A_{\mathbf{i},\mathbf{i}} & A_{\mathbf{i},\mathcal{N}_\mathbf{i}} \\ A_{\mathcal{N}_\mathbf{i},\mathbf{i}} & 0 \end{pmatrix}. \tag{3.1}$$

Compute an LU factorization of $\mathcal{F}_{\mathbf{i}}$:

$$\mathcal{F}_{\mathbf{i}} = \begin{pmatrix} L_{\mathbf{i},\mathbf{i}} & \\ L_{\mathcal{N}_{\mathbf{i}},\mathbf{i}} & I \end{pmatrix} \begin{pmatrix} K_{\mathbf{i},\mathbf{i}} & K_{\mathbf{i},\mathcal{N}_{\mathbf{i}}} \\ & \mathcal{U}_{\mathbf{i}} \end{pmatrix}, \tag{3.2}$$

where $\mathcal{U}_{\mathbf{i}}$ is the Schur complement and is called an *update matrix*. In the adjacency graph, this corresponds to the elimination of $\mathbf{i}$.

For each nonleaf node $\mathbf{i}$, $\mathcal{N}_{\mathbf{i}}$ includes all the ancestor separators that are originally connected to $\mathbf{i}$ in $\mathcal{G}$ or get connected due to lower level eliminations. Suppose the child nodes of $\mathbf{i}$ are $\mathbf{c}_1$ and $\mathbf{c}_2$. Form a *frontal matrix*

$$\mathcal{F}_{\mathbf{i}} = \mathcal{F}_{\mathbf{i}}^0 \oplus \mathcal{U}_{\mathbf{c}_1} \oplus \mathcal{U}_{\mathbf{c}_2}, \tag{3.3}$$

where the symbol $\oplus$denotes the *extend-add* operation [9, 22]. This operation performs matrix permutation and addition by matching the index sets following the ordering of the vertices in $\mathcal{G}$. Then partition $\mathcal{F}_{\mathbf{i}}$ as

$$\mathcal{F}_{\mathbf{i}} = \begin{pmatrix} F_{\mathbf{i};1,1} & F_{\mathbf{i};1,2} \\ F_{\mathbf{i};2,1} & F_{\mathbf{i};2,2} \end{pmatrix}, \tag{3.4}$$

where $F_{\mathbf{i};1,1}$ corresponds to $\mathbf{s}_{\mathbf{i}}$, and perform an LU factorization as in (3.2).

The above procedure continues along the assembly tree until the root is reached. This is known as the standard multifrontal method. A bottleneck of the method is the storage and cost associated with the local dense frontal and update matrices.

The class of structured multifrontal methods [30, 31, 33] is based on the idea that, for certain matrices A, the local dense frontal and update matrices are rank structured. For some cases, $\mathcal{F}_{\mathbf{i}}$ and $\mathcal{U}_{\mathbf{i}}$ may be approximated by rank structured forms such as HSS forms. An HSS matrix F is a hierarchical structured form that can be defined via a postordered binary tree T called an *HSS tree*. Each node i of T is associated with some local matrices called generators that are used to define F. A diagonal generator D_i is recursively defined, so that $D_k \equiv F$ for the root k of T, and for a node i with children c_1 and c_2,

$$D_i = \begin{pmatrix} D_{c_1} & U_{c_1} B_{c_1} V_{c_2}^T \\ U_{c_2} B_{c_2} V_{c_1}^T & D_{c_2} \end{pmatrix},$$

where the off-diagonal basis generators U, V are also recursively defined as

$$U_i = \begin{pmatrix} U_{c_1} R_{c_1} \\ U_{c_2} R_{c_2} \end{pmatrix}, \quad V_i = \begin{pmatrix} V_{c_1} W_{c_1} \\ V_{c_2} W_{c_2} \end{pmatrix}.$$

Suppose D_i corresponds to an index set $s_i \subset \{1 : N\}$, where N is the size of F; then $F_{\mathbf{i}}^{-} \equiv F|_{s_i \times (\{1:N\} \setminus s_i)}$ and $F_{\mathbf{i}}^{|} \equiv F|_{(\{1:N\} \setminus s_i) \times s_i}$ are called HSS blocks, whose maximum (numerical) rank is called the *HSS rank* of F. Also for convenience, suppose the root node of T is at level $l = 0$.

In the structured multifrontal method in [33], HSS approximations to the frontal matrices are constructed. The factorization of an HSS frontal matrix $\mathcal{F}_{\mathbf{i}}$ yields an HSS update matrix $\mathcal{U}_{\mathbf{i}}$. This process may be costly and complex. A simplified version in [28] uses dense $\mathcal{U}_{\mathbf{i}}$ instead. To enhance the efficiency and flexibility, randomized HSS construction [24, 35] is used in [31]. The basic idea is to compress the off-diagonal blocks of $\mathcal{F}_{\mathbf{i}}$ via randomized sampling [21]. Suppose $\mathcal{F}_{\mathbf{i}}$ has size $N \times N$ and HSS

rank r. Generate an $N \times (r+\mu)$ Gaussian random matrix X, where μ is a small integer. Compute

$$Y = \mathcal{F}_{\mathbf{i}} X. \tag{3.5}$$

The compression of an HSS block $\mathcal{F}_{\mathbf{i}}^{-} \equiv \mathcal{F}_{\mathbf{i}}|_{s_i \times (\{1:N\} \setminus s_i)}$ is done via the compression of

$$S_i = Y - D_i X|_{s_i} (= \mathcal{F}_{\mathbf{i}}^{-} X|_{\{1:N\} \setminus s_i}). \tag{3.6}$$

Applying a strong rank-revealing factorization [16] to the above quantity yields

$$S_i \approx U_i S_i|_{\hat{s}_i} \quad \text{with} \quad U_i = \Pi_i \begin{pmatrix} I \\ E_i \end{pmatrix},$$

where $\hat{s}_i$ is a subset of s_i and the entries of E_i have magnitudes around 1. This gives the U generator in the HSS construction, so that $\mathcal{F}_{\mathbf{i}}^{-} \approx U_i \mathcal{F}_{\mathbf{i}}^{-}|_{\hat{s}_i}$ with high probability [21]. Due to the feature that the row basis matrix $\mathcal{F}_{\mathbf{i}}^{-}|_{\hat{s}_i}$ is a submatrix of $\mathcal{F}_{\mathbf{i}}^{-}$, this compression is also referred to as a structure-preserving rank-revealing factorization [35]. More details for generating the other HSS generators can be found in [24, 35]. The overall HSS construction follows a bottom-up sweep of the HSS tree.

In the RSMF method [31], the product (3.5) and selected entries of $\mathcal{F}_{\mathbf{i}}$ are used to construct an HSS approximation to $\mathcal{F}_{\mathbf{i}}$. A partial ULV factorization [6, 34] of the HSS form is then computed. Unlike other multifrontal methods, here the update matrix $\mathcal{U}_{\mathbf{i}}$ does not directly participate in the extend-add operation. Instead, its products with random vectors are used in a *skinny extend-add* [31] that only involves vectors and their row permutation and addition. This significantly enhances the flexibility and efficiency of extend-add. We sketch the main framework of the method, which slightly generalizes the method in [31] to nonsymmetric matrices. A pictorial illustration is given in Figure 2. In the following, we assume $X_{\mathbf{i}}$ to be a skinny random matrix that plays the role of X in (3.5) and is used for the HSS construction for $\mathcal{F}_{\mathbf{i}}$.

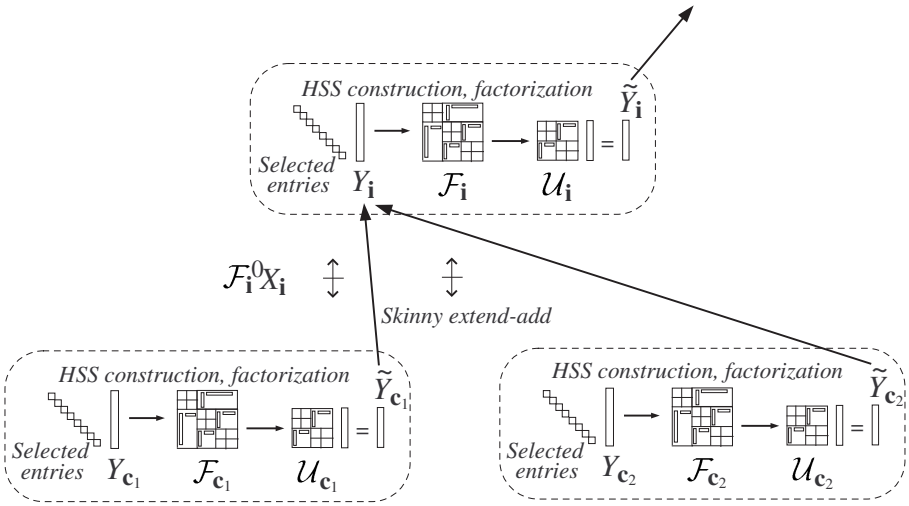


FIG. 2. *Illustration of the RSMF method in* [31]. *For simplicity, only a symmetric version is used for illustration.*

1. For nodes $\mathbf{i}$ below a certain switching level $\mathbf{l}_s$ of $\mathcal{T}$, perform the traditional multifrontal factorization.
2. For nodes $\mathbf{i}$ above level $\mathbf{l}_s$, construct an HSS approximation to $\mathcal{F}_\mathbf{i}$ using selected entries of $\mathcal{F}_\mathbf{i}$ and matrix-vector products $Y_\mathbf{i} = \mathcal{F}_\mathbf{i} X_\mathbf{i}$ and $Z_\mathbf{i} = \mathcal{F}_\mathbf{i}^T X_\mathbf{i}$. Here, $Y_\mathbf{i}$ and $Z_\mathbf{i}$ are obtained from (3.8) below.
3. Partially ULV factorize $\mathcal{F}_\mathbf{i}$ and compute $\mathcal{U}_\mathbf{i}$ with the fast Schur complement update in HSS form. Compute

$$\tilde{Y}_\mathbf{i} = \mathcal{U}_\mathbf{i} \tilde{X}_\mathbf{i}, \quad \tilde{Z}_\mathbf{i} = \mathcal{U}_\mathbf{i}^T \tilde{X}_\mathbf{i}, \tag{3.7}$$

   where $\tilde{X}_\mathbf{i}$ is a submatrix of $X_\mathbf{i}$ corresponding to the indices $\mathcal{U}_\mathbf{i}$ as in (3.2).
4. For an upper level node $\mathbf{i}$ with children $\mathbf{c}_1$ and $\mathbf{c}_2$, compute skinny extend-add operations

$$Y_\mathbf{i} = (\mathcal{F}_\mathbf{i}^0 X_\mathbf{i}) \,\ddagger\, \tilde{Y}_{\mathbf{c}_1} \,\ddagger\, \tilde{Y}_{\mathbf{c}_2}, \quad Z_\mathbf{i} = ((\mathcal{F}_\mathbf{i}^0)^T X_\mathbf{i}) \,\ddagger\, \tilde{Z}_{\mathbf{c}_1} \,\ddagger\, \tilde{Z}_{\mathbf{c}_2}. \tag{3.8}$$

5. Form selected entries of $\mathcal{F}_\mathbf{i}$ based on $\mathcal{F}_\mathbf{i}^0$, $\mathcal{U}_{\mathbf{c}_1}$, and $\mathcal{U}_{\mathbf{c}_2}$, and go to step 2.

4. Distributed-memory parallel randomized multifrontal method. Here, we present our distributed-memory parallel sparse direct solver based on the RSMF method, denoted PRSMF. The primary significance includes the following:

- We design a static mapping parallel model that (1) has two hierarchical layers of parallelism with good load balance and (2) can conveniently handle both HSS matrices and intermediate skinny matrices. The PRSMF method significantly extends the one in [28] by allowing more flexible parallel structured operations and also avoiding large dense intermediate frontal and update matrices.
- We develop some innovative parallel algorithms for a sequence of HSS operations, such as a parallel HSS Schur complement computation in partial ULV factorization, a parallel extraction of selected entries from an HSS form, and a parallel skinny extend-add algorithm for assembling local data.
- The main ideas to reduce the communication costs in the new parallel algorithms are (1) taking advantage of the tree parallelism and store HSS generators in an appropriate way; (2) assembling and passing data between processes in compact forms so that the number of messages is reduced; (3) reusing data whenever possible; (4) collecting data into messages to facilitate BLAS3 dense operations; and (5) storing a small amount of the same data in multiple processes.
- The graph reordering strategies and the convenient data assembly make the method much more general than the parallel structured multifrontal method in [28] which is restricted to a specific mesh geometry.

In this section, an overview of the parallel model is given, followed by details of the parallel algorithms.

4.1. Overview of the parallel model and data distribution. In the RSMF method, both the assembly tree and the HSS tree have nice data locality. For a distributed memory architecture, a static mapping model usually requires less communication than a dynamic scheduling model. To avoid idling as much as possible, it is vital to keep work load balance among all processes. When generating the assembly tree and local HSS trees, we try to make the trees as balanced as possible. A roughly balanced assembly tree can be obtained with Scotch [25] with appropriate strategies. More specifically, a multilevel graph partition method in [19] is used to generate the

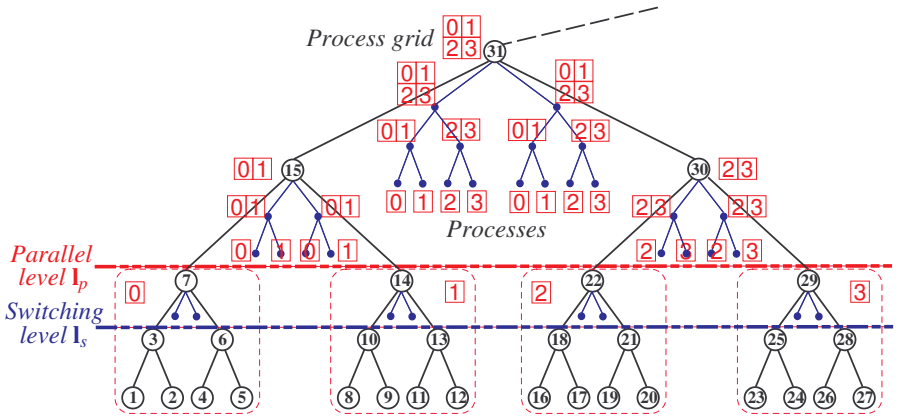


FIG. 3. *Static mapping model for our PRSMF method, where the outer-layer tree is the assembly tree, and the inner-layer trees are HSS trees.*

separators of the adjacency graph of A. In this method, each time we use the vertex greedy-graph-growing method to obtain a rough separator, and refine this separator with some strategies so that the resulting lower level subgraphs are roughly balanced. For the HSS trees, we try to keep them balanced via the recursive bipartition of the index set of a frontal matrix.

Our parallel model (Figure 3) is described as follows. The basic idea is inherited from [28], and we further incorporate more structured forms to avoid dense frontal/update matrices and also allow more flexible choices of processes for the structured operations. Below a parallel level $\mathbf{l}_p$, one process is used to handle the operations corresponding to one subtree, called a *local tree*. (The method also involves a switching level $\mathbf{l}_s$, at which the local operations switch from dense to HSS ones so as to optimize the complexity and also to avoid structured operations for very small matrices at lower levels [31].) Starting from level $\mathbf{l}_p$, processes are grouped into process grids or contexts to work on larger matrices. The standard 2D block cyclic storage scheme in ScaLAPACK [4] is adopted to store dense blocks (such as HSS generators and skinny matrices). Process grids associated with the nodes of the assembly tree $\mathcal{T}$ are generated as follows. Let $\mathbf{G_i}$ denote the set of processes assigned to a node $\mathbf{i}$. Then,

- $\mathbf{G_i} \cap \mathbf{G_j} = \varnothing$ if $\mathbf{i}$ and $\mathbf{j}$ are distinct nodes at the same level;
- $\mathbf{G_i} = \mathbf{G_{c_1}} \cup \mathbf{G_{c_2}}$ for a parent node $\mathbf{i}$ with child nodes $\mathbf{c}_1$ and $\mathbf{c}_2$. Usually, the process grids associated with $\mathbf{G_{c_1}}$ and $\mathbf{G_{c_2}}$ are concatenated vertically or horizontally to form the process grid associated with $\mathbf{G_i}$. With this approach, only pairwise exchanges between the processes are needed when redistributing matrices from the children process grids to the parent process grids. This can help reduce the number of messages exchanged and save the communication cost. See [28, 29] for more discussions. Here, since the algorithm frequently involves skinny matrices, the process grids are not necessarily square. This is different from those in [28, 29], where large local matrices are involved.

In this setting, each process can only access a subset of nodes in the assembly tree. We call these nodes *accessible nodes of a process*. The accessible nodes of a process are those in the associated local tree and in the path connecting the root of the local tree and the root of the assembly tree. For example, in Figure 3, the

accessible nodes of process 0 are nodes $1, 2, \ldots, 7, 15, 31$. A process only gets involved in the computations associated with its accessible nodes.

Along the assembly tree, child nodes pass information to parent nodes, and communication between two process groups is involved. Unlike the traditional multifrontal method and the structured one in [28] that pass dense update matrices, here, only skinny matrix-vector products and a small number of matrix entries are passed. On the process grid of a parent node, we redistribute these skinny matrices to the corresponding processes. We design a distribution scheme and a message passing method to perform the skinny extend-add operation in parallel. The result will be used in the parallel randomized HSS construction.

For a node $\mathbf{i}$ above $\mathbf{l}_p$, if HSS operations are involved, smaller process subgroups are formed by the processes in $\mathbf{G_i}$ to accommodate the parallel HSS operations for $\mathcal{F}_\mathbf{i}$, $\mathcal{U}_\mathbf{i}$, etc. For each node i of the HSS tree T of $\mathcal{F}_\mathbf{i}$, a process group $G_i \subset \mathbf{G_i}$ is formed. Initially, for the root k of T, $G_k \equiv \mathbf{G_i}$, which is used for both $\mathcal{F}_\mathbf{i}$ and $\mathcal{U}_\mathbf{i}$. Unlike the scheme in [23], G_k is also used for both children of k, so as to facilitate the computation of $\mathcal{U}_\mathbf{i}$. (See the inner-layer trees in Figure 3.) For nodes at lower levels, the same strategies to group processes as above are used along the HSS tree to form the process subgroups. That is, for any nonleaf node i of T below level 1, $G_i = G_{c_1} \cup G_{c_2}$, where children c_1 and c_2 are the children of i. Each process subgroup forms a process grid which stores the dense blocks (such as HSS generators and skinny matrices) in a 2D block cyclic scheme. The operations (such as multiplication and redistribution) on the dense blocks are performed with ScaLAPACK. As in [23], the HSS generators R_i, W_i (and also D_i, U_i for a leaf i) are stored in G_i, and B_i is stored in $G_{\mathrm{par}(i)}$. The details of these grouping and distribution schemes will become clear in the subsequent subsections.

We would also like to mention that the nonzero entries of the original sparse matrix A are needed in the formation of frontal matrices or in frontal matrix-vector multiplications if a frontal matrix is not formed explicitly. They are also needed in the formation of the D, B HSS generators. It is usually not memory efficient to store a copy of the data in each process. On the other hand, distributing the entries of A without overlapping among processes may increase communication cost dramatically. Although we know what entries of A are needed in the current front, we do not know in advance the specific destination processes of selected entries of A that are needed in the formation of the B generators in the randomized HSS construction algorithm. As a result, communications of the entries of A cannot be avoided with a nonoverlapping mapping strategy, in general. Thus, we choose a trade-off between the memory and communication costs. That is, each process within a process group $G_\mathbf{i}$ stores, besides its private data, some overlapping data. The overlapping data corresponds to the entries of A associated with the accessible nodes of the process. If the processes are in the same process group $G_\mathbf{i}$, then they store the same overlapping data. An illustration of this storage scheme is given in Figure 4. This idea can also be applied to store the neighbor information of a separator in the symbolic factorization as well as solutions in forward and backward substitutions.

To summarize, two layers of tree parallelism are naturally combined in this parallel model: the parallelism of the assembly tree and the HSS trees. Within the distributed assembly tree and HSS trees, distributed dense operations for HSS generators and skinny matrices are facilitated by ScaLAPACK. The parallel structured matrix operations and skinny matrix operations will be discussed in detail next.

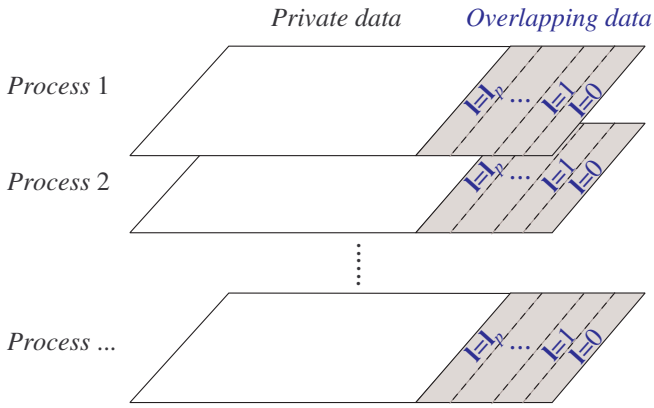


FIG. 4. *Storage scheme for the sparse matrix A. Below the parallel level $\mathbf{l}_p$, each process stores a private portion of A. Above $\mathbf{l}_p$, each process stores overlapping data associated with its accessible nodes at each level $\mathbf{l} = 0, 1, \ldots, \mathbf{l}_p$.*

4.2. Parallel randomized HSS construction, factorization, and Schur complement computation. Above the switching level $\mathbf{l}_s$, all frontal and update matrices are in HSS forms. The method in [28] is only partially structured in the sense that dense frontal matrices are formed first and then converted into HSS forms, and the updated matrices are also dense. Here, the HSS construction for a frontal matrix $\mathcal{F}_\mathbf{i}$ is done via randomization, and a partial ULV factorization of $\mathcal{F}_\mathbf{i}$ also produces an HSS form Schur complement/update matrix $\mathcal{U}_\mathbf{i}$ in (3.2).

To involve as many processes as possible, we assign all the processes in the process group $\mathbf{G}_\mathbf{i}$ to perform the HSS construction, partial ULV factorization, and Schur complement computation. The HSS tree T for $\mathcal{F}_\mathbf{i}$ has two subtrees. The left subtree T_1 corresponds to the pivot block $F_{\mathbf{i};1,1}$ in (3.4), and the right subtree T_2 corresponds to $F_{\mathbf{i};2,2}$ and thus the Schur complement $\mathcal{U}_\mathbf{i}$. Note that $\mathbf{G}_\mathbf{i}$ is used for the HSS constructions of both $F_{\mathbf{i};1,1}$ and $F_{\mathbf{i};2,2}$. This is different from the storage scheme in [23]. See the inner-layer tree associated with node $\mathbf{i} = 31$ in Figure 3 for an illustration. The individual parallel randomized HSS constructions for $F_{\mathbf{i};1,1}$ and $F_{\mathbf{i};2,2}$ follow [23]. Initially, each process stores a piece of the matrix-vector products (3.5), which are used to compress the leaf-level HSS blocks. These products are updated when the compression moves up along the HSS tree T. Intergrid redistributions are involved for pairs of siblings in the HSS tree. After the HSS constructions for $F_{\mathbf{i};1,1}$ and $F_{\mathbf{i};2,2}$, we continue to use $\mathbf{G}_\mathbf{i}$ to construct the remaining generators $B_k, \left(\begin{smallmatrix} R_{k_1} \\ R_{k_2} \end{smallmatrix}\right)$ for $k = \text{root}(T_1)$ with children k_1, k_2 and $B_q, \left(\begin{smallmatrix} W_{q_1} \\ W_{q_2} \end{smallmatrix}\right)$ for $q = \text{root}(T_2)$ with children q_1, q_2. Figure 5 shows how the HSS generators are stored among the processes.

The parallel partial ULV factorization follows a scheme in [23], except that it stops after all the nodes in T_1 are traversed. The ULV factors associated with each tree node are stored in the corresponding processes. The partial ULV factorization reduces $F_{\mathbf{i};1,1}$ to a final reduced matrix $\tilde{D}_k$ [30].

At this point, we need to form the Schur complement $\mathcal{U}_\mathbf{i}$ in an HSS form. The blocks $\tilde{D}_k, \left(\begin{smallmatrix} R_{k_1} \\ R_{k_2} \end{smallmatrix}\right), \left(\begin{smallmatrix} W_{q_1} \\ W_{q_2} \end{smallmatrix}\right)$ are stored in the process group $\mathbf{G}_\mathbf{i}$, which is also the process group G_q for node q. To obtain the HSS form of $\mathcal{U}_\mathbf{i}$, we only need to update the D, B generators of $F_{\mathbf{i};2,2}$, which are stored within the subgroups of G_q. This is done via a fast update method in [31]. (The method in [31] is symmetric and here a

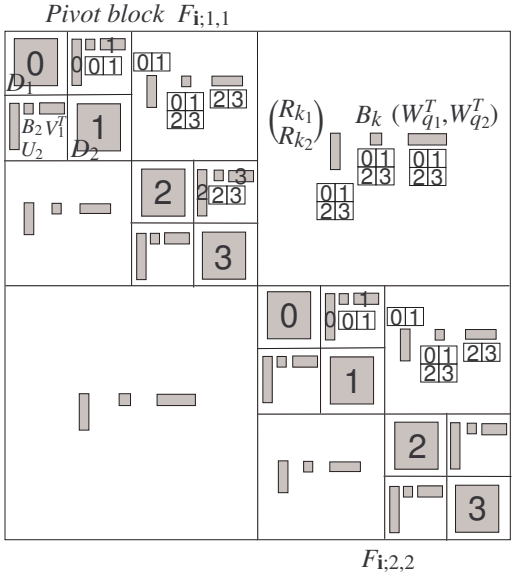


FIG. 5. *Distributed memory storage for an HSS form frontal matrix $\mathcal{F}_\mathbf{i}$ given four processes, where k_1, k_2 are the children of k (root of the HSS tree T_1 for $F_{\mathbf{i};1,1}$) and q_1, q_2 are the children of q (root of the HSS tree T_2 for $F_{\mathbf{i};2,2}$). See the inner-layer tree associated with node* $\mathbf{i}$=31 *in Figure* 3.

nonsymmetric variation is used.) The computations are performed in disjoint process groups at each level in a top-down traversal of T_2. A sequence of products S_i are computed for the update. Each S_i is computed in G_i and then redistributed for reuse at lower levels. Its submatrices are used to update the generators D_i, B_i. The details are given in Algorithm 1. An important observation from this update process is that the HSS rank of $\mathcal{U}_\mathbf{i}$ is bounded by that of $\mathcal{F}_\mathbf{i}$ [30].

After this, we then compute $\tilde{Y}_\mathbf{i}$ and $\tilde{Z}_\mathbf{i}$ in (3.7) in parallel. This may be based on direct HSS matrix-vector multiplications, or an indirect procedure [31] that updates $Y_\mathbf{i}$. The detailed parallelization is omitted here. $\tilde{Y}_\mathbf{i}$ and $\tilde{Z}_\mathbf{i}$ are passed to the parent node to participate in the skinny extend-add operation.

Remark 4.1. In our solver, the nested HSS structure has a very useful feature that significantly simplifies one major step: the formation of the Schur complement/update matrix (the mathematical scheme behind Algorithm 1). That is, the nested HSS structure enables the Schur complement computation to be conveniently done via some reduced matrices, so that only certain generators of the frontal matrix need to be quickly updated. Such a mathematical scheme was designed and proved in [30, 31]. For 3D problems, the HSS structure is indeed less efficient than more advanced structures such as $\mathcal{H}^2$ and multilayer hierarchical structures [32]. The HSS form serves as a compromise between the efficiency and the implementation simplicity.

4.3. Parallel skinny extend-add operation. The dense updated matrices used in the parallel partially structured multifrontal method in [28] need not only more computations but also more communications. In fact, the extend-add operation is a major communication bottleneck of the method in [28], just like in the standard parallel multifrontal method. In the 2D block cyclic setting, the standard dense extend-add in (3.3) requires extensive intragrid communications. Both row and column permutations are performed. Entries in the same row/column of an update

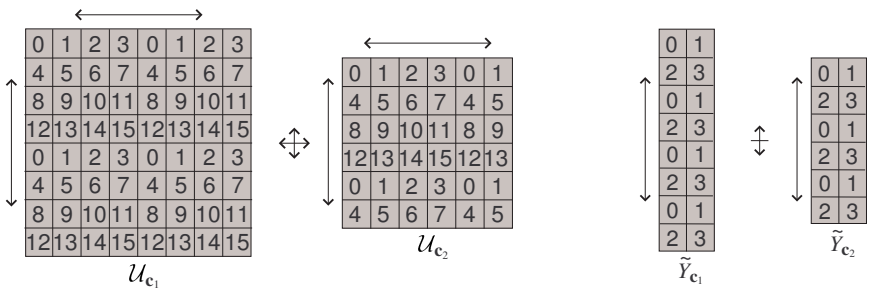
Algorithm 1. Parallel computation of the HSS Schur complement/update matrix $\mathcal{U}_{\mathbf{i}}$.

procedure PSchur
  $S_q \leftarrow B_q \begin{pmatrix} W_{q_1}^T & W_{q_2}^T \end{pmatrix} \tilde{D}_k^{-1} \begin{pmatrix} R_{k_1} \\ R_{k_2} \end{pmatrix} B_k$
    ▷ Performed in the process group G_q; q: root of the HSS tree T_2 for $F_{\mathbf{i};2,2}$
  for each level $l = 0, 1, \ldots$ of T_2 **do** ▷ *Top-down traversal of* T_2
    for each node i at level l **do** ▷ *Performing operations in the process group* G_i *and redistributing information*
      if i is a nonleaf node **then**
        $S_i = \begin{pmatrix} S_{i;1,1} & S_{i;1,2} \\ S_{i;2,1} & S_{i;2,2} \end{pmatrix}$ ▷ *Conformable partition following* $\begin{pmatrix} R_{c_1} \\ R_{c_2} \end{pmatrix}$
        Redistribute $S_{i;1,1}$ and $S_{i;2,2}$ to G_{c_1} and G_{c_2}, respectively
          ▷ G_{c_1}, G_{c_2}: *children process groups*
        $S_{c_1} \leftarrow W_{c_1}^T S_{i;1,1} R_{c_1}$, $S_{c_2} \leftarrow W_{c_2}^T S_{i;2,2} R_{c_2}$ ▷ *Performed in* G_{c_1}, G_{c_2}
        $B_{c_1} \leftarrow B_{c_1} - S_{i;1,2}$, $B_{c_2} \leftarrow B_{c_2} - S_{i;2,1}$ ▷ B_{c_1}, B_{c_2}*: stored in* G_i
      else
        $D_i \leftarrow D_i - S_i$ ▷ *Leaf level* D *generator update*
      end if
    end for
  end for
end procedure

matrix are added to one same row/column of a frontal matrix after permutations. For parallel permutations on a square process grid with P processes, each process communicates with $O(\sqrt{P})$ processes in the same column, as well as $O(\sqrt{P})$ processes in the same row.

Our parallel randomized multifrontal solver uses, instead, the skinny extend-add operations (3.8) and significantly saves the communication cost. The matrices to be permuted are tall and skinny ones instead of large square ones, and moreover, only row permutation is needed. Figure 6 illustrates the storage of the skinny matrices on a process grid.

To further reduce the communication overhead, we use a message passing method [3, 20] to reduce the number of messages. Instead of passing information row by row, we group rows according to their destination processes. The rows needed to be sent



(i) Standard parallel extend-add (ii) Skinny parallel extend-add

FIG. 6. *Parallel skinny extend-add operation as compared with the standard dense one, where the matrices are stored on a process grid. The skinny one deals with much less data and also needs only row permutations.*

Algorithm 2. Parallel skinny extend-add operation.

procedure PSExtAdd ▷ $Y_{\mathbf{i}} = (\mathcal{F}_{\mathbf{i}}^0 X_{\mathbf{i}}) \ddagger \tilde{Y}_{\mathbf{c}_1} \ddagger \tilde{Y}_{\mathbf{c}_2}$
  $Y_{\mathbf{i}} \leftarrow \mathcal{F}_{\mathbf{i}}^0 X_{\mathbf{i}}$ (stored in the process grid G)
  for $\tilde{Y} = \tilde{Y}_{\mathbf{c}_1}, \tilde{Y}_{\mathbf{c}_2}$ **do** ▷ *Processing $\tilde{Y}_{\mathbf{c}_1}$ and $\tilde{Y}_{\mathbf{c}_2}$ individually*
    for column $j = 1, \ldots,$ ncol **do** ▷ ncol: *number of process columns in G*
      for each process p in column j of G **do** ▷ *Packing message*
        $\hat{p} \leftarrow$ row coordinate of p in G
        for row $i = 1, \ldots,$ locr **do**
          ▷ locr: *number of local rows of $\tilde{Y}$ stored in p*
          $\hat{q} \leftarrow$ row coordinate of the destination process for $\tilde{Y}|_i$ in G
          Insert $\tilde{Y}|_i$ and its destination row index in $Y_{\mathbf{i}}$ into message $M_{\hat{p}\to\hat{q}}$
        end for
      end for
      for each process p in column j of G **do** ▷ *Sending & receiving messages*
        $\hat{p} \leftarrow$ row coordinate of p in G
        for $\hat{q} = 1, \ldots,$ nrow **do** ▷ nrow: *number of process rows in G*
          if $\hat{p} \neq \hat{q}$ **then** ▷ *Locally blocking send operation*
            Send message $M_{\hat{p}\to\hat{q}}$ if $M_{\hat{p}\to\hat{q}} \neq \varnothing$
          end if
        end for
        for $\hat{q} = 1, \ldots,$ nrow **do**
          if $\hat{p} \neq \hat{q}$ **then** ▷ *Globally blocking receive operation*
            Receive message $M_{\hat{q}\to\hat{p}}$ if $M_{\hat{q}\to\hat{p}} \neq \varnothing$
          end if
          Add rows in $M_{\hat{q}\to\hat{p}}$ to $Y_{\mathbf{i}}$ according to their
            destination row indices if $M_{\hat{q}\to\hat{p}} \neq \varnothing$
        end for
      end for
    end for
  end for
end procedure

from one process to another are first wrapped into a message, which is sent later. Since the update and frontal matrices are both distributed in a block-cyclic layout, it is not hard to figure out the target process of each local row by using functions indxl2g and indxg2p in ScaLAPACK. The messages are sent and received with BLACS [8] routines for point to point communications. The send operations are locally blocking. The receive operations are globally blocking. The parallel skinny extend-add algorithm is outlined in Algorithm 2.

4.4. Extracting selected entries from HSS update matrices. The matrix-vector products $Y_{\mathbf{i}}$ (and also $Z_{\mathbf{i}}$) obtained in the previous subsection are then used to construct an HSS approximation to $\mathcal{F}_{\mathbf{i}}$. This still needs selected entries of $\mathcal{F}_{\mathbf{i}}$ that will be used to form the B generators and leaf level D generators [24, 35]. Given the row and column index sets, we can extract the entries of $\mathcal{F}_{\mathbf{i}}$ from $\mathcal{F}_{\mathbf{i}}^0$, $\mathcal{U}_{\mathbf{c}_1}$, and $\mathcal{U}_{\mathbf{c}_2}$ in (3.3). $\mathcal{F}_{\mathbf{i}}^0$ is immediately available, and we focus on $\mathcal{U}_{\mathbf{c}_1}$ and $\mathcal{U}_{\mathbf{c}_2}$.

The indices of the desired entries of $\mathcal{F}_{\mathbf{i}}$ are mapped to those of $\mathcal{U}_{\mathbf{c}_1}$ and $\mathcal{U}_{\mathbf{c}_2}$ (Figure 7). Thus, we consider how to extract in parallel selected entries $\mathcal{U}|_{\mathcal{I}\times\mathcal{J}}$ of an HSS matrix $\mathcal{U}$, as specified by a row index set $\mathcal{I}$ and a column index set $\mathcal{J}$. Suppose

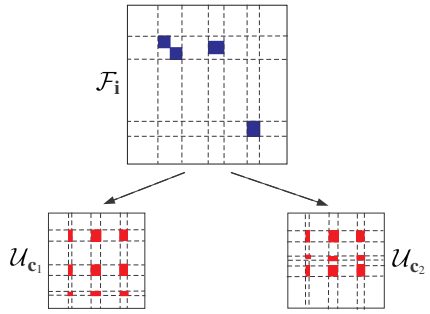


FIG. 7. *Mapping the row and column index sets of selected entries of* $\mathcal{F}_{\mathbf{i}}$ *to those of* $\mathcal{U}_{\mathbf{c}_1}$ *and* $\mathcal{U}_{\mathbf{c}_2}$.

$\mathcal{U}$ has HSS generators D_i, U_i, R_i, B_i, etc. The basic idea of the entry extraction in [31] is to reuse intermediate computations while performing matrix-vector multiplications. Here, instead of using matrix-vector multiplications like in [31], we group indices as much as possible so as to enable BLAS3 operations.

Among the leaf nodes of the HSS tree of $\mathcal{U}$, identify all the leaves i and j that correspond to $\mathcal{I}$ and $\mathcal{J}$, respectively. Suppose i and j are associated with subsets $\mathcal{I}_i \subset \mathcal{I}$ and $\mathcal{J}_j \subset \mathcal{J}$ and local index sets $\tilde{\mathcal{I}}_i$ and $\tilde{\mathcal{J}}_j$ of U_i and V_j, respectively. Let the path connecting nodes i and j in the HSS tree be

$$i - i_1 - i_2 - \cdots - i_d - j_d - \cdots - j_2 - j_1 - j, \tag{4.1}$$

where $d \equiv \text{distance}(i, j)$ is the number of ancestors of i or j in the path and is called the *distance* between i and j [31]. Then obviously,

$$\mathcal{U}|_{\mathcal{I}_i \times \mathcal{I}_j} = U_i|_{\tilde{\mathcal{I}}_i} R_{i_1} R_{i_2} \cdots R_{i_{d-1}} B_{i_d} W_{j_{d-1}}^T \cdots W_{j_2}^T W_{j_1}^T V_j^T|_{\tilde{\mathcal{I}}_j}. \tag{4.2}$$

By grouping the indices as $\mathcal{I}_i$ and $\mathcal{I}_j$, we reduce the number of paths to be traversed from $O(N)$ in [31] to $O(\frac{N}{r})$, where N is the size of $\mathcal{U}$ and r is its HSS rank.

To save computational costs, some intermediate matrix-matrix products can be precomputed for all the entries in $\mathcal{U}|_{\mathcal{I} \times \mathcal{J}}$ in the same rows/columns. That is, in (4.2), a portion of the following product is stored if the corresponding subpath $i - i_1 - i_2 - \cdots - i_{d-1}$ is shared by multiple paths like (4.1):

$$\Omega_i^d \equiv U_i|_{\tilde{\mathcal{I}}_i} R_{i_1} R_{i_2} \cdots R_{i_{d-1}}. \tag{4.3}$$

This is similar for $\Theta_j^d \equiv V_j W_{j_1} W_{j_2} \cdots W_{j_{d-1}}$. Ω_i^d and Θ_j^d are computed in process grids $G_{i_{d-1}}$ and $G_{j_{d-1}}$, respectively. Then the stored results Ω_i^d and Θ_j^d are used to compute the entries as

$$\mathcal{U}|_{\mathcal{I}_i \times \mathcal{J}_j} = \Omega_i^d B_{i_d} (\Theta_j^d)^T.$$

If it is necessary to extend the subpath $i - i_1 - i_2 - \cdots - i_{d-1}$ for larger d, then Ω_i^d and Θ_j^d will be redistributed to the parent process grids for additional computations. Thus, Ω_i^d and Θ_j^d are computed progressively. See Algorithm 3.

Since the HSS generators are distributed on different process grids, this technique also helps to reduce the number and volume of messages in the parallel implementation. This is especially attractive if many entries to be extracted are from the same rows or columns.

Algorithm 3. Parallel extraction of $\mathcal{U}|_{\mathcal{I}\times\mathcal{J}}$ from an HSS matrix $\mathcal{U}$.

1: **procedure** PHSSIJ
2: **for** each leaf i of T **do** ▷ *T: HSS tree of $\mathcal{U}$*
3: Find the corresponding subsets $\mathcal{I}_i \subset \mathcal{I}$ and $\mathcal{J}_i \subset \mathcal{J}$
4: Find the corresponding local index sets $\tilde{\mathcal{I}}_i$ of U_i and $\tilde{\mathcal{J}}_i$ of V_i
5: **end for**
6: **for** each leaf i (with $\mathcal{I}_i \neq \varnothing$) of T **do** ▷ *Precomputation*
7: $\mathbf{d} \leftarrow$ sorted vector of nonzero distance(i, j) for all leaves $j \neq i$ with $\mathcal{J}_j \neq \varnothing$
8: $\Omega_i^0 \leftarrow U_i|_{\tilde{\mathcal{I}}_i},\ c \leftarrow 0,\ k \leftarrow i$ ▷ *Performed in the process group G_i*
9: **for** $\iota = 1, 2, \ldots,$ length$(\mathbf{d})$ **do**
10: $d \leftarrow \mathbf{d}_\iota,\ \Omega_i^d \leftarrow \Omega_i^c$ ▷ *Ω_i^d is for the partial product as in* (4.3)
11: **for** $\tilde{d} = \mathbf{d}_{\iota-1}, \ldots, \mathbf{d}_\iota$ **do** ▷ *Assuming* $\mathbf{d}_0 = 1$
12: $\Omega_i^d \leftarrow \Omega_i^d R_k$ ▷ *Computing* (4.3) *step by step in the process group G_k*
13: Redistribute Ω_i^d from the process group G_k to $G_{\mathrm{par}(k)}$
14: $k \leftarrow \mathrm{par}(k)$
15: **end for**
16: $c \leftarrow d$
17: **end for**
18: **end for**
19: Repeat steps 6–18 with $\mathcal{I}_i, \mathcal{J}_j, \tilde{\mathcal{I}}_i, R_k, \Omega_i^d$ replaced by $\mathcal{J}_i, \mathcal{I}_j, \tilde{\mathcal{J}}_i, W_k, \Theta_i^d$, respectively
20: **for** each leaf i (with $\mathcal{I}_i \neq \varnothing$) of T **do** ▷ *Extracting the entries*
21: **for** each leaf j (with $\mathcal{J}_j \neq \varnothing$) of T **do**
22: **if** $i == j$ **then** ▷ *Diagonal block extraction*
23: $\mathcal{U}|_{\mathcal{I}_i\times\mathcal{J}_j} \leftarrow D_i|_{\tilde{\mathcal{I}}_i\times\tilde{\mathcal{J}}_j}$ ▷ *Performed in the process group G_i*
24: **else** ▷ *Off-diagonal block extraction*
25: $d \leftarrow$ distance(i, j)
26: $k \leftarrow$ the largest ancestor of i in the path connecting i to j
27: $\mathcal{U}|_{\mathcal{I}_i\times\mathcal{J}_j} \leftarrow \Omega_i^d B_k (\Theta_j^d)^T$ ▷ *Performed in the process group G_k*
28: **end if**
29: **end for**
30: **end for**
31: **end procedure**

5. Analysis of the communication cost. The parallel framework together with the individual parallel structured operations in the previous section give our PRSMF method. The parallel solution method is similar to that in [28]. We can conveniently study the performance of the PRSMF method. The factorization cost and storage are given in [31]. For discretized sparse matrices, when certain rank conditions are satisfied (such as in elliptic problems), the factorization costs roughly $O(n)$ flops in two dimensions and roughly $O(n)$ to $O(n^{4/3})$ in three dimensions. The storage for both two dimensions and three dimensions is near $O(n)$. Here, some polylogarithmic terms of n may be omitted.

In the following, we estimate the communication cost. For convenience, suppose the maximum HSS rank of all the frontal matrices is r. In practice, the HSS ranks often depend on the applications and the matrix size. Our analysis here then gives a reasonable upper bound. For simplicity, assume the switching level $\mathbf{l}_s$ is below the parallel level $\mathbf{l}_p$ in the assembly tree $\mathcal{T}$. Thus, HSS operations are used at every parallel level.

Let P be the total number of processes, $\hat{P}$ be the minimum number of process within the process grids. The number of processes of each process grid at level $\mathbf{l}$ of $\mathcal{T}$ is

$$P_{\mathbf{l}} = \frac{P}{2^{\mathbf{l}-1}}.$$

Suppose a process grid with $P_{\mathbf{l}}$ processes has $O(\sqrt{P_{\mathbf{l}}})$ rows and columns. Also suppose all the HSS generators of the structured frontal matrices are of sizes $O(r) \times O(r)$.

We first review the communication costs of some basic operations. Use #messages and #words to denote the numbers of messages and words, respectively.

- For a $k \times k$ matrix, redistribution between a process grid and its parent process requires #messages $= O(1)$ and #words at most $O(\frac{k^2}{\hat{P}})$.
- On the same process grid with P processes, the multiplication of two $O(r) \times O(r)$ matrices costs #messages $= O(\frac{r}{b})$ and #words $= O(\frac{r^2}{\sqrt{\hat{P}}})$, where b is the block size used in ScaLAPACK.

Suppose a frontal matrix $\mathcal{F}$ is at level $\mathbf{l}$ of $\mathcal{T}$ and has size $n_{\mathbf{l}}$. $\mathcal{F}$ corresponds to a process grid with $P_{\mathbf{l}}$ process. Then at level l of the HSS tree T for $\mathcal{F}$, the number of processes is $\frac{P_{\mathbf{l}}}{2^{l-1}}$. The number of parallel levels of the HSS tree is $O(\log P_{\mathbf{l}})$.

To analyze the communication costs at the factorization stage, we investigate the communication at each level $\mathbf{l}$ of the assembly tree.

1. Randomized HSS construction and ULV factorization.
   In [23], it is shown that both operations cost
   $$\#\text{messages} = O\left(r \log^2 P_{\mathbf{l}}\right), \quad \#\text{words} = O\left(\frac{r^2 \log^2 P_{\mathbf{l}}}{\sqrt{\hat{P}}}\right).$$
2. Structured Schur complement computation and multiplication of the Schur complement and random vectors.
   For each process, the information is passed in a top-down order along the HSS tree. The major operations are matrix redistributions and multiplications, since the additions on the same process grid involve no communication. The redistribution costs $O(\log P_{\mathbf{l}})$ messages and $O(\frac{r^2 \log P_{\mathbf{l}}}{\hat{P}})$ words. The multiplications between HSS generators at each level cost $O(\frac{r}{b})$ messages and $O(\frac{r^2}{\sqrt{\hat{P}}})$ words. Thus
   $$\#\text{messages} = O(\log P_{\mathbf{l}}) + \sum_{l=1}^{O(\log P_{\mathbf{l}})} O\left(\frac{r}{b}\right) = O\left(\frac{r}{b} \log P_{\mathbf{l}}\right),$$
   $$\#\text{words} = O\left(\frac{r^2 \log P_l}{\hat{P}}\right) + \sum_{l=1}^{O(\log P_{\mathbf{l}})} O\left(\frac{r^2}{\sqrt{\hat{P}}}\right) = O\left(\frac{r^2 \log P_{\mathbf{l}}}{\sqrt{\hat{P}}}\right).$$
   The sampling of the structured Schur complement is performed via HSS matrix-vector multiplications. The communication costs are similar to the estimates above.
3. Skinny extend-add operation and redistribution of the skinny sampling matrices.
   The skinny matrices are of size $O(n_{\mathbf{l}}) \times O(r)$. Each process communicates with processes in the same column on the process grid. Thus
   $$\#\text{messages} = O\left(\sqrt{P_{\mathbf{l}}}\right), \quad \#\text{words} = O\left(\frac{r n_{\mathbf{l}}}{P_{\mathbf{l}}}\right).$$

The redistribution of the skinny sampling matrices costs

$$\#\text{messages} = O(1), \quad \#\text{words} = O\left(\frac{rn_{\mathbf{l}}}{\hat{P}}\right).$$

4. Extracting selected entries from HSS matrices.
For each process, the information is passed in a bottom-up order along an HSS tree. The major operations are matrix redistribution and multiplications. Similar to the communication costs of the structured Schur complement computation,

$$\#\text{messages} = O\left(\log P_{\mathbf{l}}\right) + \sum_{l=1}^{O(\log P_{\mathbf{l}})} O(\frac{r}{b}) = O(r \log P_{\mathbf{l}}),$$

$$\#\text{words} = O\left(\frac{r^2 \log P_{\mathbf{l}}}{\hat{P}}\right) + \sum_{l=1}^{O(\log P_{\mathbf{l}})} O\left(\frac{r^2}{\sqrt{\hat{P}}}\right) = O\left(\frac{r^2 \log P_{\mathbf{l}}}{\sqrt{\hat{P}}}\right).$$

The total communication costs can be estimated by summing up the above costs. The total number of messages is

$$\#\text{messages} = \sum_{\mathbf{l}=1}^{O(\log P)} \left(O\left(\sqrt{P_{\mathbf{l}}}\right) + O\left(r \log^2 P_{\mathbf{l}}\right)\right) = O\left(\sqrt{P}\right) + O\left(r \log^3 P\right).$$

The total number of words is

$$\#\text{words} = \sum_{\mathbf{l}=1}^{O(\log P)} \left(O\left(\frac{rn_{\mathbf{l}}}{\hat{P}}\right) + O\left(\frac{r^2 \log^2 P_{\mathbf{l}}}{\sqrt{\hat{P}}}\right)\right),$$

which depends on specific forms of $n_{\mathbf{l}}$. For 2D and 3D problems, $n_{\mathbf{l}} = O(\sqrt{n}/2^{\lfloor \mathbf{l}/2 \rfloor})$ and $O(n^{2/3}/2^{\lfloor \mathbf{l}/2 \rfloor})$, respectively. Thus, the total number of words for 2D problems is

$$\#\text{words} = O\left(\frac{r\sqrt{n}}{\hat{P}}\right) + O\left(\frac{r^2 \log^3 P}{\sqrt{\hat{P}}}\right)$$

and for 3D problems is

$$\#\text{words} = O\left(\frac{rn^{\frac{2}{3}}}{\hat{P}}\right) + O\left(\frac{r^2 \log^3 P}{\sqrt{\hat{P}}}\right).$$

In comparison, the parallel solver in [28] has communication costs of $\#\text{messages} = O(\sqrt{P}) + O(r \log^3 P)$ and

$$\#\text{words} = O\left(\frac{n}{\hat{P}}\right) + O\left(\frac{r\sqrt{n} \log^2 P}{\sqrt{\hat{P}}}\right)$$

for 2D problems,

$$\#\text{words} = O\left(\frac{n^{\frac{4}{3}}}{\hat{P}}\right) + O\left(\frac{rn^{\frac{2}{3}} \log^2 P}{\sqrt{\hat{P}}}\right)$$

for 3D problems. In particular, the numbers of words of our new parallel solver are smaller by factors of about $O(\frac{\sqrt{n}}{r})$ and $O(\frac{n^{\frac{2}{3}}}{r})$, respectively.

6. Numerical experiments. In this section, we show some performance results of our PRSMF solver. For convenience, we simply refer to it as PRSMF in this section. A sequences of tests are performed for Poisson's equation, a linear elasticity equation,

Helmholtz equation, and a elastic wave equation in two or three dimensions. We carried out our experiments on a Cray XC30 cluster at TOTAL E&P Research & Technology USA. Each node has 20 cores and 64 GB memory. The number of nodes available for our tests is 52. The peak performance of each core is 23.30 Gflops. We show both the weak scalability and the strong scalability of the solver, as compared with the standard parallel multifrontal solver (by setting $\mathbf{l}_s = 0$ in PRSMF), denoted PMF. We report the following performance measurements:

- Time: the runtime for the factorization of A and the solution of $Ax = b$.
- Flops and flop rate: the number of floating point operations in the factorization and the corresponding flop rate.
- Factor size: the number of nonzero entries for storing the factors.
- Peak memory per process: the maximum memory occupied by one process for any process.
- Accuracy: the relative residual $\frac{||Ax-b||_2}{||b||_2}$, where b is generated via a random exact solution.

In PRSMF, the maximum HSS rank of all the frontal matrices is also reported. We also discuss how the variation of some parameters (the switching level and the sampling size) impacts the performance. In our current implementation, the number of processes used is equal to $2^{\mathbf{l}_p-1}$, where $\mathbf{l}_p$ is the parallel switching level. Thus, the strong scaling tests below indicate the dependency of the code on $\mathbf{l}_p$. The following notation will be used for convenience:

- $\tilde{r}$: the sampling size in randomized compression (the column size of X in 3.5).
- τ: the relative tolerance of rank-revealing factorizations of the matrix-vector products as in (3.6).
- $\mathbf{l}_{\max}$: the total number of levels in the assembly tree.

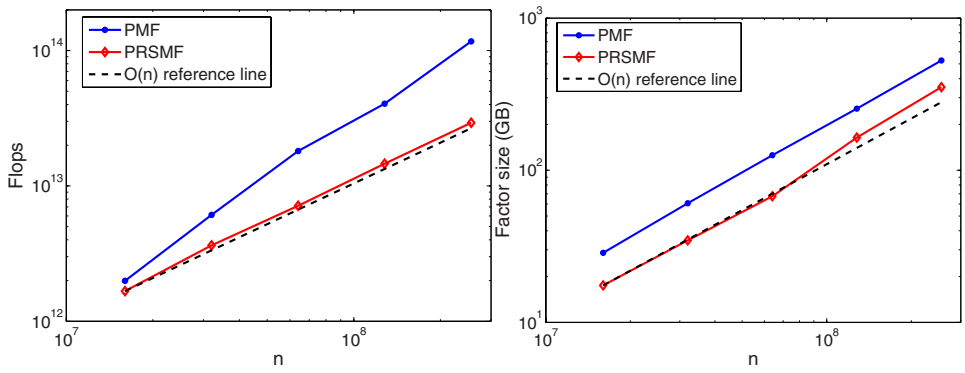
6.1. Poisson's equation and linear elasticity equation in two dimensions. We first look at two PDEs in two dimensions, Poisson's equation and a linear elasticity equation, which are known to be suitable for structured multifrontal methods [7, 33]. That is, in the multifrontal factorization, the frontal matrices have relatively small off-diagonal numerical ranks.

For 2D Poisson's equation discretized with the five-point stencil, we show the weak scalability by letting both the matrix size n and the number of processes P increase by a factor of 2. Accordingly, $\mathbf{l}_{\max}$ is increased by 1 every time so that the sizes of the smallest submeshes after nested dissection remain almost the same. The number of levels $(\mathbf{l}_{\max} - \mathbf{l}_s)$ below the switching level $\mathbf{l}_s$ remains the same, so that the factorization costs below and above $\mathbf{l}_s$ are nearly the same and the total cost is minimized [31]. (See section 6.4.) This also implies that all frontal matrices larger than a certain size are approximated by HSS forms. Here, we set $\mathbf{l}_{\max} - \mathbf{l}_s = 9$, which roughly gives the optimal factorization complexity in the tests, as shown in section 6.4. More details on the selection of $\mathbf{l}_{\max} - \mathbf{l}_s$ for structured multifrontal methods can be found in [30, 33].

In randomized compression, the sampling size is set to be $\tilde{r} = 200$, and the compression tolerance is set to $\tau = 10^{-5}$. Since our solver aims for relatively general sparse linear systems, in all these tests, it does not specifically exploit the positive definiteness of the matrices. The numerical results are shown in Table 1 and Figure 8. The CPU time for factorization and solution is shown. A weak scaling pattern can be observed. PRSMF also has a clear advantage in terms of the memory. For the matrix with $n = 256 \times 10^6$, PRSMF needs less than 1/2 of the parallel factorization time of PMF and about 1/3 of the flops. Let the upper bound of the total memory at peak

TABLE 1
Parallel weak scaling test for discretized matrices from 2D Poisson's equation.

Mesh size n		16×10^6	32×10^6	64×10^6	128×10^6	256×10^6
$\mathbf{l}_{\max}$		21	22	23	24	25
Number of processes P		32	64	128	256	512
PMF	Factorization time (s)	39.80	59.85	108.20	108.41	113.58
	Factorization flops	$1.99E12$	$6.11E12$	$1.81E13$	$4.05E13$	$1.17E14$
	Flop rate (Gflops)	50.03	102.09	167.28	373.58	1030.13
	Factor size (GB)	28.65	60.70	125.45	253.89	526.51
	Peak memory per process (GB)	3.05	3.24	3.38	3.65	4.12
	Solution time (s)	1.53	2.21	2.25	2.78	3.15
PRSMF	Factorization time (s)	32.44	37.91	50.82	49.03	55.83
	Factorization flops	$1.67E12$	$3.64E12$	$7.14E12$	$1.46E13$	$2.93E13$
	Flop rate (Gflops)	51.48	96.02	140.50	297.78	524.81
	Factor size (GB)	17.51	34.59	67.46	163.97	351.60
	Peak memory per process (GB)	2.23	2.27	2.36	2.59	3.38
	Maximum HSS rank	97	101	113	163	190
	Solution time (s)	1.58	2.05	2.89	2.75	3.39
	Relative residual	$8.34E-6$	$9.21E-6$	$5.48E-6$	$8.44E-5$	$1.20E-5$



(i) Factorization flops (ii) Factor sizes

FIG. 8. *Factorization flops and factor sizes in Table 1 for discretized matrices from the 2D Poisson's equation.*

be the product of peak memory per process and the number of processes P. Then PMF needs nearly 380 GB more memory than PRSMF when both are at peak.

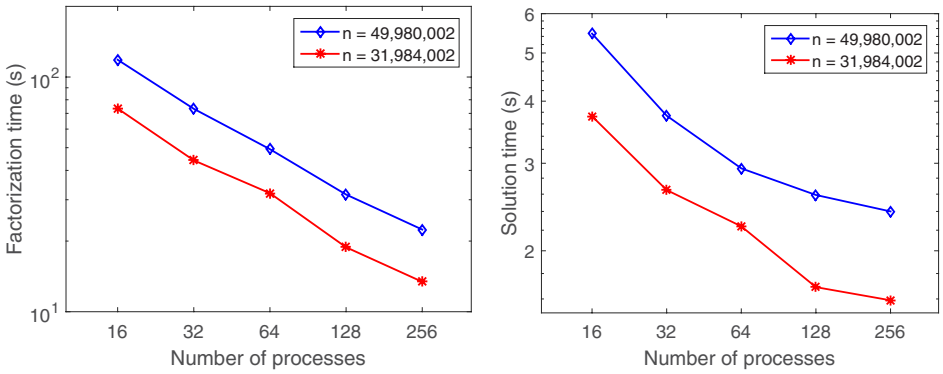
Similarly, we apply the solver to a linear elasticity equation near the incompressible limit:

$$-(\mu\Delta u + (\lambda+\mu)\nabla\nabla\cdot u) = f \text{ in } \Omega,$$
$$u = 0 \text{ on } \partial\Omega,$$

where u is the displacement vector field on a rectangular domain Ω, and λ, μ are Lamé constants. Here, $\lambda/\mu = 10^5$, which leads to ill-conditioned discretized matrices. We perform a weak scaling test similarly to the Poisson case. In PRSMF, the parameters $\tilde{r} = 200$ and $\tau = 10^{-5}$ are used, and the number of levels below the switching level is $\mathbf{l}_{\max} - \mathbf{l}_s = 12$. The test results are shown in Table 2. For the largest matrix, the

TABLE 2
Parallel weak scaling test for discretized matrices from the 2D linear elasticity equation.

Matrix size n		7,992,002	17,988,002	31,984,002	71,976,002	127,968,002
$\mathbf{l}_{\max}$		20	21	22	23	24
Number of processes P		16	32	64	128	256
PMF	Factorization time (s)	19.47	39.20	56.93	69.79	68.42
	Factorization flops	$0.58E12$	$3.11E12$	$5.35E12$	$1.23E13$	$3.27E13$
	Flop rate (Gflops)	29.79	79.34	93.98	176.24	477.93
	Factor size (GB)	11.88	31.65	56.15	137.89	250.38
	Peak memory per process (GB)	2.34	3.01	3.37	3.85	4.15
	Solution time (s)	0.92	1.51	2.59	2.94	3.58
PRSMF	Factorization time (s)	17.20	27.00	31.75	44.04	42.98
	Factorization flops	$0.53E12$	$1.61E12$	$2.68E12$	$6.27E12$	$1.13E13$
	Flop rate (Gflops)	30.81	59.63	84.40	142.36	262.93
	Factor size (GB)	9.46	22.81	43.40	95.69	180.25
	Peak memory per process (GB)	2.02	2.45	2.50	2.73	2.96
	Maximum HSS rank	126	113	122	149	158
	Solution time (s)	0.83	1.56	2.07	3.04	3.42
	Relative residual	$2.23E-4$	$4.30E-4$	$3.95E-4$	$2.88E-4$	$2.93E-4$



(i) Strong scaling of factorization time (ii) Strong scaling of solution time

FIG. 9. *Strong scaling tests with PRSMF for two discretized matrices from the 2D linear elasticity equation.*

parallel factorization time of PRSMF is less than 2/3 of the time of PMF, and the flop count is less than 1/2.

For two matrices of size 31,984,002 and 49,980,002, we display the strong scaling results of the PRSMF by changing the number of processes. Strong scaling patterns can be observed in both the factorization stage and the solution stage. See Figure 9. When the number of processes P doubles, the parallel factorization time reduces by factors close to 1.7. We also display the parallel efficiency of PRSMF in Figure 10 for matrices of sizes 7,992,002 and 17,988,002. With 16 processes, the parallel efficiency is slightly over 70%.

6.2. 3D Helmholtz equation. Next, consider the numerical solution of the 3D Helmholtz equation

$$\left(-\Delta - \frac{\omega^2}{v^2}\right) u = f,$$

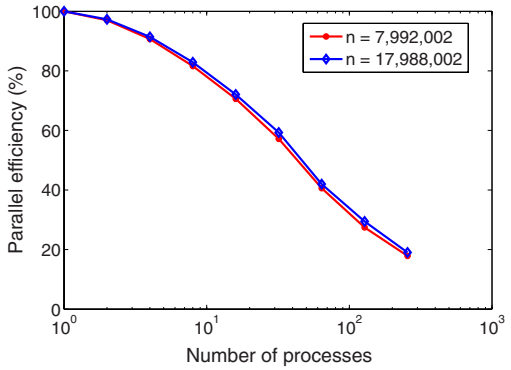


FIG. 10. *Parallel efficiency of PRSMF for two discretized matrices from the 2D linear elasticity equation.*

where ω is the angular frequency, v is the seismic velocity field, and f is the forcing term. The equation is discretized on 3D tetrahedron meshes with the continuous Galerkin method. The number of sample points per wavelength is fixed to be 5. The perfectly matched layer boundary condition is used. The P wave speed is 3.4 km/s, and ω slowly increases from about 6 Hz to 10 Hz. The meshes have 20, 40, 60, and 100 million elements, respectively. The sizes of the corresponding discretized matrices range from 3,233,795 to 15,794,036. The numbers of nonzero entries in the matrices are also reported. In seismic imaging, the Helmholtz equation is often solved with relatively low accuracies. Here, we conduct the tests in single precision. The compression tolerance in PRSMF is $\tau = 10^{-3}$. The sampling size $\tilde{r}$ is set to be 1200, 1500, 1800, and 2100 for these four matrices, respectively. The number $(\mathbf{l}_{\max} - \mathbf{l}_s)$ of levels below the switching level is 9 or 10. Table 3 shows the results.

PRSMF has a significant advantage in the flop count and storage. This can also be observed from Figure 11, and the results are consistent with the theoretical estimates in [31]. For the third matrix, PRSMF needs about 1/4 of the flops and about 1/2 of the storage for the factors. The factorization time of the new solver is always shorter. The great memory advantage of the new solver is very attractive since the memory issue plays a critical role in the design of 3D direct PDE solvers. For the largest size, PMF runs out of memory. The flop rate of PRSMF is lower due to the large number of small dense matrices (HSS generators and skinny matrices). The communication pattern of the new solver is also more complicated. On the other hand, PMF works mainly on larger dense matrices and exploits the BLAS3 operations intensively. Iterative refinements can be used to improve the accuracy of the solver. For the largest matrix, 10 steps of iterative refinements reduce the relative residual to around 10^{-6}.

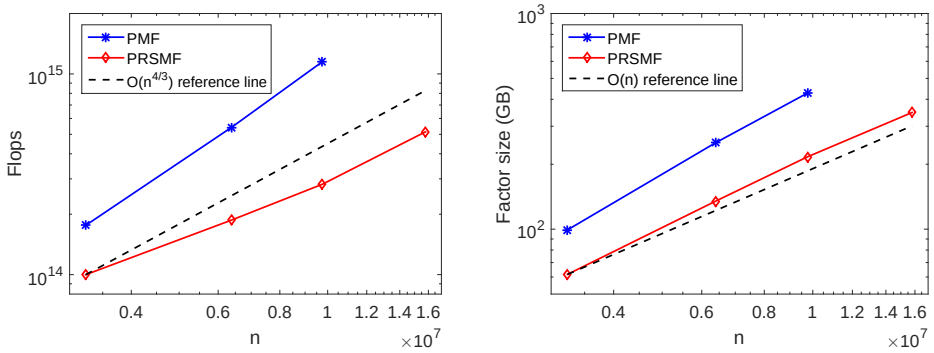
For two of the matrices, we also display the strong scaling results of PRSMF in Figure 12. For each matrix, when the number of processes P doubles, the parallel factorization time reduces by factors about $1.7 \sim 1.8$.

6.3. 3D elastic wave equation. Then consider the solution of the 3D elastic wave equation

$$-\rho\omega^2 u - \nabla \cdot (c : \nabla u) = f,$$
$$\nu \cdot (c : \nabla u) = g,$$

TABLE 3
Parallel weak scaling tests for discretized matrices from the 3D Helmholtz equation.

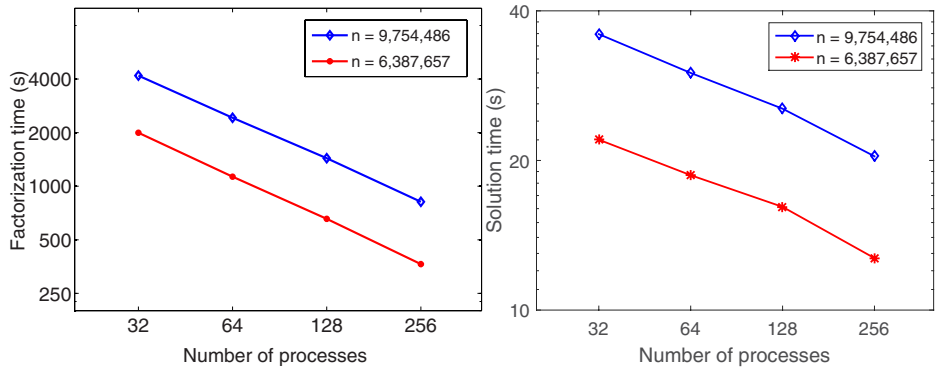
Matrix size n		3,233,795	6,387,657	9,754,486	15,794,036
Number of nonzeros		50,233,223	99,696,607	152,290,784	247,634,526
$\mathbf{l}_{\max}$		14	15	16	17
Number of processes P		64	128	256	512
PMF	Factorization time (s)	430.46	871.28	1078.77	
	Factorization flops	$1.79E14$	$5.32E14$	$1.35E15$	
	Flop rate (Gflops)	415.81	610.69	1251.48	
	Factor size (GB)	103.74	256.34	420.78	
	Peak memory per process (GB)	6.05	7.14	7.24	
	Solution time(s)	8.67	17.93	24.82	
	Relative residual	$4.50E-6$	$4.97E-5$	$4.98E-5$	
PRSMF	Factorization time (s)	376.53	655.93	817.22	981.67
	Factorization flops	$9.96E13$	$1.84E14$	$2.83E14$	$5.13E14$
	Flop rate (Gflops)	264.52	280.58	346.78	522.88
	Factor size (GB)	62.39	138.85	212.96	350.32
	Peak memory per process (GB)	5.58	5.70	5.66	6.09
	Maximum HSS rank	1077	1342	1653	2021
	Solution time (s)	7.82	15.58	25.32	33.04
	Relative residual	$2.48E-3$	$9.46E-3$	$1.76E-2$	$1.38E-2$
	Number of iterative refinement steps	3	7	9	10
	Relative residual after refinements	$1.50E-6$	$2.86E-6$	$1.76E-6$	$1.20E-6$



(i) Factorization flops (ii) Factor sizes

FIG. 11. *Factorization flops and factor sizes in Table 3 for discretized matrices from the 3D Helmholtz equation.*

where ρ is the density, ω is the frequency, c is the elastic stiffness tensor, u is the displacement, ν is the outward normal to the boundary of the domain, and the colon represents tensor contraction. The equation is discretized on meshes with 2, 5, 10, and 20 million elements in the finite element method. The sample points per wavelength are fixed to be 5 for all these meshes. Single precision is used. PRSMF uses a compression tolerance $\tau = 10^{-3}$ and a uniform sampling size $\tilde{r} = 1000$. Also, $\mathbf{l}_{\max} - \mathbf{l}_s = 9$. The results are given in Table 4, where we can observe clear advantages of the new solver in terms of the speed and memory. For the matrix with $n = 9{,}701{,}385$, PRSMF takes less than 1/3 of the flops of PMF, less than 3/5 of the parallel factorization



(i) Strong scaling of factorization time (ii) Strong scaling of solution time

FIG. 12. *Strong scaling tests with PRSMF for two discretized matrices from the 3D Helmholtz equation.*

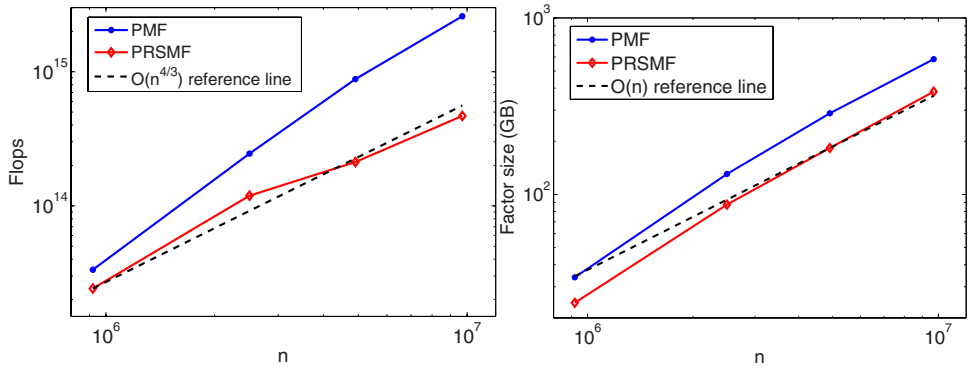
TABLE 4
Parallel weak scaling tests for discretized matrices from the 3D elastic wave equation.

Matrix size		919,683	2,496,888	4,907,088	9,701,385
Number of nonzeros		41,798,079	114,950,592	227,587,086	452,099,007
$\mathbf{l}_{\max}$		12	13	14	15
Number of processes P		64	128	256	512
PMF	Factorization time (s)	144.80	509.05	1045.07	1853.94
	Factorization flops	$3.34E13$	$2.45E14$	$8.80E14$	$2.59E15$
	Flop rate (Gflops)	230.68	481.34	842.08	1397.02
	Factor size (GB)	33.93	130.48	288.27	583.44
	Peak memory per process (GB)	2.11	4.37	5.24	5.47
	Solution time (s)	3.85	9.61	21.50	30.38
	Relative residual	$3.46E-7$	$4.84E-7$	$1.59E-7$	$1.87E-7$
PRSMF	Factorization time(s)	109.76	304.15	457.04	680.51
	Factorization flops	$2.42E13$	$1.19E14$	$2.12E14$	$4.68E14$
	Flop rate (Gflops)	220.45	391.26	463.85	687.72
	Factor size (GB)	24.34	87.64	183.38	382.23
	Peak memory per process (GB)	1.61	3.29	3.65	3.94
	Maximum HSS rank	477	813	901	976
	Solution time (s)	3.79	9.43	18.61	28.40
	Relative residual	$2.81E-6$	$4.33E-6$	$8.05E-6$	$1.35E-6$

time, and about 2/3 of the storage. Let the upper bound of the total memory at peak be the product of the peak memory per process and the number of processes P. Then PMF may need nearly 780 GB more memory than PRSMF when both are at peak. Also, when n increases from 2,496,888 to 9,701,385, the flop count of PRSMF increases by a factor around 2, while the count of PMF increases by a factor around 3.

6.4. Varying the switching level and sampling size. As mentioned in section 6.1, the switching level $\mathbf{l}_s$ is a user specified level beyond which HSS methods are applied. As shown in [31], $\mathbf{l}_s$ is chosen so that $\mathbf{l}_{\max} - \mathbf{l}_s$ is nearly constant. This means that we can use a small size n to roughly determine the optimal switching level $\mathbf{l}_s$.

For example, for the 3D Helmholtz equation in section 6.2, consider the matrix of size n =3,233,795. With 64 processes, $\tau = 10^{-3}$, $\tilde{r} = 1200$, and $\mathbf{l}_{\max} = 14$, we vary



(i) Factorization flops (ii) Factor sizes

FIG. 13. *Factorization flops and factor sizes in Table 4 for discretized matrices from the 3D elastic wave equation.*

TABLE 5

Performance of PRSMF for the discretized 3D Helmholtz equation in section 6.2 with different switching levels $\mathbf{l}_s$ *for* $n = 3{,}233{,}795$.

$\mathbf{l}_s$	3	4	5	6	7
Factorization flops	$1.25E14$	$1.06E14$	$9.96E13$	$1.03E14$	$1.09E14$
Factor size (GB)	81.31	71.45	62.39	63.01	69.28
Relative residual	$1.60E-3$	$3.64E-3$	$2.48E-3$	$1.46E-2$	$4.81E-2$

TABLE 6

Performance of PRSMF for the discretized 3D Helmholtz equation in section 6.2 with different $\tilde{r}$ *and* τ *for* $n = 3{,}233{,}795$.

Sampling size $\tilde{r}$	1000	1200	1400	1600
Compression tolerance τ	$1E-2$	$1E-3$	$1E-4$	$1E-5$
Factorization flops	$8.66E13$	$9.96E13$	$1.08E14$	$1.13E14$
Factor size (GB)	53.87	62.39	72.51	81.62
Maximum HSS rank	658	1077	1317	1544
Relative residual	$5.08E-2$	$2.48E-3$	$9.62E-4$	$1.82E-5$

$\mathbf{l}_s$ and compare the cost of the factorization and the storage. See Table 5. Clearly, $\mathbf{l}_s = 5$ or $\mathbf{l}_{\max} - \mathbf{l}_s = 9$ gives the optimal complexity and storage.

The sampling size $\tilde{r}$ is usually set to accommodate the desired accuracy of the randomized HSS approximation. An increase in $\tilde{r}$ will lead to the increase of the cost for matrix-vector multiplications and RRQR factorizations. When the compression tolerance τ reduces accordingly, higher accuracies can be achieved. To see this, we test the same matrix as in Table 5, but with varying $\tilde{r}$ and τ and with fixed $\mathbf{l}_s = 5$. The number of processes is 64. The results are given in Table 6. Hence, $\tilde{r}$ and τ can be used to control the accuracy of the solution. If we use lower accuracies for faster factorization and solution, we may use iterative refinement to improve the accuracy, or use the solver as a preconditioner.

7. Conclusions. In this paper, we have presented a distributed-memory algebraic randomized structured multifrontal solver with two levels of tree parallelism. Hierarchical structured methods are shown to be naturally scalable. Unlike previous efforts in [28], the new solver enhances both the computation and the communication

by using randomization to avoid dense intermediate matrices. It can also handle more general matrices. Several parallel HSS algorithms have been developed for the solver. The solver achieves the desired complexity and shows nice scalability in testing some discretized PDEs.

The implementation of this solver is relatively complex. It is built upon multiple components such as multifrontal methods, HSS algorithms, and randomized methods. The efficient implementations of some components are extensively studied by the scientific computing community. Here, we build a systematic fast parallel structured sparse solver, and also provide some new parallel implementations of several useful strategies in sparse and structured solutions. In the near future, we hope to make the individual components and even the entire solver work as black-box codes so that other researchers can conveniently use them for different tasks without knowing the technical details. Such codes will be made publicly available. It is also possible to simplify the implementation using simpler structures such as the HODLR form [1].

Due to the complex scale of the project, no pivoting is integrated yet. Another reason for this is that it is not clear how pivoting affects the structures since the structures depend on proper ordering. This will be thoroughly studied in future work. Out future work will also include an adaptive scheme for choosing sampling sizes at different levels of the assembly tree, the replacement of intermediate HSS forms by multilayer structures [32], the use of batched BLAS for multiple small BLAS operations, as well as more extensive direct solution and preconditioning tests.

Acknowledgments. We thank Xiao Liu, Fabien Peyruss, and Jia Shi for helping with the numerical tests and thank TOTAL E&P Research & Technology USA for providing the computing resource. We are also grateful to the three anonymous referees for the valuable suggestions.

REFERENCES

[1] S. Ambikasaran and E. F. Darve, *An $O(N \log N)$ fast direct solver for partial hierarchically semi-separable matrices*, J. Sci. Comput., 57 (2013), pp. 477–501.

[2] P. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L'Excellent, and C. Weisbecker, *Improving multifrontal methods by means of block low-rank representations*, SIAM J. Sci. Comput., 37 (2015), pp. A1451–A1474.

[3] P. Amestoy, I. S. Duff, and J. Y. L'Excellent, *Multifrontal parallel distributed symmetric and unsymmetric solvers*, Comput. Methods Appl. Mech. Engrg. 184 (2000), pp. 501–520.

[4] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, et al., *ScaLAPACK User's Guide*, SIAM, Philadelphia, PA, 1997.

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

[6] S. Chandrasekaran, P. Dewilde, M. Gu, and T. Pals, *A fast ULV decomposition solver for hierarchically semiseparable representations*, SIAM J. Matrix Anal. Appl., 28 (2006), pp. 603–622.

[7] S. Chandrasekaran, P. Dewilde, M. Gu, and N. Somasunderam, *On the numerical rank of the off-diagonal blocks of Schur complements of discretized elliptic PDEs*, SIAM J. Matrix Anal. Appl. 31 (2010), pp. 2261–2290.

[8] R. Clint Whaley, *Basic Linear Algebra Communication Subprograms: Analysis and Implementation Across Multiple Parallel Architectures*, LAPACK Working Note 73, University of Tennessee, 1994.

[9] I. S. Duff and J. K. Reid, *The multifrontal solution of indefinite sparse symmetric linear*, ACM Trans. Math. Software, 9 (1983), pp. 302–325.

[10] J. A. George, *Nested dissection of a regular finite element mesh*, SIAM J. Numer. Anal., 10 (1973), pp. 345–363.

[11] J. A. George, J. W. H. Liu, and E. Ng, *Communication results for parallel sparse Cholesky factorization on a hypercube*, Parallel Comput., 10 (1989), pp. 287–298.

[12] N. Gibbs, W. Poole, and P. Stockmeyer, *An algorithm for reducing the bandwidth and profile of a sparse matrix*, SIAM J. Sci. Comput., 13 (1976), pp. 236–250.

[13] J. R. Gilbert and S.-H. Teng, *MESHPART, A MATLAB Mesh Partitioning and Graph Separator Toolbox*, http://aton.cerfacs.fr/algor/Softs/MESHPART.
[14] A. Gillman and P. G. Martinsson, *A direct solver with $O(N)$ complexity for variable coefficient elliptic PDEs discretized via a high-order composite spectral collocation method*, SIAM J. Sci. Comput., 36 (2014), pp. A2023–A2046.
[15] L. Grasedyck, R. Kriemann, and S. Le Borne, *Domain-decomposition based $\mathcal{H}$-LU preconditioners*, in Domain Decomposition Methods in Science and Engineering XVI, O. B. Widlund and D. E. Keyes, eds., Lect. Notes. Comput. Sci. Eng. 55, Springer, New York, 2006, pp. 661–668.
[16] M. Gu and S. C. Eisenstat, *Efficient algorithms for computing a strong-rank revealing QR factorization*, SIAM J. Sci. Comput., 17 (1996), pp. 848–869.
[17] W. Hackbusch, B. N. Khoromskij, and R. Kriemann, *Hierarchical matrices based on a weak admissibility criterion*, Computing, 73 (2004), pp. 207-243.
[18] N. Halko, P.G. Martinsson, and J. Tropp, *Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions*, SIAM Rev., 53 (2011), pp. 217–288.
[19] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., 20 (1998), pp. 359–392.
[20] X. S. Li and J. W. Demmel, *SuperLU_DIST A scalable distributed-memory sparse direct solver for unsymmetric linear systems*, ACM Trans. Math. Software, 29, (2002), pp. 110–140.
[21] E. Liberty, F. Woolfe, P. G. Martinsson, V. Rokhlin, and M. Tygert, *Randomized algorithms for the low-rank approximation of matrices*, Proc. Natl. Acad. Sci. USA, 104 (2007), pp. 20167–20172.
[22] J. W. H. Liu, *The multifrontal method for sparse matrix solution: Theory and practice*, SIAM Rev., 34 (1992), pp. 82–109.
[23] X. Liu, J. Xia, and M. V. de Hoop, *Parallel randomized and matrix-free direct solvers for large structured dense linear systems*, SIAM J. Sci. Comput., 38, (2016), pp. S508–S538.
[24] P. G. Martinsson, *A fast randomized algorithm for computing a hierarchically semiseparable representation of a matrix*, SIAM. J. Matrix Anal. Appl., 32 (2011), pp. 1251–1274.
[25] F. Pellegrini and J. Roman, *Sparse matrix ordering with SCOTCH*, in Proceedings of HPCN'97, Vienna, Austria, Lecture Notes in Comput. Sci. 1225, Springer, New York, 1997, pp. 370–378.
[26] A. Pothen and C. Sun, *A mapping algorithm for parallel sparse Cholesky factorization*, SIAM J. Sci. Comput., 14 (1993), pp. 1253–1257.
[27] P. Schmitz and L. Ying, *A fast direct solver for elliptic problems on general meshes in 2D*, J. Comput. Phys., 231 (2012), pp. 1314–1338.
[28] S. Wang. X. S. Li, F. H. Rouet, J. Xia, and M. V. de Hoop, *A parallel geometric multifrontal solver using hierarchically semiseparable structure*, ACM Trans. Math. Software, 42 (2016).
[29] S. Wang, X. S. Li, J. Xia, Y. Situ, and M. V. de Hoop, *Efficient scalable algorithms for solving dense linear systems with hierarchically semiseparable structures*, SIAM J. Sci. Comput., 35 (2013), pp. C519–C544.
[30] J. Xia, *Efficient structured multifrontal factorization for general large sparse matrices*, SIAM J. Sci. Comput., 35 (2013), pp. A832–A860.
[31] J. Xia, *Randomized sparse direct solvers*, SIAM J. Matrix Anal. Appl., 34 (2013), pp. 197–227.
[32] J. Xia, *Multilayer Hierarchically Semiseparable Structures*, GMIG Report 15–16, Purdue University, 2015.
[33] J. Xia, S. Chandrasekaran, M. Gu, and X. S. Li, *Superfast multifrontal method for large structured linear systems of equations*, SIAM J. Matrix Anal. Appl., 31 (2009), pp. 1382–1411.
[34] J. Xia, S. Chandrasekaran, M. Gu, and X. S. Li, *Fast algorithms for hierarchically semiseparable matrices*, Numer. Linear Algebra Appl., 17 (2010), pp. 953–976.
[35] J. Xia, Y. Xi, and M. Gu, *A superfast structured solver for Toeplitz linear systems via randomized sampling*, SIAM J. Matrix Anal. Appl., 33 (2012), pp. 837–858.
[36] Z. Xin, J. Xia, M. V. de Hoop, S. Cauley, and V. Balakrishnan, *Parallel Randomized Structured Multifrontal Method for General Sparse Matrices*, GMIG Report 14–17, Purdue University, 2014.
[37] Z. Xin, J. Xia, M. V. de Hoop, S. Cauley, and V. Balakrishnan, *Scalable Randomized Structured Multifrontal Method for Large-Scale Sparse Direct Solutions*, GMIG Report 15–14, Purdue University, 2015.
[38] Z. Xin, J. Xia, M. V. de Hoop, S. Cauley, and V. Balakrishnan, *A Distributed-Memory Randomized Structured Multifrontal Method for Sparse Direct Solutions*, GMIG Report 9, Rice University, 2016, pp. 169–189.