SIAM J. SCI. COMPUT.
Vol. 38, No. 5, pp. S508–S538

PARALLEL RANDOMIZED AND MATRIX-FREE DIRECT SOLVERS FOR LARGE STRUCTURED DENSE LINEAR SYSTEMS*

XIAO LIU†, JIANLIN XIA‡, AND MAARTEN V. DE HOOP†

Abstract. We design efficient and distributed-memory parallel randomized direct solvers for large structured dense linear systems, including a fully matrix-free version based on matrix-vector multiplications and a partially matrix-free one. The dense coefficient matrix A has an off-diagonal low-rank structure, as often encountered in practical applications such as Toeplitz systems and discretized integral and partial differential equations. A distributed-memory parallel framework for randomized structured solution is shown. Scalable adaptive randomized sampling and hierarchical compression algorithms are designed to approximate A by hierarchically semiseparable (HSS) matrices. Systematic process grid storage schemes are given for different HSS forms. Parallel hierarchical algorithms are proposed for the resulting HSS forms. As compared with existing work on parallel HSS methods, our algorithms have several remarkable advantages, including the matrix-free schemes that avoid directly using dense A, a synchronized adaptive numerical rank detection, the integration of additional structures into the HSS generators, and much more flexible choices of the number of processes. Comprehensive analysis is conducted and shows that the communication costs are significantly reduced by up to an order of magnitude. Furthermore, we improve the original matrix-free HSS construction algorithm by avoiding some instability issues and by better revealing the nested rank structures. Tests on large challenging dense discretized matrices related to three-dimensional scattering fully demonstrate the superior efficiency and scalability of the direct solvers. For example, for a $10^6 \times 10^6$ dense discretized matrix, the partially matrix-free HSS construction takes about 4,500 seconds with 512 processes, while the solution takes only 0.63 second. The storage savings is more than 30 times. The fully matrix-free solver takes slightly longer but is more flexible and accurate.

Key words. distributed memory, scalable algorithm, randomized compression, matrix-free direct solver, HSS matrix, tree structure

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

DOI. 10.1137/15M1023774

1. Introduction.

Dense linear system solvers are fundamental and critical tools in scientific computing and numerical simulations. Many recent studies of dense direct solvers focus on exploiting the rank structures in the systems. Our work is based on hierarchically semiseparable (HSS) representations [6, 36], which are shown to be very useful for solving some dense problems such as Toeplitz matrices (in Fourier space) and certain discretized matrices (e.g., discretized integral equations and Schur complements in the factorizations of discretized PDEs) [6, 27, 28, 38, 35]. HSS matrices are closely related to other rank-structured representations such as $\mathcal{H}$-matrices [3, 13], $\mathcal{H}^2$-matrices [12], sequentially semiseparable matrices [5], and quasi-separable matrices [8].

Some key HSS algorithms include HSS construction, HSS ULV factorization [6], and ULV solution. For a given HSS form, the cost to factorize it is $O(r^2 n)$, where n is the matrix size and r is the maximum numerical rank of all the off-diagonal blocks.

*Received by the editors June 1, 2015; accepted for publication (in revised form) February 5, 2016; published electronically October 27, 2016. This work was partially supported by the sponsors (BGP, ExxonMobil, PGS, Statoil, and Total) of the Geo-Mathematical Imaging Group.
http://www.siam.org/journals/sisc/38-5/M102377.html

†Department of Mathematics, Purdue University, West Lafayette, IN 47907 (liu867@purdue.edu, mdehoop@purdue.edu).

‡Department of Mathematics and Department of Computer Science, Purdue University, West Lafayette, IN 47907 (xiaj@purdue.edu). The research of this author was supported in part by NSF CAREER Award DMS-1255416.

After the factorization, it takes $O(rn)$ flops to solve a linear system. Therefore, when r is small (we say that the matrix has a low-rank property), HSS solutions are significantly faster than standard dense ones.

Thus, it is critical to quickly construct an HSS representation or approximation for a given matrix A with the low-rank property. A direct construction from a dense matrix costs $O(rn^2)$ flops [33, 36]. To enhance the efficiency, a parallel HSS construction algorithm is designed in [28], which still requires the formation of the explicit dense matrix A. However, in many practical applications, A is usually unavailable or too expensive to be formed explicitly, while its product with vectors can be conveniently obtained. Examples of such situations include some matrices from spectral discretizations [24] and interface problems [37].

One way to avoid using the dense matrix A and to reduce the construction cost is to use randomized low-rank compression with matrix-vector products [14, 19, 29]. For a matrix block with numerical rank deficiency, an approximate low-rank decomposition can be computed with great efficiency and reliability via randomized sampling. These techniques are used to develop randomized HSS constructions, including a partially matrix-free scheme [18, 34, 38] and a fully matrix-free one [17, 32]. The partially matrix-free scheme can further take advantage of additional special structures in the HSS representation to reduce the subsequent computational cost. It uses only $O(r)$ matrix-vector multiplications and has an $O(r^2n)$ HSS construction cost. However, it needs to access $O(rn)$ entries of A during the compression process. The fully matrix-free scheme acquires data via matrix-vector multiplications only and has an $O(r^2n\log n)$ HSS construction cost. It uses $O(r\log n)$ matrix-vector multiplications. In [32], an adaptive procedure [14] to detect the numerical rank for a given tolerance is also built into the HSS construction. The factorization of the resulting HSS form also has better stability.

1.1. Main results. In this paper, we develop distributed-memory massively parallel randomized HSS constructions, as well as the factorization and solution based on the resulting structures. To obtain essential improvements, we abandon the global dense storage and the related dense operations. Distributed-memory parallelism and data distribution strategies are designed for HSS operations based on randomized sampling. The parallelization of a sequence of important computations is given, such as

- randomized compression, including a structure-preserving version and a version with orthonormalization,
- synchronized adaptive numerical rank detection,
- randomized hierarchical compression, and
- factorization of the resulting structured representations.

Full parallelism and PBLAS-3 performance are exploited without any sacrifice in the algorithm complexity. In addition, only compact structured matrices are stored. Both partially and fully matrix-free parallel direct HSS solvers are designed, which are suitable for different applications depending on the availability of A.

In the partially matrix-free version, distributed storage and computational schemes are introduced to take advantage of the interior special structures in the HSS representation. Another benefit in contrast with the parallel HSS algorithms in [28] is that we allow more flexible choices of the number of processes, not necessarily restricted by the hierarchical tree structure. The scheme in [28] instead requires the number of processes to be larger than or equal to the number of leaf nodes used in the tree structure corresponding to the HSS form.

For the fully matrix-free construction scheme, we make further improvements to the original algorithms in [17, 32]. We avoid the use of pseudoinverses not only to gain

better efficiency but also to eliminate a potential instability issue. The extraction of low-rank bases is based on stable compression techniques. In [17, 32], the hierarchical compression may yield low-rank approximations that are not compact enough. Here, a new way of conducting hierarchical compression is used so that the nested off-diagonal basis matrices fully respect the actual off-diagonal numerical ranks. We also incorporate parallel synchronized rank detection into the hierarchical compression.

Careful analysis of the parallel performance is given, including the computational costs, communication costs, and parallel runtime. As compared with the HSS compression algorithms with global dense storage in [28], the new parallel randomized approaches reduce the computation cost, storage, and communication cost by nearly a factor of $O(\frac{n}{r})$. For structured matrices with fast matrix-vector products, our methods have the capability of solving much larger problems with the same computing resources.

We test the performance of the parallel algorithms by solving challenging discretized systems arising from Foldy–Lax equations with a two-dimensional (2D) configuration of scatterers in three dimensions. The tests show a nice scalability, and the results match our theoretical estimates on both the computation and the communication costs. For example, for the partially matrix-free version with $n = 10^6$ and 512 processes, after about 4,500 seconds for the HSS construction and 280 seconds for the factorization, it needs only 0.63 second to solve a linear system, or 8.92 seconds for 128 right-hand sides. The fully matrix-free version takes slightly longer, but is more flexible and a little more accurate. The tests also suggest that our solvers can be naturally used to handle dense intermediate Schur complements in the direct factorizations of some large sparse three-dimensional (3D) discretized problems such as Helmholtz equations.

The outline of the presentation is as follows. Section 2 discusses a framework of distributed-memory HSS structures via randomization. In section 3, we show how to perform randomized low-rank compression in parallel. The two randomized matrix-free HSS direct solvers are given in detail in section 4, followed by the analysis of the performance in section 5. Some applications and numerical tests are then shown in section 6.

2. A framework of distributed-memory HSS structures via randomization. We first show the basic distributed-memory parallelism for HSS structures via randomization. An HSS matrix A with a corresponding HSS tree $\mathcal{T}$ can be understood as follows, and the reader is referred to [6, 33, 36] for a rigorous definition. Assume $\mathcal{T}$ is a postordered binary tree with nodes $i = 1, 2, \ldots$. Partition A following a sequence of index subsets $\mathbf{I}_i \subset \{1 : n\}$. Each parent/nonleaf node index subset $\mathbf{I}_i$ satisfies $\mathbf{I}_i = \mathbf{I}_{c_1} \cup \mathbf{I}_{c_2}$, where c_1 and c_2 represent the left and right children of i, respectively. The nodes are organized into L levels following the parent-child relationship, with the root at level 0. An HSS form for A is defined by a sequence of HSS generators $D_i, B_i, U_i, V_i, R_i, W_i$:

- $D_i \equiv A|_{\mathbf{I}_i \times \mathbf{I}_i}$ is a diagonal block, where $A|_{\mathbf{I}\times\mathbf{J}}$ represents a submatrix selected from A based on the row index set $\mathbf{I}$ and the column index set $\mathbf{J}$;
- U_i is a column basis matrix for $A|_{\mathbf{I}_i \times \mathbf{I}_i^c}$, where $\mathbf{I}_i^c \equiv \{1 : n\} \backslash \mathbf{I}_i$ denotes the complement of $\mathbf{I}_i$ in $\{1 : n\}$;
- V_i is a column basis matrix for $(A|_{\mathbf{I}_i^c \times \mathbf{I}_i})^T$.

The off-diagonal blocks $A|_{\mathbf{I}_i \times \mathbf{I}_i^c}$ and $A|_{\mathbf{I}_i^c \times \mathbf{I}_i}$ are said to be HSS blocks, and their maximum (numerical) rank is called the *HSS rank* of A.

These generators satisfy a nested property. That is, for any nonleaf node i, the diagonal block and off-diagonal basis can be represented in terms of those associated

with its children c_1, c_2:

- the diagonal block is recursively defined as

$$D_i = \begin{pmatrix} A|_{\mathbf{I}_{c_1}\times\mathbf{I}_{c_1}} & A|_{\mathbf{I}_{c_1}\times\mathbf{I}_{c_2}} \\ A|_{\mathbf{I}_{c_2}\times\mathbf{I}_{c_1}} & A|_{\mathbf{I}_{c_2}\times\mathbf{I}_{c_2}} \end{pmatrix} = \begin{pmatrix} D_{c_1} & U_{c_1}B_{c_1}V_{c_2}^T \\ U_{c_2}B_{c_2}V_{c_1}^T & D_{c_2} \end{pmatrix}; \tag{2.1}$$

- the off-diagonal basis matrices are recursively defined as

$$U_i = \begin{pmatrix} U_{c_1}R_{c_1} \\ U_{c_2}R_{c_2} \end{pmatrix}, \quad V_i = \begin{pmatrix} V_{c_1}W_{c_1} \\ V_{c_2}W_{c_2} \end{pmatrix}, \tag{2.2}$$

  where $U_{c_1}, U_{c_2}, V_{c_1}, V_{c_2}$ are the off-diagonal basis matrices associated with the children. Thus, upper level U, V matrices are available through the leaf-level U, V generators together with all the R, W generators.

The HSS tree can be supplied by the user or can be generated from a recursive bisection of the matrix index set. In our tests, we generate the tree in the code. If no reordering of the matrix is needed, then the tree generation is trivial. Otherwise, the matrix is reordered to enable effective compression of interactions corresponding to different partitions. Geometrically, if A is a discretized matrix and if the mesh connections are localized, we partition the domain along the most elongated direction. This is quite convenient for one-dimensional (1D) problems and for multidimensional problems on regular meshes, and the cost is negligible. For more general cases, this can be done by applying various parallel graph partitioning packages such as ParMETIS [21] and PT-Scotch [23] to bisect the mesh (or adjacency graph). This is a well-studied topic, and the parallel performance depends on the actual problem and those packages. As compared with the HSS compression, such graph partitioning cost is usually much smaller in practice.

2.1. Distribution scheme for HSS structures. We are interested in designing parallel HSS algorithms on distributed-memory computers with message-passing communications. In [28], a nested parallel strategy is proposed for distributing HSS operations. The outer layer parallelism is the distributed HSS tree, and the inner layer is the distributed HSS generators. The two distribution schemes can be formulated in a general setting.

1. *Distributed HSS tree.* Each node of the HSS tree $\mathcal{T}$ is mapped to a group of processes. Let G_i denote the set of processes used for the matrices associated with a node i. (Such matrices include the generators and the intermediate matrices used for the HSS construction.) For parallel computations between a pair of siblings c_1 and c_2, we require $G_{c_1} \cap G_{c_2} = \emptyset$. A parent set is constructed from the child sets: $G_i \subset G_{c_1} \cup G_{c_2}$. Such an idea is supported by the MPI communicator [20] and the BLACS context [2].
2. *Distributed HSS generators.* To use the standard 2D block-cyclic distribution, G_i is organized as a 2D process grid. Existing optimized dense matrix kernels can be used to speed up the dense matrix operations involving the HSS generators.

Communicating a matrix to a different process grid is called *redistribution*. To control the redistribution cost, vertical or horizontal concatenations are used to form larger process grids from smaller ones:

$$G_i = \begin{pmatrix} G_{c_1} \\ G_{c_2} \end{pmatrix} \quad \text{or} \quad G_i = \begin{pmatrix} G_{c_1} & G_{c_2} \end{pmatrix}.$$

This design reduces the redistribution among G_{c_1}, G_{c_2}, G_i to a point-to-point exchange of local storage. The process grids used in [28] are roughly square.

Here, we illustrate the redistribution in terms of more general and flexible process grid sizes. Look at a single matrix S distributed on a $g_1 \times g_2$ process grid G_{c_1}. The 2D block-cyclic scheme partitions S into multiple blocks $S_{j,k}$ and maps them cyclically to G_{c_1}. A block $S_{j,k}$ is stored in the process $p_{l,m}$ if

$$j \equiv l \mod g_1, \quad k \equiv m \mod g_2.$$

- To communicate S from G_{c_1} to another $g_1 \times g_2$ process grid G_{c_2}, a process in G_{c_1} simply sends the local submatrix to the corresponding process in G_{c_2} with the same relative position.
- To communicate S from G_{c_1} to the parent grid G_i generated by vertical concatenation and with size $2g_1 \times g_2$, the (l,m)th process in G_{c_1} sends to the (l,m)th process in G_{c_2} the concatenation of all the blocks $S_{j,k}$ satisfying

$$j \equiv l + g_1 \mod 2g_1, \quad k \equiv m \mod g_2.$$

The storage before and after the redistribution can be illustrated by the following example:

G_{c_1} :

$p_{1,1}$	$p_{1,2}$
$p_{2,1}$	$p_{2,2}$

S :

$S_{1,1}$	$S_{1,2}$	$S_{1,3}$	$S_{1,4}$	$\cdots$
$S_{2,1}$	$S_{2,2}$	$S_{2,3}$	$S_{2,4}$	$\cdots$
$S_{3,1}$	$S_{3,2}$	$S_{3,3}$	$S_{3,4}$	$\cdots$
$S_{4,1}$	$S_{4,2}$	$S_{4,3}$	$S_{4,4}$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

G_i :

$p_{1,1}$	$p_{1,2}$
$p_{2,1}$	$p_{2,2}$
$p_{3,1}$	$p_{3,2}$
$p_{4,1}$	$p_{4,2}$

S :

$S_{1,1}$	$S_{1,2}$	$S_{1,3}$	$S_{1,4}$	$\cdots$
$S_{2,1}$	$S_{2,2}$	$S_{2,3}$	$S_{2,4}$	$\cdots$
$S_{3,1}$	$S_{3,2}$	$S_{3,3}$	$S_{3,4}$	$\cdots$
$S_{4,1}$	$S_{4,2}$	$S_{4,3}$	$S_{4,4}$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

The process grids for the entire HSS tree can be constructed in the following way:

- Let the shape of all the bottom level (level-L) process grids be $g_1 \times g_2$, where g_1, g_2 are integers chosen to evenly distribute the bottom level matrices.
- The upper level process grids are formed by an alternation between α vertical and β horizontal concatenations, where α and β are integers chosen to fit the growth rate of matrix shapes across different levels. The resulting process grids have the growth rate of $2^{\frac{\alpha}{\alpha+\beta}}$ in the row size, and $2^{\frac{\beta}{\alpha+\beta}}$ in the column size. See Table 1 for some sample choices of α and β.

TABLE 1
Sample choices of the parameters α and β and the corresponding sizes of the process grids.

α	β	Level L	Level $L-1$	Level $L-2$	Level $L-3$	Level $L-4$	Level $L-5$	Level $L-6$
1	0	$g_1 \times g_2$	$2g_1 \times g_2$	$4g_1 \times g_2$	$8g_1 \times g_2$	$16g_1 \times g_2$	$32g_1 \times g_2$	$64g_1 \times g_2$
1	1	$g_1 \times g_2$	$2g_1 \times g_2$	$2g_1 \times 2g_2$	$4g_1 \times 2g_2$	$4g_1 \times 4g_2$	$8g_1 \times 4g_2$	$8g_1 \times 8g_2$
2	1	$g_1 \times g_2$	$2g_1 \times g_2$	$4g_1 \times g_2$	$4g_1 \times 2g_2$	$8g_1 \times 2g_2$	$16g_1 \times 2g_2$	$16g_1 \times 4g_2$

In order to give a quantitative study of our different parallel algorithms, we collect our main assumptions and frequently used arguments to build a uniform framework.

1. *Collective communications.* A message-passing collective broadcast or reduction algorithm requires $\log p$ messages and $m \log p$ words, where m is the size of the matrix and p is the number of processes.
2. *Parallel low-rank updates.* Low-rank updates are frequently used in the matrix factorizations and triangular matrix solutions with multiple right-hand sides. On a $g_1 \times g_2$ process grid, we assume that the communication cost of a rank-r update to an $m \times \alpha$ matrix is uniformly bounded by the number of messages (#messages) and the number of words (#words) as follows:

$$\#\text{messages} = O(r \log g_1 + r \log g_2), \quad \#\text{words} = O\left(r\frac{m}{g_1} \log g_1 + r\frac{\alpha}{g_2} \log g_2\right).$$

   These upper bounds are calculated from r rank-1 updates. At each step, the column vector is broadcasted among the process rows, the row vector is broadcasted among the process columns, and the local matrix is updated with the local vectors.
3. *Redistribution.* As shown above, the redistribution cost can be reduced to an exchange of local storage. To be precise, the redistribution of an $m \times \alpha$ matrix among node i and its children c_1, c_2 takes $O(1)$ messages and $O(\frac{m\alpha}{p})$ words, where $p = \max\{|G_i|, |G_{c_1}|, |G_{c_2}|\}$ is the maximal size of the related process grids.

2.2. Distribution scheme for randomized sampling. The randomized HSS algorithms obtain data via the multiplication of A and random sampling vectors. The data structures for the sampling vectors play a key role in our algorithms. To take advantage of BLAS-3 and PBLAS-3 kernels, the sampling vectors are grouped together into skinny *sampling matrices*. For the $n \times n$ matrix A, the global size of a typical sampling matrix X is $n \times \alpha$, where $\alpha \ll n$ is the sampling size and is usually slightly larger than the HSS rank r. In the fully matrix-free HSS construction in the next section, one such sampling matrix is involved at each level of the HSS tree $\mathcal{T}$. A given number of processes is used to handle these matrices at each level.

Here, we lay out the distributed storage scheme for these sampling matrices. The distributed storage of X needs to facilitate the multilevel off-diagonal compression in HSS constructions, and also provide a simple interface for the user-supplied matrix-vector multiplication routine. The sampling matrix X can be naturally partitioned according to the bottom level D_i generator sizes:

$$\{X_i : i \text{ is a leaf of } \mathcal{T}, \text{ and } X_i \text{ and } D_i \text{ have the same row size}\}.$$

For example, following a two-level HSS structure as follows, we can partition X accordingly:

$$A = \begin{pmatrix} \begin{pmatrix} D_1 & U_1B_1V_1^T \\ U_2B_2V_1^T & D_2 \end{pmatrix} & U_3B_3V_6^T \\ U_6B_6V_3^T & \begin{pmatrix} D_4 & U_4B_4V_5^T \\ U_5B_4V_4^T & D_5 \end{pmatrix} \end{pmatrix}, \quad X = \begin{pmatrix} X_1 \\ X_2 \\ X_4 \\ X_5 \end{pmatrix}.$$

For a leaf node i, the sampling submatrix X_i is stored in the corresponding process grid G_i for parallel HSS constructions.

1. If $|G_i| = 1$, no further distribution is performed on X_i. Then X_i actually represents the local storage of one process, and this forms a global block-row

distribution of X, where each process stores some rows of the matrix. Matrix-free computation is possible here because in many applications a matrix-vector product is readily available in block-row distribution.

2. If $|G_i| > 1$, it's not as straightforward to perform matrix-free computations because each X_i is distributed following the 2D block-cyclic scheme. Here, we can extend the HSS tree $\mathcal{T}$ so that for the extended tree $\widetilde{\mathcal{T}}$ each leaf corresponds to only one process. Some extended nodes are introduced into $\widetilde{\mathcal{T}}$ and are not associated with any HSS generator. Then we distribute X according to $\widetilde{\mathcal{T}}$, and this case becomes the previous one. Now the user-supplied matrix-vector multiplication routine is connected with the new partition

$$\{X_j : j \text{ is a leaf node of } \widetilde{\mathcal{T}}\}.$$

Since the extended nodes do not correspond to any HSS generator, after each sampling step, the matrices X_j need to be redistributed to the nodes in the original HSS tree $\mathcal{T}$. Figure 1 demonstrates the extension of the tree and the multilevel redistribution of the sampling matrices.

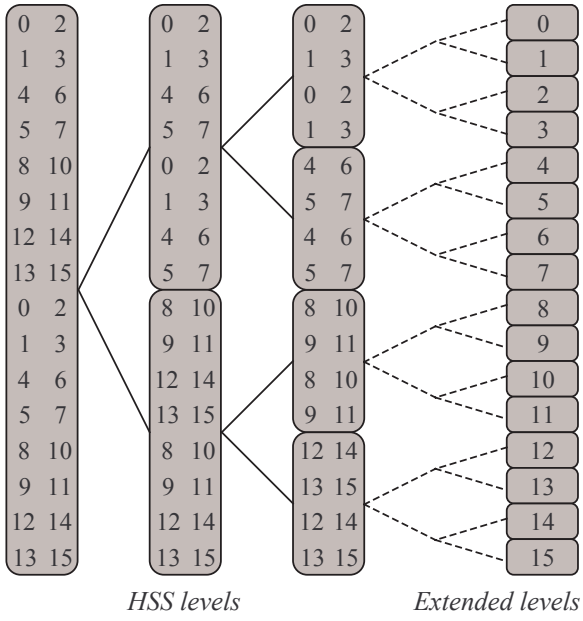


FIG. 1. *Illustration of the distribution of the sampling matrices at different HSS levels into the process grids, where the HSS tree $\mathcal{T}$ is extended to $\widetilde{\mathcal{T}}$ to accommodate a block-row distribution for user-supplied matrix-vector multiplications.*

3. Parallel randomized low-rank compression schemes. A parallel compression algorithm is used at each level of the HSS tree to obtain low-rank approximations to the corresponding off-diagonal blocks. For an $m \times k$ matrix H with (numerical) rank r, traditional compression techniques usually compute a low-rank approximation via rank-revealing factorizations. These methods are limited by the need to explicitly store and process the entire matrix. Recent randomized techniques instead are based on a randomized sampling process and the multiplication of H with random vectors. Substantial improvements can be achieved if the matrix-vector multiplication is fast.

The theoretical justification of randomized low-rank compression methods can be found in [14]. Here, we only mention some basic ideas. From an SVD representation $H = U\Sigma V^T$, a matrix-vector multiplication with a Gaussian random vector x is the combination of the weighted singular vectors $U\Sigma$ based on rotated Gaussian random coefficients $V^T x$. Different random combinations tend to be linearly independent and start to build a basis for the range space of H until the number of vectors gets close to r. By sampling H with $r + \mu$ random vectors with μ an oversampling size, the important basis vectors of H can be recovered from the sampling data, and the probability of success is high even with small μ.

Two randomized compression schemes are used in this work. One is to select important rows of H as the basis vectors from the product of H and $r+\mu$ random vectors [16]. Another way is to construct basis vectors by adaptively multiplying H and random vectors, followed by orthonormalization, until a desired accuracy is reached [14]. In this section, we present and analyze the parallel versions of these schemes. The matrix H and the skinny sampling matrices are assumed to be distributed on the same $g_1 \times g_2$ process grid.

3.1. Parallel structure-preserving rank-revealing factorization. In various applications, the low-rank compression of H may be achieved by selecting some important rows or columns via rank-revealing factorizations [11]. This can be represented by

$$H \approx UH|_{\hat{\mathbf{I}}}, \quad U \equiv P \begin{pmatrix} I \\ E \end{pmatrix}, \tag{3.1}$$

where P is a permutation matrix, $H|_{\hat{\mathbf{I}}}$ is the row basis matrix extracted from H based on the index set $\hat{\mathbf{I}}$, and E is a matrix with bounded entries. Since $H|_{\hat{\mathbf{I}}}$ is a submatrix of H and the column basis matrix is structured, (3.1) is also called a *structure-preserving rank-revealing* (SPRR) factorization [38]. Related work includes skeleton approximations [26] and interpolative decompositions [16].

When the size of H becomes large, a direct computation of (3.1) is inefficient. If the product of H and vectors can be quickly computed (through a user-supplied routine mat-vec), then randomized sampling can be used instead [16]. Given a truncation tolerance τ, a numerical rank bound r for H, and an oversampling parameter μ, we construct a Gaussian random matrix X with $r + \mu$ columns and compute

$$Y = HX.$$

An SPRR factorization is computed for Y:

$$Y \approx P \begin{pmatrix} I \\ E \end{pmatrix} Y|_{\hat{\mathbf{I}}}.$$

The same matrices P, E and row index set $\hat{\mathbf{I}}$ are then used in (3.1) for the approximation of H. For modest μ, it can be shown that this has a desired accuracy with a high probability [16].

In the parallel implementation of this factorization, row pivoting is a key element. In practice, we use existing column pivoting routines on the transpose matrix. To avoid global transpose operations, we only transpose the process grid and the local submatrix. This forms a transpose matrix with no communications. The details are given in Algorithm 1.

Algorithm 1. Parallel structure-preserving rank-revealing factorization.

Input: τ (approximation tolerance) and $r+\mu$ (number of sampling vectors)
Output: r (numerical rank) and the approximation (3.1)

1: **procedure** PSPRR(τ, l)
2: $X \leftarrow m \times (r+\mu)$ Gaussian random matrix ▷ *Sampling matrix*
3: $Y \leftarrow$ mat-vec(H, X) ▷ *PBLAS-3 user-supplied matrix-vector multiplication*
4: $T \leftarrow Y^T$ ▷ *Transpose of the process grid*
5: $T \approx QRP^T$ ▷ *Rank-revealing QR factorization of Y^T with the tolerance τ, where P^T is a permutation matrix for the column pivoting*
6: $\begin{pmatrix} R_1 & R_2 \end{pmatrix} \leftarrow R$ ▷ *Partition of R so that R_1 is $r \times r$ and corresponds to the column index set $\hat{\mathbf{I}}$*
7: $T \leftarrow R_1^{-1} R_2$ ▷ *Triangular solution*
8: $E \leftarrow T^T$ ▷ *Transpose of the process grid again*
9: **return** r and the approximation (3.1)
10: **end procedure**

The computational cost of this algorithm is $O(r^2 m)$, excluding an application-dependent sampling cost for mat-vec. The QR factorization (step 5) with column pivoting and the triangular matrix solution (step 7) share the same upper bound of the communication cost:

$$\#\text{messages} = O(r \log g_1 + r \log g_2), \quad \#\text{words} = O\left(r\frac{m}{g_1}\log g_1 + r\frac{r}{g_2}\log g_2\right),$$

where $g_1 \times g_2$ is the shape of the process grid.

3.2. Parallel adaptive randomized orthogonalization. The second scheme for computing a low-rank approximation of H is via randomized orthonormalization. We find a column basis matrix with orthonormal columns. This strategy can be combined with randomized adaptive rank estimation. That is, it does not need a numerical rank bound of H in advance. Instead, H is gradually multiplied with new random vectors x_j as needed until the desired accuracy τ is met. The criterion to quickly check the accuracy (with high probability) is [14]

$$||H - \tilde{H}||_2 \le c \max_{j=\alpha-\mu+1,\ldots,\alpha} ||(H - \tilde{H})x_j||_2 \le \tau,$$

where c is a certain real number and $\tilde{H}$ is the low-rank approximation.

Here, we perform a modified Gram–Schmidt process and the rank estimation in parallel. To take full advantage of PBLAS-3 kernels, we include a small number of additional random vectors each time and group the random vectors to perform a matrix-matrix multiplication and a block version of the modified Gram–Schmidt process with reorthogonalization. See Algorithm 2. For notational convenience, later we will refer to this algorithm as

$$Q = \mathsf{PRAO}(HX). \tag{3.2}$$

Once the column basis matrix Q is found, it is suggested in [14] that a good approximation of H is

$$H \approx Q(Q^T H). \tag{3.3}$$

Algorithm 2. Parallel adaptive randomized orthogonalization.

Input: τ (approximation tolerance), s (number of additional vectors to be included), and μ (oversampling size for accuracy check)
Output: Q (a column basis matrix with orthonormal columns for H)

procedure PARO(τ, s, μ)
  $Y \leftarrow [], \quad Q \leftarrow []$ ▷ *Empty initial matrices*
  $\alpha \leftarrow s$ ▷ *Number of random vectors for the initial round*
  loop
    $X \leftarrow m \times s$ Gaussian random matrix ▷ *Sampling matrix*
    $Y \leftarrow \begin{pmatrix} Y & \text{mat-vec}(H, X) \end{pmatrix}$
      ▷ *PBLAS-3 user-supplied matrix-vector multiplication*
    $Y \leftarrow (I - QQ^T)Y$ via block Gram–Schmidt with reorthogonalization
      ▷ *PBLAS-3*
    for $j = 1, 2, \ldots, \alpha - \mu + 1$ **do**
      if $\max\{\|(Y|_{:\times j})\|_2, \|(Y|_{:\times j+1})\|_2, \ldots, \|(Y|_{:\times j+\mu-1})\|_2\} \le \tau$ **then**
        ▷ $Y|_{:\times j}$ *denotes column* j *of* Y
        return Q ▷ *Desired accuracy reached*
      end if
      $Y|_{:\times j} \leftarrow (I - QQ^T)Y|_{:\times j}$ ▷ *Reorthogonalization of* $Y|_{:\times j}$
      $q \leftarrow Y|_{:\times j}/\|Y|_{:\times j}\|_2$ ▷ *Normalization of* $Y|_{:\times j}$
      $Y|_{:\times(j+1:\alpha)} \leftarrow (I - qq^T)Y|_{:\times(j+1:\alpha)}$ ▷ *Rank-1 update*
      $Q \leftarrow \begin{pmatrix} Q & q \end{pmatrix}$ ▷ *Expansion of the basis*
    end for
    $Y \leftarrow$ the last $\mu - 1$ columns of Y ▷ *First few vectors for the next round*
    $\alpha \leftarrow s + \mu - 1$ ▷ *Number of vectors for the next round*
  end loop
end procedure

In practice, the reorthogonalization operations can guarantee the quality of the basis matrices Q and $Q^T H$.

We give an estimate of the computational and communication costs of this algorithm. Let the output matrix Q be $m \times r$. Apart from an application-dependent sampling cost in mat-vec, the leading computational cost of this algorithm is $O(r^2 m)$ for the block Gram–Schmidt process. The computational cost of the remaining vector operations is $O(rm)$. For each column of Q, computing and broadcasting vector norms need $O(\log g_1 + \log g_2)$ messages and $O(\log g_1 + \log g_2)$ words, and the rank-1 update needs $O(\log g_1 + \log g_2)$ messages and $O(\frac{m}{g_1} \log g_1 + \frac{r}{g_2} \log g_2)$ words. The block Gram–Schmidt needs to be performed $\frac{r}{s}$ times, and each step is based on a rank-s update. From our assumptions of collective communications and low-rank updates at the end of section 2.1, the total communication cost is

$$\#\text{messages} = O(r \log g_1 + r \log g_2), \quad \#\text{words} = O\left(r \frac{m}{g_1} \log g_1 + r \frac{r}{g_2} \log g_2\right).$$

Remark 3.1. Later, for notational convenience, we write (3.1) from the scheme in section 3.1 and (3.3) from the scheme in section 3.2 as equalities (as if the blocks were exactly rank-deficient). This avoids the unnecessary confusion between the notation for A and for its HSS approximation.

Remark 3.2. In our algorithms, Gaussian random matrices are used. It is well known that these randomized low-rank approximation methods can use various types of random sampling techniques, and the approximation quality is relatively insensitive and very reliable [14]. With Gaussian random matrices, existing studies have already provided comprehensive accuracy and probability results. In particular, the stopping criterion we use in the adaptive rank detection is based on Gaussian random matrices. We have also tried other sampling techniques. For example, if the matrix to be compressed is a general dense matrix, then the subsampled random Fourier transform [16, 29] may be used to save the memory and computational costs. Here, our problems are not necessarily restricted to general dense matrices. In addition, our HSS construction algorithms below only need about $(r+\mu)n$ random numbers because of data reuse. Generating these random numbers is much faster than computing the required $O(r)$ or $O(\log n)$ matrix-vector products. Therefore, we use Gaussian random matrices for convenience. When necessary, it is not hard to replace these by other random matrices, depending on the actual applications.

4. Parallel randomized and matrix-free HSS direct solvers. In this section, we discuss our parallel randomized and matrix-free HSS direct solvers for large dense matrices with the low-rank property. The solvers consist of HSS constructions, HSS factorizations, and HSS solutions.

HSS construction algorithms provide the foundation of the HSS solutions. The traditional construction from the dense $n\times n$ matrix A is limited by the quadratic computational cost and storage. For the parallel HSS algorithms, the computing resource needs to be allocated according to the most expensive global dense matrix operations during the construction [28]. Such a choice limits other HSS related operations from reaching their peak performance. Our new parallel randomized HSS construction algorithms no longer have this limitation because each off-diagonal block is compressed from the sampling data. Also for the same reason, these algorithms are less straightforward to understand. For a better exposition of the algorithms, we will usually use part of the original matrix for illustration, but the original full dense matrix is never explicitly stored.

We present two parallel randomized HSS construction algorithms based on the two types of low-rank compression schemes in section 3. The partially matrix-free construction in [18, 38] uses SPRR factorizations for the off-diagonal compression and needs to access some entries of A. It needs fewer matrix-vector products and has a cheaper compression cost. The structures in the SPRR representations are used later to accelerate HSS factorizations and solutions.

The fully matrix-free construction in [17, 32] is based on randomized orthogonalization. It is more flexible since it uses only matrix-vector products and integrates adaptive rank detection. Because of the nice orthonormal basis matrices, this HSS structure has potentially more accurate ULV factorizations and solutions. With an additional logarithmic factor in the cost, the fully matrix-free construction is still competitive in performance. The adaptive rank detection also often yields more compact HSS representations.

Remark 4.1. The accuracies of HSS approximations with the two types of construction methods have been thoroughly studied in [30, 31]. In fact, it is shown that the HSS approximation error is well controlled by the tolerance for the off-diagonal compression, with a small magnification factor that depends on the off-diagonal numerical rank bound r and $\log n$. The HSS solution accuracy is also discussed in [30]. Our numerical tests also confirm that the accuracy is well controlled.

In our discussions of the algorithms, the following two types of parallel operations are needed, based on the distribution scheme in section 2:

1. An intergrid redistribution copies a matrix from one process grid to another. At each step, our algorithms only redistribute data among G_{c_1}, G_{c_2}, and G_i, where i is the parent of c_1 and c_2. Section 2 provides the way to control the communication cost. In practice, ScaLAPACK [22] has a general routine PxGEMR2D for such operations.
2. An intragrid operation performs dense matrix computations in a single process grid. Two low-rank compression routines PSPRR and PRAO (section 3) and a dense matrix multiplication routine PxGEMM are used in the randomized HSS construction algorithms. The LU factorization routine PxGEQPF is used in an HSS factorization algorithm.

4.1. Partially matrix-free HSS construction. In the partially matrix-free HSS construction, suppose the HSS rank is bounded by r and the entries of A are conveniently accessible. The SPRR factorization in Algorithm 1 is used to compress the HSS blocks in parallel. This construction needs only $r+\mu$ sampling vectors, where μ is the oversampling size as before.

We first describe the parallel storage scheme for the HSS form resulting from this construction, where the column basis matrices look like that in (3.1). Figure 2 shows how the generators are stored in the related process grids.

1. If i is a leaf node, the SPRR factorizations of the HSS blocks $A|_{\mathbf{I}_i \times \mathbf{I}_i^c}$ and $(A|_{\mathbf{I}_i^c \times \mathbf{I}_i})^T$ yield the column basis matrices U_i and V_i, respectively:

$$U_i = P_i \begin{pmatrix} I \\ E_i \end{pmatrix}, \quad V_i = Q_i \begin{pmatrix} I \\ F_i \end{pmatrix}, \tag{4.1}$$

   where P_i, Q_i are permutation matrices. Thus, we store E_i, F_i as well as the diagonal block D_i in the process grid G_i, which is described in section 2.1.
2. If i is a nonleaf node with children c_1 and c_2, the construction process below yields the following generators:

$$\begin{pmatrix} R_{c_1} \\ R_{c_2} \end{pmatrix} = P_i \begin{pmatrix} I \\ E_i \end{pmatrix}, \quad \begin{pmatrix} W_{c_1} \\ W_{c_2} \end{pmatrix} = Q_i \begin{pmatrix} I \\ F_i \end{pmatrix}. \tag{4.2}$$

   The matrices E_i, F_i as well as the generators B_{c_1}, B_{c_2} are stored in the process grid G_i. Note that B_{c_1}, B_{c_2} are associated with the children of i, but are stored in G_i.

We then describe our parallel implementation of the partially matrix-free HSS construction in [18, 38]. Initially, construct an $n \times (r+\mu)$ global random matrix X in a block-row distribution, and compute PBLAS-3 products

$$Y = AX, \quad Z = A^T X.$$

This needs $2(r+\mu)$ matrix-vector products. Then within the process grid G_i of each HSS leaf node i, get the partitioned sampling matrices

$$X_i = X|_{\mathbf{I}_i}, \quad Y_i = Y|_{\mathbf{I}_i}, \quad Z_i = Z|_{\mathbf{I}_i}.$$

A redistribution of X, Y, Z within G_i may be needed if G_i contains more than one process (Figure 1). Note that the sampling data Y, Z contain both full-rank diagonal contributions and low-rank off-diagonal contributions from multiple levels. A critical

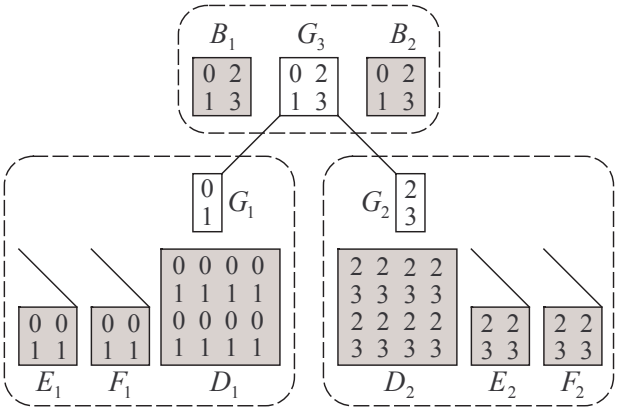


FIG. 2. *Illustration of the distributed storage of the HSS generators resulting from SPRR factorizations.*

task in the algorithm is to form the sampling information of the targeting HSS blocks. This is achieved by updating and reusing the sampling data.

The algorithm performs a parallel bottom-up traversal of the HSS tree, and we perform the following computations at each node i:

1. If i is a leaf, the diagonal block is acquired in the process grid G_i via

$$D_i = A|_{\mathbf{I}_i \times \mathbf{I}_i}.$$

The sampling data of the off-diagonal blocks are formed indirectly by the PBLAS-3 multiplications:

$$\begin{aligned} \Phi_i = &\quad A|_{\mathbf{I}_i \times \mathbf{I}_i^c} X|_{\mathbf{I}_i^c} &&= Y_i - D_i X_i, \\ \Psi_i = &\quad (A|_{\mathbf{I}_i^c \times \mathbf{I}_i})^T X|_{\mathbf{I}_i^c} &&= Z_i - D_i^T X_i. \end{aligned}$$

Then compute SPRR factorizations (as in Algorithm 1)

$$\Phi_i = U_i \Phi_i|_{\hat{\mathbf{I}}_i}, \quad \Psi_i = V_i \Psi_i|_{\hat{\mathbf{J}}_i},$$

where U_i, V_i are given in (4.1) and the index sets $\hat{\mathbf{I}}_i, \hat{\mathbf{J}}_i$ are associated with the local matrices Φ_i, Ψ_i. They correspond to the important rows of $A|_{\mathbf{I}_i \times \mathbf{I}_i^c}$ and important columns of $A|_{\mathbf{I}_i^c \times \mathbf{I}_i}$. We then find the global index sets $\tilde{\mathbf{I}}_i, \tilde{\mathbf{J}}_i$ in A corresponding to these important rows and columns (selected by $\hat{\mathbf{I}}_i, \hat{\mathbf{J}}_i$), respectively. For future reuse, we compute the products $U_i^T X_i, V_i^T X_i$ from the SPRR factors. All the intragrid operations are contained in the process grid G_i. No intergrid redistribution is performed at this step.

2. If i is a nonleaf node with children c_1, c_2, extract B_{c_1}, B_{c_2} in the process grid G_i as

$$B_{c_1} = A|_{\tilde{\mathbf{I}}_{c_1} \times \tilde{\mathbf{J}}_{c_2}}, \quad B_{c_2} = A|_{\tilde{\mathbf{I}}_{c_2} \times \tilde{\mathbf{J}}_{c_1}}. \tag{4.3}$$

B_{c_1}, B_{c_2} are used only in the parent node, so we store them in G_i to avoid unnecessary redistributions. This is one major consideration in designing the

storage scheme as in Figure 2. Then form the sampling data

$$\Phi_i = \begin{pmatrix} \Phi_{c_1}|_{\hat{\mathbf{I}}_{c_1}} \\ \Phi_{c_2}|_{\hat{\mathbf{I}}_{c_2}} \end{pmatrix} - \begin{pmatrix} B_{c_1}(\ I \quad E_{c_2}^T \ ) P_{c_2}^T X_{c_2} \\ B_{c_2}(\ I \quad E_{c_1}^T \ ) P_{c_1}^T X_{c_1} \end{pmatrix},$$

$$\Psi_i = \begin{pmatrix} \Psi_{c_1}|_{\hat{\mathbf{I}}_{c_1}} \\ \Psi_{c_2}|_{\hat{\mathbf{I}}_{c_2}} \end{pmatrix} - \begin{pmatrix} B_{c_2}^T(\ I \quad F_{c_2}^T \ ) Q_{c_2}^T X_{c_2} \\ B_{c_1}^T(\ I \quad F_{c_1}^T \ ) Q_{c_1}^T X_{c_1} \end{pmatrix},$$

where the matrices on the right-hand sides are merged by the intergrid redistribution from G_{c_1}, G_{c_2} to G_i. Then compute SPRR factorizations (as in Algorithm 1)

$$\Phi_i = \begin{pmatrix} R_{c_1} \\ R_{c_2} \end{pmatrix} \Phi_i|_{\hat{\mathbf{I}}_i}, \quad \Psi_i = \begin{pmatrix} W_{c_1} \\ W_{c_2} \end{pmatrix} \Psi_i|_{\hat{\mathbf{J}}_i},$$

where the R, W generators are given in (4.2). The derivation of this procedure is given in [38].

The algorithm then proceeds with the same strategies. It stops when the root node is reached and the generators (4.3) are retrieved.

4.2. Improved fully matrix-free HSS construction. The partially matrix-free construction above is effective when the user can provide a reasonable rank bound and a fast way of accessing the matrix entries. The fully matrix-free HSS construction algorithm does not have these restrictions. Here, other than presenting a scalable implementation, we also make some novel improvements to the original matrix-free HSS constructions in [17, 32] as follows:

1. We avoid multiple pseudoinverses used in [17, 32] to gain better stability and efficiency for the compression of the off-diagonal blocks.
2. Our compression yields more compact nested basis matrices.
3. We incorporate parallel synchronized rank detection in the hierarchical compression.

For each level $l = 0, 1, \ldots, L$ of the HSS tree, the diagonal blocks D_i form a block-diagonal matrix

$$\mathbf{D}^{(l)} = \operatorname{diag}\left(D_i = A|_{\mathbf{I}_i \times \mathbf{I}_i} : \ i \text{ at level } l\right). \tag{4.4}$$

Because of the nested index partitioning, $\mathbf{D}^{(l)}$ contains the next-level matrix $\mathbf{D}^{(l+1)}$ as smaller diagonal blocks and $\mathbf{D}^{(l)} - \mathbf{D}^{(l+1)}$ as off-diagonal blocks. At each level, the individual off-diagonal blocks within $\mathbf{D}^{(l)} - \mathbf{D}^{(l+1)}$ need to be compressed, and the compression information needs to be integrated into the HSS form. Thus, the matrix-free construction algorithm has the following major steps:

1. For level l, compress the off-diagonal blocks within $\mathbf{D}^{(l)} - \mathbf{D}^{(l+1)}$ based on the HSS form of $A - \mathbf{D}^{(l)}$ and randomization.
2. Construct the HSS form of $A - \mathbf{D}^{(l+1)}$ by combining the HSS form of $A - \mathbf{D}^{(l)}$ and the basis matrices of $\mathbf{D}^{(l)} - \mathbf{D}^{(l+1)}$.
3. Extract the diagonal blocks $\mathbf{D}^{(L)}$ at the leaf level $l = L$.

4.2.1. Off-diagonal compression at one level. Assuming we have the HSS form of $A - \mathbf{D}^{(l)}$ for $l = 0, 1, \ldots$, we show how to compress the off-diagonal blocks within $\mathbf{D}^{(l)} - \mathbf{D}^{(l+1)}$ via randomization. (For the initial case $l = 0$, the matrix $A - \mathbf{D}^{(l)}$ is empty since $A = \mathbf{D}^{(0)}$.)

The block-diagonal structure of $\mathbf{D}^{(l)}$ as in (4.4) allows us to focus on a single node i at level l with children c_1, c_2. We want to compress the off-diagonal blocks of D_i as follows:

$$D_i = \begin{pmatrix} * & A|_{\mathbf{I}_{c_1}\times\mathbf{I}_{c_2}} \\ A|_{\mathbf{I}_{c_2}\times\mathbf{I}_{c_1}} & * \end{pmatrix} = \begin{pmatrix} * & \tilde{U}_{c_1}\tilde{V}_{c_2}^T \\ \tilde{U}_{c_2}\tilde{V}_{c_1}^T & * \end{pmatrix},$$

where $A|_{\mathbf{I}_{c_1}\times\mathbf{I}_{c_2}}$ and $A|_{\mathbf{I}_{c_2}\times\mathbf{I}_{c_1}}$ are two individual off-diagonal blocks in $\mathbf{D}^{(l)} - \mathbf{D}^{(l+1)}$. The basis matrices $\tilde{U}, \tilde{V}$ are obtained simultaneously for the children of all the nodes i at level l based on randomized sampling. That is, choose a random matrix $\hat{X}$ with a global block-row distribution,

$$\hat{X}|_{\mathbf{I}_i} = \begin{pmatrix} X|_{\mathbf{I}_{c_1}} & 0 \\ 0 & X|_{\mathbf{I}_{c_2}} \end{pmatrix} \quad \text{for all } i \text{ at level } l,$$

and compute

$$(\mathbf{D}^{(l)})^T\hat{X} = A^T\hat{X} - (A - \mathbf{D}^{(l)})^T\hat{X}. \tag{4.5}$$

In this multiplication, $A^T\hat{X}$ is done through a parallel PBLAS-3 user-supplied matrix multiplication at the bottom level of the extended tree $\widetilde{\mathcal{T}}$ (Figure 1), and $(A-\mathbf{D}^{(l)})^T\hat{X}$ is done through a PBLAS-3 HSS matrix-vector multiplication routine that involves only small matrix multiplications.

Thus, we obtain from (4.5)

$$D_i^T\hat{X}|_{\mathbf{I}_i} = \begin{pmatrix} * & (A|_{\mathbf{I}_{c_2}\times\mathbf{I}_{c_1}})^T X|_{\mathbf{I}_{c_2}} \\ (A|_{\mathbf{I}_{c_1}\times\mathbf{I}_{c_2}})^T X|_{\mathbf{I}_{c_1}} & * \end{pmatrix} \equiv \begin{pmatrix} * & Y_{c_1} \\ Y_{c_2} & * \end{pmatrix}.$$

Based on Algorithm 2, we can compute

$$\tilde{V}_{c_1} = \mathsf{PRAO}(Y_{c_1}), \quad \tilde{V}_{c_2} = \mathsf{PRAO}(Y_{c_2}). \tag{4.6}$$

These computations are done in parallel in the child process grids G_{c_1}, G_{c_2}, and parallel synchronized rank detection is used. Similarly to [37], $\tilde{V}_{c_1}, \tilde{V}_{c_2}$ can then be obtained in a deterministic way:

$$\tilde{U}_{c_1} = A|_{\mathbf{I}_{c_1}\times\mathbf{I}_{c_2}}\tilde{V}_{c_2}, \quad \tilde{U}_{c_2} = A|_{\mathbf{I}_{c_2}\times\mathbf{I}_{c_1}}\tilde{V}_{c_1}, \tag{4.7}$$

where the matrix-vector multiplications on the right-hand sides are obtained via the multiplication of $\mathbf{D}^{(l)}$ and a matrix $\tilde{V}$ defined by

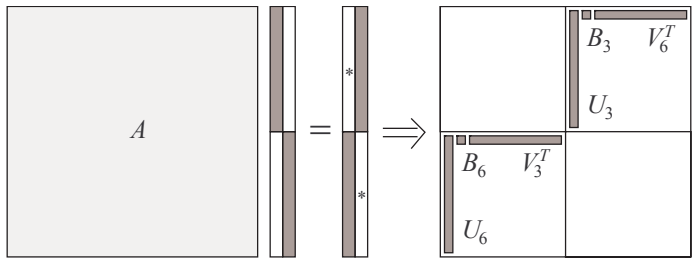
$$\tilde{V}|_{\mathbf{I}_i} = \begin{pmatrix} \tilde{V}_{c_1} & 0 \\ 0 & \tilde{V}_{c_2} \end{pmatrix} \quad \text{for all } i \text{ at level } l.$$

Thus, we obtain the low-rank forms of the off-diagonal blocks within $\mathbf{D}^{(l)} - \mathbf{D}^{(l+1)}$ for all i at level l. The number of matrix-vector multiplications used at this step is at most $4r + 2\mu$. A pictorial illustration for $l = 1, 2$ is given in Figure 3(a)–(b).

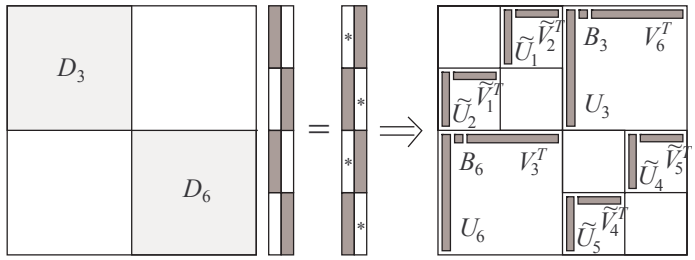
4.2.2. Nested off-diagonal basis for HSS construction. We can then get an HSS form of $A - \mathbf{D}^{(l+1)}$ based on the HSS form of $A - \mathbf{D}^{(l)}$ and the low-rank forms of the blocks within $\mathbf{D}^{(l)} - \mathbf{D}^{(l+1)}$ from the previous step. This requires converting the off-diagonal basis matrices into nested forms as in (2.2).

For $l = 1$, the off-diagonal compression scheme in the previous subsection yields

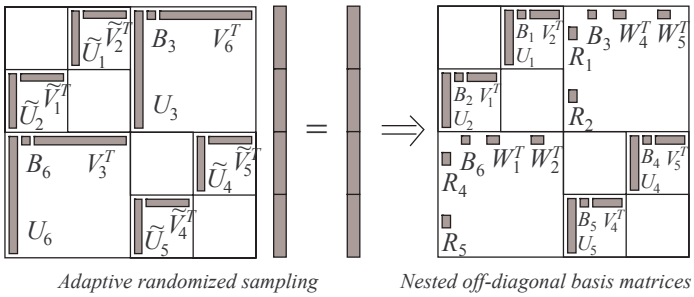
$$A - \mathbf{D}^{(1)} = \begin{pmatrix} 0 & \tilde{U}_{c_1}\tilde{V}_{c_2}^T \\ \tilde{U}_{c_2}\tilde{V}_{c_1}^T & 0 \end{pmatrix}.$$



(a) First-level off-diagonal compression



(b) Second-level off-diagonal compression



(c) Second-level nested basis construction

FIG. 3. *An example of the top-down matrix-free HSS construction, where A is used for illustration purposes and is accessed only through a user-supplied matrix-vector multiplication.*

In order to transform this form into an HSS representation

$$A - \mathbf{D}^{(1)} = \begin{pmatrix} 0 & U_{c_1} B_{c_1} V_{c_2}^T \\ U_{c_2} B_{c_2} V_{c_1}^T & 0 \end{pmatrix},$$

we simply set $V_{c_1} = \tilde{V}_{c_1}$, $V_{c_2} = \tilde{V}_{c_2}$, and compute QR factorizations

$$\tilde{U}_{c_1} = U_{c_1} B_{c_1}, \quad \tilde{U}_{c_2} = U_{c_2} B_{c_2}.$$

See Figure 3(a).

For a general level l $(1 < l < L)$ and a nonleaf node i at level l, since $\{1 : n\} =$

$\mathbf{I}_i \cup \mathbf{I}_i^c$, $\mathbf{I}_i = \mathbf{I}_{c_1} \cup \mathbf{I}_{c_2}$, the compression from the previous steps yields

$$(A - \mathbf{D}^{(l+1)})|_{\mathbf{I}_i \times \{\mathbf{I}_i \cup \mathbf{I}_i^c\}} = \left(\begin{pmatrix} 0 & \tilde{U}_{c_1} \tilde{V}_{c_2}^T \\ \tilde{U}_{c_2} \tilde{V}_{c_1}^T & 0 \end{pmatrix} \quad U_i (U_i^T A|_{\mathbf{I}_i \times \mathbf{I}_i^c}) \right),$$

$$(A - \mathbf{D}^{(l+1)})^T|_{\mathbf{I}_i \times \{\mathbf{I}_i \cup \mathbf{I}_i^c\}} = \left(\begin{pmatrix} 0 & \tilde{V}_{c_1} \tilde{U}_{c_2}^T \\ \tilde{V}_{c_2} \tilde{U}_{c_1}^T & 0 \end{pmatrix} \quad V_i (V_i^T A^T|_{\mathbf{I}_i \times \mathbf{I}_i^c}) \right).$$

$\tilde{U}, \tilde{V}$ are temporary basis matrices (Figure 3(b)), and we still need to convert them into nested forms as in (2.1)–(2.2). That is, we find orthonormal basis matrices $U_{c_1}, U_{c_2}, V_{c_1}, V_{c_2}$ from U_i, V_i and the temporary basis matrices. In [17, 32], the column basis matrix $\tilde{U}_{c_1}$ and a part of U_i are put together and then compressed to compute U_{c_1}. However, this may yield U_{c_1} that is not compact enough, and a more sophisticated recompression scheme is needed to reveal the right rank [33].

Here, we resolve this issue with randomization. Note that U_{c_1} and U_{c_2} are essentially basis matrices of some block rows of $A - \mathbf{D}^{(l+1)}$, because the diagonal blocks are zeros. We can then apply randomized sampling to the block rows of $A - \mathbf{D}^{(l+1)}$ to get U_{c_1} and U_{c_2}. For this purpose, compute

$$Y = (A - \mathbf{D}^{(l+1)})X = (A - \mathbf{D}^{(l)})X + (\mathbf{D}^{(l)} - \mathbf{D}^{(l+1)})X. \tag{4.8}$$

Based on the previous compression results, this just requires a PBLAS-3 HSS matrix-vector multiplication using the upper-level HSS structure of $A - \mathbf{D}^{(l)}$ and the low-rank forms in $\mathbf{D}^{(l)} - \mathbf{D}^{(l+1)}$.

Then compute

$$U_{c_1} = \mathsf{PRAO}(Y|_{\mathbf{I}_{c_1}}), \quad U_{c_2} = \mathsf{PRAO}(Y|_{\mathbf{I}_{c_2}}).$$

We can similarly find V_{c_1}, V_{c_2}. After these computations, we update the previous representation using the new basis. Partition U_i as $\begin{pmatrix} U_{i,1} \\ U_{i,2} \end{pmatrix}$ following the row sizes of U_{c_1} and U_{c_2}. The remaining generators can be obtained via matrix multiplications (unlike in [17, 32], no pseudoinverse is needed to compute the B generators):

$$R_{c_1} = U_{c_1}^T U_{i,1}, \quad W_{c_1} = V_{c_1}^T V_{i,1}, \quad B_{c_1} = (U_{c_1}^T \tilde{U}_{c_1})(V_{c_2}^T \tilde{V}_{c_2})^T,$$
$$R_{c_2} = U_{c_2}^T U_{i,2}, \quad W_{c_2} = V_{c_2}^T V_{i,2}, \quad B_{c_2} = (U_{c_2}^T \tilde{U}_{c_2})(V_{c_1}^T \tilde{V}_{c_1})^T.$$

The R, W generators require the redistribution between a parent and its children. The B generators involve the redistribution between two sibling nodes. It is easy to verify that this builds the HSS form of $A - \mathbf{D}^{(l+1)}$. Figure 3(c) gives an illustration. When the leaf level is reached, we have the nested off-diagonal basis for all the off-diagonal blocks as in the HSS form of A.

4.2.3. Extraction of leaf-level diagonal blocks. Finally, the only task left is to extract the leaf-level diagonal blocks. This follows from [17]. Choose a skinny matrix S that satisfies $S|_{\mathbf{I}_i} = I$ for each leaf node (though appropriate zeros may need to be inserted into $S|_{\mathbf{I}_i}$ if the leaf-level diagonal blocks do not have the same block sizes), and compute the product

$$Y = AS - (A - \mathbf{D}^{(L)})S,$$

where $(A - \mathbf{D}^{(L)})S$ is done via HSS matrix-vector multiplications. The number of matrix-vector products needed is bounded by the leaf-level diagonal block sizes, which are usually chosen to be $O(r)$. Then set $D_i = Y|_{\mathbf{I}_i}$.

4.3. HSS factorization and solution. For the HSS form resulting from the fully matrix-free construction, we can use the parallel HSS ULV factorization and solution in [28]. The HSS form from the partially matrix-free construction has additional structures in the basis generators as in (4.1) and (4.2). A ULV factorization scheme is designed in [34, 38] to take advantage of the additional structures. One basic operation is to introduce zeros to the off-diagonal block rows $A|_{\mathbf{I}_i \times \mathbf{I}_i^c}$ by noticing

$$\left[\begin{pmatrix} -E_i & I \\ I & 0 \end{pmatrix} P_i^T \right] U_i = \begin{pmatrix} 0 \\ I \end{pmatrix}. \tag{4.9}$$

This does not involve any actual computation.

We slightly extend the factorization scheme in [34, 38] for general nonsymmetric cases. That is, zeros are also introduced into the off-diagonal block column $A|_{\mathbf{I}_i^c \times \mathbf{I}_i}$ similarly to (4.9). The actual operations are performed in parallel in a bottom-up traversal of the HSS tree.

1. If i is a leaf node, update the diagonal block D_i on both sides:

$$\bar{D}_i = \begin{pmatrix} -E_i & I \\ I & 0 \end{pmatrix} P_i^T D_i Q_i \begin{pmatrix} -F_i^T & I \\ I & 0 \end{pmatrix}.$$

   Then compute a partial LU factorization

$$\bar{D}_i = \begin{pmatrix} L_{i,1} & \\ L_{i,2} & I \end{pmatrix} \begin{pmatrix} T_{i,1} & T_{i,2} \\ & \hat{D}_i \end{pmatrix},$$

   where $\hat{D}_i$ is the Schur complement. Since all the relevant matrices are stored in the process grid G_i, no intergrid communication is needed.
2. If i is a nonleaf node with children c_1 and c_2, set

$$\tilde{U}_i = P_i \begin{pmatrix} I \\ E_i \end{pmatrix}, \quad \tilde{V}_i = Q_i \begin{pmatrix} I \\ F_i \end{pmatrix},$$

   and merge the remaining blocks as

$$\tilde{D}_i = \begin{pmatrix} \hat{D}_{c_1} & B_{c_1} \\ B_{c_2} & \hat{D}_{c_2} \end{pmatrix}. \tag{4.10}$$

   Since $E_i, F_i, B_{c_1}, B_{c_2}$ are already stored in the process grid G_i (section 4.1), we just need to redistribute $\hat{D}_{c_1}, \hat{D}_{c_2}$ from the child process grids to G_i. The scheme is illustrated in Figure 4. Then remove c_1 and c_2 from the tree. As mentioned in [38, 34], i then becomes a leaf corresponding to the generators $\tilde{D}_i, \tilde{U}_i, \tilde{V}_i$, and the HSS matrix is reduced to a smaller one.

The process can then be repeated. If i is the root node, a complete LU factorization of $\tilde{D}_i$ finishes the algorithm.

Compared with the parallel HSS ULV factorization in [28], we see that many operations with the off-diagonal blocks have been avoided, and the redistribution is limited to only two matrices $\hat{D}_{c_1}, \hat{D}_{c_2}$ in (4.10). Therefore, this scheme has both lower complexity and better scalability, in general. As discussed in [32], the stability of this factorization is comparable to that of the case where the U, V generators have orthonormal columns and is only slightly worse.

Moreover, in this version, we allow each process to handle multiple nodes (or a subtree) of $\mathcal{T}$. This is also much more flexible than the scheme in [28]. The parallel

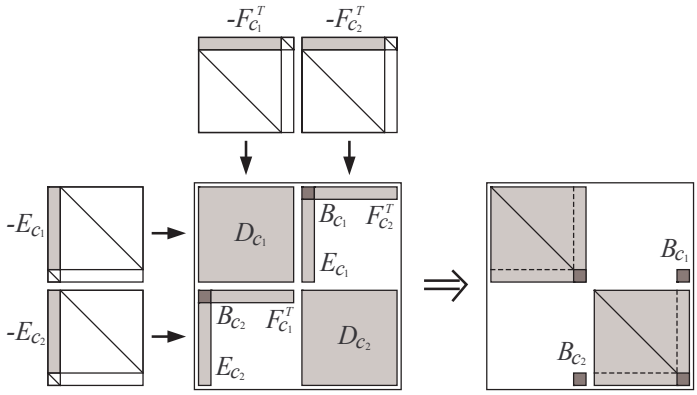


FIG. 4. *ULV factorization of the HSS form resulting from the partially matrix-free construction: introducing zeros into both the HSS block row and the HSS block column.*

scheme in [28] requires p to be larger than or equal to the number of leaves in $\mathcal{T}$. This limits the parallel performance, since either large p is required for small r, or a process needs to handle a large dense subproblem that is not structured or sufficiently compressed.

The corresponding solution algorithm for a linear system $Ax = b$ consists of multilevel forward and backward substitutions. With b partitioned into appropriate pieces b_i [38], at each step the forward substitution solves a system

$$\begin{pmatrix} L_{i,1} & \\ L_{i,2} & I \end{pmatrix} y_i = \begin{pmatrix} -E_i & I \\ I & 0 \end{pmatrix} P_i^T b_i.$$

The backward substitution solves

$$\begin{pmatrix} T_{i,1} & T_{i,2} \\ & I \end{pmatrix} z_i = y_i$$

and computes

$$x_i = Q_i \begin{pmatrix} -F_i^T & I \\ I & 0 \end{pmatrix} z_i.$$

No intergrid communications are needed for these operations. Intergrid operations occur only when appropriate intermediate solution pieces are merged or split, which is very fast.

5. Analysis of the parallel algorithms. In this section, we give a theoretical estimate of the parallel performance of our algorithms. Instead of requiring the user to test and find the right parameters, we hope to give the theoretical guidelines on how to choose the number of processes p and the shape of the process grids G_i for a given problem size to obtain the best scaling.

The optimal HSS partition occurs when the sizes of the leaf-level D_i generators are $O(r)$ [33]. This size is often chosen to be around $2r$. Thus, the height of the HSS tree is controlled by $\log \frac{n}{2r}$. To simplify the representations in our estimates, we assume that $\mathcal{T}$ is a perfect binary tree and the process grids at the same level have the same shape. The number of parallel levels L is bounded by both $\log p$ and the height of the HSS tree:

$$L \le \log p, \quad L \le \log \frac{n}{2r}.$$

Thus at level $l = 1, 2, \ldots, L$, each process grid contains p_l processes and the sizes of the D generators are n_l, where

$$p_l = \frac{p}{2^l}, \quad n_l = \frac{n}{2^l}. \tag{5.1}$$

Note that $p_L = p/2^L$ is the minimal size of all the process grids and $n_L = n/2^L$ is the leaf-level diagonal block size.

The randomized HSS construction algorithms combine the sampling operations for acquiring the data and the compression operations for processing the data. We analyze them separately as follows.

5.1. Sampling cost. Any operation for acquiring information about the matrix A is counted as sampling cost. The typical operations involved are matrix-vector multiplications and submatrix extractions. The computational costs of these operations are application-dependent, and the communication costs differ with respect to parallel distribution strategies. Still, we can compare how many times these operations are used in the two HSS construction algorithms.

The partially matrix-free construction algorithm needs

- $2(r + \mu)$ matrix-vector products for the U, V, R, W generators;
- $O(rn)$ entries of A for the B generators and the leaf-level D generators.

The fully matrix-free construction algorithm needs

- $2L(2r + \mu)$ matrix-vector products for the U, V, R, W and B generators;
- n_L matrix-vector products for the D_i generators.

The fully matrix-free version needs $2L$ times more matrix-vector products than the partially matrix-free version. This only introduces a logarithmic factor, and the impact on the scaling properties of the algorithms is minor. The benefit is that it avoids the direct access of the entries of A. The partially matrix-free version needs the extra time to extract $O(rn)$ entries of A.

In the following subsections, we then exclude these application-dependent costs for the matrix-vector multiplications and the extraction of matrix entries.

5.2. Analysis of the partially matrix-free algorithms.

5.2.1. Computational cost and storage requirement. All the local matrices involved in the partially matrix-free compression and factorization, including the partitioned sampling data within G_i, the HSS generators, and the factors, have sizes $O(r) \times O(r)$. The computational costs of randomized HSS construction (excluding the application-dependent matrix-vector multiplication cost), factorization, and solution are $O(r^2 n)$, $O(r^2 n)$, and $O(rn)$ flops, respectively. The storage cost of the skinny sampling matrices, the HSS generators, and the factors is all $O(rn)$. The detailed justifications are given in [38]. The parallel strategy based on the distributed HSS tree and the block-cyclic storage scheme of the generators can distribute the computational and storage costs evenly to each processor.

5.2.2. Communication cost. The communications during the partially matrix-free HSS construction include intergrid ones and intragrid ones.

1. *Intergrid redistribution.* The redistribution between a process grid and its parent level process grid involves only a pairwise exchange costing $O(1)$ messages and at most $O(\frac{r^2}{p_L})$ words. Taking a summation over all the levels yields

$$\#\text{messages} = \sum_{l=1}^{L} O(1) = O(\log p), \quad \#\text{words} = \sum_{l=1}^{L} O\left(\frac{r^2}{p_L}\right) = O\left(\frac{r^2}{p_L} \log p\right).$$

2. *Intragrid dense matrix kernels.* For an $O(r) \times O(r)$ matrix distributed on a $g_1 \times g_2$ grid, the SPRR algorithm (Algorithm 1) and other typical factorization algorithms [1] need $O(r \log g_1 + r \log g_2)$ messages and $O(r\frac{r}{g_1} \log g_1 + r\frac{r}{g_2} \log g_2)$ words. The expression shows that it is beneficial to choose $g_1 \approx g_2$, which means that the process grid is approximately square at each level l and g_1, g_2 are roughly equal to $\sqrt{p_l}$. The intragrid communication cost is then given by

$$\#\text{messages} = \sum_{l=1}^{L} O(r \log \sqrt{p_l}) = O(r \log^2 p),$$

$$\#\text{words} = \sum_{l=1}^{L} O\left(\frac{r^2}{\sqrt{p_l}} \log \sqrt{p_l}\right) = O\left(\frac{r^2}{\sqrt{p_L}} \log^2 p\right).$$

The communication cost of the factorization of the resulting HSS matrix is similar. These costs are summarized as the following proposition.

PROPOSITION 5.1. *Let p be the total number of processes, and let p_L be the minimum size of each process grid. The partially matrix-free randomized HSS construction and factorization both have the communication cost of $O(r \log^2 p)$ messages and $O(\frac{r^2}{\sqrt{p_L}} \log^2 p)$ words. As a comparison, recall that the direct HSS construction algorithm based on dense A needs $O(r \log^2 p)$ messages and $O(\frac{rn}{\sqrt{p_L}} \log p)$ words* [28]. *With the help of the skinny sampling matrices, the number of words in the construction is significantly reduced by nearly a factor of $O(\frac{n}{r})$.*

Remark 5.1. The construction and factorization algorithms not only have the same order of arithmetic operations but also share the same order of communication cost. In [28], the construction has both significantly higher flop counts and communication costs.

5.3. Analysis of the fully matrix-free algorithms.

5.3.1. Computational cost and storage requirement. The counts of the complexity and storage are similar to those of the original matrix-free HSS construction in [32]. Each node at level l is associated with matrix operations with costs $O(r^2 n_l)$, where n_l is given in (5.1). The total computational cost is thus $O(r^2 n \log n)$. This has an extra $\log n$ factor compared with the cost of the partially matrix-free HSS construction. The storage requirement is still $O(rn)$. The HSS factorization and solution costs remain $O(r^2 n)$ and $O(rn)$, respectively.

5.3.2. Communication cost. In the fully matrix-free HSS construction, temporary basis matrices as in (4.6) and (4.7) are computed. The process grid associated with each node at level l contains p_l processes, with p_l given in (5.1). The process grid is constructed to be of size $2^{L-l}\sqrt{p_L} \times \sqrt{p_L}$ so as to match the elongated temporary basis matrices. Then at every level, a distributed basis matrix needs $\frac{n_L}{\sqrt{p_L}} \times \frac{r}{\sqrt{p_L}}$ local storage with each process.

1. *Multilevel intergrid redistributions.* At a sampling step corresponding to a level l, a skinny sampling matrix needs to be redistributed to the block-row distribution at the bottom level of the extended tree $\widetilde{\mathcal{T}}$ (Figure 1), and the sampling result at the bottom level needs to be redistributed back to the current level. After an upper-level HSS off-diagonal matrix-vector multiplication involving $A - \mathbf{D}^{(l)}$ (e.g., (4.5) and (4.8)), the temporary result needs to be redistributed up to the top level and then back down to the current level.

The cost for such redistributions is

$$\#\text{messages} = \sum_{l=1}^{L} O(L) = O(\log^2 p),$$
$$\#\text{words} = \sum_{l=1}^{L} O\left(\frac{rn_l}{p_l} L\right) = O\left(\frac{rn}{p} \log^2 p\right).$$

2. *Intragrid dense operations.* For an $n_l \times r$ matrix distributed on a $2^{L-l}\sqrt{p_L} \times \sqrt{p_L}$ grid, the adaptive randomized orthogonalization requires $O(r \log p)$ messages and $O(\frac{rn_L}{\sqrt{p_L}} \log p)$ words. The intragrid communication cost at all the levels is

$$\#\text{messages} = \sum_{l=1}^{L} O(r \log p) = O(r \log^2 p),$$
$$\#\text{words} = \sum_{l=1}^{L} O\left(\frac{rn_L}{\sqrt{p_L}} \log p\right) = O\left(\frac{rn}{p} \sqrt{p_L} \log^2 p\right).$$

3. *Upper-level off-diagonal multiplication.* Then consider the communication at level l for the upper-level HSS off-diagonal matrix-vector multiplication involving $\left(A - \mathbf{D}^{(l)}\right)$ (e.g., (4.5) and (4.8)).
The largest matrix-matrix multiplication occurs at the current level between the transpose of an $n_l \times r$ matrix and another $n_l \times r$ matrix. This costs $O(r \log p)$ messages and $O(\frac{rn_L}{\sqrt{p_L}})$ words. For the upper levels, all the matrices are $O(r) \times O(r)$, and each multiplication costs $O(r \log p)$ messages and $O(\frac{r^2}{\sqrt{p_L}})$ words. The communications cost

$$\#\text{messages} = \sum_{l=1}^{L} \left(O(r \log p) + \sum_{j=l+1}^{L} O(r \log p) \right) = O(r \log^3 p),$$
$$\#\text{words} = \sum_{l=1}^{L} \left(O\left(\frac{rn_L}{\sqrt{p_L}}\right) + \sum_{j=l+1}^{L} O\left(\frac{r^2}{\sqrt{p_L}}\right) \right) = O\left(\frac{rn}{p} \sqrt{p_L} \log^2 p\right).$$

We can then sum up all the communication costs during the fully matrix-free HSS algorithms as the following proposition.

PROPOSITION 5.2. *Assume that larger process grids are formed by stacking smaller ones along the columns. Then the fully matrix-free HSS construction has the communication cost of $O(r \log^3 p)$ messages and $O(\frac{rn}{p} \sqrt{p_L} \log^2 p)$ words. Again, these costs are smaller than those in* [28] *by nearly a factor of $O(\frac{n}{r})$. The communication cost of the factorization is about the same as that in Proposition* 5.1.

Although the matrix-free HSS construction involves more complicated operations, by a careful choice of process grids we obtain a communication cost similar to the partially matrix-free one up to a logarithmic factor.

5.4. Parallel runtime. Combining our estimates of the computation and communication costs, we can give estimates for the parallel runtime, and with that we can find out how to achieve nearly optimal performance for a given problem size n.

The partially matrix-free HSS construction time is

$$T_1 = O\left(\frac{r^2 n \log n}{p}\right) + O(r \log^2 p) + O\left(\frac{r^2}{\sqrt{p_L}} \log^2 p\right).$$

The fully matrix-free construction HSS construction time is

$$T_2 = O\left(\frac{r^2 n \log n}{p}\right) + O(r \log^3 p) + O\left(\frac{nr}{p} \sqrt{p_L} \log^2 p\right).$$

In practice, the HSS rank bound r depends on the application and even on n, and then we can choose p accordingly to balance the computational time and communication time. Some examples are as follows:

- One case is $r = O(\log^\mu n)$, $\mu \geq 1$ (e.g., $\mu = 1$ as encountered in the discretization of some 1D kernels [10, 25, 38]). Then we choose p to be

  $$p = O(rn) = O(n \log^\mu n),$$

  so that

  $$p_L = O(\log^{2\mu} n), \quad T_1 = O(r \log^2 n), \quad T_2 = O(r \log^3 n). \tag{5.2}$$

- Another case is $r = O(n^\mu)$, $\mu \leq 1/2$ (e.g., $\mu = 1/2$ as encountered in the discretization of some 2D kernels [10]). Then we choose p to be

  $$p = O(rn) = O(n^{\mu+1}),$$

  so that

  $$p_L = O(n^{2\mu}), \quad T_1 = O(r \log^2 n), \quad T_2 = O(r \log^3 n). \tag{5.3}$$

For these cases, we come to the conclusion that we can use $O(rn)$ processes and $O(r \log^2 n)$ or $O(r \log^3 n)$ time for the HSS construction. The logarithmic terms result from HSS tree traversals and dense matrix kernels. The fully matrix-free version has an additional $\log n$ factor because of the multilevel redistribution. In addition, since the total storage is $O(rn)$, the $O(rn)$ processes also match the intuition that each process should store a constant amount of data.

The factorization time can be similarly studied and is omitted.

6. Applications and performance tests. In this section, we systematically test the performance of our massively parallel randomized direct solvers and verify our analysis. We show the direct solution of some important and challenging dense linear systems. In general, our methods can be used to solve large-scale dense systems with the low-rank structures and preferably fast matrix-vector products. Typical examples include Toeplitz problems, n-body systems, and Fredholm integral equations of the second kind. We consider a challenging discretized equation here.

6.1. Foldy–Lax equations modelling wave propagation with multiple scattering. In the acoustic scattering theory, the kernel function is related to the fundamental solution of the 3D Helmholtz equation for a given wavenumber k:

$$G(x, y) = \frac{e^{ik|x-y|}}{4\pi|x-y|}, \quad x, y \in \mathbb{R}^3.$$

Here, we consider the Foldy–Lax formulation [4, 9, 15], which is often used for analyzing multiple scattering effects among point scatterers. Given the incident field u^{inc}, the total field u satisfies

$$-\Delta u - \left(k^2 + \sum_{j=1}^{n} \sigma_j \delta(x - x_j)\right) u = 0,$$

where x_j is the location of a scatterer and σ_j is its scattering strength. The scattered field can be represented by

$$u(x) - u^{\text{inc}}(x) = \sum_{j=1}^{n} G(x_j, x)\sigma_j u(x).$$

Restricting the representation to the scattering points, we can obtain an n-body system of the form

$$(I + K)u = f, \tag{6.1}$$

where the kernel matrix K and the right-hand side vector f are defined as

$$K_{jk} = \begin{cases} 0 & \text{if } j = k, \\ -G(x_j, x_k)\sigma_k & \text{otherwise,} \end{cases} \qquad f_j = u^{\text{inc}}(x_j).$$

The solution is $u_j = u(x_j)$. The dense oscillatory kernel posts difficulties for both direct and iterative solvers.

Our matrix-free direct solver is a natural candidate. Fast matrix-vector products can be achieved via the fast multipole method (FMM) [10, 7] or the fast discrete convolution method. The performance of our direct solver is primarily determined by three quantities: the matrix size n, the HSS rank r, and the number of processes p. In order to focus our attention on these quantities, we further simplify the physical system by restricting the positions of the scatterers to a 2D $M \times N$ mesh with $M \leq N$. By choosing the mesh size we have full control of the algebraic properties based on the following relation:

$$n = MN, \quad r = O(M \log N), \tag{6.2}$$

where the estimate of r follows from standard FMM techniques. In addition, on a 2D mesh, the discretized kernel function G has a Toeplitz-block-Toeplitz (TBT) structure. A TBT matrix can be extended to a discrete 2D convolution which can be diagonalized by a fast Fourier transform (FFT). By using the well-studied parallel 2D FFT algorithm, we have an $O(n \log n)$ complexity kernel-independent way of performing parallel matrix-vector multiplications. Note that even with the restrictions to regular grids, by far a fast direct TBT matrix solver remains a challenging topic in structured matrix computations. On the other hand, our parallel matrix-free direct solvers can find the solutions with high efficiency.

We ran two types of tests on Purdue's community cluster Conte with Intel Fortran compiler and Intel Math Kernel Library. Conte has HP compute nodes each containing two 8-core Intel Xeon-E5 processors and 64GB of memory. To conveniently assess the accuracy, we generate the right-hand side vector b by computing Au for a given vector u, and then measure the relative error

$$\frac{\|u - \tilde{u}\|}{\|u\|},$$

where $\tilde{u}$ is the approximate solution obtained with our HSS solvers.

6.2. Quasi-1D weak and strong scaling tests. We first fix the mesh size M along one dimension and increase the mesh size N along the other dimension by a factor of 2. The problem still corresponds to 2D domains, but it has a quasi-1D rank structure in the sense that r in (6.2) does not grow much with respect to n. According to the estimate (5.2), if we choose $p = O(n)$, the compression times of the partially and fully matrix-free HSS constructions should be controlled by

$$T_1 = O(r \log^2 n) = O(\log^3 n), \quad T_2 = O(r \log^3 n) = O(\log^4 n),$$

respectively. (Note that this example has rank behaviors similar to those of the case of Toeplitz problems [38], but the ranks are much higher. Thus, this example is a more interesting test for our scalable solvers.)

The results are listed in Table 2. The matrix size n ranges from 40,000 to 640,000. In terms of the physical scales, the shorter side of the mesh spans 20 wavelengths, and the longer side increases from 20 to 320 wavelengths. The compression tolerance is 10^{-6}. We use the rank bound 3200 for the partially matrix-free construction, and the actual numerical rank r is returned by the rank-revealing factorization. For the fully matrix-free construction, r is determined adaptively in randomized sampling, and each sampling group has 128 vectors ($s = 128$ in Algorithm 2).

The costs and storage scale nearly linearly in n. Moreover, the desired polylogarithmic type runtime is observed for both the partially and the fully matrix-free solutions, which would not be achieved by directly compressing the global dense storage. Figure 5 shows the scaling of the construction time and the factorization time. Reference lines for our predictions are included. The actual performance of both methods matches or even exceeds the predictions.

As predicted, the partially matrix-free algorithm has a smaller storage requirement and a faster runtime due to the smaller number of matrix-vector products and the additional structures within the generators. The fully matrix-free algorithm gives a tighter HSS rank r, and the accuracy is a little higher due to the better stability.

It is worth pointing out that once the construction and the factorization are done, the solution is very fast due to the structure and the scalability. For example, for $n = 640{,}000$, the partially matrix-free version costs around 10 minutes to construct the HSS form, around 24 seconds to factorize it, and only 0.2 second to solve a system.

Furthermore, much higher flop rates can be achieved by solving multiple right-hand sides. For $n = 640{,}000$ and 128 right-hand sides, the partially matrix-free solution time is only 2.92 seconds. See the last rows of Table 2(b)–(c). Thus, it is very attractive to apply the direct solvers to problems with multiple right-hand sides or to use them (with low-accuracy compression) as preconditioners.

In addition, both methods are significantly faster and require much less memory than a standard direct solver such as the ScaLAPACK LU factorization. See Table 2(d). As the problem size increases, the ScaLAPACK LU solver becomes very slow and soon runs out of memory.

To see the tradeoff between the accuracy and the computation time/storage/r, we also test the fully matrix-free solver with different compression tolerances. See Table 3. By varying the compression tolerance, we can conveniently control the solution accuracy, cost, and storage. As expected, the HSS ranks are smaller, and the solver is faster for larger tolerances.

Last, we show the strong scaling test. For the 200×400 mesh in Table 2, we use different numbers of processes and report the runtime. See Table 4. The HSS construction and factorization scale reasonably well for both methods. (The scaling

TABLE 2

Quasi-1D weak scaling test with compression tolerance 10^{-6}*, where* mat-vec *stands for a matrix-vector multiplication and RHS stands for a right-hand side* f *in* (6.1).

(a) Problem setup

Mesh size ($M \times N$)	200×200	200×400	200×800	200×1600	200×3200
Number of wavelengths	20×20	20×40	20×80	20×160	20×320
Matrix size $n = MN$	40,000	80,000	160,000	320,000	640,000
Number of processes p	16	32	64	128	256
Total HSS tree levels L	5	6	7	8	9

(b) Partially matrix-free direct solution

Mesh size ($M \times N$)	200×200	200×400	200×800	200×1600	200×3200
Number of mat-vecs	6,400	6,400	6,400	6,400	6,400
Sampling time	8.31s	20.58s	28.12s	48.50s	90.08s
HSS construction time	295.52s	417.41s	488.64s	504.47s	570.66s
HSS construction flops	$5.31E12$	$1.15E13$	$2.42E13$	$5.01E13$	$1.02E14$
HSS rank r	1,490	1,824	2,532	2,996	3,153
HSS storage size	$2.24E8$	$4.88E8$	$1.04E9$	$2.17E9$	$4.43E9$
Factorization time	11.66s	14.92s	17.70s	21.82s	23.79s
Factorization flops	$8.08E11$	$2.54E12$	$8.92E12$	$2.59E13$	$5.89E13$
Solution time (1 RHS)	0.07s	0.10s	0.12s	0.17s	0.20s
Solution flops (1 RHS)	$4.49E8$	$9.76E8$	$2.08E9$	$4.33E9$	$8.86E9$
Relative error (2-norm)	$1.39E-7$	$1.70E-7$	$2.51E-7$	$9.56E-7$	$1.37E-5$
Relative error (∞-norm)	$4.72E-7$	$7.40E-7$	$1.66E-6$	$5.50E-6$	$5.18E-5$
Solution time (128 RHSs)	1.45s	1.77s	2.08s	2.79s	2.92s
Solution flops (128 RHSs)	$5.75E10$	$1.25E11$	$2.66E11$	$5.54E11$	$1.13E12$

(c) Fully matrix-free direct solution

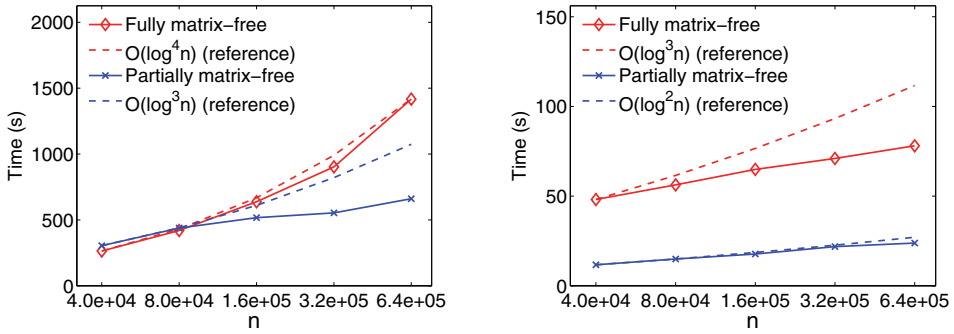
Mesh size ($M \times N$)	200×200	200×400	200×800	200×1600	200×3200
Number of mat-vecs	16,116	20,858	26,351	32,465	39,019
HSS construction time	263.36s	419.22s	637.88s	902.61s	1416.90s
HSS construction flops	$9.14E12$	$2.77E13$	$7.61E13$	$1.96E14$	$4.85E14$
HSS rank r	1,411	1,518	1,576	1,627	1,674
HSS storage	$3.08E8$	$6.82E8$	$1.45E9$	$3.03E9$	$6.20E9$
Factorization time	48.11s	56.25s	64.87s	71.02s	78.08s
Factorization flops	$1.89E12$	$4.43E12$	$9.78E12$	$2.08E13$	$4.31E13$
Solution time (1 RHS)	0.84s	1.02s	1.34s	1.63s	1.97s
Solution flops (1 RHS)	$8.68E8$	$1.92E9$	$4.11E9$	$8.54E9$	$1.75E10$
Relative error (2-norm)	$8.89E-8$	$9.58E-8$	$1.06E-7$	$1.17E-7$	$1.28E-7$
Relative error (∞-norm)	$3.28E-7$	$3.89E-7$	$6.37E-7$	$8.63E-7$	$1.34E-6$
Solution time (128 RHSs)	3.04s	3.41s	4.13s	4.60s	5.12s
Solution flops (128 RHSs)	$1.11E11$	$2.46E11$	$5.26E11$	$1.09E12$	$2.24E12$

(d) ScaLAPACK LU

Factorization time	568.88s	2739.15s	8679.70s	\
Storage	$1.60E9$	$6.40E9$	$2.56E10$	out of memory

of the sampling in the partially matrix-free solver is worse due to the overhead in the 2D FFT algorithm. This is, of course, problem dependent.)

6.3. 2D weak scaling test. We then use square domains ($M = N$) and double M, N each time. The leading factor governing the HSS rank growth is $O(\sqrt{n})$ and is not necessarily small. Our estimates suggest that speedup can be observed in the HSS operations for $O(n^{1.5})$ processes. Considering the distributed transpose algorithm used in the parallel 2D FFT, we choose $p = O(\sqrt{n})$. The partially matrix-free HSS construction time is about $O(n^{1.5})$ (with some logarithmic terms ignored), and the



(a) *HSS construction* (b) *ULV factorization*

FIG. 5. *Quasi-1D weak scaling test.*

TABLE 3

Quasi-1D weak scaling test for the fully matrix-free HSS solver with different compression tolerances.

(a) Compression tolerance 10^{-2}

Mesh size ($M \times N$)	200×200	200×400	200×800	200×1600
Number of mat-vecs	8,981	11,443	14,271	17,113
HSS construction time	58.95s	109.13s	179.62s	296.91s
HSS rank r	495	527	551	564
HSS storage	$1.46E8$	$3.04E8$	$6.25E8$	$1.27E9$
Factorization time	15.19s	15.63s	17.98s	18.06s
Relative error (∞-norm)	$3.75E-3$	$5.79E-3$	$7.82E-3$	$1.42E-2$

(b) Compression tolerance 10^{-4}

Mesh size ($M \times N$)	200×200	200×400	200×800	200×1600
Number of mat-vecs	12,364	16,057	20,373	24,967
HSS construction time	147.99s	247.47s	386.87s	591.04s
HSS rank r	946	1,020	1,072	1,109
HSS storage	$2.15E8$	$4.64E8$	$9.76E8$	$2.01E9$
Factorization time	27.06s	29.66s	34.00s	37.39s
Relative error (∞-norm)	$2.83E-5$	$4.06E-5$	$5.96E-5$	$8.14E-5$

(c) Compression tolerance 10^{-8}

Mesh size ($M \times N$)	200×200	200×400	200×800	200×1600
Number of mat-vecs	19,626	25,492	32,592	39,802
HSS construction time	433.63s	670.18s	1011.89s	1494.98s
HSS rank r	1,832	1,959	2,031	2,111
HSS storage	$4.00E8$	$9.01E8$	$1.94E9$	$4.07E9$
Factorization time	76.92s	88.60s	106.33s	117.29s
Relative error (∞-norm)	$2.61E-9$	$3.36E-9$	$5.54E-9$	$6.65E-9$

fully matrix-free construction has one more $\log n$ factor.

The results are included in Table 5. The matrix size n ranges from 15,000 to 10^6. The physical scales of the problems are from 12.5 to 100 wavelengths. The compression tolerance is 10^{-4}. The rank bound of the partially matrix-free algorithm starts from 2,000 and doubles each time. For the fully matrix-free construction, the HSS rank is determined adaptively as in the previous test. Figure 6 shows that the scaling of the parallel runtime matches very well with our prediction. Both methods give remarkable results for large systems, and the partially matrix-free version is about

TABLE 4
Quasi-1D strong scaling test with compression tolerance 10^{-6}.

(a) Partially matrix-free HSS solver

Number of processes p	16	32	64	128
Sampling time	10.68s	10.36s	10.55s	14.45s
HSS construction time	222.02s	122.37s	66.74s	34.12s
Factorization time	23.43s	16.46s	8.69s	6.05s
Solution time (1 RHS)	0.14s	0.08s	0.07s	0.07s
Solution time (128 RHS)	2.25s	1.50s	1.12s	0.82s

(b) Fully matrix-free HSS solver

Number of processes p	16	32	64	128
HSS construction time	353.13s	198.15s	174.69s	144.58s
Factorization time	69.36s	45.01s	22.15s	15.44s
Solution time (1 RHS)	1.66s	1.03s	0.75s	0.83s
Solution time (128 RHS)	4.36s	2.78s	1.64s	1.59s

two times faster in the construction and the factorization.

The largest matrix has 10^{12} entries, and the dense storage would require 7,451 GB of storage. However, with our structured solvers, we are able to give a direct solution with only 32 nodes and their limited memory. In fact, we need to store only 3.25×10^{10} nonzeros, which is a saving of over 30 times. In the partially matrix-free solution, the HSS construction and factorization take about 4,500 seconds and 280 seconds, respectively, and the solution takes only 0.63 second for one right-hand side and 8.92 seconds for 128 right-hand sides. The solution error is comparable to the compression tolerance for all the tests.

7. Conclusions. We have designed two types of distributed-memory randomized matrix-free direct solvers with high efficiency, scalability, and flexibility. We demonstrate a series of significant advantages over existing parallel HSS work. The fully matrix-free solver uses matrix-vector products only, while the partially matrix-free version provides a slightly faster option for matrices with their entries easily accessible. For the fully matrix-free solver, we improve the original algorithm by eliminating the pseudoinverses and introducing a way of constructing more compact nested off-diagonal basis matrices. The parallel implementation features a PBLAS-3 synchronized adaptive rank detection. Detailed analyses of the algorithms show a significant reduction of the computational cost, storage requirement, and communication cost as compared with traditional direct HSS constructions based on the dense matrix. The weak scaling tests verify our estimates and demonstrate a good scalability for large problems.

We test the solvers on Foldy–Lax equations with a 2D configuration of scatterers in three dimensions. For dense discretized matrices of sizes up to 10^6, we observe significant advantages in the storage and computation time. This also suggests that the randomized algorithms can be naturally integrated into structured multifrontal solvers [34] for 3D sparse problems. That is, they can be used to factorize the dense intermediate Schur complements. This strategy would lead to fast direct solutions of sparse discretized problems (e.g., Helmholtz equations) of sizes in the magnitude of 10^9. This will be addressed in separate work. For more general dense problems, the solvers may serve as preconditioners when low-accuracy off-diagonal compression is used.

TABLE 5

2D weak scaling test, where mat-vec *stands for a matrix-vector multiplication, and RHS stands for a right-hand side f in* (6.1).

(a) Problem setup

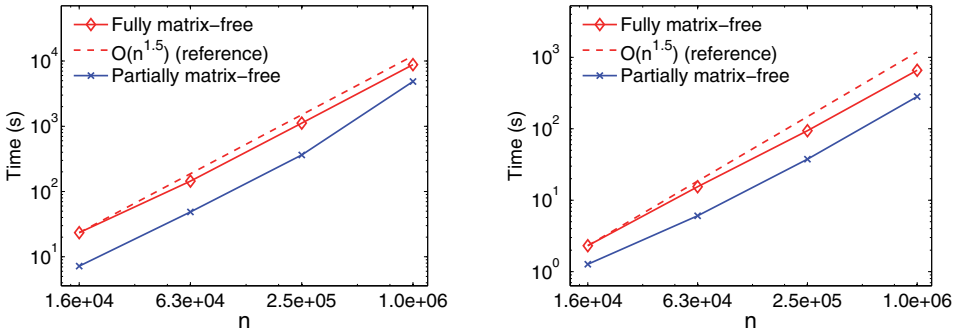
Mesh size ($M \times N$)	125×125	250×250	500×500	1000×1000
Number of wavelengths	12.5×12.5	25×25	50×50	100×100
Matrix size n	15,625	62,500	250,000	1,000,000
Number or processes p	64	128	256	512
Total HSS tree levels L	4	5	6	7

(b) Partially matrix-free direct solution

Mesh size ($M \times N$)	125×125	250×250	500×500	1000×1000
Number of mat-vecs	2,000	4,000	8,000	16,000
Sampling time	3.60s	14.45s	70.81s	345.73s
HSS construction time	3.60s	34.12s	292.89s	4509.73s
HSS construction flops	$2.48E11$	$4.22E12$	$7.07E13$	$1.17E15$
HSS rank	566	1,348	3,505	7,827
HSS storage	$4.67E7$	$3.97E8$	$3.31E9$	$2.73E10$
Factorization time	1.27s	6.05s	37.52s	282.23s
Factorization flops	$1.15E11$	$2.97E12$	$8.47E13$	$1.74E15$
Solution time (1 RHS)	0.02s	0.07s	0.29s	0.63s
Solution flops (1 RHS)	$9.34E7$	$7.93E8$	$6.63E9$	$5.45E10$
Relative error (2-norm)	$1.44E-5$	$2.10E-5$	$4.13E-5$	$3.74E-4$
Relative error (∞-norm)	$4.24E-5$	$9.42E-5$	$2.48E-4$	$2.27E-3$
Solution time (128 RHSs)	0.22s	0.82s	2.41s	8.92s
Solution flops (128 RHSs)	$1.20E10$	$1.02E11$	$8.48E11$	$6.98E12$

(c) Fully matrix-free direct solution

Mesh size ($M \times N$)	125×125	250×250	500×500	1000×1000
Number of mat-vecs	6,312	16,306	41,434	103,939
HSS construction time	23.44s	144.58s	1116.80s	8783.49s
HSS construction flops	$4.59E11$	$1.09E13$	$2.55E14$	$5.57E15$
HSS rank	577	1,222	2,559	5,450
HSS storage	$5.29E7$	$4.60E8$	$3.90E9$	$3.25E10$
Factorization time	2.31s	15.44s	93.86s	659.22s
Factorization flops	$1.91E11$	$3.38E12$	$5.79E13$	$9.71E14$
Solution time (1 RHS)	0.14s	0.83s	3.31s	16.74s
Solution flops (1 RHS)	$1.54E8$	$1.33E9$	$1.13E10$	$9.39E10$
Relative error (2-norm)	$6.52E-6$	$8.30E-6$	$1.09E-5$	$1.48E-5$
Relative error (∞-norm)	$2.23E-5$	$3.69E-5$	$5.92E-5$	$1.11E-4$
Solution time (128 RHSs)	0.35s	1.59s	5.56s	25.58s
Solution flops (128 RHSs)	$1.97E10$	$1.71E11$	$1.45E12$	$1.20E13$



(a) *HSS construction* (b) *ULV factorization*

FIG. 6. *2D weak scaling test.*

Acknowledgments. We are grateful to the associate editor and the anonymous referees for the valuable suggestions, and to Peijun Li for help with the integral equations. We also thank Yingchong Situ and Zixing Xin for some discussions about parallel computing.

REFERENCES

[1] L. S. BLACKFORD, J. CHOI, A. CLEARY, E. D'AZEUEDO, J. DEMMEL, I. DHILLON, J. DONGARRA, ET AL., *ScaLAPACK Users' Guide*, SIAM, Philadelphia, 1997.
[2] *BLACS, Basic Linear Algebra Communication Subprograms*, http://www.netlib.org/blacs.
[3] S. BÖRM, L. GRASEDYCK, AND W. HACKBUSCH, *Introduction to hierarchical matrices with applications*, Eng. Anal. Bound. Elem., 27 (2003), pp. 405–422.
[4] A. CHAI, M. MOSCOSO, AND G. PAPANICOLAOU, *Imaging strong localized scatterers with sparsity promoting optimization*, SIAM J. Imaging Sci., 7 (2014), pp. 1358–1387.
[5] S. CHANDRASEKARAN, P. DEWILDE, M. GU, T. PALS, X. SUN, A.-J. VAN DER VEEN, AND D. WHITE, *Some fast algorithms for sequentially semiseparable representations*, SIAM J. Matrix Anal. Appl., 27 (2005), pp. 341–364.
[6] S. CHANDRASEKARAN, 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] R. COIFMAN, R. VLADIMIR, AND S. WANDZURA, *The fast multipole method for the wave equation: A pedestrian prescription*, IEEE Antennas Propagat. Mag., 35 (1993), pp. 7–12.
[8] Y. EIDELMAN AND I. GOHBERG, *On a new class of structured matrices*, Integral Equations Opererator Theory, 34 (1999), pp. 293–324.
[9] L. L. FOLDY, *The multiple scattering of waves.* I. *General theory of isotropic scattering by randomly distributed scatterers*, Phys. Rev. (2), 67 (1945), pp. 107–119.
[10] L. GREENGARD AND V. ROKHLIN, *A fast algorithm for particle simulations*, J. Comput. Phys., 73 (1987), pp. 325–348.
[11] 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.
[12] W. HACKBUSCH AND S. BÖRM, *Data-sparse approximation by adaptive $\mathcal{H}^2$-matrics*, Computing, 69 (2002), pp. 1–35.
[13] W. HACKBUSCH, L. GRASEDYCK, AND S. BÖRM, *An introduction to hierarchical matrices*, Math. Bohem., 127 (2002), pp. 229–241.
[14] N. HALKO, P. G. MARTINSSON, AND J. A. TROPP, *Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions*, SIAM Rev., 53 (2011), pp. 217–288.
[15] M. LAX, *Multiple scattering of waves*, Rev. Modern Phys., 23 (1951), pp. 287–310.
[16] 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.
[17] L. LIN, J. LU, AND L. YING, *Fast construction of hierarchical matrix representation from matrix-vector multiplication*, J. Comput. Phys., 230 (2011), pp. 4071–4087.

[18] 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.
[19] P. G. Martinsson, V. Rokhlin, and M. Tygert, *A randomized algorithm for the decomposition of matrices*, Appl. Comput. Harmon. Anal., 30 (2011), pp. 47–68.
[20] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, http://www.mpi-forum.org.
[21] *ParMETIS, Parallel Graph Partitioning and Fill-Reducing Matrix Ordering*, http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.
[22] *ScaLAPACK, Scalable Linear Algebra PACKage*, http://www.netlib.org/scalapack.
[23] *Scotch and PT-Scotch, Software Package and Libraries for Sequential and Parallel Graph Partitioning, Static Mapping and Clustering, Sequential Mesh and Hypergraph Partitioning, and Sequential and Parallel Sparse Matrix Block Ordering*, http://www.labri.fr/perso/pelegrin/scotch/.
[24] J. Shen, Y. Wang, and J. Xia, *Fast structured direct spectral methods for differential equations with variable coefficients,* I. *The one-dimensional case*, SIAM J. Sci. Comput., 38 (2016), pp. A28–A54.
[25] P. Starr, *On the Numerical Solution of One-Dimensional Integral and Differential Equations*, Ph.D. thesis, Yale University, New Haven, CT, 1992.
[26] E. E. Tyrtyshnikov, *Incomplete cross approximation in the mosaic-skeleton method*, Computing, 64 (2000), pp. 367–380.
[27] S. Wang, M. V. de Hoop, J. Xia, and X. S. Li, *Massively parallel structured multifrontal solver for time-harmonic elastic waves in 3D anisotropic media*, Geophys. J. Int., 191 (2012), pp. 346–366.
[28] 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.
[29] F. Woolfe, E. Liberty, V. Rokhlin, and M. Tygert, *A fast randomized algorithm for the approximation of matrices*, Appl. Comput. Harmon. Anal., 25 (2008), pp. 335–366.
[30] Y. Xi and J. Xia, *On the stability of some hierarchical rank structured matrix algorithms*, SIAM J. Matrix Anal. Appl., submitted.
[31] Y. Xi, J. Xia, S. Cauley, and V. Balakrishnan, *Superfast and stable structured solvers for Toeplitz least squares via randomized sampling*, SIAM J. Matrix Anal. Appl., 35 (2014), pp. 44–72.
[32] Y. Xi, J. Xia, and R. Chan, *A fast randomized eigensolver with structured LDL factorization update*, SIAM J. Matrix Anal. Appl., 35 (2014), pp. 974–996.
[33] J. Xia, *On the complexity of some hierarchical structured matrix algorithms*, SIAM J. Matrix Anal. Appl., 33 (2012), pp. 388–410.
[34] J. Xia, *Randomized sparse direct solvers*, SIAM J. Matrix Anal. Appl., 34 (2013), pp. 197–227.
[35] 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.
[36] 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.
[37] J. Xia, Z. Li, and X. Ye, *Effective matrix-free preconditioning for the augmented immersed interface method*, J. Comput. Phys., 303 (2015), pp. 295–312.
[38] 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.