SIAM J. SCI. COMPUT.
Vol. 35, No. 6, pp. C519–C544

EFFICIENT SCALABLE ALGORITHMS FOR SOLVING DENSE LINEAR SYSTEMS WITH HIERARCHICALLY SEMISEPARABLE STRUCTURES*

SHEN WANG†, XIAOYE S. LI‡, JIANLIN XIA†, YINGCHONG SITU§, AND MAARTEN V. DE HOOP†

Abstract. Hierarchically semiseparable (HSS) matrix techniques are emerging in constructing superfast direct solvers for both dense and sparse linear systems. Here, we develop a set of novel parallel algorithms for key HSS operations that are used for solving large linear systems. These are parallel rank-revealing QR factorization, HSS constructions with hierarchical compression, ULV HSS factorization, and HSS solutions. The HSS tree-based parallelism is fully exploited at the coarse level. The `BLACS` and `ScaLAPACK` libraries are used to facilitate the parallel dense kernel operations at the fine-grained level. We appply our new solvers for discretized Helmholtz equations for multifrequency seismic imaging and iteratively solve time-harmonic seismic inverse boundary value problems. In particular, we use the HSS algorithms to solve the dense Schur complement systems associated with the root separator of the separator tree obtained from nested dissection of the graph of discretized Helmholtz equations. We demonstrate that the new approach is much faster and uses much less memory than the LU factorization algorithm for both two-dimensional and three-dimensional problems, using up to 8912 processing cores. This is the first work in parallelizing HSS algorithms and conducting detailed performance analysis on a large parallel machine. This also lays a good foundation for developing scalable sparse structured factorization algorithms for general sparse linear systems.

Key words. HSS matrix, parallel HSS algorithm, low-rank property, compression, HSS construction, direct solver

AMS subject classifications. 15A23, 65F05, 65F30, 65F50

DOI. 10.1137/110848062

1. Introduction.

In recent years, rank structured matrices have attracted much attention and have been widely used in the fast solutions of various partial differential equations, integral equations, and eigenvalue problems. Several useful rank structured matrix representations have been developed, such as $\mathcal{H}$-matrices [17, 15, 14], $\mathcal{H}^2$-matrices [4, 5, 16], quasi-separable matrices [1, 9], and semiseparable matrices [6, 23].

Here, we focus on a type of semiseparable structures, called *hierarchically semiseparable* (HSS) forms. Key applications of the HSS algorithms, coupled with sparse matrix techniques such as multifrontal solvers [28], have been shown very useful in solving certain large-scale discretized PDEs and computational inverse problems. For example, they can be built into parallel structured direct solvers for Helmholtz equations

*Submitted to the journal's Software and High-Performance Computing section September 15, 2011; accepted for publication (in revised form) October 15, 2013; published electronically December 11, 2013. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-05CH11231.

http://www.siam.org/journals/sisc/35-6/84806.html

†Department of Mathematics, Purdue University, West Lafayette, IN 47907 (wang273@math.purdue.edu, xiaj@math.purdue.edu, mdehoop@math.purdue.edu). The research of the third author was supported in part by NSF grants DMS-1115572 and CHE-0957024.

‡Lawrence Berkeley National Laboratory, Berkeley, CA 94720 (xsli@lbl.gov). The research of this author was supported in part by the director of the Office of Science, Office of Advanced Scientific Computing Research of the U.S. Department of Energy under contract DE-AC02-05CH11231.

§Department of Computer Science, Purdue University, West Lafayette, IN 47907 (ysitu@cs.purdue.edu).

arising from time-harmonic wave equation modeling prevailing in the energy industry [24, 25]. Direct solvers are particularly important in the time-harmonic formulation of the seismic inverse boundary value problem. Following the iterative reconstruction approach, one has to solve the Helmholtz equation for many right-hand sides on a large domain for a selected set of frequencies. The computational accuracy can be controlled, namely, in concert with the accuracy of the data.

An HSS representation has a binary tree structure, called the HSS tree, and the HSS operations can be generally conducted following the traversal of this tree in a parallel fashion. However, the existing studies of the HSS structures focus only on their mathematical aspects, and the current HSS methods are only implemented in sequential computations. Similar limitations also exist for some other rank structured methods.

Here, we present new parallel and efficient HSS algorithms and study their scalability. We concentrate on the three most significant HSS algorithms: the parallel construction of an HSS representation or approximation for a dense matrix using a parallel block compression scheme, the parallel ULV-type factorization [7] of such a matrix, and the parallel solution. The operation complexities of the HSS construction, factorization, and solution algorithms are $\mathcal{O}(rn^2)$, $\mathcal{O}(r^2n)$, and $\mathcal{O}(rn)$, respectively, where r is the maximum numerical rank and n is the size of the dense matrix [7, 29]. (The numerical rank of a matrix is the number of its singular values greater than a given tolerance.) We further analyze the communication complexity of our parallel algorithms and demonstrate parallel performance on a real machine.

Our parallel HSS construction consists of three phases: parallel block compression based on a modified Gram–Schmidt method with column pivoting, parallel row compression, and parallel column compression. The parallel HSS factorization involves the use of two children's contexts for a given parent context. The communication patterns are composed of intracontext and intercontext ones. Similar strategies are also applied to the HSS solution. Some tree techniques for symmetric positive definite HSS matrices in [30] are generalized in order to efficiently handle nonsymmetric matrices in parallel. We present analyses of the communication costs in the different procedures. For example, in the HSS construction, the number of messages and the number of words transferred are $\mathcal{O}(r \log^2 P + \log P)$ and $\mathcal{O}(rn \log P + r^2 \log^2 P + rn)$, respectively, where P is the number of processes. In our numerical experiments, we confirm the accuracy and the weak scaling of the methods when they are used as kernels for solving large (two-dimensional) and (three-dimensional) Helmholtz problems. We show that our new parallel HSS solution methods are superior to the traditional dense LU factorization kernels in all the performance metrics.

Our main contributions are summarized as follows. We are the first to develop the scalable algorithms as well as the parallel code for the HSS algorithms used in the solution of dense linear systems. We performed detailed complexity analysis of our parallel HSS algorthms, taking into account communication latency and bandwidth. We demonstrated our code performance on an actual parallel computational platform with input matrices associated with the Schur complement systems from an important application area—the Helmholtz equations in seismic inversion. This parallelization effort has paved the way for a wide spectrum of research of employing HSS structure techniques in the solution methods for solving many large-scale PDE problems on extreme scale parallel machines.

The outline of the paper is as follows. In section 2, we present an overview of HSS structures. The fundamental parallelization strategy and the performance model are introduced in section 3, where we also briefly discuss our use of `BLACS` and `ScaLAPACK` to implement the high-performance kernels. In section 4, we present our parallel HSS construction framework. The parallel HSS factorization is described in section 5. In section 6, we discuss the parallel solution strategy. Some computational experiments are given in section 7.

2. Overview of HSS structures.

We briefly summarize the key concepts of HSS structures following the definitions and notation in [32, 29]. Let A be a general $n \times n$ real or complex matrix and $\mathcal{I} = \{1, 2, \ldots, n\}$ be the set of all row and column indices. Suppose $\mathcal{T}$ is a full binary tree with $2k-1$ nodes labeled as $i = 1, 2, \ldots, 2k-1$, such that the root node is $2k-1$ and the number of leaf nodes is k. Let $\mathcal{T}$ also be a *postordered* tree. That is, for each nonleaf node i of $\mathcal{T}$, its left child c_1 and right child c_2 satisfy $c_1 < c_2 < i$. Let $t_i \subset \mathcal{I}$ be an index subset associated with each node i of $\mathcal{T}$. We use $A|_{t_i \times t_j}$ to denote the submatrix of A with row index subset t_i and column index subset t_j.

HSS matrices are designed to take advantage of the low-rank property. In particular, when the off-diagonal blocks of a matrix (with hierarchical partitioning) have small (numerical) ranks, they are represented or approximated hierarchically by compact forms. These compact forms at different hierarchical levels are also related through nested basis forms. This can be seen from the definition of an *HSS form.*

DEFINITION 2.1. *We say that A is in a postordered HSS form with the corresponding HSS tree $\mathcal{T}$ if the following conditions are satisfied:*

- $t_{c_1} \cap t_{c_2} = \emptyset, t_{c_1} \cup t_{c_2} = t_i$ *for each nonleaf node i of $\mathcal{T}$ with children c_1 and c_2, and* $t_{2k-1} = \mathcal{I}$.
- *There exist matrices $D_i, U_i, R_i, B_i, W_i, V_i$ (called HSS generators) associated with each node i of $\mathcal{T}$ such that*

$$\begin{aligned} &D_{2k-1} = A, \\ &D_i = A|_{t_i \times t_i} = \begin{pmatrix} D_{c_1} & U_{c_1} B_{c_1} V_{c_2}^H \\ U_{c_2} B_{c_2} V_{c_1}^H & D_{c_2} \end{pmatrix}, \\ &U_i = \begin{pmatrix} U_{c_1} R_{c_1} \\ U_{c_2} R_{c_2} \end{pmatrix}, \quad V_i = \begin{pmatrix} V_{c_1} W_{c_1} \\ V_{c_2} W_{c_2} \end{pmatrix}, \end{aligned} \tag{2.1}$$

where the superscript H denotes the Hermitian transpose, and U_{2k-1}, V_{2k-1}, R_{2k-1}, B_{2k-1} are not needed.

The HSS generators define the HSS form of A. The use of a postordered HSS tree enables us to use a single subscript (corresponding to the label of a node of $\mathcal{T}$) for each HSS generator [29] instead of up to three subscripts as in [7]. Thus, we also call the HSS form a postordered one. Figure 2.1 illustrates a block 8×8 HSS representation A. As a special example, its leading block 4×4 part looks like

$$A|_{t_7 \times t_7} = \begin{pmatrix} \begin{pmatrix} D_1 & U_1 B_1 V_2^H \\ U_2 B_2 V_1^H & D_2 \end{pmatrix} & \begin{pmatrix} U_1 R_1 \\ U_2 R_2 \end{pmatrix} B_3 \begin{pmatrix} W_4^H V_4^H & W_5^H V_5^H \end{pmatrix} \\ \begin{pmatrix} U_4 R_4 \\ U_5 R_5 \end{pmatrix} B_6 \begin{pmatrix} W_1^H V_1^H & W_2^H V_2^H \end{pmatrix} & \begin{pmatrix} D_4 & U_4 B_4 V_5^H \\ U_5 B_5 V_4^H & D_5 \end{pmatrix} \end{pmatrix}.$$

For each diagonal block $D_i = A|_{t_i \times t_i}$ associated with each node i of $\mathcal{T}$, we define $A_i^- = A|_{t_i \times (\mathcal{I} \backslash t_i)}$ to be the *HSS block row* and $A_i^| = A|_{(\mathcal{I} \backslash t_i) \times t_i}$ to be the *HSS block column.* They are both called *HSS blocks.* The maximum rank r (or numerical rank r for a given tolerance) of all HSS blocks is called the *HSS rank* of A. If r is small as compared with the matrix size, we say that A has a *low-rank property.*

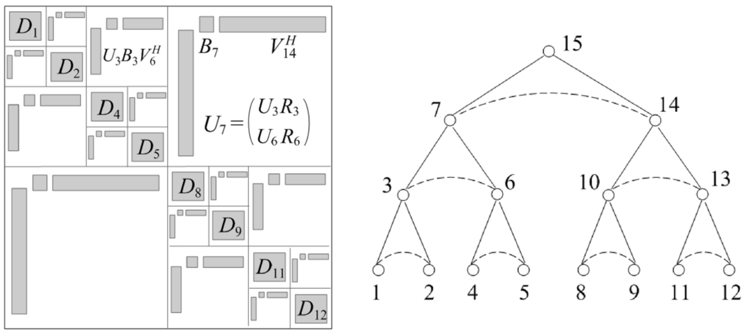


FIG. 2.1. *Pictorial illustrations of a block 8×8 HSS form and the corresponding HSS tree $\mathcal{T}$.*

Given a general dense matrix A with the low-rank property, we seek to construct an HSS representation in parallel or an HSS approximation when compression with a given tolerance is used. Our HSS factorization and solution will be conducted on the HSS forms.

3. Parallelization strategy.

For ease of exposition, we assume that all the diagonal blocks associated with the leaf nodes have the same block size m. We choose m first, which is related to the HSS rank, and then choose the HSS tree and the number of processes $P \approx n/m$. For simplicity, assume P is a power of two. Some existing serial HSS algorithms traverse the HSS tree in a postorder [29, 32]. For the HSS construction, the postordered traversal allows us to take advantage of previously compressed forms in later compression steps. However, the postordered HSS construction is serial in nature and involves global access of the matrix entries [29] and is not suitable for parallel computation.

To fully exploit parallelism, we reorganize the algorithms so that the HSS trees are traversed level by level. We use the HSS tree $\mathcal{T}$ as the primary tool to guide the matrix distribution and parallel matrix operations. Associated with each tree node i, various matrix operations are performed on the HSS generators of node i: $D_i, U_i, R_i, B_i, W_i, V_i$. We arrange the parallel framework in such a way that all the operations are performed in either an upward sweep or a downward sweep along the HSS tree. We refer to the leaf/bottom level of the tree as level 1, the next level up as level 2, and so on. We use the example in Figure 2.1 to illustrate the organization of the algorithms. The matrix is partitioned into eight block rows (Figure 3.1). We use eight processes $\{0, 1, 2, 3, 4, 5, 6, 7\}$ for the parallel operations. Each process individually works on one leaf node at level 1 of $\mathcal{T}$. At the second level, each group of two processes cooperates at a level-2 node. At the third level, each group of four processes cooperates at a level-3 node, and so on.

3.1. Using `ScaLAPACK` and `BLACS` for dense operations.

Most of the HSS algorithms consist of dense matrix kernels; although the matrix sizes are relatively small, the `ScaLAPACK` library [21] and the `BLACS` library [3] were used as much as possible. This way, we can fully benefit from the state-of-the-art high-performance dense linear algebra kernels and speed up the code development. The governing distribution scheme in `ScaLAPACK` is a 2D block cyclic matrix layout, in which the user specifies the block size of a submatrix and the shape of the 2D process grid. The blocks of matrices are then cyclically mapped to the process grid in both row and column dimensions. Furthermore, the processes can be divided into subgroups (called context

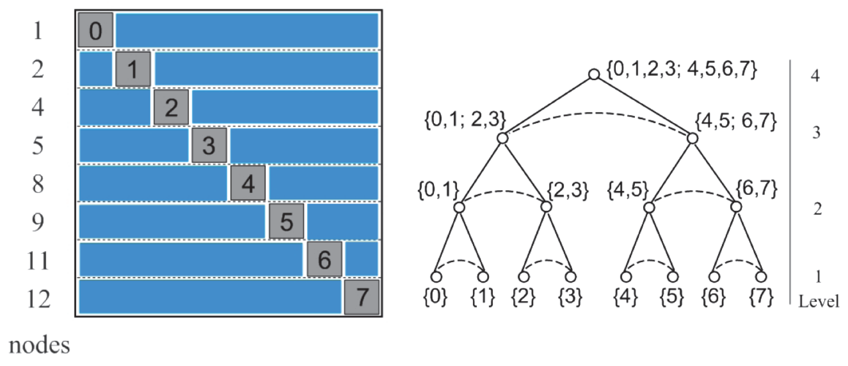


Fig. 3.1. *A block partition of a dense matrix A before the construction of the HSS form in Figure* 2.1*, where the labels inside the matrix and beside the nodes show the processes assigned to the nodes of the HSS tree in Figure* 2.1*.*

in BLACS terms) to work on independent parts of the calculations. Each subgroup is similar to the subcommunicator concept in MPI [20]. All our algorithms start with a global context created from the entire communicator, e.g., MPI_COMM_WORLD. When we move up the HSS tree, we define the other contexts encompassing the process subgroups.

For example, in Figures 2.1 and 3.1, the eight processes can be arranged as eight contexts for the leaf nodes in $\mathcal{T}$. At the second level, a group of two processes forms a context and is mapped to one node of $\mathcal{T}$. We use the notation $\{0, 1\} \leftrightarrow$ node 3 to indicate that the set of processes $\{0, 1\}$ forms a context and is mapped to node 3. Hence, four contexts are defined at the second level:

$$\{0, 1\} \leftrightarrow \text{node } 3, \{2, 3\} \leftrightarrow \text{node } 6, \{4, 5\} \leftrightarrow \text{node } 10, \text{ and } \{6, 7\} \leftrightarrow \text{node } 13.$$

Two contexts are defined at the third level:

$$\{0, 1;\ 2, 3\} \leftrightarrow \text{node } 7,\ \{4, 5;\ 6, 7\} \leftrightarrow \text{node } 14,$$

where the notation $\{0, 1;\ 2, 3\}$ means that processes 0 and 1 are stacked atop processes 2 and 3. Finally, one context is defined:

$$[0, 1, 4, 5;\ 2, 3, 6, 7] \leftrightarrow \text{node } 15.$$

We always arrange the process grid as square as possible, i.e., $P \approx \sqrt{P} \times \sqrt{P}$, and we can conveniently use $\sqrt{P}$ to refer to the number of processes in the row or column dimension. Figure 3.2 depicts a process tree associated with an HSS tree with 16 nodes. This illustrates how the 16 processes are arranged as subgroups while going up the tree.

When the algorithms move up the HSS tree, we need to perform *redistribution* to merge the submatrices in the two children's process contexts to a submatrix in the parent's context. Since the two children's contexts have the same size and shape and the parent context doubles each child's context, the parent context can always be arranged to combine the two children's contexts either side by side or one on top of the other, as shown in Figure 3.2. With this arrangement, the redistribution pattern involves only *pairwise exchanges*, that is, only the pair of processes at the same coordinate in the two children's contexts exchange data. For example, the redistribution from $\{0, 1;\ 2, 3\}$

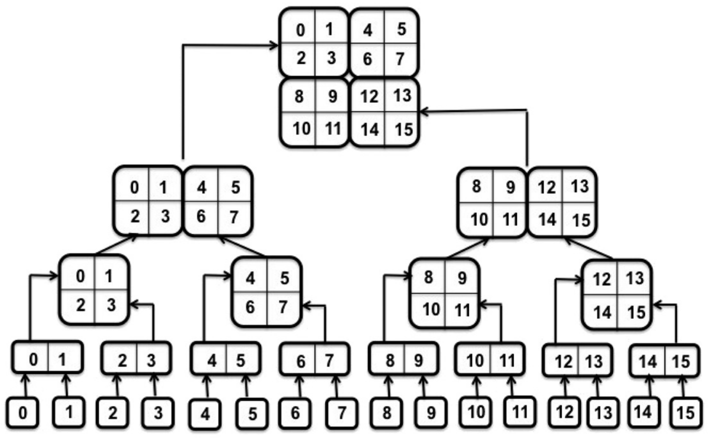


FIG. 3.2. *Illustration of the* 16 *processes which form various process subgroups/contexts along an HSS tree.*

and $\{4, 5;\ 6, 7\}$ to $\{0, 1, 4, 5;\ 2, 3, 6, 7\}$ is achieved by the following pairwise exchanges: $0 \leftrightarrow 4$, $1 \leftrightarrow 5$, $2 \leftrightarrow 6$, and $3 \leftrightarrow 7$. The redistribution from $\{0, 1, 4, 5;\ 2, 3, 6, 7\}$ and $\{8, 9, 12, 13;\ 10, 11, 14, 15\}$ to $\{0, 1, 4, 5;\ 2, 3, 6, 7;\ 8, 9, 12, 13;\ 10, 11, 14, 15\}$ is achieved by the following pairwise exchanges: $0 \leftrightarrow 8$, $1 \leftrightarrow 9$, $4 \leftrightarrow 12$, $5 \leftrightarrow 13$, $2 \leftrightarrow 10$, $3 \leftrightarrow 11$. $6 \leftrightarrow 14$, and $7 \leftrightarrow 15$.

3.2. Parallel runtime model. We will use the following notation in the analysis of the computational cost of our parallel algorithms:

- r is the HSS rank of A.
- The pair [`#messages`, `#words`] is used to count the number of messages and the number of words transferred. The parallel runtime can be modeled as the following (ignoring the overlap of communication with computation):

$$\texttt{Time} = \texttt{\#flops} \cdot \gamma + \texttt{\#messages} \cdot \alpha + \texttt{\#words} \cdot \beta,$$

  where γ, α, and β are the time taken for each flop, each message (latency), and each word transferred (reciprocal bandwidth), respectively. This model is realistic for our parallel algorithms, since our algorithms are mostly block synchronous, composed of sequences of communication phases followed by computation phases. There is very little overlap between communication and computation.
- The cost of broadcasting a message of W words among P processes is modeled as $[\log P,\ W \log P]$, assuming that a tree-based or hypercube-based broadcast algorithm is used [22]. The same cost is incurred for a reduction operation of W words.

The floating point operation counts were analyzed previously in [29, 32], which are $\mathcal{O}(rn^2)$ for HSS construction, $\mathcal{O}(r^2n)$ for ULV factorization, and $\mathcal{O}(rn)$ for solution, respectively. The flop counts between our levelwise HSS construction and the postordered one in [29, 32] are roughly the same. In fact, the flop count with the levelwise traversal is slightly higher by $\mathcal{O}(rn \log n)$, while the leading terms $\mathcal{O}(rn^2)$ in the counts are the same, or the asymptotic behavior is the same [32].

In the following sections, we will focus on analyzing the communication cost of our new parallel algorithms.

4. Parallel HSS construction.

In this section we discuss the construction of an HSS representation (or approximation) for A in parallel. The construction is composed of a row compression step (section 4.2) followed by a column compression step (section 4.3). The column compression step is applied to the intermediate matrix that is already row compressed. That is, the column compression is applied to a potentially much smaller matrix. The key kernel is a parallel compression algorithm which we discuss first.

4.1. Parallel block compression.

The key step in the HSS construction is to compress the HSS blocks of A. For example, consider an HSS block F. Truncated SVD is one option to realize such compression. That is, we drop those singular values below a prescribed threshold of all the singular values of F. SVD is generally very expensive. An efficient alternative is to use QR factorization with column pivoting and truncation, which is often sufficient. We now describe our parallel compression algorithm.

The input matrix is F of size $M \times N$ and is distributed in the process context $P \approx \sqrt{P} \times \sqrt{P}$. That is, the local dimension of F is $\frac{M}{\sqrt{P}} \times \frac{N}{\sqrt{P}}$. The second input is a prescribed tolerance τ to be used as a threshold to terminate the rank-revealing process when some $|T_{ii}|$ is relatively smaller than τ.

The following Algorithm 1, based on a modified Gram–Schmidt strategy which is revised from a QR factorization scheme in [12, 13], computes block compression in parallel: $F \approx \widetilde{Q}\widetilde{T}$, where $\widetilde{Q} = (q_1, q_2, \ldots, q_r)$ and $\widetilde{T}^H = (t_1, t_2, \ldots, t_r)$. Assuming that the HSS rank is r, each rank-revealing process takes no more than r steps, and it exits the loop when the stopping criterion is satisfied; see line 3 of Algorithm 1. Note that step 2 is used for clarity of explanation and is not done in the actual code. Step 3 is done quickly by the norm update strategy as in [12].

```
ALGORITHM 1. Parallel compression of $F$ with a relative tolerance $\tau$.
 subroutine $[Q, T, r] =$ compr($F, \tau$), such that $F \approx QT$
 for $i = 1 : \min(M, N)$
    1. In parallel, find the column $f_j = F(:, j)$ with the maximum norm
    2. Interchange $f_i$ and $f_j$
    3. Set $t_{ii} = \|f_i\|_2$, and if $t_{ii}/t_{11} \le \tau$, then EXIT with $r := i$
    4. Normalize $f_i$: $q_i = f_i/\|f_i\|_2$
    5. Broadcast $q_i$ rowwise within the context in which $F$ resides
    6. PBLAS2: $t_i^H = q_i^H(f_{i+1}, f_{i+2}, \ldots, f_N)$
    7. Compute rank-1 update: $(f_{i+1}, f_{i+2}, \ldots, f_N) = (f_{i+1}, f_{i+2}, \ldots, f_N) - q_i t_i^H$
 end
```

Communications occur in steps 1 and 5. The other steps only involve local computations. In step 1, the processes in each column group perform one reduction of size $\frac{N}{\sqrt{P}}$ to compute the column norms, with communication cost $[\log \sqrt{P},\ \frac{N}{\sqrt{P}} \log \sqrt{P}]$. This is followed by another reduction among the processes in the row group to find the maximum norm among all the columns, with communication cost $[\log \sqrt{P},\ \log \sqrt{P}]$. In step 5, the processes among each row group broadcast q_i of size $\frac{M}{\sqrt{P}}$, costing $[\log \sqrt{P},\ \frac{M}{\sqrt{P}} \log \sqrt{P}]$.

Summing the leading terms for (at most) r steps, we obtain the following communication cost:

$$\texttt{Comm}_{compr} = \left[\log\sqrt{P},\ \frac{M+N}{\sqrt{P}}\log\sqrt{P}\right]\cdot r\ . \tag{4.1}$$

To achieve higher performance, a block strategy can be adopted similarly, like the serial `LAPACK` subroutine `xGEQP3` [19]. The parallel blocked algorithm remains our future work.

4.2. Parallel row compression stage. Algorithm 2 gives an overview of the main steps of parallel row compression stage [29]:

- At the leaf level, compute the compression of the HSS block rows with a QR factorization with column pivoting. Then ignore the resulting orthogonal basis (which are the U generators).
- At an upper level, merge the remaining factors from the child compression, and form a block to compress. The compression yields the R generators, which are ignored in upper level compression.

(The remaining blocks participate in the column compression later, which has a similar framework.)

We use the block 8×8 matrix in Figures 2.1 and 3.1 to illustrate the algorithm step by step.

```
ALGORITHM 2. Parallel row compression of $A$ with $P$ processes.
  Let $L = \log P$ be the number of leaves of the HSS tree $\mathcal{T}$ and $L+1$ be the
  number of levels of $\mathcal{T}$.
  1. At level-1 leaf nodes,
    (1.1) In parallel, each process $i$ performs local compression on node $i$:
          $F_i \approx U_i\widehat{F}_i$, where, $F_i \equiv A_i^- = A|_{t_i\times(\mathcal{I}\backslash t_i)}$.
    (1.2) Redistribution to prepare for level-2 compression:
      For $i = 0\ldots L/2$, pairs of processes $\{2i\} \leftrightarrow \{2i+1\}$ do pairwise exchange,
      $2r\times n$ matrix $[\widehat{F}_{2i};\widehat{F}_{2i+1}]$ is distributed in new process subgroup $\{2i, 2i+1\}$.
  2. For levels $l = 2\ldots L+1$ in $\mathcal{T}$,
    (2.1) Each 2D process subgroup $2^{\lceil \frac{l}{2}\rceil -1}\times 2^{\lfloor \frac{l}{2}\rfloor}$ performs parallel compression
          on node $i$, with $c_1$ and $c_2$ being two children of $i$:
          $F_i \equiv \begin{pmatrix}\widehat{F}_{c_1}\\ \widehat{F}_{c_2}\end{pmatrix} \approx \begin{pmatrix}R_{c_1}\\ R_{c_2}\end{pmatrix}\widehat{F}_i$
    (2.2) (If $l \neq L+1$) Redistribution to prepare for level $l+1$ compression:
      For $i = 0\ldots L/2$, pairs of processes $\{2i\} \leftrightarrow \{2i+2^l\}$ do pairwise exchange,
      $2r\times n$ matrix $[\widehat{F}_{2i};\widehat{F}_{2i+1}]$ is distributed in new process subgroup
      $2^{\lceil \frac{l+1}{2}\rceil -1}\times 2^{\lfloor \frac{l+1}{2}\rfloor}$.
    Endfor
```

4.2.1. Row compression—level 1. In the first step, all the leaves 1, 2, 4, 5, 8, 9, 11, and 12 of $\mathcal{T}$ have their own processes $\{0\},\{1\},\{2\},\{3\},\{4\},\{5\},\{6\}$, and $\{7\}$, respectively. Each process owns a block row of the global matrix A, given by $D_i = A|_{t_i\times t_i}$ and $A_i^- = A|_{t_i\times(\mathcal{I}\backslash t_i)}$, as illustrated in Figure 3.1. The off-diagonal block to be compressed is $F_i \equiv A_i^-$. Each process performs a sequential operation of compression on F_i:

$$F_i \approx U_i\widehat{F}_i.$$

For notational convenience, we write $\widehat{F}_i \equiv A|_{\hat{t}_i \times (\mathcal{I}\backslash t_i)}$, which can be understood as $\widehat{F}_i$ is stored in the space of F_i or in A with row index set $\hat{t}_i$. That is, we can write the above factorization as[1]

$$A|_{t_i \times (\mathcal{I}\backslash t_i)} \approx U_i A|_{\hat{t}_i \times (\mathcal{I}\backslash t_i)}. \tag{4.2}$$

No communication is involved in this step. One of the HSS generators U_i is obtained here.

We now prepare for the compression at the upper level, level 2 of $\mathcal{T}$. The upper level compression must be carried out among a pair of processes in each context. For this purpose, we need a redistribution phase prior to the compression. That is, we perform pairwise exchange of data: $\{0\} \leftrightarrow \{1\}$, $\{2\} \leftrightarrow \{3\}$, $\{4\} \leftrightarrow \{5\}$, and $\{6\} \leftrightarrow \{7\}$. The level-2 nodes on $\mathcal{T}$ are 3, 6, 10, and 13, whose contexts are $\{0,1\}$, $\{2,3\}$, $\{4,5\}$, and $\{6,7\}$, respectively. For each node i at level 2 with children c_1 and c_2, we have

$$A|_{t_{c_1} \times t_{c_2}} \approx U_{c_1} A|_{\hat{t}_{c_1} \times t_{c_2}},\ A|_{t_{c_2} \times t_{c_1}} \approx U_{c_2} A|_{\hat{t}_{c_2} \times t_{c_1}},\ A_i^- \approx \begin{pmatrix} U_{c_1} A|_{\hat{t}_{c_1} \times (\mathcal{I}\backslash t_i)} \\ U_{c_2} A|_{\hat{t}_{c_2} \times (\mathcal{I}\backslash t_i)} \end{pmatrix}.$$

Ignoring the basis matrices U_{c_1} and U_{c_2}, the block to be compressed in the next step is

$$F_i \equiv \begin{pmatrix} A|_{\hat{t}_{c_1} \times (\mathcal{I}\backslash t_i)} \\ A|_{\hat{t}_{c_2} \times (\mathcal{I}\backslash t_i)} \end{pmatrix}. \tag{4.3}$$

This procedure is illustrated in Figure 4.1(a). Two communication steps are used during redistribution. First, exchange $A|_{\hat{t}_{c_1} \times t_{c_2}}$ and $A|_{\hat{t}_{c_2} \times t_{c_1}}$ between c_1's and c_2's contexts. This prepares for the column compression in the future (section 4.3). Next, redistribute the newly merged off-diagonal block F_i, $i = 3, 6, 10, 13$, into the process grid contexts $\{0,1\}$, $\{2,3\}$, $\{4,5\}$, and $\{6,7\}$ corresponding to nodes 3, 6, 10, and 13, respectively. Here we use a `ScaLAPACK` subroutine `PxGEMR2D` to realize the data exchange and redistribution steps.

During the redistribution phase, the number of messages is 2, and the number of words exchanged is $\frac{rn}{2} \cdot 2$. The communication cost is $[2,\ \frac{rn}{2} \cdot 2]$. We recall that n is the number of columns of F_i and each compressed block row is assumed to have the same rank r.

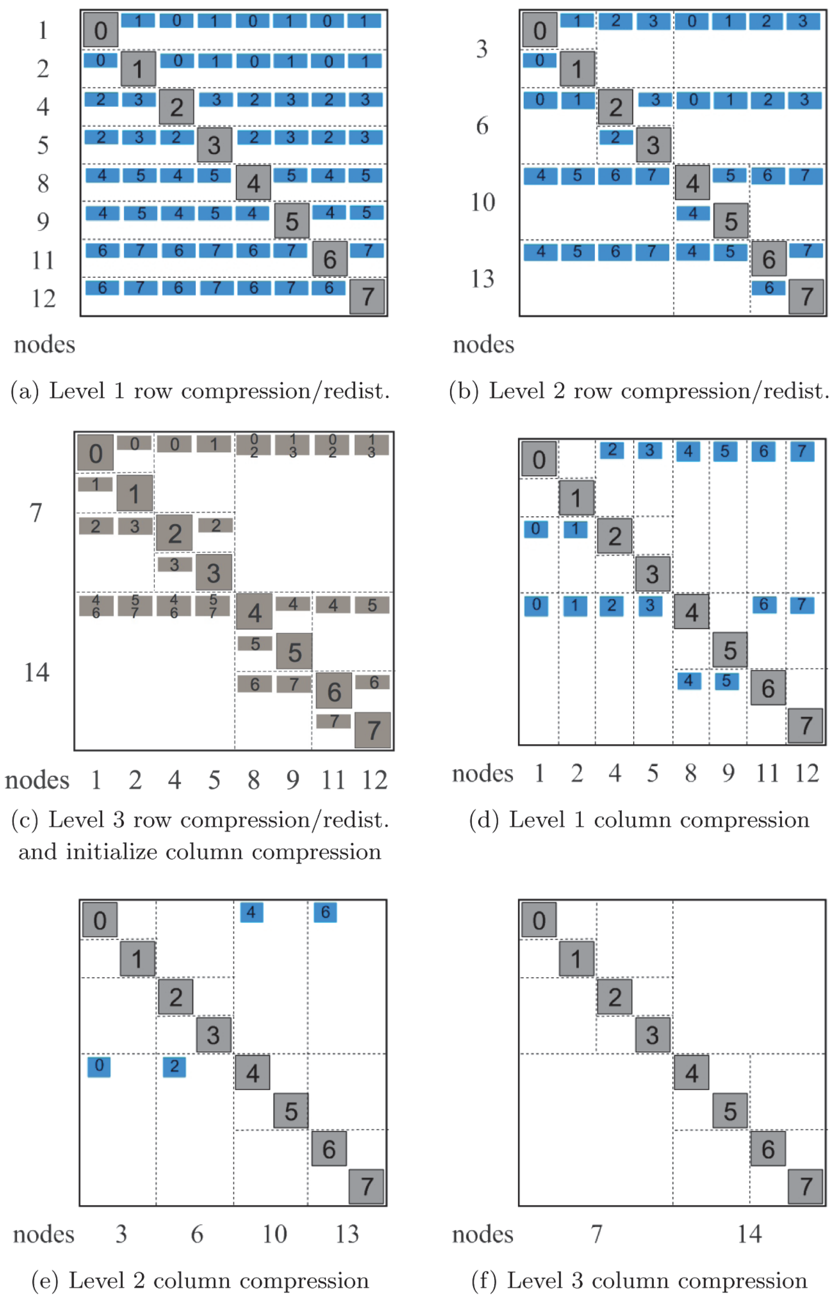
4.2.2. Row compression—level 2. At level 2 of $\mathcal{T}$, within the context for each node $i = 3, 6, 10, 13$, we perform parallel compression for F_i in (4.3):

$$F_i \approx \begin{pmatrix} R_{c_1} \\ R_{c_2} \end{pmatrix} A|_{\hat{t}_i \times (\mathcal{I}\backslash t_i)}, \tag{4.4}$$

where $A|_{\hat{t}_i \times (\mathcal{I}\backslash t_i)}$ is defined similar to the one in (4.2). The R generators associated with the child level are then obtained. Since the size of each F_i is bounded by $2r \times n$ and two processes are used for the compression, we obtain the communication cost $[\log\sqrt{2},\ \frac{2r+n}{\sqrt{2}} \log\sqrt{2}] \cdot r$ using (4.1).

To prepare for the compression at level 3 of $\mathcal{T}$, again, we need a redistribution phase, performing the following pairwise data exchange: $\{0,1\} \leftrightarrow \{2,3\}$ and $\{4,5\} \leftrightarrow$

[1]This is only a way to simplify notation in the presentation and is not the actual storage used in our implementation.



(a) Level 1 row compression/redist.

(b) Level 2 row compression/redist.

(c) Level 3 row compression/redist. and initialize column compression

(d) Level 1 column compression

(e) Level 2 column compression

(f) Level 3 column compression

FIG. 4.1. *Illustration of data distribution in the row and column compressions, where the labels inside the matrix mean processes, and those outside mean the nodes of $\mathcal{T}$.*

$\{6, 7\}$. In this notation, the exchanges $0 \leftrightarrow 2$ and $1 \leftrightarrow 3$ occur simultaneously. There is no need for data exchanges between processes 0 and 3, nor 1 and 2. The level-3 nodes are 7 and 14 with the process contexts $\{0, 1;\ 2, 3\}$ and $\{4, 5;\ 6, 7\}$, respectively. There are also communications similar to the procedure for forming (4.3) in the previous step, so that each of the two off-diagonal blocks F_i, $i = 7, 14$, is formed and distributed onto the respective process context. This is illustrated in Figure 4.1(b).

During the redistribution phase, the number of messages is 2, and the number of words exchanged is $\frac{rn}{2^2} \cdot 2$. Thus the communication cost is $[2,\ \frac{rn}{2^2} \cdot 2]$.

4.2.3. Row compression—level 3. At level 3, we perform the parallel compression within each context for each $F_i,\ i = 7, 14$, similar to (4.4). The R generators associated with the child level are also obtained here. The communication cost of the compression is given by $[\log \sqrt{4},\ \frac{2r+n}{\sqrt{4}} \log \sqrt{4}] \cdot r$.

Since the upper level node has only one node, the root node 15 of $\mathcal{T}$, there is no off-diagonal block associated with it. Thus, to prepare for the column HSS constructions, only one pairwise exchange step is needed between the two contexts: $\{0, 1;\ 2, 3\} \leftrightarrow \{4, 5;\ 6, 7\}$, meaning $0 \leftrightarrow 4, 1 \leftrightarrow 5, 2 \leftrightarrow 6$, and $3 \leftrightarrow 7$. This is similar to (4.3) except that there is no merging step to form F_{15}. The procedure is illustrated in Figure 4.1(c). The communication cost in the redistribution phase is $[2,\ \frac{rn}{2^3} \cdot 2]$.

4.2.4. Communication costs in row compression. In general, the compression and communication for an HSS matrix with more blocks can be similarly shown, following Algorithm 2. Here, we sum the messages and number of words communicated at all the levels of the tree in this row compression stage. For simplicity, assume there are P leaves and about $L \approx \log P$ levels in $\mathcal{T}$. Then the total communication cost is summed up by the following:[2]

(1) Redistributions:

$$\texttt{\#messages} = \sum_{i=1}^{L} 2 \approx 2 \log P,$$

$$\texttt{\#words} = \sum_{i=1}^{L} \left(\frac{rn}{2^i} 2 \right) = \mathcal{O}(2rn).$$

(2) RRQR compression (Algorithm 1):

$$\texttt{\#messages} = \sum_{i=1}^{L} (\log \sqrt{2^i}) \cdot r = r \log 2 \sum_{i=1}^{L} \frac{i}{2} = \mathcal{O}(r \log^2 P),$$

$$\begin{aligned}\texttt{\#words} &= \sum_{i=1}^{L} \left(\frac{2r+n}{\sqrt{2^i}} \log \sqrt{2^i} \right) \cdot r \\ &= r \left(\frac{2r+n}{2} \right) \log 2 \sum_{i=1}^{L} \frac{i}{2^{i/2}} = \mathcal{O}(rn \log P).\end{aligned}$$

The total flop count of all the row compression is

$$\sum_{i=1}^{L} \mathcal{O} \left(r \frac{n}{2^{L-i}} \left(n - \frac{n}{2^i} \right) \right) = \mathcal{O}(rn^2),$$

where we assume the bottom level diagonal blocks have sizes $\mathcal{O}(r)$, as commonly done [32].

[2] In most situations in which we are interested, we can assume $n \gg r$.

4.3. Parallel column compression stage. After the row compression, the blocks $A|_{\hat{t}_j \times (\mathcal{I} \backslash t_j)}$ that remain to be compressed in the column compression stage are much smaller. In addition, in this stage, pieces of the blocks $A|_{\hat{t}_j \times (\mathcal{I} \backslash t_j)}$ for nodes j at different levels may be compressed together to get a V generator. For example, for node 11 in Figure 4.1(c), to form the block to be compressed, we need to stack the off-diagonal pieces (marked with process 6) resulting from row compression at three different levels:

$$G_{11}^H = \begin{pmatrix} A|_{\hat{t}_{12} \times t_{11}} \\ A|_{\hat{t}_{10} \times t_{11}} \\ A|_{\hat{t}_7 \times t_{11}} \end{pmatrix}.$$

Clearly, $A|_{\hat{t}_7 \times t_{11}}$ and $A|_{\hat{t}_{10} \times t_{11}}$ are pieces associated with nodes 7 and 10, respectively, which are previously visited.

To systematically keep track of the nodes which are previously visited and need to be considered in a column compress step, we use the following definition, which generalizes the concept of visited sets in [30] for symmetric positive definite matrices to nonsymmetric ones. (Please see [30] for a related illustration.)

Definition 4.1. *The* left visited set *associated with a node i of a postordered full binary tree $\mathcal{T}$ is*

$$\mathcal{V}_i = \{j \mid j \textit{ is a left node and } \mathrm{sib}(j) \in \mathrm{ances}(i)\},$$

where $\mathrm{sib}(j)$ *is the sibling of j in $\mathcal{T}$ and* $\mathrm{ances}(i)$ *is the set of ancestors of node i including i. Similarly, the* right visited set *associated with i is*

$$\mathcal{W}_i = \{j \mid j \textit{ is a right node and } \mathrm{sib}(j) \in \mathrm{ances}(i)\}.$$

$\mathcal{V}_i$ and $\mathcal{W}_i$ are essentially the stacks before the visit of i in the postordered and reverse-postordered traversals of $\mathcal{T}$, respectively.

We now describe how the column compression works. We use the same 8×8 block matrix example after the row compression for illustration.

4.3.1. Column compression—level 1. After the row compression, the updated off-diagonal blocks $\widehat{F}_j \equiv A|_{\hat{t}_j \times (\mathcal{I} \backslash t_j)}$, $j = 1, 2, \ldots, 14$, are stored in the individual contexts, at different levels of the HSS tree. For example, $\widehat{F}_1$ is stored in the context $\{0\}$, $\widehat{F}_3$ is stored in the context $\{0, 1\}$, and $\widehat{F}_7$ is stored in the context $\{0, 1;\ 2, 3\}$. Similar to the row compression phase, the column compression proceeds upward along $\mathcal{T}$ level by level. The difference is that, here, associated with each tree node is a block column of the matrix, and it was already compressed during the row compression stage. To prepare for the compression associated with the leaf nodes, we first need a redistribution phase to transfer the $A|_{\hat{t}_j \times \mathrm{sib}(j)}$ blocks for nodes j at the higher levels to the bottom leaf level. This is achieved in $\log P$ steps of communication in a top-down fashion. In each step, we redistribute $A|_{\hat{t}_j \times \mathrm{sib}(j)}$ in the context of j to the contexts of the leaf nodes. These j indices of the row blocks that need to be redistributed downward are precisely those in the visited sets in Definition 4.1. For instance, the block column associated with leaf node 2 needs the pieces corresponding to the nodes $\mathcal{V}_2 \cup \mathcal{W}_2 = \{1\} \cup \{6, 14\}$. For all the leaf nodes, the redistribution procedure achieves the following blocks which need to be compressed in this stage:

(4.5)

$$G_1^H = \begin{pmatrix} A|_{\hat{t}_2\times t_1} \\ A|_{\hat{t}_6\times t_1} \\ A|_{\hat{t}_{14}\times t_1} \end{pmatrix},\ G_2^H = \begin{pmatrix} A|_{\hat{t}_1\times t_2} \\ A|_{\hat{t}_6\times t_2} \\ A|_{\hat{t}_{14}\times t_2} \end{pmatrix},\ G_4^H = \begin{pmatrix} A|_{\hat{t}_3\times t_4} \\ A|_{\hat{t}_5\times t_4} \\ A|_{\hat{t}_{14}\times t_4} \end{pmatrix},\ G_5^H = \begin{pmatrix} A|_{\hat{t}_4\times t_5} \\ A|_{\hat{t}_3\times t_5} \\ A|_{\hat{t}_{14}\times t_5} \end{pmatrix},$$

$$G_8^H = \begin{pmatrix} A|_{\hat{t}_9\times t_8} \\ A|_{\hat{t}_{13}\times t_8} \\ A|_{\hat{t}_7\times t_8} \end{pmatrix},\ G_9^H = \begin{pmatrix} A|_{\hat{t}_8\times t_9} \\ A|_{\hat{t}_{13}\times t_9} \\ A|_{\hat{t}_7\times t_9} \end{pmatrix},\ G_{11}^H = \begin{pmatrix} A|_{\hat{t}_{12}\times t_{11}} \\ A|_{\hat{t}_{10}\times t_{11}} \\ A|_{\hat{t}_7\times t_{11}} \end{pmatrix},\ G_{12}^H = \begin{pmatrix} A|_{\hat{t}_{11}\times t_{12}} \\ A|_{\hat{t}_{10}\times t_{12}} \\ A|_{\hat{t}_7\times t_{12}} \end{pmatrix}.$$

We can use $\mathcal{V}_i$ and $\mathcal{W}_i$ to simplify the notation. For example, we write

$$\bar{t}_1 = \hat{t}_2 \cup \hat{t}_6 \cup \hat{t}_{14},\ G_1^H = A|_{\bar{t}_1\times t_1}.$$

We still use the `ScaLAPACK` subroutine `PxGEMR2D` to perform these intercontext communications. In this redistribution phase, the number of messages sent is $\log P$, and the number of words is $\frac{r\ n}{\sqrt{P}} \log P$. Thus the communication cost is $[\log P,\ \frac{rn}{\sqrt{P}} \log P]$.

After the redistribution, the layout of the off-diagonal blocks is illustrated by Figure 4.1(c), which initiates the parallel column construction. At the bottom level, the contexts $\{0\}$, $\{1\}$, $\{2\}$, $\{3\}$, $\{4\}$, $\{5\}$, $\{6\}$, and $\{7\}$ are associated with the leaf nodes 1, 2, 4, 5, 8, 9, 11, and 12, respectively. G_i^H for all leaves i are indicated by the shaded areas in Figure 4.1(c). We carry out parallel compression on G_i:

$$G_i \approx V_i \widetilde{G}_i.$$

This can be denoted as

$$G_i \approx A|_{\bar{t}_i\times\tilde{t}_i} V_i^H,$$

where $\tilde{t}_i$ is a subset of t_i and we can understand that $\widetilde{G}_i = \left(A|_{\bar{t}_i\times\tilde{t}_i}\right)^H$ can be stored in the space of G_i. (This is solely for notational convenience. See the remark for (4.2).) We note that this step is done locally within each process. The V generators are obtained here. See Figure 4.1(c)–(d).

To prepare for the upper level column compression, communications occur pairwise: $\{0\} \leftrightarrow \{1\}$, $\{2\} \leftrightarrow \{3\}$, $\{4\} \leftrightarrow \{5\}$, $\{6\} \leftrightarrow \{7\}$. The upper level blocks $G_i,\ i = 3, 6, 10, 13$, for compression are formed by ignoring the V basis matrices and merging parts of $A|_{\bar{t}_{c_1}\times\tilde{t}_{c_1}}$ and $A|_{\bar{t}_{c_2}\times\tilde{t}_{c_2}}$. That is, we set

$$G_i = \begin{pmatrix} A|_{\bar{t}_i\times\tilde{t}_{c_1}}, & A|_{\bar{t}_i\times\tilde{t}_{c_2}} \end{pmatrix},\ B_{c_1} = A|_{\hat{t}_{c_1}\times\tilde{t}_{c_2}},\ B_{c_2} = A|_{\hat{t}_{c_2}\times\tilde{t}_{c_1}}. \tag{4.6}$$

This procedure is illustrated in Figure 4.1(d). Two communication steps are needed. In the first step B_{c_1} and B_{c_2} are generated by exchanging $A|_{\hat{t}_{c_2}\times\tilde{t}_{c_1}}$ and $A|_{\hat{t}_{c_1}\times\tilde{t}_{c_2}}$ pairwise between c_1's and c_2's contexts. We note that some B generators are obtained here. The second step is to redistribute the newly merged off-diagonal block G_i onto the process grid associated with the contexts for nodes $i = 3, 6, 10, 13$.

We note that during the column compression stage, the number of nodes in $\mathcal{V}_i \cup \mathcal{W}_i$ needed to form G_i is the same as the number of levels in the HSS tree, which is $\log(\frac{n}{m}) \approx \log P$. See, e.g., (4.6). Therefore, the row dimension of G_i is bounded by $r \log(\frac{n}{m})$, which is much smaller than the column dimension n during the row compression stage. Similar to the level-1 row compression, during this redistribution phase, the number of messages is 2 and the number of words exchanged is $\frac{r^2 \log P}{2} \cdot 2$. The communication cost is then $[2,\ \frac{r^2 \log P}{2} \cdot 2]$.

4.3.2. Column compression—level 2. At level 2, the contexts $\{0,1\}$, $\{2,3\}$, $\{4,5\}$, and $\{6,7\}$ are associated with the nodes 3, 6, 10, and 13, respectively. Each off-diagonal block $G_i,\ i=3,6,10,13$, has already been distributed onto the respective process context, as illustrated in Figure 4.1(d). That is, G_i is formed by merging appropriate blocks associated with the children c_1 and c_2:

$$G_i = \begin{pmatrix} (A|_{\bar{t}_i\times\tilde{t}_{c_1}})^H \\ (A|_{\bar{t}_i\times\tilde{t}_{c_2}})^H \end{pmatrix}.$$

Then we perform parallel compression on each G_i:

$$G_i = \begin{pmatrix} W_{c_1} \\ W_{c_2} \end{pmatrix} \widetilde{G}_i,\ \widetilde{G}_i \equiv \left(A|_{\bar{t}_i\times\tilde{t}_i}\right)^H. \tag{4.7}$$

See Figure 4.1(d)–(e). Some W generators are obtained. Since each G_i is bounded by the size $r\log P\times 2r$ and two processes are used for its compression, using (4.1), we obtain the communication cost $[\log\sqrt{2},\ \frac{2r+r\log P}{\sqrt{2}}\log\sqrt{2}]\cdot r$.

To enable the upper level column HSS construction, communication occurs pairwise: $\{0,1\}\leftrightarrow\{2,3\}$ and $\{4,5\}\leftrightarrow\{6,7\}$. The procedure is illustrated by Figure 4.1(e). Similar to (4.6), two communication steps are needed. During the distribution phase, the number of messages is 2, and the number of words exchanged is $\frac{r^2\log P}{4}\cdot 2$. The communication cost is $[2,\ \frac{r^2\log P}{4}\cdot 2]$.

4.3.3. Column compression—level 3. At level 3, the two contexts $\{0,1;\ 2,3\}$ and $\{4,5;\ 6,7\}$ are associated with the nodes 7 and 14, respectively. Each off-diagonal block $F_i,\ i=7,14$, has already been distributed onto the respective process contexts, as shown in Figure 4.1(e). Then we perform the compression similarly to (4.7). See Figure 4.1(e)–(f). The communication cost of the compression is given by $[\log\sqrt{4},\ \frac{2r+r\log P}{\sqrt{4}}\log\sqrt{4}]\cdot r$.

Since the level-4 node is the root node 15 of $\mathcal{T}$, there is no off-diagonal block F_{15} associated with it. Thus, the entire parallel HSS construction is finalized at this step. There is only one stage of communications occurring: $\{0,1;\ 2,3\}\leftrightarrow\{4,5;\ 6,7\}$, which is similar to (4.6) except there is no merging step. Figure 4.1(f) indicates that after this final communication, all the HSS generators are obtained. The communication cost is $[2,\ \frac{r^2\log P}{8}\cdot 2]$.

4.3.4. Communication costs in column compression. We now sum all the messages and words communicated at all the levels of the tree during the column compression and obtain the total communication costs as follows, where $L\approx\log P$:

(1) Redistributions:

$$\texttt{\#messages} = \log P + \sum_{i=1}^{L} 2 \approx 3\log P,$$

$$\texttt{\#words} = \frac{rn}{\sqrt{P}}\log P + \sum_{i=1}^{L}\left(\frac{r^2\log P}{2^i}2\right) = \mathcal{O}(rn).$$

(2) RRQR compression (Algorithm 1):

$$\texttt{\#messages} = \sum_{i=1}^{L} \log \sqrt{2^i} \cdot r = r \log 2 \sum_{i=1}^{L} \frac{i}{2} = \mathcal{O}(r \log^2 P),$$

$$\begin{aligned}\texttt{\#words} &= \sum_{i=1}^{L} \left(\frac{2r + r\log P}{\sqrt{2^i}} \log \sqrt{2^i} \right) \cdot r \\ &= r \cdot \frac{2r + r \log P}{2} \log 2 \sum_{i=1}^{L} \frac{i}{2^{i/2}} = \mathcal{O}(r^2 \log^2 P).\end{aligned}$$

The flop count of all the column compression is less straightforward. Just like in [32], with the aid of visited sets (see Definition 4.1), it can be shown that the cost is $\mathcal{O}(rn \log n)$.

4.4. Total communication cost in parallel HSS construction. After the two stages of compression, all the HSS generators $D_i, U_i, R_i, B_i, W_i, V_i$ are obtained. The following formulas summarize the total communication costs for the entire parallel HSS construction, including both the row construction and the column construction:

(1) Redistributions:

$$\texttt{\#messages} = \mathcal{O}(\log P), \tag{4.8}$$

$$\texttt{\#words} = \mathcal{O}(rn). \tag{4.9}$$

(2) RRQR compression:

$$\texttt{\#messages} = \mathcal{O}(r \log^2 P), \tag{4.10}$$

$$\texttt{\#words} = \mathcal{O}(rn \log P + r^2 \log^2 P). \tag{4.11}$$

Comparing (4.8)–(4.11), we see that the compression dominates the communication costs both in message count and in message volume.

Remarks. Putting this in perspective, we compare the communication complexity to the flop count. It was analyzed in [29] that the total #flops of the sequential HSS construction algorithm is $\mathcal{O}(rn^2)$. Then, assuming a perfect load balance, the flop count per process is $\mathcal{O}(\frac{rn^2}{P})$. Taking (4.11) to be the dominant communication part for large problems, the *flop-to-byte ratio* is roughly $\frac{n}{P \log P}$, which is very small. This indicates that our parallel algorithm is very much communication bound, and its parallel performance is more sensitive to the network speed than the CPU speed.

When the HSS-based method is used to solve a linear system, the construction cost dominates those of the ULV factorization (section 5) and solution (section 6). Therefore, we can now compare this complexity to that of Gaussian elimination commonly used for solving a dense linear system. We recall that the parallel LU factorization implemented in `ScaLAPACK` needs $\mathcal{O}(n^3)$ flops and $[\mathcal{O}(n \log P), \mathcal{O}(\frac{n^2 \log P}{\sqrt{P}})]$ communication cost [2]. They are both higher than the HSS counterparts. For the `ScaLAPACK` LU algorithm, the *flop-to-byte ratio* is $\mathcal{O}(\frac{n}{\sqrt{P} \log P})$. Therefore, the new HSS algorithm has a lower flop-to-byte ratio than the classical dense LU algorithm, indicating less potential for data reuse and more communication bound.

If a (relative) tolerance τ is used in the HSS construction, the Frobenius-norm relative approximation error is $\mathcal{O}(\tau \sqrt{r} \log n)$ [33]. In practice, the errors are often much smaller.

From our experiences with seismic applications using the Helmholtz equations [27, 26], the HSS ranks r are usually on the order of 10 s for 2D problems and 100 s $\sim$ 1000 s for 3D problems. The typical values of n/r are about 100 s for 2D problems and 10 s for 3D problems (see Figure 7.1 in section 7). This corroborates the prior analysis that for 2D problems, r is about $\mathcal{O}(\log n)$ [10], and for 3D, it is about $\mathcal{O}(\sqrt{n})$ [31]. Thus, n/r is larger in 2D than in 3D. Therefore, from the #flops viewpoint, HSS has greater advantage over LU in 2D than in 3D.

5. Parallel ULV HSS factorization. After the HSS approximation to A is constructed, we are ready to factorize it via the generators. In these two sections, we discuss a type of factorization and solution strategies, called ULV HSS factorization and solution [7, 29]. That is, the factorization on an HSS matrix generates a sequence of orthogonal (U, V) and triangular (L) local matrices. These local matrices are also obtained during the traversal of the HSS tree and are also associated with the nodes. The ULV HSS factorization generally involves these steps:

- Introduce zeros into off-diagonal block rows using the U generators.
- Partially factorize and eliminate the diagonal blocks.
- Merge the remaining blocks and repeat.

(A framework for the ULV solution can be similarly outlined.)

Here, we present our parallel strategy in terms of a block 2×2 HSS form which is a submatrix of the HSS matrix and corresponds to two leaves in the HSS tree (Figure 5.1(a)):

$$\begin{pmatrix} D_{c_1} & U_{c_1} B_{c_1} V_{c_2}^H \\ U_{c_2} B_{c_2} V_{c_1}^H & D_{c_2} \end{pmatrix}, \tag{5.1}$$

where c_1 and c_2 are children of a node i and are leaves of the HSS tree $\mathcal{T}$, and the generators associated with c_1 and c_2 are distributed on the process grids corresponding to the contexts of c_1 and c_2, respectively. The context of i is the union of the contexts of c_1 and c_2. We assume that the sizes of U_{c_1} and U_{c_2} are $m \times r$.

We start with the QL factorization of U_{c_1} and U_{c_2}:

$$U_{c_1} = Q_{c_1} \begin{pmatrix} 0 \\ \widetilde{U}_{c_1} \end{pmatrix}, \quad U_{c_2} = Q_{c_2} \begin{pmatrix} 0 \\ \widetilde{U}_{c_2} \end{pmatrix}, \tag{5.2}$$

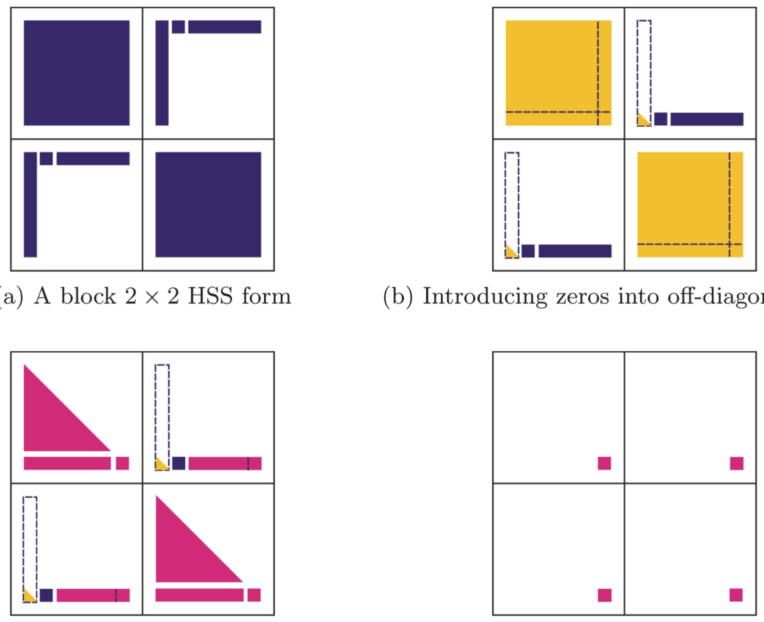
where $\widetilde{U}_{c_1}$ and $\widetilde{U}_{c_2}$ are lower triangular matrices of size $r \times r$, respectively. (In fact, since U_{c_1} and U_{c_2} have orthonormal columns in our HSS construction, we can directly derive orthogonal matrices so that $\widetilde{U}_{c_1}$ and $\widetilde{U}_{c_2}$ become identity matrices.) We note that there is no intercontext communication at this stage. We multiply $Q_{c_1}^H$ and $Q_{c_2}^H$ to the block rows independently within each context and obtain

$$\begin{pmatrix} Q_{c_1}^H & 0 \\ 0 & Q_{c_2}^H \end{pmatrix} \begin{pmatrix} D_{c_1} & U_{c_1} B_{c_1} V_{c_2}^H \\ U_{c_2} B_{c_2} V_{c_1}^H & D_{c_2} \end{pmatrix} = \begin{pmatrix} \widehat{D}_{c_1} & \begin{pmatrix} 0 \\ \widetilde{U}_{c_1} \end{pmatrix} B_{c_1} V_{c_2}^H \\ \begin{pmatrix} 0 \\ \widetilde{U}_{c_2} \end{pmatrix} B_{c_2} V_{c_1}^H & \widehat{D}_{c_2} \end{pmatrix},$$

where

$$\widehat{D}_{c_1} = Q_{c_1}^H D_{c_1}, \quad \widehat{D}_{c_2} = Q_{c_2}^H D_{c_2}.$$

This is illustrated by Figure 5.1(b).



(a) A block 2×2 HSS form (b) Introducing zeros into off-diagonal blocks

(c) Introducing zeros into diagonal blocks (d) After partial diagonal elimination

FIG. 5.1. (a) *The ULV factorization of a block* 2×2 *HSS form and the illustration of the intercontext communication to form* (5.4).

Next, we partition the diagonal blocks conformably as

$$\widehat{D}_{c_1} = \begin{pmatrix} \widehat{D}_{c_1;1,1} & \widehat{D}_{c_1;1,2} \\ \widehat{D}_{c_1;2,1} & \widehat{D}_{c_1;2,2} \end{pmatrix}, \quad \widehat{D}_{c_2} = \begin{pmatrix} \widehat{D}_{c_2;1,1} & \widehat{D}_{c_2;1,2} \\ \widehat{D}_{c_2;2,1} & \widehat{D}_{c_2;2,2} \end{pmatrix},$$

where $\widehat{D}_{c_1;2,2}$ and $\widehat{D}_{c_2;2,2}$ are of size $r \times r$ and correspond to the rows of $\tilde{U}_{c_1}$ and $\tilde{U}_{c_2}$, respectively. Compute an LQ factorization independently within each context:

$$\begin{aligned} \begin{pmatrix} \widehat{D}_{c_1;1,1} & \widehat{D}_{c_1;1,2} \end{pmatrix} &= \begin{pmatrix} \widetilde{D}_{c_1;1,1} & 0 \end{pmatrix} P_{c_1}, \\ \begin{pmatrix} \widehat{D}_{c_2;1,1} & \widehat{D}_{c_2;1,2} \end{pmatrix} &= \begin{pmatrix} \widetilde{D}_{c_2;1,1} & 0 \end{pmatrix} P_{c_2}. \end{aligned}$$

We multiply P_{c_1} and P_{c_2} to the block columns independently within each context and obtain

$$\begin{aligned} &\begin{pmatrix} Q_{c_1}^H & 0 \\ 0 & Q_{c_2}^H \end{pmatrix} \begin{pmatrix} D_{c_1} & U_{c_1} B_{c_1} V_{c_2}^H \\ U_{c_2} B_{c_2} V_{c_1}^H & D_{c_2} \end{pmatrix} \begin{pmatrix} P_{c_1}^H & 0 \\ 0 & P_{c_2}^H \end{pmatrix} \\ &= \begin{pmatrix} \begin{pmatrix} \widetilde{D}_{c_1;1,1} & 0 \\ \widetilde{D}_{c_1;2,1} & \widetilde{D}_{c_1;2,2} \end{pmatrix} & \begin{pmatrix} 0 \\ \widetilde{U}_{c_1} B_{c_1} \begin{pmatrix} \widetilde{V}_{c_2;1}^H & \widetilde{V}_{c_2;2}^H \end{pmatrix} \end{pmatrix} \\ \begin{pmatrix} 0 \\ \widetilde{U}_{c_2} B_{c_2} \begin{pmatrix} \widetilde{V}_{c_1;1}^H & \widetilde{V}_{c_1;2}^H \end{pmatrix} \end{pmatrix} & \begin{pmatrix} \widetilde{D}_{c_2;1,1} & 0 \\ \widetilde{D}_{c_2;2,1} & \widetilde{D}_{c_2;2,2} \end{pmatrix} \end{pmatrix}, \end{aligned} \tag{5.3}$$

where the blocks are partitioned conformably. See Figure 5.1(c). We note that there is still no intercontext communication up to this stage.

After this, we can assign new generators to the parent node i of c_1 and c_2:
(5.4)

$$D_i = \begin{pmatrix} \widetilde{D}_{c_1;2,2} & \widetilde{U}_{c_1} B_{c_1} \widetilde{V}_{c_2;2}^H \\ \widetilde{U}_{c_2} B_{c_2} \widetilde{V}_{c_1;2}^H & \widetilde{D}_{c_2;2,2} \end{pmatrix}, \quad U_i = \begin{pmatrix} \widetilde{U}_{c_1} R_{c_1} \\ \widetilde{U}_{c_2} R_{c_2} \end{pmatrix}, \quad V_i = \begin{pmatrix} \widetilde{V}_{c_1;2} W_{c_1} \\ \widetilde{V}_{c_2;2} W_{c_2} \end{pmatrix}.$$

These generators are formed via intercontext communications. See Figure 5.1(d). Equation (5.4) maintains the form of the recursive definition (2.1) of the HSS generators, except that the size has been reduced due to the HSS compression introduced in section 4. Then we remove c_1 and c_2 from $\mathcal{T}$ (the related processes are finished), so that i becomes a leaf and we can repeat the above steps on i.

Such a step is then performed recursively. When the root node is reached, an LU factorization with partial pivoting is performed on D_i.

We now examine the communication cost in the HSS factorization. In the first step corresponding to the leaf level, each process performs local QL and LQ factorizations with U_i of size bounded by $m \times r$. No communication is involved. In the subsequent higher levels, the sizes of all the matrices are bounded by $2r \times 2r$, as in Figure 5.1(d) and (5.4). The `ScaLAPACK` QL/LQ factorization and matrix multiplication routines all have the communication cost $[\mathcal{O}(\frac{2r}{b}),\ \mathcal{O}(\frac{(2r)^2}{\sqrt{P_i}})]$ [2], where b is the block size used in `ScaLAPACK`, and P_i is the number of processes used for node i of $\mathcal{T}$. Summing over all the levels, the total cost is bounded by

$$[\mathcal{O}(\frac{r}{b}\log P),\ \mathcal{O}(r^2 \log P)].$$

Since $r \ll n$, this cost is much smaller than that incurred during the HSS compression phase (see (4.10)–(4.11)).

Clearly, there are $\mathcal{O}(r^3)$ operations associated with each node of the HSS tree. Thus, the total cost for the factorization is [29]

$$\mathcal{O}\left(\frac{n}{r}\right) \times \mathcal{O}(r^3) = \mathcal{O}(r^2 n).$$

Note that in the HSS ULV factorization, the two major operations (introducing zeros into the off-diagonal blocks and partially factorizing the diagonal blocks) are both done by QR factorizations. Thus, the overall ULV algorithm can be considered as a generalized QR algorithm, and generally no pivoting is necessary. This can also be understood as follows. If a local diagonal block is ill-conditioned, the intermediate QR factorizations pass the ill-conditioning all the way up along the tree, until the root node is reached, where the final reduced matrix [31] has the same condition number as the original HSS matrix, but is small and can be factorized accurately. In fact, it is shown in [33] that the stability of the ULV factorization is significantly better than standard LU factorizations with partial pivoting.

6. Parallel HSS solution.

We solve the linear system of equations $Ax = b$ after obtaining an HSS approximation to A in section 4 and the ULV factorization in section 5. We continue the discussion for the block 2×2 HSS form in section 5, and the HSS system looks like

$$\begin{pmatrix} D_{c_1} & U_{c_1} B_{c_1} V_{c_2}^H \\ U_{c_2} B_{c_2} V_{c_1}^H & D_{c_2} \end{pmatrix} \begin{pmatrix} x_{c_1} \\ x_{c_2} \end{pmatrix} = \begin{pmatrix} b_{c_1} \\ b_{c_2} \end{pmatrix}. \tag{6.1}$$

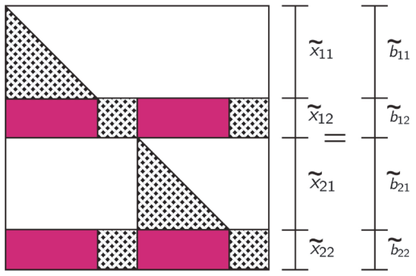


FIG. 6.1. *Illustration of the linear system of* (6.2) *when* $i = 3$, $c_1 = 1$, *and* $c_2 = 2$.

With the aid of (5.3), we can rewrite (6.1) into the following form:

$$\begin{pmatrix} \begin{pmatrix} \widetilde{D}_{c_1;1,1} & 0 \\ \widetilde{D}_{c_1;2,1} & \widetilde{D}_{c_1;2,2} \end{pmatrix} & \begin{pmatrix} 0 \\ \widetilde{U}_{c_1} B_{c_1} \begin{pmatrix} \widetilde{V}^H_{c_2;1} & \widetilde{V}^H_{c_2;2} \end{pmatrix} \end{pmatrix} \\ \begin{pmatrix} 0 \\ \widetilde{U}_{c_2} B_{c_2} \begin{pmatrix} \widetilde{V}^H_{c_1;1} & \widetilde{V}^H_{c_1;2} \end{pmatrix} \end{pmatrix} & \begin{pmatrix} \widetilde{D}_{c_2;1,1} & 0 \\ \widetilde{D}_{c_2;2,1} & \widetilde{D}_{c_2;2,2} \end{pmatrix} \end{pmatrix} \begin{pmatrix} \widetilde{x}_{c_1;1} \\ \widetilde{x}_{c_1;2} \\ \widetilde{x}_{c_2;1} \\ \widetilde{x}_{c_2;2} \end{pmatrix} = \begin{pmatrix} \widetilde{b}_{c_1;1} \\ \widetilde{b}_{c_1;2} \\ \widetilde{b}_{c_2;1} \\ \widetilde{b}_{c_2;2} \end{pmatrix}, \tag{6.2}$$

where

$$x_{c_1} = P^H_{c_1} \widetilde{x}_{c_1} = P^H_{c_1} \begin{pmatrix} \widetilde{x}_{c_1;1} \\ \widetilde{x}_{c_1;2} \end{pmatrix}, \quad x_{c_2} = P^H_{c_2} \widetilde{x}_{c_2} = P^H_{c_2} \begin{pmatrix} \widetilde{x}_{c_2;1} \\ \widetilde{x}_{c_2;2} \end{pmatrix},$$
$$b_{c_1} = Q_{c_1} \widetilde{b}_{c_1} = Q_{c_1} \begin{pmatrix} \widetilde{b}_{c_1;1} \\ \widetilde{b}_{c_1;2} \end{pmatrix}, \quad b_{c_2} = Q_{c_2} \widetilde{b}_{c_2} = Q_{c_2} \begin{pmatrix} \widetilde{b}_{c_2;1} \\ \widetilde{b}_{c_2;2} \end{pmatrix}.$$

Equation (6.2) is illustrated by Figure 6.1. We point out that the solution to (6.1) is converted into the solution to (6.2). We can easily compute the original solution x once $\widetilde{x}_{c_1}$ and $\widetilde{x}_{c_2}$ are obtained as follows.

First, the following two triangular systems can be efficiently solved locally within each context:

$$\widetilde{D}_{c_1;1,1}\ \widetilde{x}_{c_1;1} = \widetilde{b}_{c_1;1}, \quad \widetilde{D}_{c_2;1,1}\ \widetilde{x}_{c_2;1} = \widetilde{b}_{c_2;1}.$$

Then a local update of the right-hand side is conducted:

$$\widetilde{b}_{c_1;2} = \widetilde{b}_{c_1;2} - \widetilde{D}_{c_1;2,1}\ \widetilde{x}_{c_1;1}, \quad \widetilde{b}_{c_2;2} = \widetilde{b}_{c_2;2} - \widetilde{D}_{c_2;2,1}\ \widetilde{x}_{c_2;1}.$$

Up to this stage, there is no intercontext communication between c_1's and c_2's contexts.

Next, we update the right-hand side via intercontext communication:

$$\widetilde{b}_{c_1;2} = \widetilde{b}_{c_1;2} - \widetilde{U}_{c_1} \left[B_{c_1} \left(\widetilde{V}^H_{c_2;1}\ \widetilde{x}_{c_2;1} \right) \right],$$
$$\widetilde{b}_{c_2;2} = \widetilde{b}_{c_2;2} - \widetilde{U}_{c_2} \left[B_{c_2} \left(\widetilde{V}^H_{c_1;1}\ \widetilde{x}_{c_1;1} \right) \right].$$

Finally, we solve a system on the i context:

$$\begin{pmatrix} \widetilde{D}_{c_1;2,2} & \widetilde{U}_{c_1} B_{c_1} \widetilde{V}^H_{c_2;2} \\ \widetilde{U}_{c_2} B_{c_2} \widetilde{V}^H_{c_1;2} & \widetilde{D}_{c_2;2,2} \end{pmatrix} \begin{pmatrix} \widetilde{x}_{c_1;2} \\ \widetilde{x}_{c_2;2} \end{pmatrix} = \begin{pmatrix} \widetilde{b}_{c_1;2} \\ \widetilde{b}_{c_2;2} \end{pmatrix}.$$

This can be done by two triangular solutions after the LU factorization of the coefficient matrix.

For the general case, the above idea is applied recursively. Clearly, there are $\mathcal{O}(r^2)$ operations associated with each node of the HSS tree. Thus, the total cost for the solution is [29]

$$\mathcal{O}\left(\frac{n}{r}\right) \times \mathcal{O}(r^2) = \mathcal{O}(rn).$$

7. Performance tests and numerical results. In this section, we present the performance results of our parallel HSS solver when applied to some matrices arising from real applications. We carry out the experiments on the cluster Cray XE6 (hopper.nersc.gov) at the National Energy Research Scientific Computing Center. Each node has two 12-core AMD MagnyCours 2.1-GHz processors with 24 cores in total on a node. Each node has 32 GB memory.

Our primary application testbed is the modeling of time-harmonic seismic waves, for which we need to solve the Helmholtz equation of the form

$$\left(-\Delta - \frac{\omega^2}{v(x)^2}\right) u(x,\omega) = s(x,\omega), \tag{7.1}$$

where Δ is the Laplacian, ω is the angular frequency, $v(x)$ is the wavespeed (possibly complex), and $u(x,\omega)$ is called the time-harmonic wavefield solution to the forcing term $s(x,\omega)$. Helmholtz equations arise frequently in real applications such as seismic imaging. The discretization of the Helmholtz operator commonly leads to very large sparse matrices $\mathcal{A}$ which are indefinite and ill-conditioned. The matrices are complex and unsymmetric. It has been observed that in the direct factorization of $\mathcal{A}$ the dense intermediate matrices are highly compressible [10, 24].

Previously, we developed a parallel multifrontal factorization method for sparse matrix $\mathcal{A}$ based on the multifrontal method [8] with nested dissection reordering [11]. Some performance results of the multifrontal solver for this application were presented in [25]. Now, we apply our parallel HSS construction/ULV factorization/solution algorithms to the dense Schur complement matrix corresponding to the root of the separator tree in the nested dissection partitioning. In the following experiments, we appy nested dissection to either a 2D $k \times k$ mesh or a 3D $k \times k \times k$ mesh, where the mesh is recursively partitioned into submeshes by the separators. The dimension of the *last* Schur complement matrix A is $n = k$ in two dimensions and $n = k^2$ in three dimensions. The frequency $\omega = 5Hz$ is used to obtain $\mathcal{A}$, and A is obtained after partial elimination of all the variables prior to the top level separator.

For these seismic applications, often the initial data has low accuracy (one to two digits) in approximating the actual problems. Thus, two to three digits are usually sufficient for compression accuracy. In our tests, we use $\tau = 10^{-3}$ in the compression algorithm (Algorithm 1), and single precision is used in the code. Detailed error and stability analysis is performed in [33], which indicates that we can usually get nice accuracy as desired. To validate the accuracy, we present the normwise relative residual, $\frac{\|Ax-b\|_\infty}{\|A\|_\infty \|x\|_\infty + \|b\|_\infty}$, in the last row of Tables 7.1 and 7.2, when we solve $Ax = b$ using the HSS compression and ULV factorization of A. Recall that $\varepsilon \approx 5.96 \times 10^{-8}$ in single precision.

In this paper, for convenience, we assume the dense matrix is stored first. In practice, for problems such as solving Toeplitz problems with HSS methods in Fourier space [33], there is no need to store the dense matrix in advance, and the blocks can be formed dynamically based on some analytical representations. Even if we need to

Table 7.1

Parallel runtime, gigaflops rate, and memory usage of the HSS algorithms applied to the largest dense frontal matrices A from the exact multifrontal factorization of 2D discretized Helmholtz operators in (7.1). n is the order of A, P is the number of processes, and the relative tolerance in HSS construction is $\tau = 10^{-3}$. Each addition (multiplication) of two complex numbers is counted as two (six) regular flops.

k (mesh: $k \times k$; $n = k$)			5,000	10,000	20,000	40,000	80,000
P			16	64	256	$1,024$	$4,096$
Dense LU	LU factorization (s)		1.96	4.27	9.35	22.47	63.06
	Gflops/s		170.3	625.1	2280.4	7596.4	21649.9
	Triangular solve (s)		0.01	0.03	0.05	0.10	0.30
	Gflops/s		27.7	63.1	133.5	265.8	336.7
	Total memory (GB)		0.2	0.8	3.2	12.8	51.2
HSS	HSS rank r		9	9	11	12	17
	Constr.	Total (s)	0.24	0.31	0.49	0.97	2.31
		RRQR (s)	0.16	0.19	0.23	0.34	0.41
		Redist. (s)	0.06	0.05	0.10	0.19	0.32
	Gflops/s		12.3	44.1	120.4	260.8	462.8
	ULV factorization (s)		0.11	0.09	0.11	0.21	0.45
	Gflops/s		95.6	243.5	402.3	427.4	399.4
	Solution (s)		0.01	0.02	0.12	0.53	2.76
	Gflops/s		1.9	2.1	0.8	0.4	0.1
	Total memory (GB)		0.05	0.05	0.11	0.21	0.43
	Relative residual		5.2e-4	8.1e-4	2.5e-3	4.7e-3	5.5e-3

Table 7.2

Parallel runtime and gigaflops rate of the HSS algorithms applied to the largest dense frontal matrices A from the exact multifrontal factorization of 3D discretized Helmholtz operators in (7.1), where n^2 is the size of A, P is the number of processes, and the relative tolerance in HSS construction is $\tau = 10^{-3}$. Each addition (multiplication) of two complex numbers is counted as two (six) regular flops.

k (mesh: $k \times k \times k$; $n = k^2$)			100	200	300	400	500
P			64	256	1,024	4,096	8,192
Dense LU	LU factorization (s)		4.21	57.62	175.85	312.87	540.28
	Gflops/s		633.3	2961.8	11054.6	34911.6	77120.2
	Triangular solve (s)		0.02	0.12	0.30	0.80	1.34
	Gflops/s		65.6	215.2	434.6	511.3	748.4
	Total memory (GB)		0.8	12.8	64.8	204.8	500
HSS	HSS rank r		335	618	894	1226	1497
	Constr.	Total (s)	8.30	51.49	193.40	207.72	259.46
		RRQR (s)	7.94	50.09	189.86	203.94	251.56
		Redist. (s)	0.24	1.05	2.21	2.11	3.22
	Gflops/s		100.8	382.2	595.5	2041.5	4050.1
	ULV factorization (s)		0.40	1.42	1.76	2.46	4.16
	Gflops/s		166.3	495.3	1286.5	2863.8	2513.7
	Solution (s)		0.04	0.20	0.55	2.29	9.52
	Gflops/s		1.1	1.6	1.5	0.9	0.3
	Total memory (GB)		0.12	0.76	1.96	4.55	6.84
	Relative residual		9.7e-3	1.1e-2	1.7e-2	1.5e-2	1.7e-2

form the dense matrix first, we may work by panels on part of the matrix, and the panel storage can be reused by the panels at different times. In the results presented below, the memory we consider is the memory for the HSS factors, as compared with the dense LU factors.

Example 7.1. The 2D Helmholtz equations are discretized on $\mathrm{k} \times \mathrm{k}$ mesh. We apply the parallel HSS algortihms to the last Schur complement A corresponding to the top level separator in the nested dissection ordering.

For the weak scaling test, we increase the number of cores by a factor of four upon doubling the mesh size k. Table 7.1 shows the various performance statistics of the parallel HSS methods for k ranging from 5,000 to 80,000. As a comparison, we include the results of parallel dense LU factorization and triangular solution from the routines `PCGETRF` and `PCGETRS` in `ScaLAPACK` [21]. Recall that the HSS rank r is defined as the maximum rank of all the HSS blocks. Here, for the 2D problems, the HSS ranks are more or less constant (less than 20) and are independent of the problem size. We split the total HSS compression time into an RRQR factorization (Algorithm 1) part and a data redistribution part. As predicted, the HSS factorization and solution are faster than the HSS compression. Inside the compression phase, the redistribution part takes less time than the RRQR factorization. This validates our earlier analysis on communication complexity; see section 4.4.

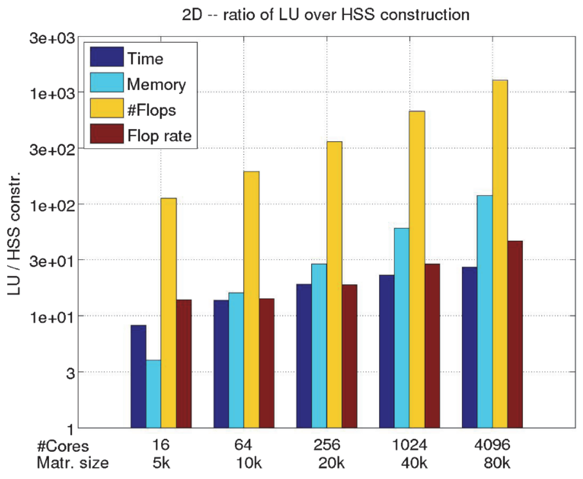
Figure 7.1(a) visually compares the weak scaling performance of the new HSS approach to the LU approach. Here we focus only on the HSS construction algorithm, because it dominates the ULV factorization and solution. For each performance metric, time, storage, #flops, and gigaflop rate, we plot the ratio of the LU algorithm over the HSS construction algorithm. In all these metrics, the HSS method is always better than the LU approach and scales better in parallel runtime, memory usage, and #flops. For the largest mesh size 80,000 using 4096 cores, the HSS algorithm is about 30 times faster than the LU algorithm and memory usage is reduced by 100-fold. The #flops increases faster for the LU than for the HSS. For the largest problem, the LU #flops is about 1000 times more than the HSS flops. On the other hand, the LU Gflops/s rate is much higher than the HSS Gflops/s rate, as much as 40 times higher. This is because the HSS algorithm involves operations with much smaller matrices and spends a larger fraction of the time in memory access and communication, as analyzed in section 4. (See remarks on the flop-to-byte ratio in section 4.4.) Therefore, the flop count alone is not a very good predictive metric about parallel runtime for this class of algorithms.

Example 7.2. The 3D Helmholtz equations discretized on $\mathrm{k} \times \mathrm{k} \times \mathrm{k}$ mesh. We apply the parallel HSS algortihms to the last Schur complement A corresponding to the top level separator in a nested dissection ordering.

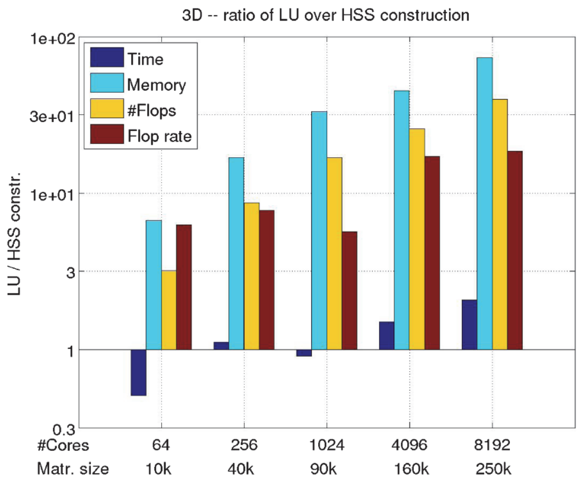
We now repeat the same experiments for the 3D problems. We increase the number of cores used while increasing the mesh size k. Table 7.2 shows the various performance statistics of the parallel HSS methods for k ranging from 100 to 500 (i.e., n ranging from 10,000 to 250,000. As a comparison, we include the results of parallel dense LU factorization and triangular solution from the routines `PCGETRF` and `PCGETRS` in `ScaLAPACK`.

In contrast to the 2D case, now the HSS ranks increase with the problem size. The growth is linear with respect to one side of the mesh (i.e., k), which seems to corroborate the theory predicted in [10]. The larger HSS ranks give less advantage for the HSS method over the LU method than the 2D case. Figure 7.1(b) compares the weak scaling performance of the HSS construction to the LU factorization. The HSS time is faster than that of LU for large problems. For the largest mesh size 500^3 using 8192 cores, the HSS algorithm is over 2 times faster and memory usage is reduced by 70-fold. The #flops increases faster for the LU than for the HSS. For the largest problem, the LU #flops is about 40 times more than the HSS flops. On the other hand, the LU Gflops/s rate is 18 times higher than the HSS Gflops/s rate.

Notice that for the 3D case, the RRQR compression time much more dominates the entire HSS construction time. One reason is that the HSS rank r is much larger now, which represents a larger factor to both the computation cost ($\mathcal{O}(rn^2)$) and the communication cost associated with the RRQR compression algorithm (see (4.8) to (4.11)). Another reason is that the ranks of different HSS blocks (i.e., different branches of the HSS tree) are not of similar values. We observed over 2 times the difference between the ranks of some branches. With our current data distribution scheme (see Figure 3.2), this causes significant load imbalance among the processors



(a) 2D



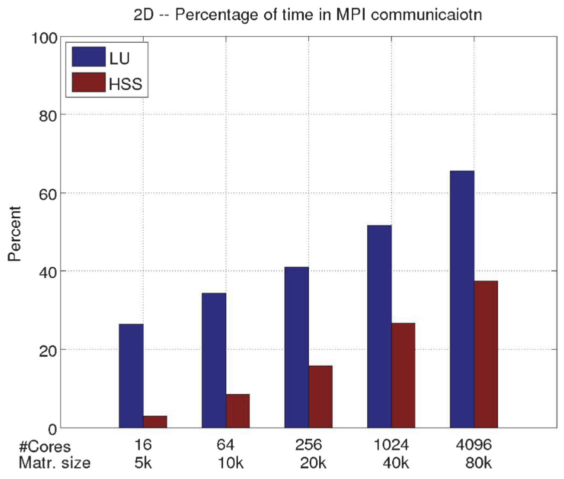
(b) 3D

FIG. 7.1. *Performance ratio of LU over HSS.*

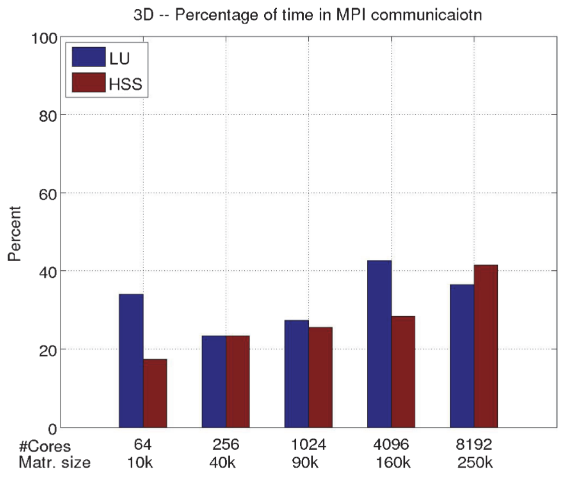
on the two branches of the tree. Overall, the RRQR time for the slowest processor can be 2 to 3 times larger than that of the fastest processor. We will address this load imbalance issue in our future work.

7.1. Communication cost. Finally, we examine the communication overhead for the same 2D and 3D matrices. The communication statistics are collected using the IPM (integrated performance monitoring) performance profiling tool [18].

Figure 7.2 compares the time spent in communication for the HSS construction and for the LU factorization. In the 2D case, Figure 7.2(a), the fraction of time in communication increases with the increasing matrix size and the number of processing cores used. Furthermore, the LU algorithm spends a larger fraction of time in communication than the HSS algorithm. In the 3D case, Figure 7.2(b), the fraction



(a) 2D



(b) 3D

FIG. 7.2. *Percentage time spent in MPI communication.*

of time in communication also increases with the increasing matrix size and number of cores. But now, both the LU algorithm and the HSS algorithm spend a similar fraction of time in communication. This is because the HSS ranks in the 3D case are much larger, which contributes to a larger amount of communication (see (4.10) and (4.11)).

8. Conclusions.

We have designed and implemented novel parallel algorithms for the HSS structured matrix algorithms in parallel computation. We are able to conduct classical structured compression, factorizations, and solutions in parallel. We performed a detailed analysis of the communication costs of the parallel algorithms and found that the algorithms are more communication bound than the other algorithms without using the rank structures. This is mainly because the amount of floating point operations is drastically reduced by exploiting the low rankness, but the reduction in communication is moderate. The future challenge is to design novel communication-avoiding algorithms for this class of rank structured methods.

Our implementations are portable by using the two well-established libraries `BLACS` and `ScaLAPACK`. The computational results for weak scaling, strong scaling, and accuracy demonstrate the high efficiency and scalability of our algorithms. The algorithms are very useful in solving large dense and sparse linear systems that arise in real-world applications. For example, we have applied our new parallel HSS-embedded multifrontal solver to the Helmholtz equation for time-harmonic seismic inverse boundary value problems, and we were able to solve a linear system with 6.4 billion unknowns using 4096 processors in about 20 minutes. The classical multifrontal solver simply failed due to high demand for memory. Our techniques can also benefit the development of fast parallel methods using the other rank structures.

Acknowledgments. We thank Francois-Henry Rouet for insightful discussions about the parallel performance bottleneck of the code and for providing the code for computing the residual. We thank the members of the Geo-Mathematical Imaging Group at Purdue University, BGP, ConocoPhillips, ExxonMobil, PGS, Statoil, and Total for partial financial support.

REFERENCES

[1] T. Bella, Y. Eidelman, V. Gohberg, and I. Olshevsky, *Computations with quasiseparable polynomials and matrices*, Theoret. Comput. Sci., 409 (2008), pp. 158–179.
[2] L. S. Blackford, J. Choi, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*, Software Environ. Tools, SIAM, Philadelphia, 1997.
[3] *BLACS*. http://www.netlib.org/blacs.
[4] S. Börm, L. Grasedyck, and W. Hackbusch, *Introduction to hierarchical matrices with applications*, Eng. Anal. Bound. Elem, 27 (2003), pp. 405–422.
[5] S. Börm and W. Hackbusch, *Data-Sparse Approximation by Adaptive $\mathcal{H}^2$-Matrices*, Technical report, Max Planck Institute for Mathematics, Leipzig, Germany, 2001.
[6] 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.
[7] 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.
[8] I. S. Duff and J. K. Reid, *The multifrontal solution of indefinite sparse symmetric linear equations*, ACM Trans. Math. Softw., 9 (1983), pp. 302–325.
[9] Y. Eidelman and I. Gohberg, *On a new class of structured matrices*, Integral Equations Operator Theory, 34 (1999), pp. 293–324.
[10] B. Engquist and L. Ying, *Sweeping preconditioner for the Helmholtz equation: Hierarchical matrix representation*, Comm. Pure Appl. Math., 64 (2011), pp. 697–735.

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

[12] G. H. Golub and C. V. Loan, *Matrix Computations*, 3rd ed., Johns Hopkins University Press, Baltimore, MD, 1996.

[13] 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.

[14] W. Hackbusch, L. Grasedyck, and S. Börm, *An introduction to hierarchical matrices*, Math. Bohem., 127 (2002), pp. 229–241.

[15] W. Hackbusch and B. N. Khoromskij, *A sparse $\mathcal{H}$-matrix arithmetic. Part* II*: Application to multi-dimensional problems*, Computing, 64 (2000), pp. 21–47.

[16] W. Hackbusch, B. Khoromskij, and S. A. Sauter, *On $\mathcal{H}^2$-Matrices*, Lectures on Appl. Math., Springer, Berlin, 2000, pp. 9–29.

[17] W. Hackbusch, *A sparse matrix arithmetic based on $\mathcal{H}$-matrices. Part* I*: Introduction to $\mathcal{H}$-matrices*, Computing, 62 (1999), pp. 89–108.

[18] *Integrated Performance Monitoring: IPM,* http://ipm-hpc.sourceforge.net.

[19] *LAPACK—Linear Algebra PACKage*, http://www.netlib.org/lapack.

[20] *Message Passing Interface Forum*, http://www.mpi-forum.org.

[21] *ScaLAPACK—Scalable Linear Algebra PACKage*, http://www.netlib.org/scalapack.

[22] R. Thakur, R. Rabenseifner, and W. Gropp, *Optimization of collective communication operations in MPICH*, Int. J. High Performance Comput. Appl., 19 (2005), pp. 49–66.

[23] R. Vandebril, M. Van Barel, G. Golub, and N. Mastronardi, *A bibliography on semiseparable matrices*, Calcolo, 42 (2005), pp. 249–270.

[24] S. Wang, M. V. de Hoop, and J. Xia, *Seismic inverse scattering via Helmholtz operator factorization and optimization*, J. Comput. Phys., 229 (2010), pp. 8445–8462.

[25] S. Wang, M. V. de Hoop, and J. Xia, *On* 3*D modeling of seismic wave propagation via a structured massively parallel multifrontal direct Helmholtz solver*, Geophys. Prospect., 59 (2011), pp. 857–873.

[26] S. Wang, M. V. de Hoop, J. Xia, and X. S. Li, *Massively parallel structured multifrontal solver for time-harmonic elastic waves in* 3*D anisotropic media*, Geophys. J. Int., 91 (2012), pp. 346–366.

[27] S. Wang, J. Xia, M. V. de Hoop, and X. S. Li, *Massively parallel structured direct solver for equations describing time-harmonic qP-polarized waves in TTI media*, Geophysics, 77 (2012), pp. T69–T82.

[28] 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.

[29] J. Xia, S. Chandrasekaran, M. Gu, and X. S. Li, *Fast algorithms for hierarchically semiseparable matrices*, Numer. Linear Algebra Appl., 2010 (2010), pp. 953–976.

[30] J. Xia and M. Gu, *Robust approximate cholesky factorization of rank-structured symmetric positive definite matrices*, SIAM J. Matrix Anal. Appl., 31 (2010), pp. 2899–2920.

[31] J. Xia, *Efficient structured multifrontal factorization for general large sparse matrices*, SIAM J. Sci. Comput., 35 (2012), pp. A832–A860.

[32] J. Xia, *On the complexity of some hierarchical structured matrix algorithms*, SIAM J. Matrix Anal. Appl., 33 (2012), pp. 388–410.

[33] 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., submitted.