NUMERICAL LINEAR ALGEBRA WITH APPLICATIONS
Numer. Linear Algebra Appl. 2010; **17**:953–976
Published online 22 December 2009 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/nla.691

Fast algorithms for hierarchically semiseparable matrices

Jianlin Xia[1,*,†], Shivkumar Chandrasekaran[2], Ming Gu[3] and Xiaoye S. Li[4]

[1]*Department of Mathematics, Purdue University, West Lafayette, IN 47907, U.S.A.*
[2]*Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106, U.S.A.*
[3]*Department of Mathematics, University of California, Berkeley, CA 94720, U.S.A.*
[4]*Lawrence Berkeley National Laboratory, MS 50F-1650, One Cyclotron Rd., Berkeley, CA 94720, U.S.A.*

SUMMARY

Semiseparable matrices and many other rank-structured matrices have been widely used in developing new fast matrix algorithms. In this paper, we generalize the hierarchically semiseparable (HSS) matrix representations and propose some fast algorithms for HSS matrices. We represent HSS matrices in terms of general binary HSS trees and use simplified postordering notation for HSS forms. Fast HSS algorithms including new HSS structure generation and HSS form Cholesky factorization are developed. Moreover, we provide a new linear complexity explicit ULV factorization algorithm for symmetric positive definite HSS matrices with a low-rank property. The corresponding factors can be used to solve the HSS systems also in linear complexity. Numerical examples demonstrate the efficiency of the algorithms. All these algorithms have nice data locality. They are useful in developing fast-structured numerical methods for large discretized PDEs (such as elliptic equations), integral equations, eigenvalue problems, etc. Some applications are shown. Copyright © 2009 John Wiley & Sons, Ltd.

Received 10 January 2009; Revised 11 November 2009; Accepted 13 November 2009

KEY WORDS: HSS matrix; postordering HSS tree; low-rank property; fast HSS algorithms; generalized HSS Cholesky factorization

1. INTRODUCTION

Rank-structured matrices have attracted much attention in recent years. These matrices have been shown very effective in solving linear systems [1–4], least squares problems [5], eigenvalue problems [6–11], PDEs [12–15], integral equations [16–18], etc. Examples of rank-structured matrices include $\mathscr{H}$-matrices [19–21], $\mathscr{H}^2$-matrices [22–24], quasiseparable matrices [25], and semiseparable matrices [26–28]. It has been noticed that in the factorizations of some discretized PDEs and integral equations, dense intermediate matrices have off-diagonal blocks with small numerical ranks. By exploiting this *low-rank property*, fast numerical methods can be developed. The property is also observed in some eigenvalue problems [6–8].

*Correspondence to: Jianlin Xia, Department of Mathematics, Purdue University, West Lafayette, IN 47907, U.S.A.
†E-mail: xiaj@math.purdue.edu

Contract/grant sponsor: NSF; contract/grant numbers: CCF-0515034, CCF-0830764

Here, we consider a type of semiseparable structures called hierarchically semiseparable (HSS) matrices proposed in [1, 2, 29]. HSS structures have been used to develop fast direct or iterative solvers for both dense and sparse linear systems [1, 2, 15, 29–31]. For dense matrices with the low-rank property and represented in HSS forms, the linear system solution needs only linear complexity based on implicit *ULV*-type factorizations, where U and V are orthogonal matrices and L is lower-triangular [2, 29]. HSS matrices can also be used in many other applications where the low-rank property exists. The HSS representation features a nice hierarchical multi-level tree structure (HSS tree), and is efficient in capturing the semiseparable low-rank property. HSS structures are closely related to $\mathscr{H}$-matrices [19–21] and $\mathscr{H}^2$-matrices [22–24], and is a special case of the representations in the fast multipole method [18, 32, 33]. It is also a generalization of the sequentially semiseparable representation [5, 26].

In the meantime, the implementation of the current HSS algorithms is still not very convenient, partly due to the tedious notation and the level-wise (global) tree traversal scheme. Moreover, some existing HSS algorithms can be highly improved, such as the structure construction and system solution, especially for symmetric problems.

Thus in this paper, we present simplifications and generalizations of HSS representations, which are easier to use and have better data locality. We also develop some HSS algorithms with better efficiency. With the improved HSS representation, most HSS algorithms can be done more conveniently by traversing postordering HSS trees. Our generalizations of HSS structures also allow more flexibility in dealing with arbitrary HSS tree patterns.

The algorithms we propose can be used to provide nearly linear complexity direct solvers for sparse-discretized PDEs [15], such as elliptic equations and others [12–14, 34, 35]. Note that these dense HSS techniques apply to discretized PDEs in one-dimensional (1D) domains, but can be used in sparse matrix schemes to work on 2D problems. For example, with the nested dissection ordering of a sparse discretized PDE followed by a multifrontal factorization, we can use HSS matrices to approximate the dense intermediate fill-in [15]. Such a scheme can also possibly be used to solve or to precondition higher dimensional problems.

In addition, the postordering HSS tree notation we use simulates the assembly tree structure [36–38] in sparse matrix solutions and has a good potential to be parallelized. The ideas presented here are also useful for developing new HSS algorithms.

1.1. Review of HSS matrices

We first briefly review the standard HSS structure and give some concise definitions based on the discussions in [1, 2, 15, 29–31].

The low-rank property is concerned with the ranks or numerical ranks of certain types of off-diagonal blocks. Here, by *numerical ranks* we mean the ranks obtained by rank revealing QR factorizations or τ-accurate SVD (SVD with an absolute or relative tolerance τ for the singular values). The off-diagonal blocks used in HSS representations are called *HSS blocks* as shown in Figure 1(i), (ii). They are block rows or columns without diagonal parts and are hierarchically defined for different levels of splittings of the matrix.

Definition 1.1 (*HSS blocks*)
For an $N\times N$ matrix H and a partition sequence $\{m_{k;j}\}_{j=1}^{2^k}$ satisfying $\sum_{i=1}^{2^k} m_{k;j}=N$, partition H into 2^k block rows so that block row j has row dimension $m_{k;j}$. Similarly, partition the columns. Denote the $m_{k;j}\times m_{k;j}$ intersection block of block row j and block column j by $D_{k;i}$. Then,

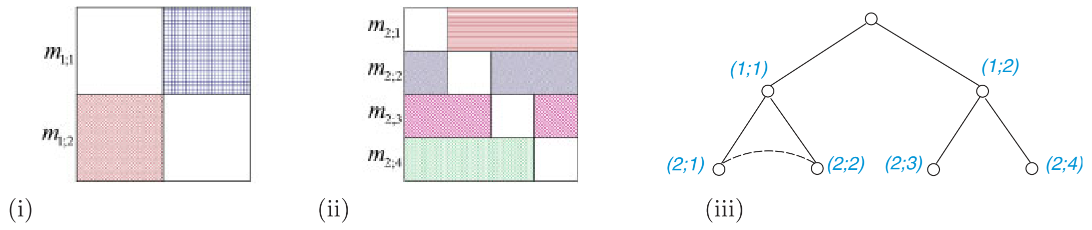

Figure 1. Two levels of HSS off-diagonal blocks: (i) first-level HSS blocks; (ii) second-level HSS blocks; and (iii) corresponding binary tree.

block row (column) j with $D_{k;j}$ removed is called the j-th HSS block row (column) at the k-th level (bottom level) partition. Upper level HSS blocks are similarly defined using the partition sequences $\{m_{i;j}\}_{j=1}^{2^i}$, which are given recursively by

$$m_{i-1;j} = m_{i;2j-1} + m_{i;2j}, \quad j = 1, 2, \ldots, 2^{i-1}-1, 2^{i-1}, \quad i = k, k-1, \ldots, 2, 1.$$

Thus, the jth HSS block row at level i has dimensions $m_{i;j} \times (N - m_{i;j})$. Clearly, these blocks can be associated with a binary tree. For example, we can have a binary tree as shown in Figure 1(iii) corresponding to Figure 1(i), (ii), where each tree node is associated with an HSS block row and column. Later, we use $(i;j)$ to denote the j-th node at level i of the tree.

Definition 1.2 (*HSS tree and HSS representation*)
For a matrix H and its HSS blocks as defined in Definition 1.1, let $\mathbf{T}$ be a perfect binary tree where each node is associated with an HSS block row (column). H is said to have an HSS representation if there exists matrices $D_{i;j}$, $U_{i;j}$, $V_{i;j}$, $R_{i;j}$, $W_{i;j}$, $B_{i;j,j\pm1}$ associated with each tree node $(i;j)$ which satisfy the recursions

$$D_{i-1;j} = \begin{pmatrix} D_{i;2j-1} & U_{i;2j-1} B_{i;2j-1,2j} V_{i;2j}^{\mathrm{T}} \\ U_{i;2j} B_{i;2j,2j-1} V_{i;2j-1}^{\mathrm{T}} & D_{i;2j} \end{pmatrix}, \qquad j = 1, 2, \ldots, 2^{i-1}-1, 2^{i-1}, \quad (1)$$
$$U_{i-1;j} = \begin{pmatrix} U_{i;2j-1} R_{i;2j-1} \\ U_{i;2j} R_{i;2j} \end{pmatrix}, \quad V_{i-1;j} = \begin{pmatrix} V_{i;2j-1} W_{i;2j-1} \\ V_{i;2j} W_{i;2j} \end{pmatrix},$$

$$i = k, \quad k-1, \ldots, 2, 1,$$

so that $D_{0;1} \equiv H$, corresponding to the root of $\mathbf{T}$. The matrices in (1) are called generators of H. If node $(i;j)$ of $\mathbf{T}$ is associated with the generators $D_{i;j}$, $U_{i;j}$, $V_{i;j}$, $R_{i;j}$, $W_{i;j}$, $B_{i;j,j\pm1}$, we say $\mathbf{T}$ is an HSS tree of H.

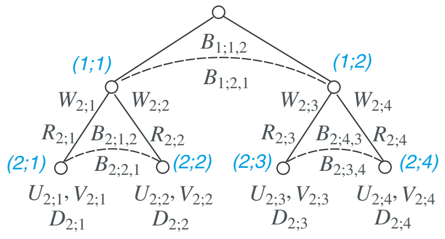

Figure 2. HSS tree for (2).

Note that due to this hierarchical structure, only R, W generators and bottom level D, U, V generators need to be stored. As an example, a block 4×4 HSS matrix looks like

$$\begin{array}{c} m_{2;1} \\ m_{2;2} \\ m_{2;3} \\ m_{2;4} \end{array} \left(\begin{array}{cc} \begin{pmatrix} D_{2;1} & U_{2;1}B_{2;1,2}V_{2;2}^{\mathrm{T}} \\ U_{2;2}B_{2;2,1}V_{2;1}^{\mathrm{T}} & D_{2;2} \end{pmatrix} & \begin{pmatrix} U_{2;1}R_{2;1} \\ U_{2;2}R_{2;2} \end{pmatrix} B_{1;1,2}(W_{2;3}^{\mathrm{T}}V_{2;3}^{\mathrm{T}} \quad W_{2;4}^{\mathrm{T}}V_{2;4}^{\mathrm{T}}) \\ \begin{pmatrix} U_{2;3}R_{2;3} \\ U_{2;4}R_{2;4} \end{pmatrix} B_{1;2,1}(W_{2;1}^{\mathrm{T}}V_{2;1}^{\mathrm{T}} \quad W_{2;2}^{\mathrm{T}}V_{2;2}^{\mathrm{T}}) & \begin{pmatrix} D_{2;3} & U_{2;3}B_{2;3,4}V_{2;4}^{\mathrm{T}} \\ U_{2;4}B_{2;4,3}V_{2,3}^{\mathrm{T}} & D_{2;4} \end{pmatrix} \end{array} \right), \tag{2}$$

which corresponds to the second-level block partition in Figure 1. The hierarchical structure of HSS matrices can be seen by writing (2) in terms of the first-level block partition

$$\begin{array}{c} m_{1;1} \equiv m_{2;1}+m_{2;2} \\ m_{1;2} \equiv m_{2;3}+m_{2;4} \end{array} \begin{pmatrix} D_{1;1} & U_{1;1}B_{1;1,2}V_{1;2}^{\mathrm{T}} \\ U_{1;2}B_{1;2,1}V_{1;1}^{\mathrm{T}} & D_{1;2} \end{pmatrix}. \tag{3}$$

The corresponding HSS tree of H is shown in Figure 2.

HSS trees can be used to conveniently represent HSS matrices. As an example, the (2, 3) block of (2) can be identified by the directed path connecting the second and third nodes at level 2:

$$\overset{U_{2;2}}{(2;2)} \xrightarrow{R_{2;2}} (1;1) \xrightarrow{B_{1;1,2}} (1;2) \xrightarrow{W_{2;3}^{\mathrm{T}}} \overset{V_{2;3}^{\mathrm{T}}}{(2;3)},$$

where U and R generators are used for outgoing directions, and V and W generators are used for incident directions. HSS trees also allow the operations on HSS matrices to be done conveniently via tree operations.

In addition, we can see that the second-level HSS block rows in (2) are given by

$$U_{2;1}(B_{2;1,2} \quad R_{2;1}B_{1;1,2})\mathrm{diag}(V_{2;2}^{\mathrm{T}}, V_{1;2}^{\mathrm{T}}), \quad U_{2;2}(B_{2;2,1} \quad R_{2;2}B_{1;1,2})\mathrm{diag}(V_{2;2}^{\mathrm{T}}, V_{1;2}^{\mathrm{T}}), \text{ etc.,}$$

where each $U_{2;i}$ is an appropriate column basis matrix and $\mathrm{diag}(V_{2;2}^{\mathrm{T}}, V_{1;2}^{\mathrm{T}})$ (denoting a diagonal matrix formed with diagonal blocks $V_{2;2}^{\mathrm{T}}$ and $V_{1;2}^{\mathrm{T}}$) is a row basis.

The effectiveness of HSS structures relies on the low-rank property.

Definition 1.3 (*low-rank property*)
A matrix with given partition sequences as in Definition 1.1 is said to have the low-rank property (in terms of HSS blocks) if all its HSS blocks have small ranks or numerical ranks.

Definition 1.4 (*HSS rank and compact HSS representation*)
The *HSS rank of a matrix H* is the maximum (numerical) rank of all the HSS off-diagonal blocks at all levels of HSS partitions of H. An HSS representation of (or approximation to) H is said to be compact if all its R and B generators have sizes close to the HSS rank which is small when compared with the matrix size.

When numerical ranks are used, the HSS form approximates the original matrix.

1.2. Main results

In this work, we first simplify and generalize HSS representations. The original HSS notation in [2, 29], as in (2) and (3), uses up to three subscripts for the generators, and existing HSS algorithms are mainly concerned with perfect binary HSS trees. The existing HSS operations are also mostly based on level-wise traversal of the trees, which may reduce the data locality. Here, we use only one subscript and allow the HSS tree to be a general (partial) binary tree. The nodes are ordered following the postordering of the tree. This simplifies the manipulation of HSS matrices and brings more flexibility to HSS operations. Postordering representations fit the patterns of many HSS algorithms and are more natural for both the notation and the data structure. They also have good data locality and are very suitable for parallel computations.

On the basis of the simplified postordering HSS notation, we provide some new HSS algorithms which are fast and stable. They include:

1. *Fast HSS structure generation algorithm* which has better complexity than the one in [29]. The cost is $O(N^2)$ flops for an order N dense matrix with the low-rank property. Comparison with [29] is discussed.
2. *Quadratic complexity explicit Cholesky factorization* of a symmetric positive definite (SPD) matrix in compact HSS form. The idea of this algorithm is useful in computing Schur complements, when an HSS matrix is partially factorized [15].
3. *Linear complexity explicit ULV factorization* of an SPD matrix in compact HSS form. The new algorithm is called a generalized HSS Cholesky factorization algorithm, because the factors consist of orthogonal transformations and triangular matrices. Numerical experiments are used to demonstrate the stability and linear complexity. Comparisons with the fast solver in [29] are given in terms of system solution.
4. *System solution* using the generalized HSS Cholesky factors. Traditional forward and backward substitutions are replaced by forward (postordering) and backward (reverse-postordering) traversals of a solution tree, respectively. This solution process also has linear complexity but with a small constant.
5. *Applications* of the new algorithms in more advanced schemes such as factorizing or preconditioning discretized PDEs.

These algorithms are both cost and memory efficient. They are useful in solving more complicated problems including discretized PDEs [15], integral equations, least squares problems, etc. For some discretized PDEs, a nearly linear complexity direct solver is proposed based on these algorithms [15].

The remaining sections are organized as follows. The next section shows the generalizations and simplifications of HSS representations. The fast HSS construction algorithm is presented in Section 3. Section 4 discusses the two types of HSS factorizations and also the generalized HSS solver. Numerical experiments are included. Some applications of these HSS algorithms are shown in Section 5. Section 6 gives some concluding remarks.

2. GENERALIZATIONS OF HSS REPRESENTATIONS

In this section, we first simplify the HSS notation and then introduce partial HSS forms.

2.1. Postordering HSS representation

Tree structures are very useful in numerical problems like direct solutions of sparse linear systems where they appear as assembly trees [36–38]. In an assembly tree, the nodes are often ordered following the postordering, which gives the actual elimination order. Similarly in the situation of HSS operations, it often needs to traverse HSS trees. A postordering of the HSS tree nodes brings more flexibility and convenience.

For example, the nodes of the HSS tree in Figure 2 can be relabeled in the postordering form as in Figure 3(i). Accordingly, the generators are relabeled with only one index each. We call this HSS notation the *postordering HSS notation*. Accordingly, the HSS off-diagonal blocks at different levels are also ordered and associated with the tree nodes. With the postordering notation, the matrix (2) now looks like

$$\begin{pmatrix} D_1 & U_1B_1V_2^{\mathrm{T}} & U_1R_1B_3W_4^{\mathrm{T}}V_4^{\mathrm{T}} & U_1R_1B_3W_5^{\mathrm{T}}V_5^{\mathrm{T}} \\ U_2B_2V_1^{\mathrm{T}} & D_2 & U_2R_2B_3W_4^{\mathrm{T}}V_4^{\mathrm{T}} & U_2R_2B_3W_5^{\mathrm{T}}V_5^{\mathrm{T}} \\ U_4R_4B_6W_1^{\mathrm{T}}V_1^{\mathrm{T}} & U_4R_4B_6W_2^{\mathrm{T}}V_2^{\mathrm{T}} & D_4 & U_4B_4V_5^{\mathrm{T}} \\ U_5R_5B_6W_1^{\mathrm{T}}V_1^{\mathrm{T}} & U_5R_5B_6W_2^{\mathrm{T}}V_2^{\mathrm{T}} & U_5B_5V_4^{\mathrm{T}} & D_5 \end{pmatrix}. \tag{4}$$

The postordering HSS notation simplifies the HSS representation and is convenient in HSS structure transformations, data manipulations, parallelization, etc. Moreover, the postordering HSS representation keeps good data locality by limiting direct communications to be between parent and child nodes only.

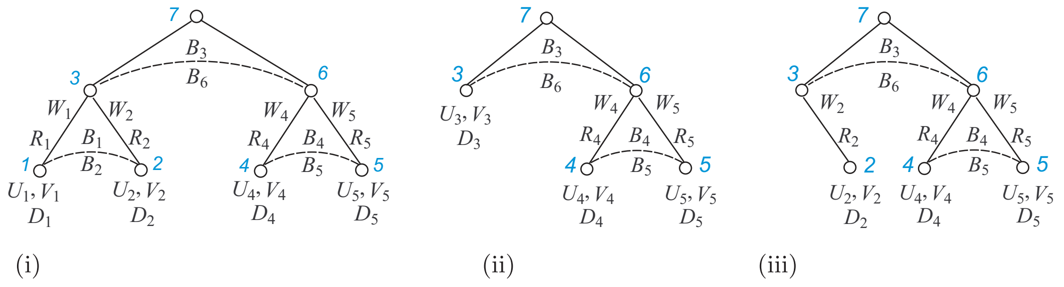

Figure 3. Examples of a postordering HSS tree and partial HSS trees: (i) postordering form of Figure 2; (ii) full HSS tree; and (iii) general partial HSS tree.

2.2. *Partial HSS form*

In Figure 3(i), the HSS tree is a perfect binary tree, that is, the tree has $2^{l+1}-1$ nodes if it has l levels (the root is at level 0). But HSS trees can be more general. For example, if we merge the first two block rows and columns of the matrix (4), we get an HSS form corresponding to Figure 3(ii). We may have even more general cases like Figure 3(iii).

An HSS tree which is not perfect is said to be a *partial HSS tree*, and the corresponding HSS matrix is in partial HSS form. In various HSS operations like solving HSS systems, it is often more convenient and practical to use partial HSS trees [15]. Thus, we consider operations on general partial HSS matrices, not necessarily restricted to complete HSS matrices as in [29]. However, a tree in Figure 3(iii) can be transformed to Figure 3(ii) by merging certain nodes and edges, so it usually suffices to consider partial HSS trees which are *full binary trees*. That is, each non-leaf node i has two children c_1 and c_2 with $c_1<c_2$, called the *left* and *right children* of i, respectively. Furthermore, it suffices to use full binary trees due to the following simple fact.

Theorem 2.1
For any integer $k>0$, there always exists a full binary tree with exactly k leaf nodes and $2k-1$ nodes in total.

Proof
A straightforward proof is to construct an unbalanced full binary tree where each right (or left) node is only a leaf node . Such a tree with k leaves has exactly $k-1$ non-leaf nodes. A generally better choice is to recursively construct a tree, which is as balanced as possible or has as small depth as possible. □

3. FAST AND STABLE CONSTRUCTION OF HSS REPRESENTATIONS

For a dense matrix H and a partition sequence $\{m_i\}$, there always exists a full HSS tree according to Theorem 2.1. We assume that an HSS tree corresponding to $\{m_i\}$ is also given. The paper [29] provides an HSS construction algorithm based on (τ-accurate) SVDs. That method traverses HSS trees by levels and, in general, can only generate HSS matrices with perfect HSS trees. Here, we provide a new algorithm which follows a general postordering (partial) HSS tree. It is fully stable and costs much less than the one in [29]. We *compress* the HSS off-diagonal blocks associated with the tree nodes. Here, by compression we mean a QR factorization or SVD of a low-rank block, or a rank-revealing QR or τ-accurate SVD of a numerically low-rank block when approximations are used.

3.1. *A block* 4×4 *example*

We first demonstrate the procedure of constructing a 4×4 block HSS form (4) for H using the postordering HSS tree in Figure 3. Initially, we partition the matrix H into a 4×4 block form

$$H=\begin{pmatrix} D_1 & T_{12} & T_{14} & T_{15} \\ T_{21} & D_2 & T_{24} & T_{25} \\ T_{41} & T_{42} & D_4 & T_{45} \\ T_{51} & T_{52} & T_{54} & D_5 \end{pmatrix},$$

where the subscripts follow the node ordering, that is T_{ij} denotes the block corresponding to nodes i and j. Moreover, we use $T_{i,:}$ ($T_{:,i}$) to denote the HSS block row (column) corresponding to node i. As an example, the HSS block rows corresponding to nodes 1 and 3 are $T_{1,:} \equiv (T_{12} \;\; T_{14} \;\; T_{15})$ and

$$T_{3,:} \equiv \begin{pmatrix} T_{14} & T_{15} \\ T_{24} & T_{25} \end{pmatrix},$$

respectively. The HSS construction is done following the traversal of the HSS tree. We demonstrate the first few steps.

(a) *Node* 1. At the beginning, we compress the HSS off-diagonal block row and column corresponding to node 1 by QR factorizations

$$T_{1,:}(\equiv (T_{12} \;\; T_{14} \;\; T_{15})) = U_1(\tilde{T}_{12} \;\; \tilde{T}_{14} \;\; \tilde{T}_{15}),$$

$$T_{:,1}^{\mathrm{T}}(\equiv (T_{21}^{\mathrm{T}} \;\; T_{41}^{\mathrm{T}} \;\; T_{51}^{\mathrm{T}})) = V_1(\tilde{T}_{21}^{\mathrm{T}} \;\; \tilde{T}_{41}^{\mathrm{T}} \;\; \tilde{T}_{51}^{\mathrm{T}}),$$

where $\tilde{T}_{ij}$ denotes a temporary matrix (so does $\bar{T}_{ij}$ below).

(b) *Node* 2. Now, compress the HSS block row and column for node 2. Parts of these blocks have been compressed in the previous step. As U and V matrices are bases of appropriate off-diagonal blocks, the compression of those blocks can be done without these basis matrices. For example, this can be justified by writing

$$T_{2,:} = (\tilde{T}_{21} V_1^{\mathrm{T}} \;\; T_{24} \;\; T_{25}) = (\tilde{T}_{21} \;\; (T_{24} \;\; T_{25})) \begin{pmatrix} V_1^{\mathrm{T}} & \\ & I \end{pmatrix}.$$

Thus, V_1^{T} can be ignored in further compressions. This is essential in saving the cost. Compute QR factorizations

$$(\tilde{T}_{21} \;\; T_{24} \;\; T_{25}) = U_2(B_2 \;\; \tilde{T}_{24} \;\; \tilde{T}_{25}),$$

$$(\tilde{T}_{12}^{\mathrm{T}} \;\; T_{42}^{\mathrm{T}} \;\; T_{52}^{\mathrm{T}}) = V_2(B_1^{\mathrm{T}} \;\; \tilde{T}_{42}^{\mathrm{T}} \;\; \tilde{T}_{52}^{\mathrm{T}}).$$

Then H becomes

$$H = \begin{pmatrix} D_1 & U_1 B_1 V_2^{\mathrm{T}} & U_1 \tilde{T}_{14} & U_1 \tilde{T}_{15} \\ U_2 B_2 V_1^{\mathrm{T}} & D_2 & U_2 \tilde{T}_{24} & U_2 \tilde{T}_{25} \\ \tilde{T}_{41} V_1^{\mathrm{T}} & \tilde{T}_{42} V_2^{\mathrm{T}} & D_4 & T_{45} \\ \tilde{T}_{51} V_1^{\mathrm{T}} & \tilde{T}_{52} V_2^{\mathrm{T}} & T_{54} & D_5 \end{pmatrix}.$$

(c) *Node* 3. The HSS block row and column corresponding to node 3 can be obtained by merging appropriate pieces of the HSS blocks of nodes 1 and 2 (Figure 1). We identify and compress them (ignoring any U, V-bases)

$$\begin{pmatrix} \tilde{T}_{14} & \tilde{T}_{15} \\ \tilde{T}_{24} & \tilde{T}_{25} \end{pmatrix} = \begin{pmatrix} R_1 \\ R_2 \end{pmatrix} (\bar{T}_{34} \;\; \bar{T}_{35}), \quad \begin{pmatrix} \tilde{T}_{41}^{\mathrm{T}} & \tilde{T}_{51}^{\mathrm{T}} \\ \tilde{T}_{42}^{\mathrm{T}} & \tilde{T}_{52}^{\mathrm{T}} \end{pmatrix} = \begin{pmatrix} W_1 \\ W_2 \end{pmatrix} (\bar{T}_{43}^{\mathrm{T}} \;\; \bar{T}_{53}^{\mathrm{T}}).$$

H then has a more compact form.

Similarly, we can continue the compression of the HSS blocks, but with appropriate U, V-bases ignored in the QR factorizations.

3.2. General algorithm

In general, we compress the HSS block row and column corresponding to each tree node i. The major idea of the efficient construction is that we can ignore appropriate U, V-bases so that some portions of the HSS blocks can be replaced by sub-blocks of previously compressed HSS blocks ($\tilde{T}$, $\bar{T}$ blocks above). In the following, c_1 and c_2 denote the children of a node i.

ALGORITHM 1 (*Fast and stable HSS construction*)

```
1. For a matrix $H$ and an HSS tree with $n$ nodes, assign bottom level blocks to the leaves.
2. For nodes $i=1,\dots,n-1$
   (a) If node $i$ is a leaf, locate the appropriate HSS block row $T_{i,:}$ and column $T_{:,i}$ in $H$ with
       any previous $U$, $V$ bases ignored. Compute (rank-revealing) QR factorizations
           $T_{i,:}=U_i\tilde{T}_{i,:}, \quad T_{:,i}^{\mathrm{T}}=V_i\tilde{T}_{:,i}^{\mathrm{T}}.$
       Push $\tilde{T}_{i,:}$ and $\tilde{T}_{:,i}$ onto the stack. The $\tilde{T}$ notation will be replaced by $T$ later.
   (b) Otherwise, pop $T_{c_2,:}$, $T_{:,c_2}$, $T_{c_1,:}$, $T_{:,c_1}$ from the stack.
       (i) Form the HSS block row $T_{i,:}$ based on $T_{c_1,:}$ and $T_{c_2,:}$. See Figure 4. Any $U,V$ bases
           are automatically ignored. Similarly form $T_{:,i}$.
       (ii) Compress $T_{i,:}$ and $T_{:,i}$ and obtain the generators $R_{c_1}$, $R_{c_2}$, $W_{c_1}^{\mathrm{T}}$, $W_{c_2}^{\mathrm{T}}$
           $T_{i,:}=\begin{pmatrix}R_{c_1}\\R_{c_2}\end{pmatrix}\tilde{T}_{i,:}, \quad T_{:,i}^{\mathrm{T}}=\begin{pmatrix}W_{c_1}\\W_{c_2}\end{pmatrix}\tilde{T}_{:,i}^{\mathrm{T}}.$
           Push $\tilde{T}_{i,:}$ and $\tilde{T}_{:,i}$ onto the stack.
       (iii) Identify $B_{c_1}$ and $B_{c_2}$ from $T_{c_1,:}$ and $T_{c_2,:}$, respectively. At step 2(b)i the columns of
           $T_{c_1}$ that do not go to $T_{i,:}$ form $B_{c_1}$. Similarly, form $B_{c_2}$. See Figure 4.
```

This new algorithm is stable at all steps due to the use of orthogonal transformations. If H has a small HSS rank, this new algorithm costs $O(N^2)$ flops but with a hidden constant smaller than that in the construction algorithm in [29]. For example, we consider the cost for constructing the above block 4×4 HSS matrix. For simplicity, assume all $m_i \equiv m = \frac{N}{4}$, the matrix H has HSS rank $r \ll m$, and all matrices to be factorized have ranks r. The main costs are for the QR factorizations of the matrices as listed in Table I. The total cost is about $3rN^2+6r^2N$ flops. On the other hand, the construction algorithm in [29] needs SVDs of eight $m\times 3m$ matrices and several multiplications of matrices with various sizes. The SVDs alone are already much more expensive than our new algorithm.

The reader is also referred to [24] for detailed complexity counts and error analysis of an $\mathscr{H}^2$-matrix construction algorithm, which has a similar hierarchical scheme as our method. Note that in real applications like solving discretized PDEs [15], the HSS forms are constructed by recursive accumulation along the elimination. The construction cost can then be much less than

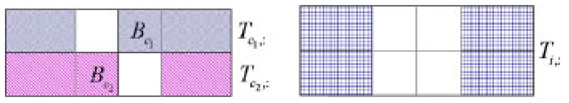

Figure 4. Forming $T_{i,:}$ and identifying B_{c_1} and B_{c_2}.

Table I. Matrices for QR factorizations in the construction of the block 4×4 HSS matrix example.

Matrix size	$m\times 3m$	$r\times(2r+r)$	$2r\times 2m$	$m\times(m+r)$	$m\times 2r$	$2m\times r$
Number of matrices	2	2	2	2	2	2

$O(N^2)$. It is also possible to generalize HSS constructions to higher dimensional problems using the techniques in [24].

4. FAST QUADRATIC COMPLEXITY AND LINEAR COMPLEXITY HSS FACTORIZATIONS

In this section, we discuss explicit factorizations of HSS matrices. For simplicity, we only consider SPD matrices. We look at two types of factorizations: the Cholesky factorization of an SPD HSS matrix and a generalized HSS Cholesky factorization.

4.1. *Fast Cholesky factorization of SPD HSS matrices*

Given the HSS form of an SPD matrix, we can conveniently compute its HSS form Cholesky factor. As the matrix is symmetric, the generators satisfy

$$D_i^{\mathrm{T}}=D_i,\quad U_i=V_i,\quad R_i=W_i,\quad \text{and}\quad B_i=B_j^{\mathrm{T}} \text{ for siblings } i \text{ and } j.$$

Assume H has an HSS tree like Figure 3(i) but with more nodes. The factorization consists of two major operations, eliminating the principal diagonal block and updating the Schur complement. Correspondingly, there are two operations on the HSS tree: removing a node and updating the remaining ones.

First, we look at the situation of eliminating node 1. Factorize $D_1=L_1L_1^{\mathrm{T}}$ and compute the Schur complement $\tilde{H}$ as follows:

$$H=\begin{pmatrix} L_1 & 0\\ l_1 & I\end{pmatrix}\begin{pmatrix} L_1^{\mathrm{T}} & l_1^{\mathrm{T}}\\ 0 & \tilde{H}\end{pmatrix},$$

where

$$l_1^{\mathrm{T}} = (\tilde{U}_1 B_1 U_2^{\mathrm{T}} \;\; \tilde{U}_1 R_1 B_3 R_4^{\mathrm{T}} U_4^{\mathrm{T}} \;\; \tilde{U}_1 R_1 B_3 R_5^{\mathrm{T}} U_5^{\mathrm{T}} \;\; \ldots),$$

$$\tilde{H} = \begin{pmatrix} \tilde{D}_2 & U_2 \tilde{R}_2 B_3 R_4^{\mathrm{T}} U_4^{\mathrm{T}} & U_2 \tilde{R}_2 B_3 R_5^{\mathrm{T}} U_5^{\mathrm{T}} & \ldots \\ U_4 R_4 B_3^{\mathrm{T}} \tilde{R}_2^{\mathrm{T}} U_2^{\mathrm{T}} & \tilde{D}_4 & U_4 \tilde{B}_4 U_5^{\mathrm{T}} & \ldots \\ U_5 R_5 B_3^{\mathrm{T}} \tilde{R}_2^{\mathrm{T}} U_2^{\mathrm{T}} & U_5 \tilde{B}_4^{\mathrm{T}} U_4^{\mathrm{T}} & \tilde{D}_5 & \\ \vdots & \vdots & & \ddots \end{pmatrix},$$

with $\tilde{U}_1 = L_1^{-1} U_1$ and

$$\tilde{D}_2 = D_2 - U_2 B_1^{\mathrm{T}} \tilde{U}_1^{\mathrm{T}} \tilde{U}_1 B_1 U_2^{\mathrm{T}}, \quad \tilde{R}_2 = R_2 - B_1^{\mathrm{T}} \tilde{U}_1^{\mathrm{T}} \tilde{U}_1 R_1$$

$$\tilde{D}_4 = D_4 - U_4 R_4 B_3^{\mathrm{T}} R_1^{\mathrm{T}} \tilde{U}_1^{\mathrm{T}} \tilde{U}_1 R_1 B_3 R_4^{\mathrm{T}} U_4^{\mathrm{T}}, \quad \tilde{B}_4 = B_4 - R_4 B_3^{\mathrm{T}} R_1^{\mathrm{T}} \tilde{U}_1^{\mathrm{T}} \tilde{U}_1 R_1 B_3 R_5^{\mathrm{T}}, \ldots$$

We can see that the Schur complement $\tilde{H}$ takes a form similar to the original matrix but with its first block row and column removed.

In general, the update of the generators can be clearly seen by using appropriate paths in the tree. Following the postordering of the nodes $i = 1, \ldots, n$, we can perform two steps to remove each node i during the elimination. At the first step, eliminate node i by computing

$$D_i = L_i L_i^{\mathrm{T}}, \quad \tilde{D}_i = L_i, \quad \tilde{U}_i = L_i^{-1} U_i.$$

At the second step, compute the Schur complement by updating the remaining nodes. This means, we consider each node $j = i+1, \ldots, n$ according to the rules below.

1. If node j is a leaf node, locate the path connecting j and i: $j \to \cdots \to i \to \cdots \to j$, and update D_j as
$$\tilde{D}_j = D_j - U_j R_j \ldots R_i^{\mathrm{T}} \tilde{U}_i^{\mathrm{T}} \tilde{U}_i R_i \ldots R_j^{\mathrm{T}} U_j^{\mathrm{T}}.$$
2. If node j is a left child, locate the path connecting j to i and then to s, the sibling of j: $j \to \cdots \to i \to \ldots \to s$, and update B_j as
$$\tilde{B}_j = B_j - R_j \ldots R_i^{\mathrm{T}} \tilde{U}_i^{\mathrm{T}} \tilde{U}_i R_i \ldots R_s^{\mathrm{T}}.$$
3. If node j is the right child of p which is an ascendant of i, locate the path connecting j to i and then to s, the sibling of j: $j \to \cdots \to i \to \cdots \to s$, and update R_j as
$$\tilde{R}_j = R_j - B_s^{\mathrm{T}} \ldots R_i^{\mathrm{T}} \tilde{U}_i^{\mathrm{T}} \tilde{U}_i R_i \ldots R_s.$$

Nodes of the HSS tree are removed along the progress of the elimination. This algorithm gives an explicit HSS form of the Cholesky factor. The idea is also useful for finding Schur complements in partial HSS factorizations [15]. This algorithm costs $O(N^2)$ flops where N is the dimension of H. As our main concern is the linear complexity factorization below, we skip the detailed flop count.

4.2. *Linear complexity generalized Cholesky factorization of HSS matrices*

There exist $O(N)$ complexity algorithms for solving a compact HSS system [29]. The HSS solver in [29] computes an implicit *ULV* factorization. However, sometimes an explicit factorization of an HSS matrix may be convenient. Furthermore, various simplifications and improvements can be made. Here, we provide an improved linear complexity factorization scheme. As our algorithm computes an explicit *ULV* factorization instead of the traditional Cholesky factorization, we call it a generalized HSS Cholesky factorization. In the following, we factorize a compact SPD HSS matrix H such as the one represented by the HSS tree in Figure 3(i).

4.2.1. *Introducing zeros into off-diagonal blocks.*

We consider to partially eliminate node i in the HSS tree. The generator U_i is a basis matrix for the i-th off-diagonal block row. It is directly available for any leaf node i, and can be formed recursively for a non-leaf node. By bringing zeros into U_i, we can quickly introduce zeros into the HSS blocks corresponding to i.

Assume that U_i has size $m_i \times k_i$. In a compact HSS form, we should have $m_i \geqslant k_i$. Here, we leave the case $m_i = k_i$ to Subsection 4.2.3 and assume $m_i > k_i$. In such a situation, we can introduce a QL factorization with an orthogonal transformation Q_i such that

$$U_i = Q_i \begin{pmatrix} 0 \\ \tilde{U}_i \end{pmatrix}, \quad \tilde{U}_i : k_i \times k_i. \tag{5}$$

Multiply Q_i^{T} to the entire block row i. Then the first $m_i - k_i$ rows of the off-diagonal block become zeros. See Figure 5 for a pictorial representation. Similarly, we apply Q_i on the right to the off-diagonal column corresponding to node i. As the HSS form is symmetric, this will also introduce $m_i - k_i$ zero columns in the i-th off-diagonal block column. The diagonal block is now changed to

$$\hat{D}_i = Q_i^{\mathrm{T}} D_i Q_i. \tag{6}$$

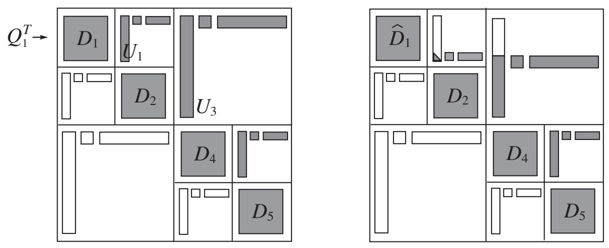

Figure 5. A pictorial representation for introducing zeros into the off-diagonal block rows of an HSS matrix. Dark blocks represent the nonzero portions of the generators in the block upper triangular part. The nonzero pattern for the block lower triangular part comes from symmetry and is not shown.

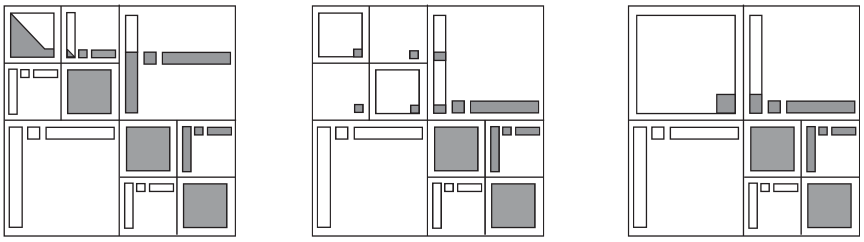

Figure 6. Partial factorization of a diagonal block and merging generators of siblings: (i) partial factorization; (ii) after partial elimination; and (iii) merging blocks.

4.2.2. Partially factorizing diagonal blocks. We partition the diagonal block $\hat{D}_i$ conformally and partially factorize it

$$\hat{D}_i = \begin{array}{c c} & \begin{array}{cc} m_i-k_i & k_i \end{array} \\ \begin{array}{c} m_i-k_i \\ k_i \end{array} & \begin{pmatrix} D_{i;1,1} & D_{i;1,2} \\ D_{i;2,1} & D_{i;2,2} \end{pmatrix} \end{array} = \begin{pmatrix} L_i & 0 \\ D_{i;2,1}L_i^{-\mathrm{T}} & I \end{pmatrix} \begin{pmatrix} L_i^{\mathrm{T}} & L_i^{-1}D_{i;1,2} \\ 0 & \tilde{D}_i \end{pmatrix}, \tag{7}$$

where $\tilde{D}_i$ is the Schur complement

$$\tilde{D}_i = D_{i;2,2} - D_{i;2,1}L_i^{-\mathrm{T}}L_i^{-1}D_{i;1,2}. \tag{8}$$

See Figure 6(i). We see that the block $D_{i;1,1}$ can then be eliminated (Figure 6(ii)).

4.2.3. Merging child blocks. At this point, $\tilde{U}_i$ is a square matrix (corresponding to the situation $m_i = k_i$ mentioned in Subsection 4.2.1). We then merge sibling blocks and move to the parent node (instead of doing level-wise elimination as in [29]). For notational convenience, we use i to mean a parent node, with its children c_1 and c_2 partially eliminated as above. Then, we can merge blocks to obtain upper level generators

$$D_i = \begin{pmatrix} \tilde{D}_{c_1} & \tilde{U}_{c_1}B_{c_1}\tilde{U}_{c_2}^{\mathrm{T}} \\ \tilde{U}_{c_2}B_{c_1}^{\mathrm{T}}\tilde{U}_{c_1}^{\mathrm{T}} & \tilde{D}_{c_2} \end{pmatrix}, \quad U_i = \begin{pmatrix} \tilde{U}_{c_1}R_{c_1} \\ \tilde{U}_{c_2}R_{c_2} \end{pmatrix}. \tag{9}$$

We emphasize that these D_i and U_i are the generators of the reduced HSS matrix, instead of the original H.

Now, we can remove nodes c_1 and c_2 from the HSS tree. We repeat these steps following the postordering of the tree to eliminate other nodes, until we reach the root n where we can factorize its reduced-size D_n generator directly. Note that a node only communicates with its parent (and children). This is essential for the linear complexity of the factorization and is useful for parallelization.

4.2.4. Algorithm and complexity. We organize the steps in the following algorithm.

ALGORITHM 2 (*Generalized HSS Cholesky factorization*)

```
1. Let $H$ be an SPD HSS matrix with $n$ nodes in the HSS tree. Allocate space for a stack.
2. For each node $i=1,2,\dots,n-1$
   (a) If $i$ is a non-leaf node
       (i) Pop $\tilde{D}_{c_2}$, $\tilde{U}_{c_2}$, $\tilde{D}_{c_1}$, $\tilde{U}_{c_1}$ from the stack, where $c_1, c_2$ are the children of $i$.
       (ii) Obtain $D_i$ and $U_i$ with (9).
   (b) Compress $U_i$ with (5). Push $\tilde{U}_i$ onto the stack.
   (c) Update $D_i$ with (6). Factorize $\hat{D}_i$ with (7) and obtain the Schur complement $\tilde{D}_i$ as in
       (8). Push $\tilde{D}_i$ onto the stack.
3. For the root node $n$, form $D_n$ and compute the Cholesky factorization $D_n = L_n L_n^{\mathrm{T}}$.
```

Remark 1
For each step, we can also replace the compression step (5) and the partial Cholesky factorization step (7) by a different process. That is, we first compute a full Cholesky factorization $D_i = L_i L_i^{\mathrm{T}}$ and then QR factorize $L_i^{-1} U_i$. This way, D_i is updated to an identity matrix and remains so that step (6) is avoided, which can save some work.

Note that the factors after the generalized Cholesky factorization include lower triangular matrices L_i, orthogonal transformations Q_i in the compressions, and applicable permutations P_i during the merge step. They together form the generalized HSS Cholesky factor. To clearly see the roles of these matrices in the actual factorization, we look at an example with three nodes in the tree. The off-diagonal compression and the partial diagonal factorization leads to

$$H = \begin{pmatrix} Q_1 & 0 \\ 0 & Q_2 \end{pmatrix} \hat{L}_3 \begin{pmatrix} \begin{pmatrix} I & 0 \\ 0 & \tilde{D}_1 \end{pmatrix} & \begin{pmatrix} 0 & 0 \\ 0 & \tilde{U}_1 B_1 \tilde{U}_2^{\mathrm{T}} \end{pmatrix} \\ \begin{pmatrix} 0 & 0 \\ 0 & \tilde{U}_2 B_1^{\mathrm{T}} \tilde{U}_1^{\mathrm{T}} \end{pmatrix} & \begin{pmatrix} I & 0 \\ 0 & \tilde{D}_2 \end{pmatrix} \end{pmatrix} \hat{L}_3^{\mathrm{T}} \begin{pmatrix} Q_1^{\mathrm{T}} & 0 \\ 0 & Q_2^{\mathrm{T}} \end{pmatrix},$$

where the notation I for identity matrices and 0 for zero matrices may have different sizes, and

$$\hat{L}_3 = \mathrm{diag}\left(\begin{pmatrix} L_1 & 0 \\ T_1 & I \end{pmatrix}, \begin{pmatrix} L_2 & 0 \\ T_2 & I \end{pmatrix} \right) \quad \text{with } T_1 = D_{1;2,1} L_1^{-\mathrm{T}},\ T_2 = D_{2;2,1} L_2^{-\mathrm{T}}. \tag{10}$$

The merge process then uses a permutation matrix P_3 to bring together appropriate dense blocks to form D_3 as shown in (9) (there is no U_3 as there are only two bottom-level blocks here). Then another factorization step $D_3 = L_3 L_3^{\mathrm{T}}$ follows. Thus,

$$H = L_H L_H^{\mathrm{T}} \quad \text{with } L_H = Q_3 \hat{L}_3 P_3 \begin{pmatrix} I & 0 \\ 0 & L_3 \end{pmatrix}. \tag{11}$$

The matrix L_H is the actual generalized HSS Cholesky factor formed by $\{L_i\}$, $\{T_i\}$, $\{Q_i\}$, and $\{P_i\}$. This procedure is recursive and we can easily generalize the example. In addition, Q_i and P_i can be stored in terms of (Householder) vectors and scalars (matrix dimensions), respectively.

Table II. Flop counts of some basic matrix operations, with low-order terms dropped.

Operation	Flops
Cholesky factorization of an $n \times n$ matrix	$n^3/3$
Inverse of an $n \times n$ lower triangular matrix times an $n \times k$ matrix	$n^2 k$
QR factorization of an $m \times k$ tall matrix $(m > k)$	$2k^2(m - \frac{k}{3})$
Product of Q and an $m \times n$ vector	$2nk(2m-k)$
Product of a general $m \times n$ matrix and an $n \times k$ matrix	$2mnk$

Table III. Sizes of the generators in Algorithm 2, where i and j are siblings.

Generator	U_i	R_i	B_i
Size	$m_i \times k_i$	$k_i \times k_p$	$k_i \times k_j$

The algorithm has linear complexity as shown in the following theorem.

Theorem 4.1
Assume that an $N \times N$ SPD matrix H is in compact HSS form with a full HSS tree. Moreover, assume that the bottom-level HSS block row dimensions are $O(r)$, where r is the HSS rank of H. Then the generalized Cholesky factorization of H with Algorithm 2 has complexity $O(r^2 N)$ flops.

Proof
To show the complexity, we use the flop counts of some basic matrix operations as listed in Table II. They can be found, say, in [39, 40].

Consider node i of the HSS tree corresponding to each step i of Algorithm 2. Assume node i (except the root) has a sibling j, a parent p, and two children c_1 and c_2 if applicable. We further assume the U_i basis being compressed in (5) has dimension $m_i \times k_i$. Note that for a non-leaf i, the matrices D_i and U_i are generators of a reduced HSS matrix after an intermediate merge process (9), and is not a generator of the original H. This means, each U_i in the compression step (5) has row dimension $m_i = O(r)$. Also, let R_i and B_i have dimensions as indicated in Table III. Clearly, all $k_i = O(r)$.

The major operations in the factorization are as follows.

- For a non-leaf i, the merge step (9) needs four matrix-matrix products $\tilde{U}_{c_1} B_{c_1}$, $(\tilde{U}_{c_1} B_{c_1})\tilde{U}_{c_2}^{\mathrm{T}}$, $\tilde{U}_{c_1} R_{c_1}$, and $\tilde{U}_{c_2} R_{c_2}$, with costs $2m_{c_1}k_{c_1}k_{c_2}$, $2m_{c_1}k_{c_2}m_{c_2}$, $2m_{c_1}k_{c_1}k_i$, and $2m_{c_2}k_{c_2}k_i$, respectively.
- For each i, the compression (QR) step (5) costs $2k_i^2(m_i - \frac{k_i}{3})$.
- For each i, the diagonal update (6) costs $4m_i k_i(2m_i - k_i)$ due to two products involving Q_i.
- For each i, the partial factorization in (7) costs $\frac{1}{3}(m_i - k_i)^3$, $(m_i - k_i)^2 k_i$, and $(m_i - k_i)k_i^2$, which are for factorizing the pivot block, updating the lower triangular part, and computing the Schur complement, respectively.

These counts are summarized in the third column of Table IV.

To simplify the calculations, we assume each bottom-level U_i has the same row dimension m. The counts are then simplified as in the third column of Table IV. The HSS tree has N/m leaf

Table IV. Cost of generalized HSS Cholesky factorization (leading terms only).

Node	Operation	Flops		Number of nodes
Leaf node i	Compression (5)	$2k_i^2(m_i-\frac{k_i}{3})$	$=O(mr^2)$	$\times N/m$
	Diagonal update (6)	$4m_ik_i(2m_i-k_i)$	$=O(m^2r)$	
	Factorization (7)	$\frac{1}{3}(m_i-k_i)^3+(m_i-k_i)^2k_i+(m_i-k_i)k_i^2$	$=O((m-r)^3)$	
Non-leaf node i	Merge step (9)	$2(m_{c_1}k_{c_1}k_{c_2}+m_{c_1}k_{c_2}m_{c_2}+m_{c_1}k_{c_1}k_i+m_{c_2}k_{c_2}k_i)$	$=O(r^3)$	$\times(N/m-1)$
	Diagonal update (6)	$4m_ik_i(2m_i-k_i)$	$=O(r^3)$	
	Compression (5)	$2k_i^2(m_i-\frac{k_i}{3})$	$=O(r^3)$	
	Factorization (7)	$\frac{1}{3}(m_i-k_i)^3+(m_i-k_i)^2k_i+(m_i-k_i)k_i^2$	$=O(r^3)$	

Table V. Computation time (in seconds) of the generalized HSS Cholesky factorization and system solution (denoted `NEW`), when compared with `DPOTRF`, where $\times$ means insufficient memory.

Matrix size		256	512	1024	2048	4096	8192	16 384
	`DPOTRF`	0.074	0.765	11.339	105.068	845.855	6857.316	$\times$
`NEW`	Factorization	0.068	0.076	0.104	0.172	0.280	0.520	0.953
	Solution	0.003	0.006	0.013	0.029	0.054	0.109	0.227
Matrix size		32 768	65 536	131 072	262 144	524 288	1 048 576	
	`DPOTRF`	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	
`NEW`	Factorization	1.855	3.773	7.453	14.914	32.797	59.547	
	Solution	0.457	0.871	1.746	3.566	7.211	13.875	

nodes, and $N/m-1$ non-leaf nodes. Therefore, the total cost is

$$[O(mr^2)+O(m^2r)+O((m-r)^3)]\times\frac{N}{m}+O(r^3)\times\frac{N}{m}=O(r^2N),$$

as $m=O(r)$. □

Numerical experiments for the algorithm are displayed in Table V of Subsection 4.4 together with system solution results.

4.3. HSS linear system solver with the generalized HSS Cholesky factor

After we compute generalized Cholesky HSS factorization, we can solve HSS systems with substitutions. Assume we solve the system $Hx=b$, where $H=L_HL_H^{\mathrm{T}}$ and the generalized Cholesky factor L_H is formed by $\{L_i\}$, $\{T_i\}$, $\{Q_i\}$, $\{P_i\}$ as obtained in Algorithm 2. Just like the traditional triangular system solution, our new HSS solver also has two stages, forward substitution and backward substitution, for the following two systems, respectively

$$L_Hy=b, \tag{12}$$

$$L_H^{\mathrm{T}}x=y. \tag{13}$$

Here, the substitutions are done along the HSS tree, following the postordering (or bottom-up, forward) and reverse-postordering (or top-down, backward) traversals, respectively.

4.3.1. Forward substitution. We solve (12) first. If we have, say, an explicit expression like (11), then we can explicitly write

$$y=\begin{pmatrix} I & 0 \\ 0 & L_3^{-1} \end{pmatrix} P_3^{\mathrm{T}} \hat{L}_3^{-1} \begin{pmatrix} Q_1^{\mathrm{T}} & 0 \\ 0 & Q_2^{\mathrm{T}} \end{pmatrix} b \tag{14}$$

which involves matrix–vector multiplications and standard triangular system solution. But in general, we do this implicitly with the HSS factorization tree whose structure has good data locality. The solution vector y is obtained in the following way.

We use the space of b for y. First, partition b conformally according to the bottom-level block sizes or the partition sequence $\{m_i\}$. Denote the vector pieces by $\{y_i\}$. Associate y_i with each leaf i.

Next, apply Q_i^{T} to y_i (see (14))

$$\hat{y}_i = Q_i^{\mathrm{T}} y_i = \begin{array}{c} m_i - r_i \\ r_i \end{array} \begin{pmatrix} \hat{y}_{i;1} \\ \hat{y}_{i;2} \end{pmatrix},$$

where $\hat{y}_i$ is partitioned according to (5) and (7). Then, we solve for

$$\tilde{y}_i = \begin{pmatrix} L_i & 0 \\ T_i & I \end{pmatrix}^{-1} \hat{y}_i = \begin{pmatrix} \tilde{y}_{i;1} \\ \hat{y}_{i;2} - T_i \tilde{y}_{i;1} \end{pmatrix} \equiv \begin{array}{c} m_i - r_i \\ r_i \end{array} \begin{pmatrix} \tilde{y}_{i;1} \\ \tilde{y}_{i;2} \end{pmatrix}, \tag{15}$$

where $\tilde{y}_{i;1} = L_i^{-1} \hat{y}_{i;1}$. The vector piece y_i is now replaced by $\tilde{y}_{i;1}$, and $\tilde{y}_{i;2}$ is passed to the parent node p of i. That is, $\tilde{y}_{i;2}$ is the contribution from i to p. Thus, if i and j are the left and right children of p, respectively, then essentially

$$y_p = \begin{pmatrix} \tilde{y}_{i;2} \\ \tilde{y}_{j;2} \end{pmatrix}.$$

The formation of y_p eventually finishes the operation

$$P_p^{\mathrm{T}} \begin{pmatrix} \tilde{y}_i \\ \tilde{y}_j \end{pmatrix}$$

(see (14)). Note that no extra storage is necessary for y_p as it can use the original storage of its child solution vector pieces y_i and y_j, except that pointers are used for the locations.

We repeat this procedure along the postordering HSS factorization tree, until finally, for the root node n we are ready to apply L_n^{-1} to the generated y_n. All solution vector pieces are stored in the space of b, and at the end of the procedure b is transformed into y.

4.3.2. Backward substitution. In this stage, we solve (12). Associate with each node of the HSS factorization tree a solution piece x_i which is initially y_i. For the root n, we first get

$$x_n = L_n^{-\mathrm{T}} y_n \equiv \begin{array}{c} m_{c_1}-r_{c_1} \\[6pt] m_{c_2}-r_{c_2} \end{array} \begin{pmatrix} \hat{x}_{c_1} \\[6pt] \hat{x}_{c_2} \end{pmatrix},$$

where the partition essentially applies the permutation P_n. Next for each node $i<n$, compute

$$\tilde{x}_i = \begin{pmatrix} L_i & 0 \\ T_i & I \end{pmatrix}^{-\mathrm{T}} \begin{pmatrix} y_i \\ \hat{x}_i \end{pmatrix} = \begin{pmatrix} L_i^{-\mathrm{T}}(y_i - T_i^{\mathrm{T}} \hat{x}_i) \\ \hat{x}_i \end{pmatrix}, \tag{16}$$

where $\hat{x}_i$ is inherited from p, the parent of i. Now set $x_i = Q_i \tilde{x}_i$. The formation of x_i is then completed. Partition x_i according to the children $\hat{c}_1$ and $\hat{c}_2$ of i as

$$x_i = \begin{array}{c} m_{\hat{c}_1}-r_{\hat{c}_1} \\[6pt] m_{\hat{c}_2}-r_{\hat{c}_2} \end{array} \begin{pmatrix} \hat{x}_{\hat{c}_1} \\[6pt] \hat{x}_{\hat{c}_2} \end{pmatrix}.$$

Then, the procedure repeats along the reverse postordering of the HSS factorization tree.

After the backward substitution, the vector y is transformed into the solution x. That is, by using b as the solution storage, it automatically becomes x after the two substitutions. If H is a compact $N \times N$ HSS matrix with HSS rank r, it is easy to verify that the cost of the above solver is $O(rN)$. Therefore, the overall complexity for solving $Hx=b$ is linear in N.

4.4. Performance of the generalized Cholesky factorization and system solution

We implement the generalized HSS Cholesky factorization and solution algorithms in Fortran 90 and test them on some nearly random SPD HSS matrices with sizes from 256 to 1,048,576. These matrices have small HSS ranks r, and the bottom HSS block rows have the same row dimension m. For convenience, we choose $m \equiv 2r$ so that the factorization associated with each node starts with a compression step instead of merging. Results for $m=16$ are reported. We ran the code on a Sun UltraSPARC-II 248 Mhz server. The CPU timing is shown in Table V. We also include the timing for the standard Cholesky factorization routine, `DPOTRF` from LAPACK [41], applied to the original dense matrices. The results are consistent with the complexity. The HSS algorithm is also memory efficient. The generalized HSS Cholesky factors are then used to solve linear systems. The timing is also included in Table V.

Now, we consider the stability of the overall procedure for solving SPD HSS systems using the generalized HSS factorization and solution. This overall procedure has similar stability as the solver in [29], that is, it is stable when $\|R_i\|<1$ for a submultiplicative norm. We can verify that the construction algorithm in Section 3 provides HSS matrices satisfying this condition for the 2-norm. The claimed stability is due to the use of orthogonal and triangular transformations. For some test matrices in Table V, we report the experimental backward errors in Table VI, which indicate the backward stability of the overall procedure.

4.5. Comparison with the implicit ULV solver in [*29*]

Both the generalized HSS factorization Algorithm 2 and the implicit *ULV* algorithm in [29] (denoted `Implicit` *ULV*) are based on partial factorizations and off-diagonal compressions.

Table VI. Backward errors of the generalized HSS solver corresponding to Table V.

Matrix size	256	512	1024	2048	4096
$\dfrac{\|Hx-b\|_1}{\varepsilon_{\text{mach}}(\|H\|_1\|x\|_1+\|b\|_1)}$	0.38	0.47	0.39	0.53	0.62

Table VII. Computational costs (flops$/N$) of the generalized HSS Cholesky factorization and system solution (denoted `NEW`), when compared with the implicit *ULV* solver in [29] (denoted `Implicit` *ULV*), where N is the order of each HSS matrix $\tilde{F}$.

N			560	1023	1477	2777	4249	6143	10 038
$\dfrac{\text{flops}}{N}$	`Implicit` *ULV*		1.37E5	1.48E5	1.53E5	1.57E5	1.59E5	1.60E5	1.61E5
	`NEW`	Factorization	4.41E4	4.73E4	4.88E4	4.99E4	5.04E4	5.07E4	5.09E4
		Solution	1.24E3	1.32E3	1.35E3	1.37E3	1.38E3	1.39E3	1.39E3

Although these two algorithms both have linear complexity for compact HSS matrices, there are some major differences between them.

1. `Implicit` ULV uses level-wise traversal of the HSS tree. On the other hand, Algorithm 2 uses local traversal of the HSS tree following the postordering, and all the operations are done with local communications between nodes and their parents (see the loop in Algorithm 2). The new algorithm thus has better data locality, and is easier to implement and analyze in general.
2. `Implicit` *ULV* is designed to work on general (nonsymmetric) matrices and does not specifically preserve symmetry for symmetric problems. In terms of the detailed total flop counts, `Implicit` *ULV* costs about $46r^2N$ flops, with an assumption $m=2r$ [29]. Under the same assumption, Algorithm 2 with the idea in Remark 1 costs no more than $20r^2N$ flops.
3. In system solutions, there is also a major difference in updating the right-hand side vector b after the partial factorization of a diagonal block. When the generalized HSS Cholesky factor is used in system solution, the update of b is only done through local communication between a node and its parent. See (15) and (16). On the other hand, `Implicit` *ULV* uses an HSS matrix–vector multiplication algorithm to update the right-hand side. This could involve all the rest uneliminated variables. This multiplication algorithm needs to be carefully implemented to reuse information at different levels. Otherwise, the total complexity can be more than $O(N)$. See [29] for more details.
4. Moreover, for systems with many right-hand sides, the new explicit algorithm is even more efficient, because the solution stage is usually much faster than the one-time factorization stage. Thus, when used as preconditioners, the new algorithm is especially more efficient than `Implicit` *ULV*.

A numerical comparison of the two algorithms is given in Table VII of the next section in terms of a practical application.

5. APPLICATIONS

The algorithms developed in this work can be applied to dense matrices in more complicated problems and to quickly obtain approximate solutions or to precondition difficult problems. As an example, the generalized HSS Cholesky algorithm is used to factorize dense Schur complements in the direct solution of some sparse discretized PDEs [15]. The main idea of the solver in [15] is as follows.

First, it has been noticed that during the direct factorization of some discretized PDEs like elliptic equations, the fill-in has the low-rank property [12–14, 34, 35]. Thus, we can approximate the intermediate dense matrices by HSS matrices [15].

Second, this low-rank property can be revealed by organizing the factorization carefully. The discretized matrix is first ordered to reduce fill-in, which corresponds to the reordering of the mesh points in the discretization. Nested dissection [42] is used in [15], and the mesh points are recursively put into different levels of separators. These separators are then eliminated bottom up. The elimination is conducted with a supernodal version of the multifrontal method [43–45]. All separators are ordered following an assembly tree. The multifrontal method reorganizes the overall sparse elimination into partial factorizations of many smaller dense intermediate matrices called *frontal matrices*. The Schur complement from the partial factorization of a frontal matrix is called an *update matrix*. Following the traversal of the assembly tree, lower level update matrices are assembled (called *extend-add*) into the parent frontal matrix in the tree.

Thus, when the multifrontal method is used to solve those discretized PDEs with the low-rank property, the intermediate frontal and update matrices can be approximated by HSS matrices. At certain elimination level, simple HSS approximations are constructed with Algorithm 1. At later elimination levels, the HSS form frontal and update matrices are obtained with recursive accumulation. The partial factorizations and extend-add operations are then done in HSS forms. This leads to a structured approximate multifrontal method, as illustrated in Figure 7. Such a structured multifrontal method reduces the factorization cost from $O(n^{3/2})$ to nearly $O(rn)$, and the storage from $O(n\log n)$ to $O(n\log r)$, where r is the maximum of all applicable HSS ranks, and n is size of the sparse matrix.

For problems such as elliptic equations and linear elasticity equations in 2D domains, the low-rank property has been observed and leads to efficient direct-structured factorization [15].

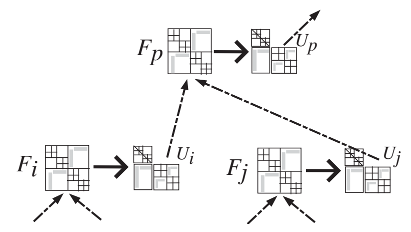

Figure 7. Illustration of the structured multifrontal method in [15], where F and U represent HSS form frontal and update matrices, respectively, a solid dark arrow represents the partial factorization, and a broken arrow represents the pass of child contributions to the parent along the assembly tree.

Table VIII. Preconditioning a system $Fx = b$ with $\tilde{F}$ from Table VII and $N = 6143$, where a relative residual accuracy 10^{-16} is used, and the storage means the number of double precision nonzero entries.

	Dense F		With $\tilde{F}$ as the preconditioner	
Storage	Storage of F	3.77E7	Storage of $\tilde{F}$	2.12E6
	Cost of F times a vector	7.55E7	Cost of preconditioning by $\tilde{F}$	8.52E6
Solving $Fx = b$	Number of CG iterations	1375	Number of PCG iterations	96
	Total CG cost	1.04E11	Total PCG cost	8.07E9
Condition number	$\kappa_2(F)$	7.89E8	$\kappa_2(\tilde{F}^{-1}F)$	1.70E2

Here, we do not show the details of factorizing those 2D examples, and instead, we consider the preconditioning of an ill-conditioned 3D problem with a large jump in the coefficient [46]

$$\begin{gathered} -\nabla\cdot(a\nabla u) = f \text{ in } \Omega = (-1,1)^3 \equiv (-1,1)\times(-1,1)\times(-1,1), \\ u = g_D \text{ on } \Gamma_D, \quad u = g_N \quad \text{on } \Gamma_N, \\ a(x) = 1 \text{ if } x \in (-0.5,0)^3 \text{ or } (0,0.5)^3, \quad a(x) = \varepsilon \text{ otherwise,} \end{gathered} \tag{17}$$

where Γ_D and Γ_N are Dirichlet and Neumann boundaries, respectively, and appropriate boundary conditions are defined as in [46]. When the parameter ε is small, classical iterative methods including multigrid deteriorate quickly. Here we use $\varepsilon = 10^{-7}$. The problem is discretized by an adaptive finite element method in the package *i*FEM [47].

As the current paper focuses on dense HSS operations, we demonstrate the performance of the generalized HSS Cholesky factorization and solution algorithms in preconditioning intermediate matrices in the multifrontal method for solving (17). Each exact frontal matrix F corresponding to the last separator in nested dissection is calculated and is used as our model matrix. F is dense and does not necessarily have significant low-rank property. Thus, we construct an HSS approximation matrix $\tilde{F}$ by manually setting an upper bound r for the numerical ranks during the off-diagonal compression. In the following tests, r is about 50, and so are the bottom-level HSS block row dimensions. Then, HSS factorizations and solutions are tested on these HSS matrices $\tilde{F}$.

Both the implicit ULV solver in [29] and the generalized HSS factorization/solution are used to solve the HSS systems. See Table VII for the computational costs. Flop counts instead of timing are used when only a Matlab code for the implicit *ULV* solver in [29] is available. The Matlab timing results compare similarly. On the other hand, the Fortran timing of our new algorithms is illustrated for another example in Table V. The flop counts are roughly certain constants times N. Furthermore, the generalized HSS factorization/solution algorithms are more efficient than the implicit *ULV* solver.

We then consider the effectiveness of precondition F with $\tilde{F}$. For $N = 6143$, Table VIII lists some statistics of using $\tilde{F}$ (in its factorized form) as a preconditioner in the conjugate gradient method (CG) for solving $Fx = b$. The preconditioned matrix has significantly better condition.

This example is used to illustrate the potential of our HSS algorithms for difficult problems. In future work, we expect to conduct more comprehensive comparisons with other advanced solvers, and to exploit the possibility of direct solutions of complicated 3D problems with HSS techniques. Moreover, the reader is referred to [15] for more details on the structured multifrontal method using HSS matrices.

6. CONCLUDING REMARKS

We present generalizations of HSS representations and some new HSS algorithms in this work. Existing work involving HSS matrices in [1, 2, 29, 30] can potentially benefit from the simplified HSS representation. Furthermore, the new HSS generation and matrix factorization algorithms have better performance than existing ones. These new algorithms have been used in a nearly linear complexity direct solver for sparse-discretized PDEs [15]. Together with additional HSS algorithms in [2, 15], an entire set of HSS operations can be defined. These HSS operations are useful for many other problems where semiseparable structures can be used or the low-rank property exists. In addition, existing rank-structured methods for problems such as Toeplitz systems [3] and companion eigenproblems [7] can be possibly improved by using this work to obtain better performance and scalability.

ACKNOWLEDGEMENTS

The authors are very grateful to the anonymous referees for their valuable comments. The authors would also like to thank Long Chen for some test codes and data. The work of Ming Gu was supported in part by NSF grants CCF-0515034 and CCF-0830764.

REFERENCES

1. Chandrasekaran S, Gu M, Lyons W. A fast adaptive solver for hierarchically semiseparable representations. *CALCOLO* 2005; **42**:171–185.
2. Chandrasekaran S, Gu M, Pals T. Fast and stable algorithms for hierarchically semi-separable representations. *Technical Report*, Department of Mathematics, University of California, Berkeley, 2004.
3. Chandrasekaran S, Gu M, Sun X, Xia J, Zhu J. A superfast algorithm for Toeplitz systems of linear equations. *SIAM Journal on Matrix Analysis and Applications* 2007; **29**:1247–1266.
4. Delvaux S, Van Barel M. A QR-based solver for rank structured matrices. *SIAM Journal on Matrix Analysis and Applications* 2008; **30**:464–490.
5. Chandrasekaran S, Dewilde P, Gu M, Pals T, Sun X, van der Veen AJ, White D. Fast stable solvers for sequentially semi-separable linear systems of equations and least squares problems. *Technical Report*, University of California, Berkeley, CA, 2003.
6. Bini DA, Gemignani L, Pan VY. Fast and stable QR eigenvalue algorithms for generalized companion matrices and secular equations. *Numerische Mathematik* 2005; **100**:373–408.
7. Chandrasekaran S, Gu M, Xia J, Zhu J. A fast QR algorithm for companion matrices. *Operator Theory*: *Advances and Applications*, vol. 179. Birkhäuser: Basel, 2007; 111–143.
8. Eidelman Y, Gohberg I, Olshevsky V. The QR iteration method for Hermitian quasiseparable matrices of an arbitrary order. *Linear Algebra and its Applications* 2005; **404**:305–324.
9. Laub AJ, Xia J. Statistical condition estimation for the roots of polynomials. *SIAM Journal on Scientific Computing* 2008; **31**:624–643.
10. Laub AJ, Xia J. Fast condition estimation for a class of structured eigenvalue problems. *SIAM Journal on Matrix Analysis and Applications* 2009; **30**:1658–1676.
11. Van Barel M, Vandebril R, Mastronardi N, Delvaux S, Vanberghen Y. Rank structured matrix operations. *Workshop on State-of-the-Art in Scientific and Parallel Computing*, *PARA06*, Umea, Sweden, June 2006.
12. Bebendorf M, Hackbusch W. Existence of $\mathscr{H}$-matrix approximants to the inverse FE-matrix of elliptic operators with L^∞-Coefficients. *Numerische Mathematik* 2003; **95**:1–28.
13. Grasedyck L, Kriemann R, Le Borne S. Parallel black box domain decomposition based H-LU preconditioning. *Technical Report 115*, Max Planck Institute for Mathematics in the Sciences, Leipzig, 2005.
14. Grasedyck L, Kriemann R, Le Borne S. Domain-decomposition based H-LU preconditioners. In *Domain Decomposition Methods in Science and Engineering XVI*, Widlund OB, Keyes DE (eds). Lecture Notes in Computational Science and Engineering, vol. 55. Springer: Berlin, 2006; 661–668.

15. Xia J, Chandrasekaran S, Gu M, Li XS. Superfast multifrontal method for structured linear systems of equations. *SIAM Journal on Matrix Analysis and Applications* 2009; **31**:1382–1411.
16. Gohberg I, Kailath T, Koltracht I. Linear complexity algorithms for semiseparable matrices. *Integral Equations and Operator Theory* 1985; **8**:780–804.
17. Martinsson PG, Rokhlin V. A fast direct solver for boundary integral equations in two dimensions. *Journal of Computational Physics* 2005; **205**:1–23.
18. Rokhlin V. Rapid solution of integral equations of scattering theory in two dimensions. *Journal of Computational Physics* 1990; **86**:414–439.
19. Hackbusch W. A Sparse matrix arithmetic based on $\mathscr{H}$-matrices. Part I: introduction to $\mathscr{H}$-matrices. *Computing* 1999; **62**:89–108.
20. Hackbusch W, Grasedyck L, Börm S. An introduction to hierarchical matrices. *Mathematica Bohemica* 2002; **127**:229–241.
21. Hackbusch W, Khoromskij BN. A sparse $\mathscr{H}$-matrix arithmetic. Part-II: Application to multi-dimensional problems. *Computing* 2000; **64**:21–47.
22. Börm S, Grasedyck L, Hackbusch W. Introduction to hierarchical matrices with applications. *Engineering Analysis with Boundary Elements* 2003; **27**:405–422.
23. Hackbusch W, Khoromskij B, Sauter S. On $\mathscr{H}^2$-matrices. In *Lectures on Applied Mathematics*, Bungartz H, Hoppe RHW, Zenger C (eds). Springer: Berlin, 2000; 9–29.
24. Hackbusch W, Börm S. Data-sparse approximation by adaptive $\mathscr{H}^2$-matrices. *Computing* 2002; **69**:1–35.
25. Eidelman Y, Gohberg I. On a new class of structured matrices. *Integral Equations and Operator Theory* 1999; **34**:293–324.
26. Chandrasekaran S, Dewilde P, Gu M, Pals T, Sun X, van der Veen AJ, White D. Some fast algorithms for sequentially semiseparable representations. *SIAM Journal on Matrix Analysis and Applications* 2005; **27**:341–364.
27. Vandebril R, Van Barel M, Mastronardi N. A note on the representation and definition of semiseparable matrices. *Numerical Linear Algebra with Applications* 2005; **12**:839–858.
28. Vandebril R, Van Barel M, Golub G, Mastronardi N. A bibliography on semiseparable matrices. *CALCOLO* 2005; **42**:249–270.
29. Chandrasekaran S, Gu M, Pals T. A fast *ULV* decomposition solver for hierarchically semiseparable representations. *SIAM Journal on Matrix Analysis and Applications* 2006; **28**:603–622.
30. Dewilde P, Chandrasekaran S. A hierarchical semi-separable Moore–Penrose equation solver. *Operator Theory*: *Advances and Applications* 2006; **167**:69–85.
31. Sheng Z, Dewilde P, van der Meijs N. Iterative solution methods based on the hierarchically semi-separable representation. *Proceedings of 17th Annual Workshop on Circuits*, *Systems and Signal Processing* (*ProRISC*), Veldhoven, NL, November 2006; 343–349.
32. Carrier J, Greengard L, Rokhlin V. A fast adaptive multipole algorithm for particle simulations. *SIAM Journal on Scientific and Statistical Computing* 1988; **9**:669–686.
33. Greengard L, Rokhlin V. A fast algorithm for particle simulations. *Journal of Computational Physics* 1987; **73**:325–348.
34. Chandrasekaran S, Dewilde P, Gu M. On the numerical rank of the off-diagonal blocks of Schur complements of discretized elliptic PDEs. Preprint, 2009.
35. Martinsson PG. A fast direct solver for a class of elliptic partial differential equations. *Journal of Scientific Computing* 2009; **38**:316–330.
36. Eisenstat SC, Liu JWH. The theory of elimination trees for sparse unsymmetric matrices. *SIAM Journal on Matrix Analysis and Applications* 2005; **26**:686–705.
37. Gilbert JR, Liu JWH. Elimination structures for unsymmetric sparse LU factors. *SIAM Journal on Matrix Analysis and Applications* 1993; **14**:334–352.
38. Liu JWH. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications* 1990; **18**:134–172.
39. Demmel J. *Applied Numerical Linear Algebra*. SIAM: Philadelphia, PA, 1997.
40. Golub G, Loan CV. *Matrix Computations*. The John Hopkins University Press: San Francisco, CA, 1989.
41. Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J, Du Croz J, Greenbaum A, Hammarling S, McKenney A, Sorensen D. *LAPACK Users' Guide* (3rd edn). SIAM: Philadelphia, PA, 1999.
42. George JA. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis* 1973; **10**:345–363.

43. Duff IS, Reid JK. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software* 1983; **9**:302–325.
44. Eisenstat SC, Liu JWH. A tree-based dataflow model for the unsymmetric multifrontal method. *Electronic Transactions on Numerical Analysis* 2005; **21**:1–19.
45. Liu JWH. The multifrontal method for sparse matrix solution: theory and practice. *SIAM Review* 1992; **34**:82–109.
46. Chen L, Xu J, Zhu Y. Local multilevel preconditioners for elliptic equations. Preprint, 2009.
47. Chen L. *i*FEM: An innovative finite element methods package in MATLAB. 2009; submitted.