NUMERICAL LINEAR ALGEBRA WITH APPLICATIONS
Numer. Linear Algebra Appl. 2011; **00**:1–0 *Prepared using* nlaauth.cls *[Version: 2002/09/18 v1.02]*

A robust inner-outer HSS preconditioner

Jianlin Xia[1*†]

[1]*Department of Mathematics, Purdue University, West Lafayette, IN 47907, U.S.A.*

SUMMARY

This paper presents an inner-outer preconditioner for symmetric positive definite matrices based on hierarchically semiseparable (HSS) matrix representation. A sequence of new HSS algorithms are developed, including some ULV-type HSS methods and triangular HSS methods. During the construction of this preconditioner, off-diagonal blocks are compressed in parallel, and an approximate HSS form is used as the outer HSS representation. In the meantime, the internal dense diagonal blocks are approximately factorized into HSS Cholesky factors. Such inner factorizations are guaranteed to exist for any approximation accuracy. The inner-outer preconditioner combines the advantages of both direct HSS and triangular HSS methods, and are both scalable and robust. Systematic complexity analysis and discussions of the errors are presented. Various tests on some practical numerical problems are used to demonstrate the efficiency and robustness. In particular, the effectiveness of the preconditioner in iterative solutions has been shown for some ill-conditioned problems. This work also gives a practical way of developing inner-outer preconditioners using direct rank structured factorizations (other than iterative methods).

KEY WORDS: robust preconditioner; hierarchically semiseparable (HSS) matrix; inner-outer HSS algorithm; ULV factorization

1. INTRODUCTION

In this paper, we present a robust and scalable inner-outer preconditioner based on hierarchically semiseparable (HSS) matrix structures [7, 8, 35]. It is known that dense intermediate matrices in many practical applications have certain low-rank structure (or low-rank off-diagonal blocks), based on the early studies by Rokhlin [28], Gohberg, Kailath, and Koltracht [12], and on the ideas of the fast multipole method (FMM) by Greengard and Rokhlin [17]. The HSS representation by Chandrasekaran, Dewilde, Gu, et al. is a stable special form of FMM representations and $\mathcal{H}$- and $\mathcal{H}^2$-matrices by Hackbusch, et al [5, 19, 20]. HSS matrices have been widely used in the development of fast numerical methods for sparse matrices, PDEs, integral equations, Toeplitz problems, and more [7, 22, 23, 26, 31, 34, etc.]. HSS methods can often be used to solve related problems with nearly

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

The research of Jianlin Xia was supported in part by NSF grants DMS-1115572 and CHE-0957024.

linear complexity. An HSS form has a nice binary tree structure and the operations are conducted locally at multiple hierarchical levels.

The applications of HSS methods (and similar) as preconditioners have been considered by Dewilde, Gu, et al. [18, 30, 36]. These methods can be considered as structured incomplete factorizations. Robustness enhancement has also been studied. In previous studies for general symmetric positive definite (SPD) matrices, it is know that incomplete Cholesky factorizations exist for M-matrices and certain H-matrices. See, e.g., [27] by Meijerink and van der Vorst and [25] by Manteuffel. Some robust or stabilized methods have been proposed for general SPD matrices. See, e.g., [3, 4] by Benzi, Cullum, and Tuma, and [21] by Kaporin. In fact, rank structured matrix techniques have also been shown very useful in robust preconditioning. Robustness techniques for $\mathcal{H}$-matrices are discussed by Bebendorf and Hackbusch [2]. For an SPD matrix, the structured methods in [18] by Gu, Li, and Vassilevski and in [36] by Xia and Gu guarantee that the structured preconditioners are always positive definite. In particular, it is discussed in [36] that HSS methods have the potential to work as effective preconditioners for problems where the low-rank structure is insignificant, or when the problems have only weak low-rank property.

However, both methods in [18, 36] involve Schur complement computations in approximate triangular factorizations, and are not easily parallelizable. Recently, the scalability of HSS methods has been investigated by Wang, Li, et al. [32]. It is shown that both the direct construction of an HSS form and the solution of an HSS system with an ULV factorization procedure are highly parallelizable. (In ULV, U and V represent orthogonal matrices and L represents triangular ones.) On the other hand, the direct HSS construction may not preserve positive definiteness so that the ULV factorization may break down.

Direct HSS methods and triangular HSS factorizations also have different efficiency. For example, the direct construction of an HSS approximation to a general SPD matrix costs $3rn^2$ flops under certain assumptions, where r is a related off-diagonal rank bound [33]. Then the ULV factorization and solution in [35] costs at least $\frac{56}{3}r^2n$. In contrast, an HSS Cholesky factorization costs about $\frac{11}{2}rn^2$, but the triangular HSS solution is more efficient and needs only $10rn$.

1.1. Main results

The task of this work is to combine the benefits of both direct HSS and triangular HSS methods. That is, we propose an efficient and effective preconditioner for SPD matrices which is both scalable and robust. Motivated by the ideas of inner-outer iterative preconditioning by Saad [29], Golub and Ye [14], Bai, Benzi, and Chen [1], et al., we design an inner-outer HSS factorization scheme. We give a sequence of new HSS algorithms, including both direct and triangular HSS ones. Then the inner-outer preconditioner uses a direct HSS construction and ULV factorization as the outer scheme, where an inner triangular HSS factorization is applied to the diagonal blocks. Both the inner and the outer schemes approximate certain off-diagonal blocks with compression or rank-revealing factorizations.

Just like some inner-outer iteration-based preconditioners [1, 14], we can also use lower accuracies in the inner HSS factorizations so as to save costs. Moreover, we allow the flexibility of adjusting the levels of hierarchical operations in both the inner and the outer HSS methods. The level of outer operations can be designed to fit the available parallel computing resource, and the inner one can then be adjusted to enhance the *robustness*. Here, the robustness means that our inner scheme guarantees the existence of a local Cholesky HSS factorization regardless of the approximation accuracy, unlike ULV HSS methods which may break down at certain stages of the hierarchical operations (see, e.g.,

Example 2 below). At the outer HSS levels, even if breakdown occurs, we can use simple diagonal shifting. Since the number of outer HSS levels is often small, the effect of diagonal shifting on the approximation accuracy is limited, as illustrated in Section 5.

The individual new HSS algorithms here are compared with similar existing ones, and generally have better efficiency and scalability. For example, for ULV HSS factorizations, we use local triangular factorizations for diagonal blocks, which not only are more efficient than QR factorizations as used in [8], but also enable us to use the new triangular HSS algorithms as inner operations. Our triangular HSS factorization only needs to compress off-diagonal blocks about half of the size of those used in [36], while keeping about the same accuracy. Unlike the triangular HSS solution procedure in [24] which is sequential, a scalable triangular ULV factorization scheme with similar cost is proposed.

The detailed complexity of these new HSS algorithms is analyzed. The analysis can help us compare different methods and provide improvements and optimization. (See, e.g., (39) and (40)). Discussions on the errors are given. Several numerical examples are used to demonstrate the efficiency and robustness of the algorithms. We show the effectiveness of the inner-outer HSS preconditioning for some ill-conditioned problems, such as a Toeplitz problem generated by a Gaussian radial basis function and a linear elasticity PDE. We observe that, with efficient inner-outer HSS preconditioning, the preconditioned conjugate gradient method converges quickly for all the examples, and the convergence is often much faster than with block diagonal preconditioning. The overall inner-outer preconditioning cost is often insignificant when we manually set the off-diagonal rank bounds to be small. Our methods can also be built into sparse factorization or preconditioning frameworks as in [15, 16, 34] by Grasedyck, Le Borne, Kriemann, Xia, et al.

1.2. Outline and notation

The remaining sections are organized as follows. Section 2 provides some new HSS factorization algorithms for SPD matrices. Additional HSS solution algorithms and our inner-outer HSS preconditioner is given in Section 3. The complexity and error analysis are presented in Section 4. Sections 5 and 6 are devoted to some numerical experiments and concluding remarks, respectively.

For convenience, the following notation is used in the presentation:

- For an index set $t_i \subset \mathcal{I} \equiv \{1, 2, \ldots, n\}$, let $t_i^c \cup t_i \cup t_i^r = \mathcal{I}$, where all indices in t_i^c are smaller than those in t_i, and all indices in t_i^r are larger than those in t_i.
- $A|_{t_i \times t_j}$ is the submatrix of a matrix A with row index set t_i and column index set t_j.
- If a rank-revealing QR (RRQR) factorization is computed for $A|_{t_i \times t_j}$, we sometimes write $A|_{t_i \times t_j} \approx U_i A|_{\hat{t}_i \times t_j}$, which can be understood as that the factor $A|_{\hat{t}_i \times t_j}$ is still stored in A with row index set $\hat{t}_i$.
- $\mathcal{T}$ is a postordered full binary tree with nodes ordered as $1, 2, \ldots$ That is, each non-leaf node i and its children $c_{i,1}, c_{i,2}$ are ordered following $c_{i,1} < c_{i,2} < i$. Sometimes we just write the children as c_1, c_2 when no confusion is caused. For a node i, denote its parent and sibling by $\mathrm{par}(i)$ and $\mathrm{sib}(i)$, respectively.
- $\mathrm{diag}(A_1, \ldots, A_k)$ is a diagonal matrix with diagonal blocks $A_1, \ldots, A_k$.

The definition and notation for HSS structures are also briefly reviewed as follows, following a postordering HSS form [35, 36].

Definition 1.1. *Assume A be an $n \times n$ dense matrix. Let $\mathcal{T}$ be a postordered full binary tree with $2k-1$ nodes, and $t_i \subset \mathcal{T}$ be an index set associated with each node i of $\mathcal{T}$. We say $\mathcal{T}$ is a* postordered HSS tree *and A is in an* HSS form *if:*

1. *For each non-leaf node i with children c_1 and c_2, $t_{c_1} \cup t_{c_2} = t_i$, $t_{c_1} \cap t_{c_2} = \phi$, and $t_{2k-1} = \mathcal{I} = \{1, 2, \ldots, n\}$.*
2. *There exist matrices $D_i, U_i, V_i, R_i, W_i, B_i$ (called* HSS generators*) associated with each node i satisfying*

$$D_i = \begin{pmatrix} D_{c_1} & U_{c_1} B_{c_1} V_{c_2}^T \\ U_{c_2} B_{c_2} V_{c_1}^T & D_{c_2} \end{pmatrix}, \; U_i = \begin{pmatrix} U_{c_1} R_{c_1} \\ U_{c_2} R_{c_2} \end{pmatrix}, \; V_i = \begin{pmatrix} V_{c_1} W_{c_1} \\ V_{c_2} W_{c_2} \end{pmatrix}, \tag{1}$$

so that $D_{2k-1} \equiv A|_{t_i \times t_i}$. Here, $U_{2k-1}, V_{2k-1}, R_{2k-1}, W_{2k-1}, B_{2k-1}$ are empty matrices.

A is in a data-sparse form given be the generators. For a non-leaf node i, the generators D_i, U_i, V_i are recursively defined and are not explicitly stored. Define the following *HSS blocks*:

$$A_i^- = A|_{t_i \times (\mathcal{I} \setminus t_i)}, \; A_i^| = A|_{(\mathcal{I} \setminus t_i) \times t_i}.$$

Clearly, U_i gives a column basis for A_i^- (HSS block row), and V_i^T gives a row basis for $A_i^|$ (HSS block column). U_i and V_i are also called cluster bases in [5]. The maximum (numerical) rank of all HSS blocks is called the HSS rank of A. The following is a block 4×4 HSS matrix example:

$$A = \begin{pmatrix} D_1 & U_1 B_1 V_2^T & U_1 R_1 B_3 W_4^T V_4^T & U_1 R_1 B_3 W_5^T V_5^T \\ U_2 B_2 V_1^T & D_2 & U_2 R_2 B_3 W_4^T V_4^T & U_2 R_2 B_3 W_5^T V_5^T \\ U_4 R_4 B_6 W_1^T V_1^T & U_4 R_4 B_6 W_2^T V_2^T & D_4 & U_4 B_4 V_5^T \\ U_5 R_5 B_6 W_1^T V_1^T & U_5 R_5 B_6 W_2^T V_2^T & U_5 B_5 V_4^T & D_5 \end{pmatrix}. \tag{2}$$

See Figure 1. Obviously, if A is symmetric, we can set [35]

$$D_i = D_i^T, \; V_i = U_i, \; B_j = B_i^T \text{ with } j = \text{sib}(i). \tag{3}$$

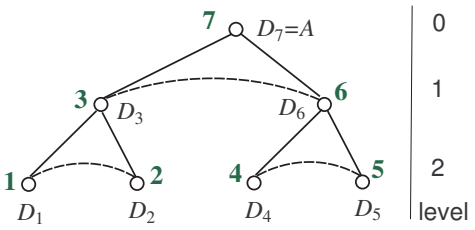


Figure 1. HSS tree for the block 4×4 HSS matrix (2).

A given HSS form generally enables us to conduct related matrix operations such as matrix inversions and multiplications in linear complexity, if the HSS rank is small. Otherwise, we can use compact HSS forms as efficient preconditioners.

2. NEW SPD HSS FACTORIZATION ALGORITHMS

We show how to convert a dense SPD matrix into data-sparse forms, by either direct HSS construction with ULV factorization, or Cholesky HSS factorization. In this section, for convenience, each subsection uses the same notation for the generators of a related HSS form. To help the reader understand the algorithms, we briefly summarize the major framework of each algorithm in Table I.

HSS construction (Section 2.1) • Leaf level: off-diagonal block row compression for U generators; • Non-leaf levels: off-diagonal block row compression for R generators; • Similar off-diagonal block column compression.
ULV HSS factorization (Section 2.2, and similar for Section 2.4) • Diagonal block factorization; • Introducing zeros into off-diagonal blocks and preserving diagonal identity matrices; • Partial diagonal elimination; • Merging and recursion.
Robust HSS Cholesky factorization (Section 2.3) • Leaf level: – Diagonal block factorization; – Computation of the off-diagonal block row of the factor and its compression for U generators; – Partial Schur complement formation; • Non-leaf level: off-diagonal block row compression for R generators; • Similar off-diagonal block column compression.

Table I. Framework of the major algorithms in Section 2.

2.1. Scalable symmetric HSS construction

The HSS construction algorithm in [35] uses nested compression of the HSS blocks of a given matrix A. However, it is a sequential process since early compression results participate in later compression. The algorithm can be modified into a new scalable one. In particular, for a symmetric matrix A, we need only to compress the block rows. Moreover, we can further reduce the compression cost. The method has two stages and is briefly explained as follows. For simplicity, assume the row size and numerical rank of all HSS block rows are m and r, respectively.

During the first stage, all HSS block rows are compressed. If i is a leaf, compute an RRQR

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

If i is a non-leaf node with children c_1 and c_2, compute an RRQR of the matrix

$$\begin{pmatrix} A|_{\hat{t}_{c_1}\times(\mathcal{I}\setminus t_i)} \\ A|_{\hat{t}_{c_2}\times(\mathcal{I}\setminus t_i)} \end{pmatrix} \approx \begin{pmatrix} R_{c_1} \\ R_{c_2} \end{pmatrix} A|_{\hat{t}_i\times(\mathcal{I}\setminus t_i)}.$$

Then by recursion,

$$A|_{t_i\times(\mathcal{I}\setminus t_i)} \approx \begin{pmatrix} U_{c_1}A|_{\hat{t}_{c_2}\times(\mathcal{I}\setminus t_i)} \\ U_{c_2}A|_{\hat{t}_{c_2}\times(\mathcal{I}\setminus t_i)} \end{pmatrix} = \begin{pmatrix} U_{c_1} & \\ & U_{c_2} \end{pmatrix} \begin{pmatrix} R_{c_1} \\ R_{c_2} \end{pmatrix} A|_{\hat{t}_i\times(\mathcal{I}\setminus t_i)} = U_i A|_{\hat{t}_i\times(\mathcal{I}\setminus t_i)}.$$

That is, we obtain the orthonormal column basis U_i for each HSS block row A_i^-. These compression steps are done independently of other nodes at the same level of the HSS tree. After this stage, all U and R generators are available.

In the next stage, we compute the B generators. Let $j = \mathrm{sib}(i)$. According to the first stage, we have

$$A|_{t_i\times t_j} \approx U_i A|_{\hat{t}_i\times t_j}.$$

On the other hand, in the HSS approximation, $A|_{t_i \times t_j}$ has a form $U_i B_i U_j^T$. Thus, we can let

$$B_i^T = \begin{cases} A|_{\hat{t}_j \times t_i} U_j, & \text{if } i \text{ is a leaf,} \\ A|_{\hat{t}_j \times t_i} \begin{pmatrix} R_{c_{j1}} \\ R_{c_{j2}} \end{pmatrix}, & \text{otherwise.} \end{cases} \tag{4}$$

Note that the method in [35] needs additional compression steps to get B_i, which are generally more expensive than (4).

2.2. Improved ULV-type HSS factorization

The ULV HSS Cholesky factorization in [35] computes an explicit ULV-type factorization of an HSS matrix A. The basic idea is similar to the implicit ULV solution in [8]. However, for SPD matrices, as pointed out in [35], the efficiency of these ULV algorithms can be improved. Moreover, we can take advantage of certain special forms of the generators. The main steps are illustrated in Figure 2 and are explained as follows. For simplicity, assume all leaf level HSS block row sizes are $m_i \equiv m$, and the ranks of all HSS blocks are r. Since U_i forms a basis for the column space of A_i^-, sometimes we write $A_i^- = U_i T_i$.

If node i is a leaf, compute a Cholesky factorization $D_i = L_i L_i^T$, and then a full QL factorization of $L_i^{-1} U_i$ as

$$L_i^{-1} U_i = Q_i \begin{pmatrix} 0 \\ \tilde{U}_i \end{pmatrix} \begin{matrix} m-r \\ r \end{matrix}. \tag{5}$$

See Figure 2(i)–(ii). Then apply Q_i^T to the HSS block row $A_i^- = U_i T_i$ (on the left):

$$Q_i^T A_i^- = \begin{pmatrix} 0 \\ \tilde{U}_i T_i \end{pmatrix} \begin{matrix} m-r \\ r \end{matrix}$$

This also means Q_i is applied to the HSS block column (on the right). Notice that the diagonal block remains to be an identity matrix. See Figure 2(iii). After this, the leading $m-r$ rows of block row i can be eliminated. This is done for all the leaves.

If node i is a non-leaf node with its children c_1 and c_2 being leaves, assume the previous operations are applied to both c_1 and c_2. That is,

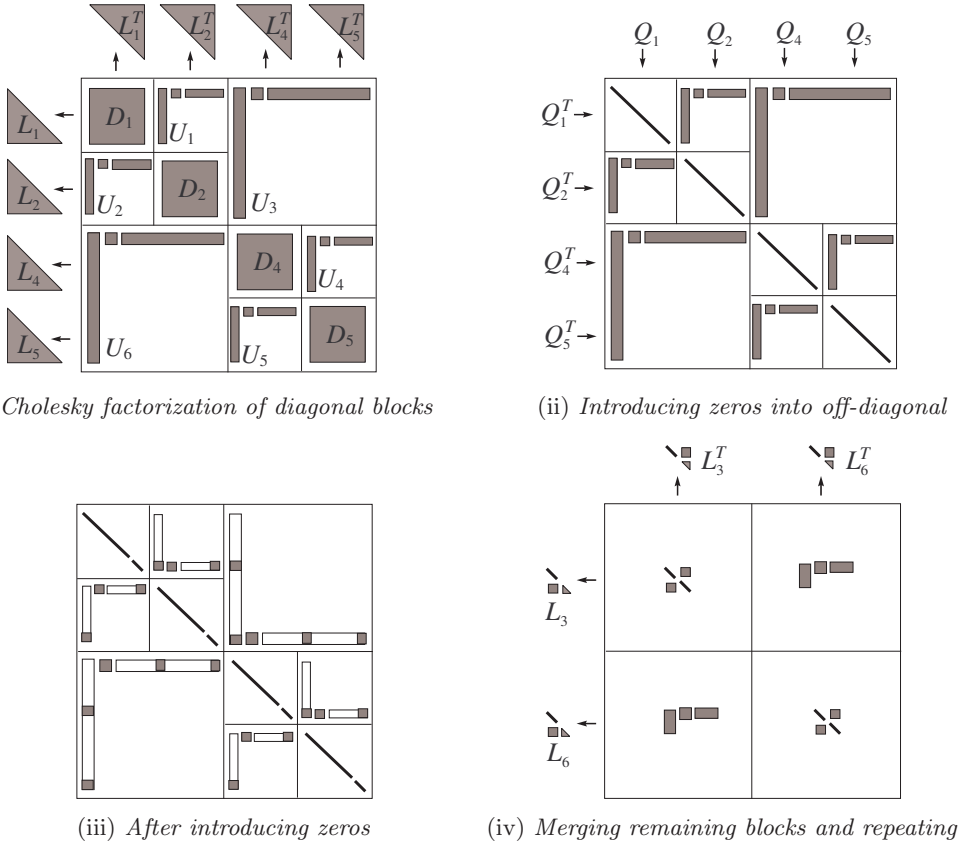
$$\begin{aligned} &\begin{pmatrix} Q_{c_1}^T L_{c_1}^{-1} & \\ & Q_{c_2}^T L_{c_2}^{-1} \end{pmatrix} \begin{pmatrix} D_{c_1} & U_{c_1} B_{c_1} U_{c_2}^T \\ U_{c_2} B_{c_1}^T U_{c_1}^T & D_{c_2} \end{pmatrix} \begin{pmatrix} L_{c_1}^{-T} Q_{c_1} & \\ & L_{c_2}^{-T} Q_{c_2} \end{pmatrix} \\ &= \begin{pmatrix} I & \operatorname{diag}\left(0,\ \tilde{U}_{c_1} B_{c_1} \tilde{U}_{c_2}^T\right) \\ \operatorname{diag}\left(0,\ \tilde{U}_{c_2} B_{c_1}^T \tilde{U}_{c_1}^T\right) & I \end{pmatrix}, \end{aligned} \tag{6}$$

where $\operatorname{diag}(\cdots)$ denotes a block diagonal matrix with the given blocks on the diagonal. Then we merge appropriate blocks on the right-hand side of (6) and remove c_1 and c_2 from the HSS tree so that i becomes a leaf with D and U generators

$$D_i = \begin{pmatrix} I & \tilde{U}_{c_1} B_{c_1} \tilde{U}_{c_2}^T \\ \tilde{U}_{c_2} B_{c_1}^T \tilde{U}_{c_1}^T & I \end{pmatrix},\ U_i = \begin{pmatrix} \tilde{U}_{c_1} R_{c_1} \\ \tilde{U}_{c_2} R_{c_2} \end{pmatrix}. \tag{7}$$

(Note that we are using D_i and U_i to mean the generators of a smaller matrix after elimination, instead of A. This smaller matrix is called a *reduced HSS matrix*). See Figure 2(iv).

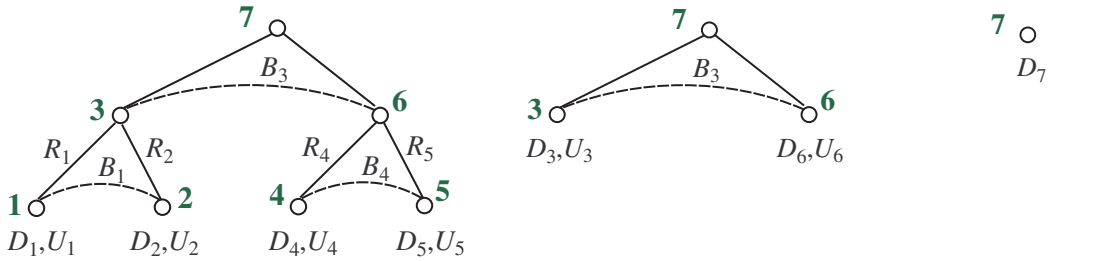
The same operations then apply recursively. At the root level, a direct Cholesky factorization is computed. Figure 3 shows how an HSS tree is reduced along the factorization.



(i) *Cholesky factorization of diagonal blocks* (ii) *Introducing zeros into off-diagonal*

(iii) *After introducing zeros* (iv) *Merging remaining blocks and repeating*

Figure 2. An improved ULV HSS factorization scheme for an SPD matrix.



(i) *Symmetric HSS tree* (ii) *After eliminating one level* (iii) *Until the root*

Figure 3. How the HSS tree is reduced along the ULV HSS factorization.

Note that due to the special form of D_i in (7), the intermediate Cholesky factorization $D_i = L_i L_i^T$ at non-leaf levels can be done quickly. That is, we have

$$L_i = \begin{pmatrix} I & \\ \tilde{U}_{c_2} B_{c_1}^T \tilde{U}_{c_1}^T & \tilde{L}_i \end{pmatrix}, \quad \tilde{L}_i \tilde{L}_i^T = I - (\tilde{U}_{c_2} B_{c_1}^T)(\tilde{U}_{c_2} B_{c_1}^T)^T. \tag{8}$$

Similarly, the update $L_i^{-1} U_i$ for (7)–(8) can also be done quickly as

$$L_i^{-1} U_i = \begin{pmatrix} I & \\ -\tilde{L}_i^{-1} \tilde{U}_{c_2} B_{c_1}^T \tilde{U}_{c_1}^T & \tilde{L}_i^{-1} \end{pmatrix} \begin{pmatrix} \tilde{U}_{c_1} R_{c_1} \\ \tilde{U}_{c_2} R_{c_2} \end{pmatrix} = \begin{pmatrix} \tilde{U}_{c_1} R_{c_1} \\ \tilde{L}_i^{-1} \tilde{U}_{c_2} [R_{c_2} - (B_{c_1}^T \tilde{U}_{c_1}^T)(\tilde{U}_{c_1} R_{c_1})] \end{pmatrix}. \tag{9}$$

(8)–(9) can be carefully organized so that they only need six matrix multiplications and one triangular solution.

This improved algorithm is thus more efficient than the one in [35], which requires two dense square matrix products in the form of $Q_i^T \tilde{D}_i Q_i$ and the full QR factorization of a size $(m-r)\times r$ dense block for each node i. In fact, according to Table VI in Section 4, this new ULV factorization with $m = 2r$ costs $\frac{35}{3}r^2 n$ flops, while the one in [35] costs $\frac{56}{3}r^2 n$ flops.

Our algorithm computes an ULV-type HSS factorization

$$A = \hat{\mathbf{L}}\hat{\mathbf{L}}^T, \tag{10}$$

where $\hat{\mathbf{L}}$ is given by a sequence of orthogonal and lower triangular matrices. We call $\hat{\mathbf{L}}$ an ULV factor, which is formed recursively. Let $A^{(l)}$ be the reduced matrix at level l obtained from $A^{(l+1)}$ after the elimination of all nodes at level $l+1$, where $A^{(l_{\max})} \equiv A$ for the leaf level $l_{\max}$. Assume the nodes at level l are $j_1, j_2, \ldots, j_\alpha$. Also let P_l be a permutation matrix which performs all the merge steps during the elimination of level $l+1$ and also forms the new reduced HSS matrix. That is,

$$P_l X_l \mathrm{diag}\left(\begin{pmatrix}\bar{U}_{c_{j_1,1}} R_{c_{j_1,1}}\\ \bar{U}_{c_{j_1,2}} R_{c_{j_1,2}}\end{pmatrix}, \ldots, \begin{pmatrix}\bar{U}_{c_{j_\alpha,1}} R_{c_{j_\alpha,1}}\\ \bar{U}_{c_{j_\alpha;2}} R_{c_{j_\alpha,2}}\end{pmatrix}\right) = \begin{pmatrix} 0 \\ \mathrm{diag}\left(\bar{U}_{j_1}, \ldots, \bar{U}_{j_\alpha}\right)\end{pmatrix}, \tag{11}$$

$$P_l X_l A^{(l+1)} X_l^T P_l^T = \mathrm{diag}\left(I, A^{(l)}\right), \tag{12}$$

where

$$X_l = \mathrm{diag}\left(\mathrm{diag}\left(Q^T_{c_{j_1,1}} L^{-1}_{c_{j_1,1}},\ Q^T_{c_{j_1,2}} L^{-1}_{c_{j_1,2}}\right), \ldots, \mathrm{diag}\left(Q^T_{c_{j_\alpha,1}} L^{-1}_{c_{j_\alpha,1}}, Q^T_{c_{j_\alpha,2}} L^{-1}_{c_{j_\alpha,2}}\right)\right).$$

Then we let

$$\hat{\mathbf{L}} \equiv \hat{\mathbf{L}}^{(l_{\max})},\ \text{and}\ A^{(l)} = \hat{\mathbf{L}}^{(l)}(\hat{\mathbf{L}}^{(l)})^T,\ l = l_{\max}, l_{\max}-1, \ldots, 1, 0,$$

where $\hat{\mathbf{L}}^{(0)}$ is the Cholesky factor associated with the root, and

$$\hat{\mathbf{L}}^{(l+1)} = X_l^{-1} P_l^T \,\mathrm{diag}\left(I, \hat{\mathbf{L}}^{(l)}\right) P_l,\ l = l_{\max}-1, l_{\max}-2, \ldots, 1, 0. \tag{13}$$

2.3. Robust HSS Cholesky factorization

2.3.1. Motivation of robust and effective HSS preconditioning The factorization method in [36] computes an approximate Cholesky factorization for a dense SPD matrix $A \approx \mathbf{R}^T\mathbf{R}$, where $\mathbf{R}$ is an upper triangular HSS matrix. The approximation $\mathbf{R}^T\mathbf{R}$ is guaranteed to exist and to be positive definite, regardless of the approximation accuracy. The basic idea is called *Schur compensation* and can be illustrated as follows. Consider a block 2×2 SPD matrix and the Cholesky factorization of its $(1,1)$ block:

$$A \equiv \begin{pmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22}\end{pmatrix} = \begin{pmatrix} D_1^T & \\ A_{21}D_1^{-1} & I\end{pmatrix}\begin{pmatrix} D_1 & D_1^{-T}A_{21}^T \\ & S\end{pmatrix}, \tag{14}$$

where $S = A_{22} - \left(A_{21}D_1^{-1}\right)\left(D_1^{-T}A_{21}^T\right)$ is the Schur complement. Compute an SVD: $D_1^{-T}A_{21}^T = U_1\Sigma U_2^T + \hat{U}_1\hat{\Sigma}\hat{U}_2^T$, where the singular values in Σ are larger than a tolerance τ, and those in $\hat{\Sigma}$ are smaller than τ. (A relative tolerance τ may be used.) If all the singular values in $\hat{\Sigma}$ are dropped, then S is approximated by

$$\tilde{S} = A_{22} - \hat{U}_2\Sigma^2\hat{U}_2^T = S + O(\tau^2). \tag{15}$$

That is, a positive semidefinite term is automatically added to the Schur complement. Compute a Cholesky factorization $\tilde{S} = D_2D_2^T$. We then get an approximate Cholesky factorization $A \approx \mathbf{R}^T\mathbf{R}$.

It is also shown in [36] that, with certain additional techniques, a modification to $\mathbf{R}$ can work as an effective preconditioner when the low-rank property is not significant. That is, if the numerical rank for a given tolerance is large, we can still manually set a small rank to get a low accuracy $\mathbf{R}$. The idea is then generalized so that the Cholesky factor of A is approximated by an HSS matrix. RRQR factorizations are often used to replace SVDs in the compression, since they are generally more efficient.

2.3.2. Improved HSS Cholesky factorization The HSS Cholesky factorization algorithm in [36] uses a form for $\mathbf{R}$ where only D, U, R, B generators are needed. $\mathbf{R}$ is computed block rowwise. The computed off-diagonal block column $\mathbf{R}|_{t_i^c \times t_i}$ and off-diagonal block row $\mathbf{R}|_{t_i \times t_i^r}$ associated with node i are put together as $\begin{pmatrix} (\mathbf{R}|_{t_i^c \times t_i})^T & \mathbf{R}|_{t_i \times t_i^r} \end{pmatrix}$, which is compressed to provide the column basis U_i. This is equivalent to constructing a form for a symmetric full matrix whose block upper triangular part is $\mathbf{R}$. Here, we propose a new procedure which has several advantages:

(1) Instead of compressing $\begin{pmatrix} (\mathbf{R}|_{t_i^c \times t_i})^T & \mathbf{R}|_{t_i \times t_i^r} \end{pmatrix}$, we compress $\mathbf{R}|_{t_i^c \times t_i}$ and $\mathbf{R}|_{t_i \times t_i^r}$ independently so as to get their row and column bases U_i and V_i^T, respectively. This implementation is much simpler.

(2) The HSS form of the Cholesky factor we obtained is more compact, since the individual numerical ranks of $\mathbf{R}|_{t_i^c \times t_i}$ and $\mathbf{R}|_{t_i \times t_i^r}$ are usually smaller than that of $\begin{pmatrix} (\mathbf{R}|_{t_i^c \times t_i})^T & \mathbf{R}|_{t_i \times t_i^r} \end{pmatrix}$, if the same compression accuracy is used.

(3) The algorithm in [36] maintains a compressed form (see (22) below) only for Schur complement computations. Here, this compressed form is also used to speed up the computation of V_i generators.

The new HSS Cholesky factorization procedure is described with the aid of the following definition.

Definition 2.1. *[36] For a node i of a postordered binary tree $\mathcal{T}$, the* set of predecessors *of i is defined to be*

$$\mathrm{pred}(i) = \begin{cases} \{i\}, & \text{if } i \text{ is the root of } \mathcal{T}, \\ \{i\} \cup \mathrm{pred}(\mathrm{par}(i)), & \text{otherwise.} \end{cases}$$

A visited set *$\mathcal{V}_i$ (set of visited nodes before i whose siblings have not been visited) is defined to be*

$$\mathcal{V}_i = \{j : j \text{ is a left node and } \mathrm{sib}(j) \in \mathrm{pred}(i)\}. \tag{16}$$

We can still use the HSS tree $\mathcal{T}$ as in Definition 1.1 for $\mathbf{R}$, but some generators do not exist. The following result can be easily verified.

Observation 1. For an HSS matrix to be block triangular, some of its generators are empty matrices, as specified in Table II.

	Lower triangular	Upper triangular
$\mathrm{par}(i) \in \mathrm{pred}(1)$	$U_i,\ R_i$	$V_i,\ W_i$
$\mathrm{par}(i) \in \mathrm{pred}(j)$	$V_i,\ W_i$	$U_i,\ R_i$

Table II. Empty generators of triangular HSS matrices, where j is the last (largest-labeled) leaf of the HSS tree of the matrix.

The factorization is conducted along the postordering traversal of the HSS tree for nodes $i = 1, 2, \ldots$

If i is a leaf, let $A^{(i-1)}|_{t^r_{i-1}\times t^r_{i-1}}$ denote the Schur complement due to the previous $i-1$ factorization steps. $A^{(i-1)}|_{t^r_{i-1}\times t^r_{i-1}}$ is partially formed, or, its first block row $A^{(i-1)}|_{t_i\times t^r_{i-1}}$ is available (see (23) below). Note $t^r_{i-1} = t_i \cup t^r_i$. Compute a block row $\begin{pmatrix} D_i & \mathbf{R}|_{t_i\times t^r_i}\end{pmatrix}$ of $\mathbf{R}$ as in the usual Cholesky factorization. That is, compute the following Cholesky factorization and update the off-diagonal block as

$$A^{(i-1)}|_{t_i\times t_i} = D_i^T D_i,\ \mathbf{R}|_{t_i\times t^r_i} = D_i^{-T} A^{(i-1)}|_{t_i\times t^r_i}. \tag{17}$$

Then compress $\mathbf{R}|_{t_i\times t^r_i}$ with an RRQR factorization

$$\mathbf{R}|_{t_i\times t^r_i} \approx U_i \mathbf{R}|_{\hat{t}_i\times t^r_i}. \tag{18}$$

The off-diagonal block column $\mathbf{R}|_{t^c_i\times t_i}$ is already partially compressed in previous steps so that $\mathbf{R}|_{t_j\times t_i} \approx U_j \mathbf{R}|_{\hat{t}_j\times t_i}$ for all $j \in \mathcal{V}_i \equiv \{j_1, j_2, \ldots, j_s\}$, where $t^c_i = t_{j_1} \cup t_{j_2} \cup \cdots \cup t_{j_s}$. Ignoring previously computed U bases, we only need to compress

$$\left((\mathbf{R}|_{\hat{t}_{j_1}\times t_i})^T \quad (\mathbf{R}|_{\hat{t}_{j_2}\times t_i})^T \quad \cdots \quad (\mathbf{R}|_{\hat{t}_{j_s}\times t_i})^T\right) \approx V_i \left((\mathbf{R}|_{\hat{t}_{j_1}\times \hat{t}_i})^T \quad (\mathbf{R}|_{\hat{t}_{j_2}\times \hat{t}_i})^T \quad \cdots \quad (\mathbf{R}|_{\hat{t}_{j_s}\times \hat{t}_i})^T\right). \tag{19}$$

Note that (19) can be replaced by a more efficient step as illustrated in (26) and (27) below.

If i is a non-leaf node, let c_1 and c_2 be its children. Clearly, $t^r_{c_2} = t^r_i$. The blocks $\mathbf{R}|_{t_{c_1}\times t^r_i}$ and $\mathbf{R}|_{t_{c_2}\times t^r_i}$ are compressed in previous steps. With the basis matrices ignored, we compress the following matrix

$$\begin{pmatrix} \mathbf{R}|_{\hat{t}_{c_1}\times t^r_i} \\ \mathbf{R}|_{\hat{t}_{c_2}\times t^r_i}\end{pmatrix} \approx \begin{pmatrix} R_{c_1} \\ R_{c_2}\end{pmatrix} \mathbf{R}|_{\hat{t}_i\times t^r_i}. \tag{20}$$

Similarly, the further compression of $\mathbf{R}|_{t^c_i\times t_i} = \begin{pmatrix} \mathbf{R}|_{t^c_i\times t_{c_1}} & \mathbf{R}|_{t^c_i\times t_{c_2}}\end{pmatrix}$ is done by ignoring certain existing V basis matrices. That is, compress

$$\begin{pmatrix} \begin{pmatrix} (\mathbf{R}|_{\hat{t}_{j_1}\times \hat{t}_{c_1}})^T & \cdots & (\mathbf{R}|_{\hat{t}_{j_s}\times \hat{t}_{c_1}})^T \end{pmatrix} \\ \begin{pmatrix} (\mathbf{R}|_{\hat{t}_{j_1}\times \hat{t}_{c_2}})^T & \cdots & (\mathbf{R}|_{\hat{t}_{j_s}\times \hat{t}_{c_2}})^T \end{pmatrix}\end{pmatrix} \approx \begin{pmatrix} W_{c_1} \\ W_{c_2}\end{pmatrix} \left((\mathbf{R}|_{\hat{t}_{j_1}\times \hat{t}_i})^T \quad \cdots \quad (\mathbf{R}|_{\hat{t}_{j_s}\times \hat{t}_i})^T\right). \tag{21}$$

After these compression steps, U_i and V_i are implicitly available due the recursions in (1). According to Observation 1, if $i \in \text{pred}(1)$, (18) and (20) are not needed. Also, if $i \in \text{pred}(j)$ where j is the last (largest-labeled) leaf, (19) and (21) are not needed.

Furthermore, if i is a right node with its sibling j, we set

$$B_j = \mathbf{R}|_{\hat{t}_j\times \hat{t}_i}.$$

Before the traversal of each leaf i, we partially form $A^{(i-1)}|_{t_i\times t^r_{i-1}}$, which is the first block row of the new Schur complement. Similar to [36], we also maintain a compact approximation

$$\mathbf{R}|_{t^c_i\times t^r_{i-1}} \approx Q_{i-1} Y_{i-1}, \tag{22}$$

where Q_{i-1} has orthonormal columns and is not stored. Once (22) holds, then

$$A^{(i-1)}|_{t_i\times t^r_{i-1}} = A|_{t_i\times t^r_{i-1}} - (\mathbf{R}|_{t^c_i\times t_i})^T \mathbf{R}|_{t^c_i\times t^r_{i-1}} \approx A|_{t_i\times t^r_{i-1}} - Y^T_{i-1;1} Y_{i-1}, \tag{23}$$

where a partitioned form of (22) is used:

$$\begin{pmatrix} \mathbf{R}|_{t^c_i\times t_i} & \mathbf{R}|_{t^c_i\times t^r_i}\end{pmatrix} \approx Q_{i-1} \begin{pmatrix} Y_{i-1;1} & Y_{i-1;2}\end{pmatrix}. \tag{24}$$

Here, Y_{i-1} is obtained approximately by recursion. For any $i-1 \in \mathrm{pred}(1)$, set $Y_{i-1} \equiv \mathbf{R}|_{\hat{t}_{i-1} \times t^r_{i-1}}$. Otherwise, let j be the largest leaf before i so that Y_{j-1} is previously available. Then Y_{i-1} can be approximately computed with an RRQR factorization

$$\begin{pmatrix} Y_{j-1;2} \\ \mathbf{R}|_{\hat{t}_j \times t^r_j} \end{pmatrix} \approx \tilde{Q}_{i-1} Y_{i-1}. \tag{25}$$

See [36] for the detailed derivation.

Furthermore, unlike [36], here we can use Y_{i-1} to improve the efficiency of computing V_i. Due to (24), the HSS block column $\mathbf{R}|_{t^c_i \times t_i}$ is already in a compressed form

$$\mathbf{R}|_{t^c_i \times t_i} \approx Q_{i-1} Y_{i-1;1}.$$

Thus, to get the V_i^T which gives a row basis for the row space of $\mathbf{R}|_{t^c_i \times t_i}$, we just compress $Y^T_{i-1;1}$ with an RRQR factorization

$$Y^T_{i-1;1} \approx V_i T_i, \tag{26}$$

where T_i is not stored. Then set

$$\left((\mathbf{R}|_{\hat{t}_{j_1} \times \hat{t}_i})^T \quad (\mathbf{R}|_{\hat{t}_{j_2} \times \hat{t}_i})^T \quad \cdots \quad (\mathbf{R}|_{\hat{t}_{j_s} \times \hat{t}_i})^T \right) = V_i \left((\mathbf{R}|_{\hat{t}_{j_1} \times t_i})^T \quad (\mathbf{R}|_{\hat{t}_{j_2} \times t_i})^T \quad \cdots \quad (\mathbf{R}|_{\hat{t}_{j_s} \times t_i})^T \right). \tag{27}$$

Equations (26) and (27) can be used to replace (19). This generally reduces the compression cost, since the difference between the costs of (19) and (26)–(27) is

$$\begin{aligned} &\left(4m(sr)r - 2r^2(sr+m) + \frac{4}{3}r^3 \right) - \left(2(sr)mr + 4mrr - 2r^2(r+m) + \frac{4}{3}r^3 \right) \\ &= 2r^2 \left((m-r)s + r - 2m \right). \end{aligned}$$

Here, $m > r$ in general, and s is as large as $O(\log n)$.

2.4. Triangular ULV HSS factorization

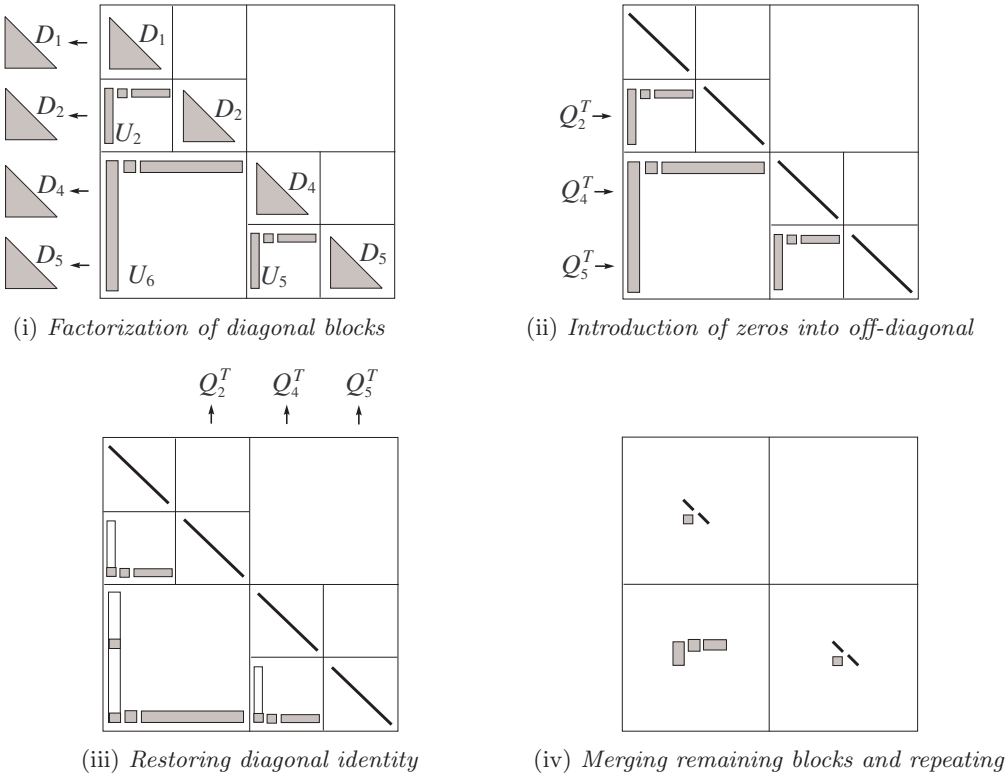
Next, we study the ULV factorization of triangular HSS matrices. Notice that the ULV algorithm in Section 2.2 cannot be applied in a straightforward way, due to the nonsymmetry. We consider a (block) lower triangular HSS matrix L. (An upper triangular HSS factorization procedure can be similarly derived.) A triangular HSS solution algorithm is given in [23, 24]. The algorithm traverses the HSS tree with depth-first sweeps so that newly computed solution entries can be used to update the right-hand side, or information is propagated to all related nodes of the HSS tree. This algorithm implicitly computes a triangular HSS matrix-vector multiplication by recursion. It is thus not suitable for parallelization.

Here, we provide a new triangular ULV HSS solver which is scalable. Again, we convert the diagonal blocks to identity matrices so as to avoid multiplications of square dense blocks. The main steps are illustrated in Figure 4 and explained briefly as follows.

If node i is a leaf, multiply D_i^{-1} to the block row i so that the diagonal block becomes an identity matrix, and update U_i as $D_i^{-1}U_i$. Compute a full QL factorization of $D_i^{-1}U_i$

$$D_i^{-1}U_i = Q_i \begin{pmatrix} 0 \\ \tilde{U}_i \end{pmatrix} \begin{matrix} m-r \\ r \end{matrix}. \tag{28}$$

Then let $Q_i^T V_i = \begin{pmatrix} \tilde{V}_i \\ \tilde{V}_i \end{pmatrix}$, where the partition follows that of (28). Clearly, the leading $(m-r) \times (m-r)$ subblock of the diagonal block becomes an identity matrix, which is the only nonzero block in those rows and can be eliminated.



(i) *Factorization of diagonal blocks* (ii) *Introduction of zeros into off-diagonal*

(iii) *Restoring diagonal identity* (iv) *Merging remaining blocks and repeating*

Figure 4. A triangular ULV HSS factorization scheme.

If node i is a non-leaf node with its children c_1 and c_2 being leaves, assume the previous operations are applied to both c_1 and c_2. That is,

$$\begin{pmatrix} Q_{c_1}^T D_{c_1}^{-1} & \\ & Q_{c_2}^T D_{c_2}^{-1} \end{pmatrix} \begin{pmatrix} D_{c_1} & \\ U_{c_2} B_{c_2} V_{c_1}^T & D_{c_2} \end{pmatrix} = \begin{pmatrix} I & \\ \operatorname{diag}\left(0,\ \tilde{U}_{c_2} B_{c_2} \tilde{V}_{c_1}^T\right) & I \end{pmatrix}. \tag{29}$$

Then we merge appropriate blocks on the right-hand side of (29) and remove c_1 and c_2 from the HSS tree so that i becomes a leaf with D and U generators

$$D_i = \begin{pmatrix} I & \\ \tilde{U}_{c_2} B_{c_2} \tilde{V}_{c_1}^T & I \end{pmatrix},\ U_i = \begin{pmatrix} \tilde{U}_{c_1} R_{c_1} \\ \tilde{U}_{c_2} R_{c_2} \end{pmatrix}. \tag{30}$$

L is reduced to a smaller matrix. The same operations then apply recursively.

Note that due to the special form of D_i in (30), the intermediate update step $D_p^{-1}\tilde{U}_p$ can be done quickly as

$$D_i^{-1}\tilde{U}_i = \begin{pmatrix} \tilde{U}_{c_1} R_{c_1} \\ \tilde{U}_{c_2}[R_{c_2} - (B_{c_2}\tilde{V}_{c_1}^T)(\tilde{U}_{c_1} R_{c_1})] \end{pmatrix}. \tag{31}$$

The products in (30)–(31) can be carefully organized so that only five multiplications of $r \times r$ matrices are needed.

3. INNER-OUTER HSS PERCONDITIONER

3.1. Improved ULV HSS solution

Due to the improved ULV-type HSS factorization in Section 2.2, the solution procedure with the ULV factor is much simpler than the one in [35]. For example, for $\hat{\mathbf{L}}$ in (10), we consider the solution of the following system by forward substitution:

$$\hat{\mathbf{L}}y = \mathbf{b}.$$

In the process, each node i of the HSS tree is associated with $\mathbf{b}_i$, a piece of $\mathbf{b}$, and y_i, a piece of y. These pieces are defined recursively. First, $\mathbf{b}$ is partitioned into $\mathbf{b}_i$ pieces according to the leaf level L_i sizes. For each leaf i, let

$$\tilde{\mathbf{b}}_i = Q_i^T L_i^{-1} \mathbf{b}_i. \tag{32}$$

Partition $\tilde{\mathbf{b}}_i$ as $\tilde{\mathbf{b}}_i = \begin{pmatrix} \tilde{\mathbf{b}}_{i,1} \\ \tilde{\mathbf{b}}_{i,2} \end{pmatrix} \begin{matrix} m-r \\ r \end{matrix}$, and set

$$\tilde{y}_i = \tilde{\mathbf{b}}_{i,1}.$$

Assume similar operations are also conducted for $j = \mathrm{sib}(i)$. Then for $p = \mathrm{par}(i)$, set $\mathbf{b}_p = \begin{pmatrix} \tilde{\mathbf{b}}_{i,2} \\ \tilde{\mathbf{b}}_{j,2} \end{pmatrix}$ if $i < j$, or $\mathbf{b}_p = \begin{pmatrix} \tilde{\mathbf{b}}_{j,2} \\ \tilde{\mathbf{b}}_{i,2} \end{pmatrix}$ otherwise. Similar operations can then be applied to upper level nodes. These steps proceed until the root node $2k-1$ is reached, where

$$\tilde{y}_{2k-1} = L_{2k-1}^{-1} \mathbf{b}_k. \tag{33}$$

After all these $\tilde{y}_i$ pieces are computed, we traverse the HSS tree top-down. Let $y_{2k-1} = \tilde{y}_{2k-1}$. For a non-leaf node i with children c_1, c_2, partition $\tilde{y}_i$ as $\tilde{y}_i = \begin{pmatrix} \tilde{y}_{i,1} \\ \tilde{y}_{i;2} \end{pmatrix} \begin{matrix} r \\ r \end{matrix}$ and compute

$$y_{c_1} = L_{c_1}^{-T} Q_{c_1} \begin{pmatrix} \tilde{y}_{c_1} \\ \tilde{y}_{i;1} \end{pmatrix}, \; y_{c_2} = L_{c_2}^{-T} Q_{c_2} \begin{pmatrix} \tilde{y}_{c_2} \\ \tilde{y}_{i;2} \end{pmatrix}. \tag{34}$$

When all the nodes are visited, the pieces y_i associated with all the leaves can be assembled into y. See Figure 6 below.

3.2. Triangular ULV HSS solution

We then consider the ULV solution for a lower triangular HSS system

$$Lx = b.$$

Initially, partition b conformably following the sizes of the leaf level diagonal blocks D_i in L.

For each node i of the HSS tree, compute

$$\tilde{b}_i = Q_i^T (D_i^{-1} b_i)$$

Similarly, the structure of D_i in (30) can be used to accelerate the computation of $D_i^{-1} b_i$. Partition $\tilde{b}_i$ as $\tilde{b}_i = \begin{pmatrix} \tilde{b}_{i,1} \\ \tilde{b}_{i,2} \end{pmatrix} \begin{matrix} m-r \\ r \end{matrix}$, and set

$$\tilde{x}_i = \tilde{b}_{i,1}.$$

Then similar to the previous subsection, we form b_p by stacking $\tilde{b}_{i,2}$ and $\tilde{b}_{j,2}$ for $j = \mathrm{sib}(i)$. The process then proceeds, until the root node $2k-1$ is reached, where

$$\tilde{x}_{2k-1} = D_{2k-1}^{-1} b_k.$$

After all solution pieces $\tilde{x}_i$ are computed, we traverse the HSS tree top-down. Let $x_{2k-1} = \tilde{x}_{2k-1}$. For a non-leaf node i with children c_1, c_2, partition $\tilde{x}_i$ as $\tilde{x}_i = \begin{pmatrix} \tilde{x}_{i,1} \\ \tilde{x}_{i;2} \end{pmatrix} \begin{matrix} r \\ r \end{matrix}$ and compute

$$x_{c_1} = Q_{c_1} \begin{pmatrix} \tilde{x}_{c_1} \\ \tilde{x}_{i;1} \end{pmatrix}, \; x_{c_2} = Q_{c_2} \begin{pmatrix} \tilde{x}_{c_2} \\ \tilde{x}_{i;2} \end{pmatrix}. \tag{35}$$

When all the nodes are visited, the pieces x_i associated with all the leaves can be assembled into x.

3.3. Inner-outer HSS solution

With all the new HSS algorithms above, we are ready to present our inner-outer HSS algorithms. In the inner-outer factorization, A is approximated by an HSS form with the HSS tree $\mathcal{T}$, where each diagonal block D_i is approximately factorized as $D_i \approx L_i L_i^T$, and L_i is an inner lower triangular HSS form with the HSS tree $\mathcal{T}_0$. This is illustrated with Figure 5. Then ULV factorizations are computed for both inner and outer HSS forms. Sections 2.3 and 2.2 are used. In the solution stage, we follow an outer solution procedure given by the ULV solution in Section 3.1. In the meantime, for any solution steps involving L_i associated with a leaf i, such as (32), we use the triangular HSS solution in Section 3.2. The solution algorithm is summarized in Table III.

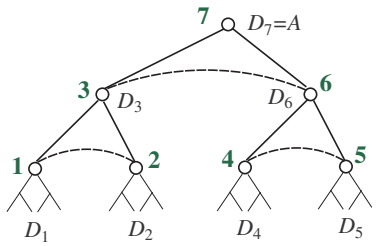


Figure 5. A pictorial illustration of the inner-outer HSS process, where each leaf D_i block of the outer HSS tree (in Figure 1) is approximately factorized into an HSS Cholesky factor as the inner HSS matrix.

In practical implementations of the algorithm as a preconditioner, additional techniques can be utilized. One is that we can traverse the HSS trees levelwise, so that the algorithms can be implemented in parallel. For example, when all the y_i pieces are obtained as in (34), they can be first assembled together at each level. Assume the solution pieces provided at each level l form a vector $\bar{y}^{(l)}$. Then the solution y is formed by collecting all $\bar{y}^{(l)}$ with appropriate permutations (Figure 6). That is, we define

$$y^{(l)} = P_l^T \begin{pmatrix} \bar{y}^{(l)} \\ y^{(l-1)} \end{pmatrix}, \; l = l_{\max}, l_{\max} - 1, \ldots, 2, 1 \tag{36}$$

with $\bar{y}^{(0)} = y_{2k-1}$. Then $y \equiv y^{(0)}$.

Another techniques is that we can use diagonal shifts in the outer factorization step to enhance robustness. (This is not needed for the inner factorizations.) Since D_i has the form in (7) for a

ALGORITHM 1. (*Inner-outer HSS solution*)

`subroutine io_hsssol`(A, b)
Input: A in factorized form, where the outer ULV factor is given by Q_i and L_i, and L_i is given by its inner ULV factor; b: right-hand side (initially, $y \equiv b$)
Output: Solution y
`for` all nodes i at the bottom level $l_{\max}$, partition y into y_i pieces
`for` each leaf i, compute $\tilde{y}_i = Q_i^T \cdot [\texttt{triulvsol}(L_i, b_i)]$
`for` $l = l_{\max} - 1, l_{\max} - 2, \ldots, 1$ `do`

`for` all nodes i at level l `do`

`for` each child c of i, compute $\tilde{b}_c = Q_c^T L_c^{-1} b_c \equiv \begin{pmatrix} \tilde{y}_{c,1} \\ \tilde{y}_{c,2} \end{pmatrix} \begin{matrix} m-r \\ r \end{matrix}$

Let $b_i = \begin{pmatrix} \tilde{y}_{c_1,2} \\ \tilde{y}_{c_2,2} \end{pmatrix}$, $\tilde{y}_{c_1} = \tilde{y}_{c_1,1}$, $\tilde{y}_{c_2} = \tilde{y}_{c_2,1}$

`end for`

`end for`
Compute $\tilde{y}_{2k-1} = L_{2k-1}^{-1} b_{2k-1}$
`for` $l = 0, 1, \ldots, l_{\max} - 1$ `do`

`for` all nodes i at level l `do`

Partition $\tilde{y}_i$ as $\tilde{y}_i = \begin{pmatrix} \tilde{y}_{i,1} \\ \tilde{y}_{i;2} \end{pmatrix} \begin{matrix} r \\ r \end{matrix}$

`for` $j = 1, 2$, compute $y_{c_j} = L_{c_j}^{-T} Q_{c_j} \begin{pmatrix} \tilde{y}_{c_j} \\ \tilde{y}_{i;j} \end{pmatrix}$

`end for`

`end for`
`for` each leaf i, compute $y_{c_j} = \texttt{triulvsol}\left(L_{c_j}^T,\ Q_{c_j} \begin{pmatrix} \tilde{y}_{c_j} \\ \tilde{y}_{i;j} \end{pmatrix}\right)$
`end subroutine`

Table III. An inner-outer HSS solution algorithm, where `triulvsol` represents the solution scheme in Section 3.2.

non-leaf node i in $\mathcal{T}$, we can add the following shift to D_i:

$$s_i = \varepsilon I. \tag{37}$$

Notice that this diagonal shift is only needed for non-leaf nodes in $\mathcal{T}$. In parallel computing, the number of non-leaf nodes may be set to be twice of the number of processors, and the affect on the accuracy is limited.

4. ANALYSIS

We collect all the new HSS algorithms in this work as follows:

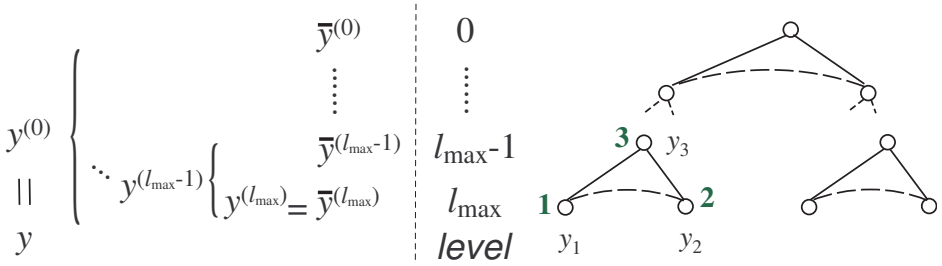


Figure 6. How the solution pieces generated in Algorithm 1 are assembled into the solution y, where the pieces y_i associated with the nodes at each level l of the HSS tree $\mathcal{T}$ are assembled into $\bar{y}^{(l)}$, and then into $y^{(l)}$ as in (36).

- `Direct-HSS`: The new HSS algorithms based on direct HSS construction and ULV operations:
  - `Direct-HSS`(constr): The HSS construction procedure in Section 2.1.
  - `Direct-HSS`(fact): The ULV HSS factorization procedure in Section 2.2.
  - `Direct-HSS`(sol): The ULV HSS solution procedure in Section 3.1.
- `Tri-HSS`: The new HSS algorithms based on triangular HSS operations:
  - `Tri-HSS`(constr): The HSS Cholesky factorization procedure in Section 2.3.
  - `Tri-HSS`(sol): The triangular HSS factorization procedure in Section 2.4 together with solution in Section 3.2.
- `I/O-HSS`: The new inner-outer HSS algorithms:
  - `I/O-HSS`(constr): The procedure with outer `Direct-HSS`(constr) and inner `Tri-HSS`(constr).
  - `I/O-HSS`(fact): The procedure with outer `Direct-HSS`(fact).
  - `I/O-HSS`(sol): The procedure with outer `Direct-HSS`(sol) and inner `Tri-HSS`(sol).

For convenience, we use the following notation in the analysis and the numerical experiments:

- For the outer HSS algorithms, n, r, m, and $\mathcal{T}$ are the matrix size, HSS rank, leaf level diagonal block size, and outer HSS tree, respectively. The number of leaves in $\mathcal{T}$ is k.
- For the inner HSS algorithms, we similarly use the notation n_0, r_0, m_0, $\mathcal{T}_0$, and k_0. Here, $n_0 \equiv m$.
- $\xi^{(1)}$, $\xi^{(2)}$, and $\xi^{(3)}$ are the flop counts of the above three types of algorithms. For example, $\xi^{(2)}_{\text{constr}}$ is the cost of `Tri-HSS`(constr).

4.1. Complexity count: the new HSS Cholesky factorization as an example

As an example, we count the flops of `Tri-HSS`(constr), and then other algorithms can be easily studied. For simplicity, assume the inner HSS tree $\mathcal{T}_0$ is a perfect binary tree, and $m_0 = O(r_0)$. Since there are k_0 leaves, $k_0 m_0 = n_0$. We also assume the root of $\mathcal{T}_0$ is at level 0 so that there are totally about $\log_2 k_0$ levels. Our complexity analysis uses the flop counts of some basic matrix operations in Table IV.

Also two useful formulas are needed:

$$\sum_{i \text{ at level } l} n_i = \sum_{j=1}^{2^l} (k_0 - j\frac{k_0}{2^l}) m_0 = \frac{1}{2} k_0 m_0 (2^l - 1), \qquad \sum_{i \text{ at level } l} s_i = \sum_{j=1}^{l} \binom{l}{j} = 2^l - 1, \tag{38}$$

Operation	Flops
Cholesky factorization of an $m_0 \times m_0$ matrix	$\frac{1}{3}m_0^3$
Product of an $m_0 \times q$ matrix and an $q \times r_0$ matrix	$2m_0qr_0$
RRQR factorization of an $m_0 \times q$ matrix with rank r_0	$4m_0qr_0 - 2r_0^2(m_0+q) + \frac{4}{3}r_0^3$
Full QR factorization of an $m_0 \times r_0$ tall matrix ($m_0 > r_0$)	$2r_0^2\left(m_0 - \frac{r_0}{3}\right)$
Product of the Q factor and an $m \times q$ matrix	$2r_0q(2m_0 - r_0)$
Solution of an order m_0 triangular system $Lx = b$	m_0^2

Table IV. Flop counts (leading terms only) of some basic matrix operations, which can be found in, say, [10, 13] or can be derived based on those. The RRQR factorization in our work is based on the modified Gram-Schmidt process with column pivoting [13].

where n_i is the column size of $\mathbf{R}|_{t_i \times t_i^r}$, and s_i is the cardinality of $\mathcal{V}_i$ in (16). The first formula is because there are 2^l nodes at each level l, with the n_i values $(k_0 - j\frac{k_0}{2^l})m_0$, $j = 1, 2, \ldots, 2^l$. The second formula is derived in [33].

(a) *Elimination cost.* For each leaf i, the elimination step (17) costs $\frac{m_0^3}{3} + m_0^2 n_i$ flops. Thus, the total elimination cost for all leaves is

$$\sum_{i:\text{ leaf}} \left(\frac{m_0^3}{3} + m_0^2 n_i\right) = \frac{k_0 m_0^3}{3} + m_0^2 \sum_{i \text{ at level } \log_2 k_0} n_i = \frac{1}{2}k_0^2 m_0^3 + \frac{1}{3}k_0 m_0^3.$$

(b) *Compression cost.* For each leaf i, the RRQR step (18) is applied to an $m_0 \times n_i$ block to get U_i. For each non-leaf node i at level $l > 0$, the RRQR step (20) is applied to a $2r_0 \times n_i$ block to get R_{c_1}, R_{c_2}. The total for all HSS block row compression is then

$$\begin{aligned}&\sum_{i:\text{ leaf}} (4m_0 n_i r_0 - 2r_0^2(m_0+n_i) + \frac{4}{3}r_0^3) + \sum_{l=1}^{\log_2 k_0 - 1} \sum_{i \text{ at level } l} [4(2r_0)n_i r_0 - 2r_0^2(2r_0+n_i) + \frac{4}{3}r_0^3] \\ &\approx 2r_0 k_0^2 m_0^2 + 2r_0^2 k_0^2 m_0 - 2r_0^2 k m_0 + \frac{4}{3}k_0 r_0^3,\end{aligned}$$

where (38) is used.

For each leaf i, to compute V_i with (26)–(27), we need the compression of a size $r_0 \times m_0$ block and the multiplication of a $m_0 \times r_0$ and an $r_0 \times s_i r_0$ matrix. For each non-leaf node i at level $l > 0$, the RRQR step (21) is applied to an $2r_0 \times s_i r_0$ block to get the W generators. The total cost at all HSS block column compression is then

$$\begin{aligned}&\sum_{i:\text{ leaf}} [4m_0 r_0^2 - 2r^2(m_0+r_0) + \frac{4}{3}r_0^3 + 2(s_i r_0)r_0 m_0] + \sum_{l=1}^{\log_2 k - 1} \sum_{i \text{ at level } l} [8(s_i r_0)r_0^2 - 2r_0^2(2r_0 + s_i r_0) + \frac{4}{3}r_0^3] \\ &\approx 4k_0 r_0^2 m_0 - \frac{8}{3}r_0^3 m_0 + \frac{10}{3}k_0 r_0^3.\end{aligned}$$

(c) *Schur complement computation cost.* Similarly, we can count the cost of (23) and (25) for each leaf i as $\left(m_0 r_0 + 3r_0^2\right) k_0^2 m_0$.

Therefore, the total factorization cost is roughly

$$\xi_{\text{constr}}^{(2)} \approx \left(\frac{1}{2}m_0 + 3r_0 + 5\frac{r_0^2}{m_0}\right) n_0^2. \tag{39}$$

This formula can be used to minimize $\xi_{\text{constr}}^{(2)}$. It can be easily shown that the optimal cost is

$$\xi_{\text{constr}}^{(2)} \approx \left(3 + \sqrt{10}\right) r_0 n_0^2, \tag{40}$$

which is achieved when $m_0 = \sqrt{10}r_0$. If we choose $m_0 = 2r_0$ as often used, $\xi^{(2)}_{\text{constr}} \approx \frac{13}{2} r_0 n_0^2$, which is slightly larger than the optimal cost. On the other hand, the original version in [36] with $m_0 = 2r_0$ costs at least $11\tilde{r}_0 n_0^2$ flops [33], where $\tilde{r}_0$ may be as large as $2r_0$.

Similarly, the costs of the new ULV triangular HSS factorization and solution are

$$\xi^{(2)}_{\text{fact}} \approx r_0 \left(m_0 + 2r_0 + \frac{50}{3}\frac{r_0^2}{m_0} \right) n_0 \quad \text{and} \quad \xi^{(2)}_{\text{sol}} \approx \left(m_0 + 4r_0 + 8\frac{r_0^2}{m_0} \right) n_0,$$

respectively.

4.2. Complexity of the inner-outer HSS algorithm and other algorithms

The complexity of other algorithms involved can be counted similarly. See Table V.

Type	Operation	Complexity
`Direct-HSS`	Construction	$2\left(1+\frac{r}{m}\right) rn^2$
	ULV factorization	$\left(\frac{1}{3}m^2 + mr + 2r^2 + \frac{38}{3}\frac{r^3}{m}\right) n$
	ULV solution	$\left(m + 4r + 8\frac{r^2}{m}\right) n$
`Tri-HSS`	Factorization	$\left(\frac{1}{2}m_0 + 3r_0 + 5\frac{r_0^2}{m_0}\right) n_0^2$
	ULV factorization	$r_0\left(m_0 + 2r_0 + \frac{50}{3}\frac{r_0^2}{m_0}\right) n_0$
	ULV solution	$\left(m_0 + 4r_0 + 8\frac{r_0^2}{m_0}\right) n_0$
`I/O-HSS`	Construction	$2\left(1+\frac{r}{m}\right) rn^2 + \left(\frac{1}{2}m_0 + 3r_0 + 5\frac{r_0^2}{m_0}\right) mn$
	Factorization	$\left[r_0\left(m_0 + 2r_0 + \frac{50}{3}\frac{r_0^2}{m_0}\right) + r\left(m_0 + 4r_0 + 8\frac{r_0^2}{m_0}\right) + 2r^2 + \frac{38}{3}\frac{r^3}{m}\right] n$
	Solution	$\left(m_0 + 4r_0 + 8\frac{r_0^2}{m_0} + 4r + 8\frac{r^2}{m}\right) n$

Table V. Flop counts (leading terms only) of the HSS algorithms proposed in this work.

In particular, with certain values of m and m_0, we get some sample counts in Table VI. It can then be verified that the new HSS algorithms proposed in this work are generally much faster than similar existing ones. See, e.g., Section 2.2.

Type	Operation	Complexity
`Direct-HSS`	Construction	$3rn^2$
	ULV factorization	$\frac{35}{3}r^2 n$
	ULV solution	$10rn$
`Tri-HSS`	Factorization	$\frac{13}{2}r_0 n_0^2$
	ULV factorization	$\frac{37}{3}r_0^2 n_0$
	ULV solution	$10r_0 n_0$
`I/O-HSS`	Construction	$\frac{23}{8}rn^2$
	Factorization	$\frac{73}{3}r^2 n$
	Solution	$14rn$

Table VI. Flop counts (leading terms only) as in Table V, with $r = r_0,\ m_0 = 2r_0,\ m = \frac{n}{4}$. (These assumptions may not be satisfied in the numerical experiments in the next section.)

4.3. Approximation errors

The off-diagonal compression introduces errors into the approximations, which can be roughly measured by the following result.

Proposition 4.1. *For a given SPD matrix A, let τ be the relative tolerance in the off-diagonal compression (by truncated SVD). Then after the outer HSS construction for A, the HSS approximation $\tilde{A}$ satisfies*

$$\frac{||A-\tilde{A}||_2}{||A||_2} \leq \tau O(\log n).$$

Sketch of the proof. The proof uses the result that the U, V generators have orthonormal columns. The HSS tree $\mathcal{T}$ has $O(\log n)$ levels, and the approximation error accumulates additively when the compression level moves bottom-up along $\mathcal{T}$. ∎

This proposition can be applied to HSS constructions with a given relative tolerance τ for the off-diagonal singular values. For an off-diagonal block B, the singular values σ_i that satisfy $\sigma_i < \sigma_1\tau$ are dropped, where σ_1 is the largest singular value of B. The number of remaining singular values is the numerical rank of B. The proposition gives a bound for the error introduced in the approximation of A.

The error analysis for the HSS Cholesky factorization is less straightforward. As in [18], the compression of $\mathbf{R}|_{t_i\times t_i^r} = D_i^{-T}A^{(i-1)}|_{t_i\times t_i^r}$ in (17) with an absolute error τ_0 introduces an error of $e = O\left(\|D_1\|_2\,\tau_0\right) = O\left(\sqrt{\|A\|_2}\tau_0\right)$ into $A^{(i-1)}|_{t_i\times t_i^r}$. (If an relative tolerance is used, this becomes more complicated.) With hierarchical compression, the error introduced into the original off-diagonal block $A|_{t_i\times t_i^r}$ can be as large as $O(\log n)\cdot e$. Moreover, the compression of $\mathbf{R}|_{t_i\times t_i^r}$ adds an error of $\tilde{e} = O(\tau_0^2)$ to the Schur complement S_i, which continues to be factorized approximately, and $\tilde{e}$ may be magnified by $O(n)$. Such analysis may be too pessimistic, and the detailed study is not the main focus of this work. In practice, the error is often well controlled. Since our objective is preconditioning, a lower accuracy is allowed in the inner HSS Cholesky factorization. The feasibility is verified by the numerical experiments.

5. NUMERICAL EXPERIMENTS

The inner-outer HSS algorithms and others are implemented in Matlab, and are tested on various examples. For convenience, we list additional notation as follows, other than that summarized at the beginning of Section 4:

Notation	Meaning	Notation	Meaning
$\kappa_2(A)$	2-norm condition number of A	e	Relative residual $\frac{\|\|A_1x-b\|\|_2}{\|\|b\|\|_2}$
ε	Size of the shift in (37)	N_{iter}	Number of preconditioned conjugate gradient (PCG) iterations
N_{shift}	Number of times when shifts (37) are used to avoid breakdown in `I/O-HSS`	ξ_{iter}	Total PCG cost (flops)

Example 1. Let

$$A_0 = \left(\sqrt{|x_i - x_j|}\right)_{n\times n}, \tag{41}$$

where the points $x_i = \cos((2i+1)\pi/2n)$ are the zeros of the nth Chebyshev polynomial, as used in [7, 9].

We demonstrate the performance of the algorithms for a linear system with an SPD coefficient matrix:

$$A_1 x = b, \text{ with } A_1 = A_0^T A_0 + 2I. \tag{42}$$

(The $2I$ diagonal matrix ensures that A_1 remains positive definite in our numerical tests with the presence of numerical errors on the computer. This nearly has no effect on the off-diagonal numerical ranks in our factorizations, and does not change the nature of the tests.) A_1 has size n ranging from 500 to 160,000, and b is generated with random exact x. Several aspects are tested.

First, both `Tri-HSS` and `Direct-HSS` algorithms proposed here are faster than the original versions in [7, 36]. For example, for $n = 4000$, $m_0 = 50$, and compression accuracy $\tau_0 = 10^{-6}$, our `Tri-HSS`(constr) and `Tri-HSS`(sol) (directly applied to A_1) cost about $8.00E8$ and $6.09E5$ flops, respectively. While the original versions in [36] cost $1.24E9$ and $6.11E5$ flops, respectively.

In the second test, we use `I/O-HSS` algorithms to directly solve the system (42). Different inner and outer off-diagonal compression accuracies are used, which are τ_0 and τ, respectively. No diagonal shift is involved here, or $\varepsilon = 0$ in (37). The performance is demonstrated in Table VII. It can be seen that, for reasonable accuracy, `I/O-HSS`(constr) has about $O(n^2)$ complexity, and `I/O-HSS`(fact) and `I/O-HSS`(sol) have nearly $O(n)$ complexity.

n	500	1000	2000	4000	8000	16000
$\xi^{(3)}_{\text{constr}}$	$4.84E6$	$2.07E7$	$8.56E7$	$3.47E8$	$1.42E9$	$5.49E9$
$\xi^{(3)}_{\text{fact}}$	$3.30E5$	$6.88E5$	$1.40E6$	$2.79E6$	$5.77E6$	$1.06E7$
$\xi^{(3)}_{\text{sol}}$	$7.91E4$	$1.70E5$	$3.51E5$	$7.07E5$	$1.43E6$	$2.78E6$
e	$1.68E-8$	$3.06E-8$	$5.68E-8$	$2.94E-8$	$5.32E-8$	$4.67E-8$

Table VII. *Example 1:* Flop counts $\xi^{(3)}$ and relative residuals e of the inner-outer HSS algorithms for solving (42), where $m = 1000$, $m_0 = 50$, $\varepsilon = 0$, $\tau = 10^{-8}$, $\tau_0 = 10^{-6}$.

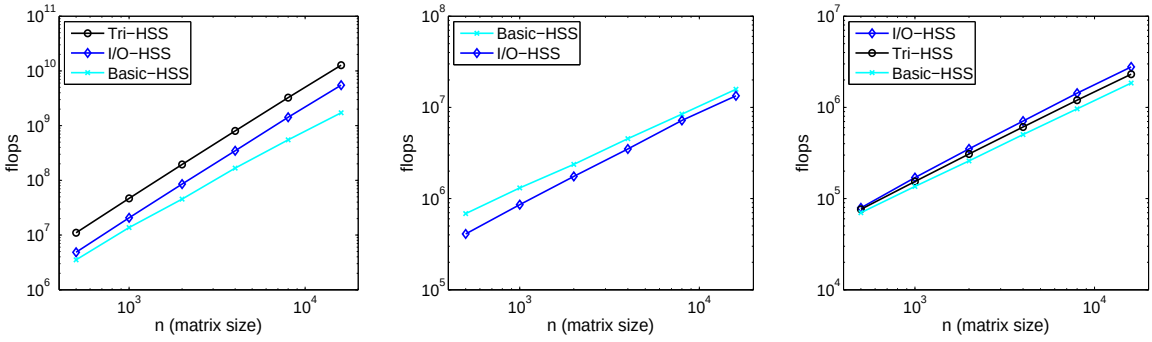
The `I/O-HSS` algorithms are also compared with `Tri-HSS` and `Direct-HSS` algorithms, when they are directly applied to A_1 to get about the same accuracy. The same set of parameters m_0, τ_0 are used by `Tri-HSS` and `Direct-HSS`. (Note that the ULV factorization in `Tri-HSS` is intended for parallel implementation and may be slower than the sequential triangular solver in [24]. Thus, here we use the one in [24], which does not use a factorization stage.) From Figure 7, we observe that the ξ_{sol} costs of all three types of algorithms are comparable to each other, while `I/O-HSS` can take into consideration both the scalability and the robustness.

In the third test, we examine the effectiveness of `I/O-HSS` as a preconditioner in PCG for A_1 and also

$$A_2 = (A_0^T A_0)^T (A_0^T A_0) + 2I.$$

The matrices A_1 and A_2 have 2-norm condition numbers $\kappa_2 = 5.4e6$ and $\kappa_2 = 6.2e13$, respectively. In `I/O-HSS`(constr), we only keep very few columns in the off-diagonal compression, or we manually set the numerical rank r to be a small number (and set $r_0 = r$). Thus, it is very efficient to obtain and use the preconditioner. The results are shown in Table VIII. The details of the iteration processes are shown in Figure 8. We can see that, PCG with `I/O-HSS` converges much faster and costs far less than with block diagonal preconditioning.

Lastly, we test the performance of the inner-outer preconditioning by using lower accuracies in the inner `Tri-HSS` factorization. For A_1 in Table VIII, we use outer HSS rank bound $r = 7$, and the inner HSS rank bound r_0 varies from 3 to 7. See Table 9 for the performance. We see that fast

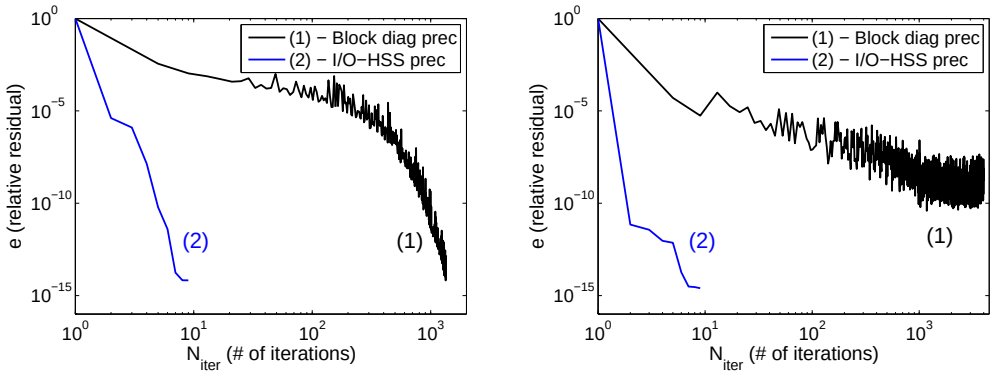


(i) ξ_{constr} (ii) ξ_{fact} (iii) ξ_{sol}

Figure 7. *Example 1:* Flop counts of `I/O-HSS`, `Tri-HSS`, and `Direct-HSS` for solving (42), where $m = 1000$, $m_0 = 50$, $\tau = 10^{-8}$, $\tau_0 = 10^{-6}$, and $\varepsilon = 0$ in (37).

Matrix A_1 ($\kappa_2 = 5.4E6$)				Matrix A_2 ($\kappa_2 = 6.2E13$)			
	N_{iter}	ξ_{iter}	e		N_{iter}	ξ_{iter}	e
Block diagonal	1354	$4.39E10$	$9.10E-15$	Block diagonal	3793	$1.23E11$	$1.64E-9$
I/O-HSS	9	$6.57E8$	$6.63E-15$	I/O-HSS	9	$6.57E8$	$2.48E-15$

Table VIII. *Example 1:* Performance of PCG with block diagonal preconditioning and `I/O-HSS` preconditioning for A_1 and A_2 with sizes $n = 4000$, where $m = 1000$, $m_0 = 50$, $r = 7$, and $\varepsilon = 0$ in (37).



(i) A_1 (ii) A_2

Figure 8. *Example 1:* Convergence of PCG with block diagonal preconditioning and `I/O-HSS` preconditioning for A_1 and A_2 with sizes $n = 4000$, where $m = 1000$, $m_0 = 50$, $r = 7$, and $\varepsilon = 0$.

convergence is achieved for all these r_0 values. In fact, the convergence behaviors for $r_0 = 4, 5, 6, 7$ are almost the same.

EXAMPLE 2. Then we consider a Toeplitz matrix A generated by a Gaussian radial basis function, as

$$v(t_i) = e^{-\delta^2 t_i^2},$$

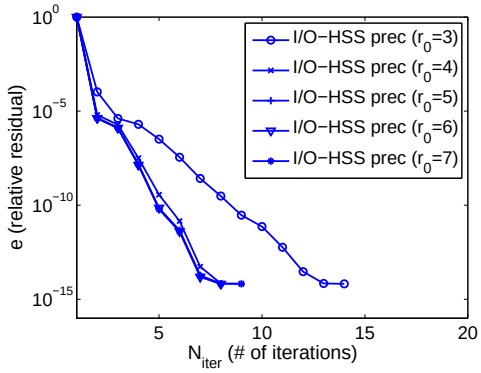


Figure 9. *Example 1:* Convergence of PCG with `I/O-HSS` preconditioning for A_1 in Table VIII with different inner rank bounds r_0.

where the $t_i = i - 1$. (The Toeplitz structure is not our focus and is ignored.) The matrix is highly ill conditioned for small δ. In fact, its condition number is about $e^{\pi^2/(4\delta^2)}$ [6].

We consider the matrix with size $n = 16000$. If regular `Direct-HSS` methods are directly applied with a low accuracy τ, they actually break down since some intermediate diagonal blocks (corresponding to non-leaf nodes of $\mathcal{T}$) fail to be positive definite. Then we may use shifts as in (37). The number of times (nodes) of such breakdown or shifting is denoted N_{shift}. However, `I/O-HSS` can significantly enhance the robustness. The larger m is, the more robust the method is. See Figure IX. If no inner factorization is used, then too many shifting steps are needed, and the preconditioner fails to be useful in practice (Table X). PCG with `I/O-HSS` preconditioning also costs much less than with block diagonal preconditioning. The detailed convergence is shown in Figure 10.

	`Direct-HSS`	`I/O-HSS`							
m	50 ($m = m_0$)	250	500	750	1000	1250	1500	1750	2000
$\xi^{(3)}_{\text{constr}}$	$6.11E9$	$5.42E9$	$5.49E9$	$5.64E9$	$5.79E9$	$5.95E9$	$6.11E9$	$6.33E9$	$6.48E9$
$\xi^{(3)}_{\text{fact}}$	$2.98E7$	$1.88E7$	$1.95E7$	$1.98E7$	$1.99E7$	$2.01E7$	$2.01E7$	$2.02E7$	$2.02E7$
$\xi^{(3)}_{\text{sol}}$	$2.62E6$	$1.38E6$	$1.33E6$	$1.31E6$	$1.30E6$	$1.30E6$	$1.29E6$	$1.29E6$	$1.29E6$
N_{shift}	160	32	16	10	8	6	5	4	4

Table IX. *Example 2:* Costs $\xi^{(3)}$ of `I/O-HSS` algorithms, and the number N_{shift} of shifts, where of PCG with `I/O-HSS` preconditioning, where $m_0 = 50$, $r = 10$, and $\varepsilon = 10^{-4}$ in (37).

	N_{iter}	ξ_{iter}	e
Block diagonal	124	$7.00E10$	$4.16E{-}14$
`Direct-HSS`	*Failure*		
`I/O-HSS` ($r = 10$)	16	$1.40E10$	$3.45E{-}16$

Table X. *Example 2:* Convergence of PCG with block diagonal preconditioning and `I/O-HSS` preconditioning, where $m_0 = 50$, $r = 10$, and $\varepsilon = 10^{-4}$ in (37).

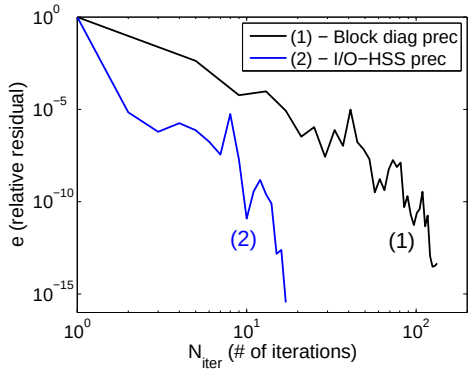


Figure 10. *Example 2:* Convergence of PCG with block diagonal preconditioning and `I/O-HSS` preconditioning, where $m_0 = 50$, $r = 10$, and $\varepsilon = 10^{-4}$ in (37).

EXAMPLE 3. Next, we consider the dense Schur complements in the direct factorization of sparse matrices arising from the discretization of the following two-dimensional linear elasticity equation:

$$-(\mu\Delta\mathbf{u} + (\lambda+\mu)\nabla\nabla\cdot\mathbf{u}) = \mathbf{f} \text{ in } \Omega = (0,1)\times(0,1),$$

where λ and μ are the Lamé constants, $\mathbf{u}$ is the displacement vector field, and a Dirichlet boundary condition is used.

It is well known that the discretized matrix is ill conditioned when the ratio λ/μ is large, which is referred to be the incompressible limit. As in [18, 36], here we consider a matrix A which is the largest Schur complement in the factorization of the discretized matrix after nested dissection reordering. As illustrated in Table XI, when λ/μ increases, the 2-norm condition number κ_2 increases, and the number of iterations of PCG with block diagonal preconditioning increases significantly. But with `I/O-HSS` preconditioning, the convergence is fast for all λ/μ. Here we keep the numerical ranks to be at most $r = 5$. The total PCG costs compare similarly to the previous examples.

λ/μ		1	10^2	10^4	10^6	10^8	10^{10}
κ_2		$1.71E3$	$2.17E4$	$2.02E6$	$2.02E8$	$2.02E10$	$2.02E12$
Block diagonal	N_{iter}	127	117	136	449	1137	1512
	e	$9.81E-15$	$7.63E-15$	$9.89E-15$	$9.82E-15$	$1.02E-14$	$9.90E-15$
`I/O-HSS`	N_{iter}	23	25	37	33	34	35
	e	$2.35E-15$	$6.42E-15$	$2.07E-15$	$7.54E-15$	$8.30E-15$	$3.57E-15$
	ε	10^{-1}	10^{-1}	10^{-1}	10^{-3}	10^{-5}	10^{-7}

Table XI. *Example 3:* Convergence of PCG with block diagonal preconditioning and `I/O-HSS` preconditioning, where $n = 2002$, $m = 500$, $m_0 = 20$, $r = 5$, and $N_{\text{shift}} = 2$.

REMARK 1. In Table XI, if the small shifts ε are all replaced by 10^{-1}, the convergence is nearly unchanged. In fact, in all these tests, and the effect of shifting on the approximation is limited.

EXAMPLE 4. Finally, we demonstrate the performance of `I/O-HSS` as a preconditioner for A without rapidly decaying off-diagonal singular values. A is constructed as follows. Take an SPD discretized matrix $\mathcal{A}$ defined on a 3D tetrahedral finite element mesh, as generated by the routine

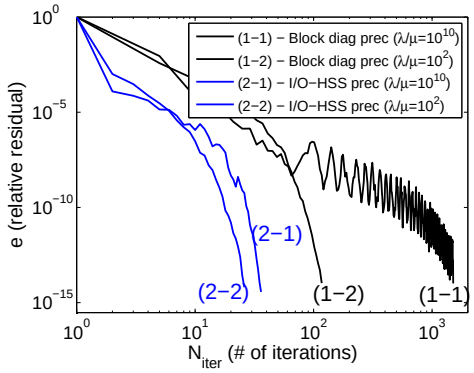


Figure 11. *Example 3:* Convergence of PCG with block diagonal preconditioning and `I/O-HSS` preconditioning, where the parameters are given in Table XI.

`grid3dt` in [11]. Similar to the previous example, let S be the largest Schur complement in the factorization of $\mathcal{A}$ after nested dissection reordering. S corresponds to the largest 2D separator and generally do not have the low-rank property for a small relative tolerance τ. Then let $A = S^3$. (Note: we use S^3 in order for the decay of the off-diagonal singular values of A to be sufficiently slow and for A to be ill conditioned.) Such matrix powers are frequently used in matrix functions.

As pointed out in [36], HSS factorization methods have a good potential to work as effective preconditioners, even if the decay of the off-diagonal singular values is not rapid. In fact, the HSS ranks of A for different relative tolerances are shown in Table XII. The ranks are relatively high even for relatively large tolerances. Also, $\kappa_2(A) = 5.790 \times 10^7$.

τ	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}
HSS rank	104	144	196	256	305	356

Table XII. *Example 4:* HSS ranks of A of order $n = 2500$ for different relative tolerances τ.

However, by specifying a small numerical rank r, `I/O-HSS` can still work as an effective preconditioner. For different parameters, the convergence of PCG with `I/O-HSS` preconditioning and block diagonal preconditioning is shown in Table XIII. With `I/O-HSS`, we observe similar convergence behaviors when different inner HSS ranks r_0 are used. In particular, the detailed convergence for $r_0 = 23$ is shown in Figure 12. Clearly, with a small inner HSS rank $r_0 = 23$ and outer HSS rank $r = 25$ in `I/O-HSS`, PCG converges quickly. To reach the relative residual $e = 10^{-14}$, PCG needs 144 steps with `I/O-HSS` preconditioning, while 746 steps with block diagonal preconditioning.

Preconditioner	I/O-HSS								Block diagonal
r_0	18	19	20	21	22	23	24	25	
N_{iter}	167	160	154	150	147	144	143	142	746
ξ_{iter}	$2.94E9$	$2.87E9$	$2.82E9$	$2.80E9$	$2.79E9$	$2.78E9$	$2.80E9$	$2.82E9$	$9.37E9$

Table XIII. *Example 4:* Numbers of iterations N_{iter} and total costs (flops) ξ_{iter} for PCG to reach an accuracy of $e = 10^{-14}$, where in `I/O-HSS`, $n = 2500$, $m = 625$, $m_0 = 25$, $r = 25$, $\varepsilon = 0.2$, and $N_{\text{shift}} = 2$, and in block diagonal preconditioning, the diagonal block sizes are 25.

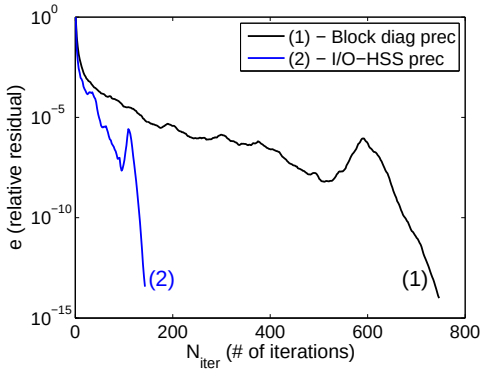


Figure 12. *Example 4:* Convergence of PCG with block diagonal preconditioning and `I/O-HSS` preconditioning, where $r_0 = 23$ and the rest parameters are the same as in Table XIII.

6. CONCLUSIONS

This work presents a robust and scalable inner-outer HSS preconditioner based on a series of new HSS algorithms which improve the efficiency or scalability of existing ones. Systematic complexity analysis and an error bound are given. The performance of the inner-outer preconditioner is illustrated with some ill-conditioned numerical problems. Due to the hierarchical structured operations, the algorithms have nice data locality and good potential for parallel implementations. This work also gives a practical way of developing inner-outer preconditioners using direct rank structured factorizations (other than iterative methods). The method is useful for dealing with dense intermediate matrices in sparse preconditioning. In our future work, we expect to analyze the dependence of the convergence on the tolerance, and to design a parallel Fortran code based on the scalable HSS package from [32].

ACKNOWLEDGEMENTS

The author is very grateful to Ming Gu and Shen Wang for some useful discussions and to Zhiqiang Cai for a test example. The author would also like to thank the anonymous referees for their valuable suggestions.The research of Jianlin Xia was supported in part by NSF grants DMS-1115572 and CHE-0957024.

REFERENCES

1. Bai Z-Z, Benzi M, Chen F. Modified HSS iteration methods for a class of complex symmetric linear systems. *Computing* 2010; **87**:93–111.
2. Bebendorf M, Hackbusch W. Stabilized rounded addition of hierarchical matrices. *Numerical Linear Algebra with Applications* 2007; **14**:407–423.
3. Benzi M, Cullum JK, Tuma M. Robust approximate inverse preconditioning for the conjugate gradient method. *SIAM Journal on Scientific Computing* 2000; **22**:1318–1332.
4. Benzi M, Tuma M. A robust incomplete factorization preconditioner for positive definite matrices. *Numerical Linear Algebra with Applications* 2003; **10**:385–400.
5. Börm S, Hackbusch W. Data-sparse approximation by adaptive $\mathcal{H}^2$-matrices. *Computing* 2002; **69**:1–35.

6. Boyd JP, Gildersleeve KW. Numerical experiments on the condition number of the interpolation matrices for radial basis functions. *Applied Numerical Mathematics* 2011; **61**:443–459.
7. Chandrasekaran S, Dewilde P, Gu N, Lyons W, Pals T. A fast solver for HSS representations via sparse matrices. *SIAM Journal on Matrix Analysis and Applications* 2006; **29**:67–81.
8. 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.
9. Chandrasekaran S, Gu N, Lyons W. A fast adaptive solver for hierarchically semiseparable representations. *CALCOLO* 2005; **42**:171–185.
10. Demmel J. *Applied Numerical Linear Algebra*. SIAM: Philadelphia, PA, 1997.
11. Gilbert JR, Teng S-H. MESHPART, A Matlab Mesh Partitioning and Graph Separator Toolbox, http://aton.cerfacs.fr/algor/Softs/MESHPART/.
12. Gohberg I, Kailath T, Koltracht I. Linear complexity algorithms for semiseparable matrices. *Integral Equations and Operator Theory* 1985; **8**:780–804.
13. Golub G, Loan CV. *Matrix Computations*. The John Hopkins University Press, 1989.
14. Golub GH, Ye Q. Inexact preconditioned conjugate gradient method with inner-outer iteration. *SIAM Journal on Scientific Computing* 1999; **21**:1305–1320.
15. Grasedyck L, Le Borne S, Kriemann R. Parallel blackbox H-LU preconditioning for elliptic boundary value problems. *Computing and Visualization in Science* 2008; **11**:273–291.
16. Grasedyck L, Kriemann R, Le Borne S. Domain decomposition based $\mathcal{H}$-LU preconditioning. *Numerische Mathematik* 2009; **112**:565–600.
17. Greengard L, Rokhlin V. A fast algorithm for particle simulations. *Journal of Computational Physics* 1987; **73**:325–348.
18. Gu M, Li XS, Vassilevski P. Direction-preserving and Schur-monotonic semiseparable approximations of symmetric positive definite matrices. *SIAM Journal on Matrix Analysis and Applications* 2010; **31**:2650–2664.
19. Hackbusch W. A Sparse matrix arithmetic based on $\mathcal{H}$-matrices. Part I: introduction to $\mathcal{H}$-matrices. *Computing* 1999; **62**:89–108.
20. Hackbusch W, Khoromskij BN. A sparse $\mathcal{H}$-matrix arithmetic. Part-II: Application to multi-dimensional problems. *Computing* 2000; **64**:21–47.
21. Kaporin IE. High quality preconditioning of a general symmetric positive definite matrix based on its $U^TU + U^TR + R^TU$-decomposition. *Numerical Linear Algebra with Applications* 1998; **5**:483–509.
22. Lin L, Lu J, Ying L. Fast construction of hierarchical matrix representation from matrix-vector multiplication. *Journal of Computational Physics* 2001; **230**: 4071–4087.
23. Lyons W. *Fast Algorithms with Applications to PDEs*. PhD thesis, University of California, Santa Barbara, June. 2005.
24. Lyons W, Chandrasekaran S, Gu M. Fast LU decomposition for operators with hierarchically semiseparable structure. *UCSB Technical Report*, 2005.
25. Manteuffel T. An incomplete factorization technique for positive definite linear systems. *Mathematics of Computation* 1980; **34**:473–497.
26. Martinsson PG. A fast randomized algorithm for computing a hierarchically semi-separable representation of a matrix. Submitted, http://amath.colorado.edu/faculty/martinss/Pubs/2010_randomhudson.pdf.
27. Meijerink JA, van der Vorst HA. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Mathematics of Computation* 1977; **31**:148–162.
28. Rokhlin V. Rapid solution of integral equations of classical potential theory. *Journal of Computational Physics* 1985; **60**:187–207.
29. Saad Y. A flexible inner-outer preconditioned GMRES algorithm. *SIAM Journal on Scientific Computing* 1993; **14**:461–469.
30. Sheng Z, Dewilde P, van der Meijs N. Iterative solution methods based on the hierarchically semi-separable representation. In *Proceedings of the 17th annual workshop on Circuits, Systems and Signal Processing (ProRISC)* 2006; Veldhoven (NL):343–349.
31. Sheng Z, Dewilde P, Chandrasekaran S. Algorithms to solve hierarchically semi-separable systems. *Operator Theory: Advances and Applications* 2007; Birkhauser Basel, **176**:255–294.
32. Wang S, Li XS, Xia J, Situ Y, de Hoop MV. Efficient scalable algorithms for hierarchically semiseparable matrices. Preprint, 2011, http://www.math.purdue.edu/~xiaj/work/parhss.pdf.
33. Xia J. On the complexity of some hierarchical structured matrices. *SIAM Journal on Matrix Analysis and Applications*. Submitted, 2011, http://www.math.purdue.edu/~xiaj/work/hsscost.pdf.
34. Xia J, Chandrasekaran S, Gu M, Li XS. Superfast multifrontal method for large structured linear systems of equations. *SIAM Journal on Matrix Analysis and Applications* 2009; **31**:1382–1411.
35. Xia J, Chandrasekaran S, Gu M, Li XS. Fast algorithms for hierarchically semiseparable matrices. *Numerical Linear Algebra with Applications* 2010; **17**:953–976.
36. Xia J, Gu M. Robust approximate Cholesky factorization of rank-structured symmetric positive definite matrices. *SIAM Journal on Matrix Analysis and Applications* 2010; **31**:2899–2920.