*IMA Journal of Numerical Analysis* (2026) **00**, 1–23
https://doi.org/10.1093/imanum/drag030

# Linear complexity $\mathscr{H}^2$ direct solver for fine-grained parallel architectures

Wajih Boukaram* and David Keyes
*Applied Mathematics and Computational Science, King Abdullah University of Science and Technology, 4700 KAUST, 23955-6900 Thuwal, KSA*
*Corresponding author: wajih.boukaram@kaust.edu.sa

Xiaoye Li and Yang Liu
*Applied Mathematics and Computational Research Division, Lawrence Berkeley National Laboratory, 1 Cyclotron Road, MS 50A-3111, 94710 Berkeley, CA, USA*

and

George Turkiyyah
*Applied Mathematics and Computational Science, King Abdullah University of Science and Technology, 4700 KAUST, 23955-6900 Thuwal, KSA*

[Received on 11 September 2025; revised on 19 February 2026]

Dedicated to the memory of Nicholas J. Higham.

We present factorization and solution phases for a new linear complexity direct solver designed for concurrent batch operations on fine-grained parallel architectures, for matrices amenable to hierarchical representation. We focus on the strong-admissibility-based $\mathscr{H}^2$ format, where strong recursive skeletonization factorization compresses remote interactions. We build upon previous implementations of $\mathscr{H}^2$ matrix construction for efficient factorization and solution algorithm design, which are illustrated graphically in stepwise detail. The algorithms are 'blackbox' in the sense that the only inputs are the matrix and right-hand side, without analytical or geometrical information about the origin of the system. We demonstrate linear complexity scaling in both time and memory on four representative families of dense matrices up to one million in size. Parallel scaling up to 16 threads is enabled by a multi-level matrix graph coloring and avoidance of dynamic memory allocations thanks to prefix-sum memory management. An experimental backward error analysis is included. We break down the timings of different phases, identify phases that are memory-bandwidth limited, and discuss alternatives for phases that may be sensitive to the trend to employ lower precisions for performance.

*Keywords*: linear complexity direct linear solver; hierarchically low-rank matrices; fine-grained concurrent implementation.

## 1. Introduction

Hierarchically off-diagonal low-rank structure has been used to develop fast algebraic solvers for many systems, including various structured matrices, e.g., Toeplitz and Cauchy (Chandrasekaran *et al.*, 2008), as well as broader problems, such as integral equation (IE) methods for acoustics, electromagnetics (Hackbusch, 1999; Bebendorf & Hackbusch, 2003; Börm *et al.*, 2003), differential equation-based PDE solvers (Xia, 2013; Ghysels *et al.*, 2017), machine learning methods like kernel ridge regression (Chávez *et al.*, 2020) and Gaussian processes (Ambikasaran *et al.*, 2015).

There is a diverse family of hierarchical matrix techniques, including the $\mathscr{H}/\mathscr{H}^2$ format (Hackbusch, 1999; Bebendorf & Hackbusch, 2003; Börm *et al.*, 2003), the hierarchically off-diagonal low-rank (HODLR) format (Ambikasaran & Darve, 2013), the hierarchically semiseparable (HSS) format (Chandrasekaran *et al.*, 2007) or hierarchically block separable (HBS) format (Gillman *et al.*, 2012), the inverse fast multipole method (IFMM) (Coulier *et al.*, 2017) and the hierarchical interpolative factorization (HIF) algorithms (Ho & Ying, 2016). These formats can be characterized by the so-called admissibility condition, which determines how much separated interactions can be low-rank compressed. The optimal choice of hierarchical format depends on the particular application, including the dimensionality and discretization scheme. For high-dimensional problems, weak-admissibility-based formats such as HODLR, HSS, HBS typically cannot achieve quasi-linear complexities with the exception of HIF, which, however, has been demonstrated only with regular grid-based discretization. On the other hand, strong-admissibility-based formats can attain quasi-linear (e.g., $\mathscr{H}$) and linear complexities (e.g., $\mathscr{H}^2$ and IFMM) for high-dimensional problems. This paper addresses the strong-admissibility-based $\mathscr{H}^2$ format.

In earlier work (Boukaram *et al.*, 2025), we developed a novel linear-complexity bottom-up sketching-based algorithm for constructing an $\mathscr{H}^2$ matrix, and presented its high performance GPU implementation. This paper builds upon that work and further develops efficient factorization and solution algorithms likewise suitable for fine-grained architectures (Keyes et al., 2020). It is worth mentioning that once the $\mathscr{H}^2$ matrix has been constructed, efficient (i.e., low-prefactor) inversion algorithms have also been recently developed (Minden *et al.*, 2017; Ma & Jiao, 2019) and parallelized (Ma *et al.*, 2022; Liang *et al.*, 2024). Our algorithms thereby fill a gap in GPU computing and GPU code of the algorithm presented herein is in progress, to follow the CPU implementation.

### 1.1 *Background on hierarchical matrices*

Hierarchical matrices offer a compact and efficient way to represent dense matrices that possess data sparsity, where specific sub-blocks can be accurately approximated using low-rank structures. Various formulations of hierarchical matrices have emerged, distinguished primarily by their strategies for block partitioning and the methods used to construct and store the basis vectors for low-rank approximations. The block partitioning process typically begins by hierarchically clustering the indices of a matrix $A$ into a cluster tree $I$. A dual tree traversal is then performed on the cluster tree, generating pairs of clusters $(s, t)$ that are evaluated against an admissibility condition. This condition determines whether the matrix block corresponding to the cluster pair can be effectively approximated by a low-rank matrix. In this work, we adopt the general admissibility condition, denoted as `adm`, which assesses the compressibility of a block based on the distance `Dist` between the bounding boxes of $s$ and $t$, and the average of their diameters $D$:

$$\texttt{adm}(s,t) = 1, \text{ if } \frac{D(s) + D(t)}{2} \leq \eta \cdot \texttt{Dist}(s,t) \tag{1.1}$$

The parameter $\eta$ controls the strength of the admissibility condition, starting from what is usually referred to as strong admissibility for values of $\eta \geq 0.5$ and weakening as it is increased, coarsening the matrix structure. The general admissibility condition is used to guide a dual tree traversal, which constructs a matrix tree in which each node represents a cluster pair $(s, t)$ at the same level of the cluster tree. If a cluster pair fails to satisfy the admissibility criterion, the traversal proceeds recursively on the four child pairs derived from $s$ and $t$. This process continues until the corresponding matrix block becomes small enough to store in its original dense form. The complete set of leaf nodes in the resulting matrix tree defines the final block partitioning of the matrix.

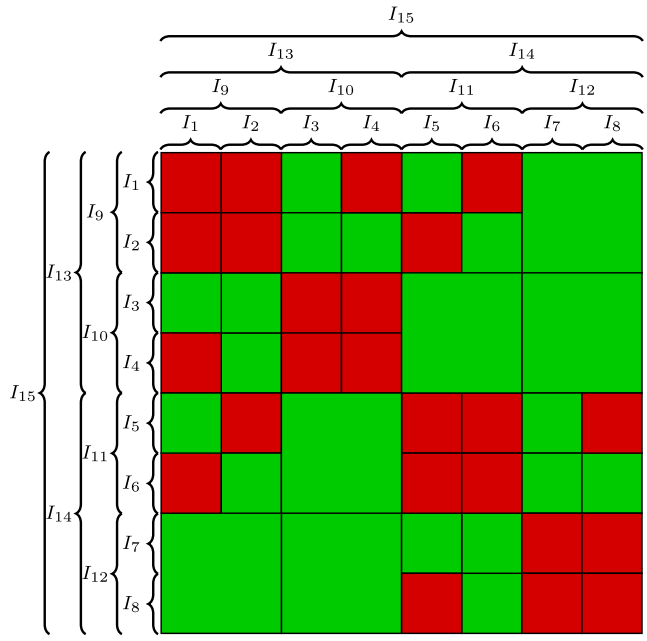



FIG. 1. A hierarchically partitioned matrix and the cluster tree $I$.

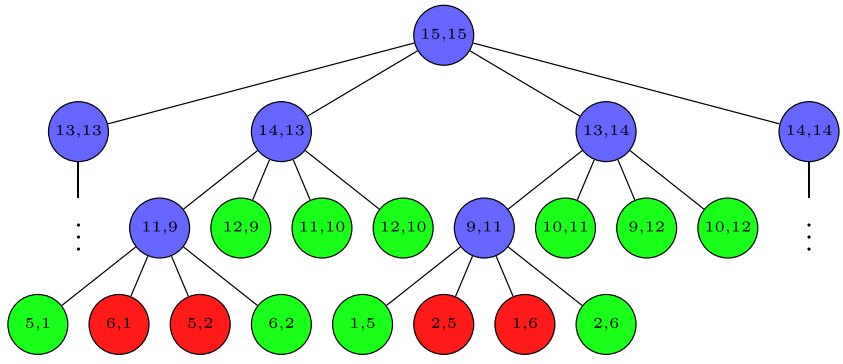



FIG. 2. The matrix tree for the hierarchical matrix in Fig. 1 representing the inadmissible blocks [(15,15), (13,13), (14,13), (13,14), (14,14), (11,9), (9,11)], the admissible leaves [(12,9), (11,10), (12,10), (10,11), (9,12), (10,12)] and the inadmissible leaves [(6,1), (5,2), (2,5), (1,6)]. In general, the matrix tree is not a complete tree. The ellipses represent complete subtrees, omitted for brevity.

Figure 1 shows the hierarchically partitioned matrix $A$ and its row and column indices hierarchically clustered into a cluster tree $I$. This block partitioning of the matrix is formed from the leaves of the hierarchical matrix tree of Fig. 2, generated from a dual tree traversal on the cluster tree.

The complete subtrees $A_{13,13}$ and $A_{14,14}$ produced from the diagonal blocks are omitted for brevity. The inadmissible leaves of the tree are shown in red, the inadmissible inner nodes are shown in blue and admissible leaves are shown in green. All the inadmissible nodes can be interpreted as forming a block sparse matrix. A key property of hierarchical matrices, ultimately leading to log-linear or linear

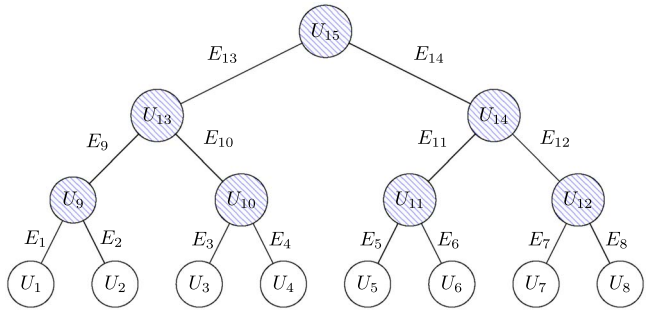



FIG. 3. The basis tree for the hierarchical matrix in Fig. 1 where leaves $U_\tau$ are stored explicitly and the shaded inner nodes are implicitly represented by the nested bases using the transfer matrices $E$.

complexity estimates for storage and factorization and solution operations, is that the number of nonzero blocks in any row or column of such a block sparse matrix is bounded by a constant that remains independent of the overall problem size. This constant is referred to as the *sparsity constant* $C_{\text{sp}}$.

We denote by $A_{s,t}$ the submatrix of $A$ associated with the cluster pair $(s, t)$. Once the block partitioning is established, admissible blocks $A_{s,t}$ are approximated using low-rank representations. In the $\mathscr{H}$-matrix format, each admissible block of size $m \times n$ and rank $k$ is represented as $A_{s,t} = U_{s,t} V_{s,t}^{\mathrm{T}}$, where $U_{s,t}$ and $V_{s,t}$ are $m \times k$ and $n \times k$ matrices, respectively. This factorization leads to a storage complexity of $O(n \log n)$ for the entire matrix. The $\mathscr{H}^2$-matrix variant further reduces this complexity to $O(n)$ by employing a nested basis structure that enables shared representations across blocks.

Instead of storing independent $U$ and $V$ matrices for each admissible block, the $\mathscr{H}^2$-matrix format introduces a shared basis for block rows and columns associated with each cluster in the cluster tree. Each admissible block is then represented as $A_{s,t} = U_s S_{s,t} V_t^{\mathrm{T}}$, where $U_s$ and $V_t$ are the basis matrices associated with clusters $s$ and $t$, respectively, and a small $k \times k$ coupling matrix $S_{s,t}$.

The bases for the leaf clusters are stored explicitly, while the basis for an internal node $\tau$ in the cluster tree is defined recursively in terms of the bases of its children $\tau_1$ and $\tau_2$ using transfer matrices. Specifically, the basis at $\tau$ is constructed via transfer matrices $E$ for $U$ and $T$ for $V$, resulting in a nested basis structure that enables efficient storage and computation:

$$U_\tau = \begin{bmatrix} U_{\tau_1} & \\ & U_{\tau_2} \end{bmatrix} \begin{bmatrix} E_{\tau_1} \\ E_{\tau_2} \end{bmatrix} \qquad V_\tau = \begin{bmatrix} V_{\tau_1} & \\ & V_{\tau_2} \end{bmatrix} \begin{bmatrix} T_{\tau_1} \\ T_{\tau_2} \end{bmatrix} \tag{1.2}$$

These notations are summarized in Table 1. We also use MATLAB notation where convenient. Figure 3 illustrates the basis tree corresponding to the cluster hierarchy shown in Fig. 1. In this representation, the clear nodes at the leaf level denote basis matrices that are stored explicitly, while the shaded internal nodes are defined implicitly via the nested basis property. For simplicity of exposition, we assume $A$ is symmetric and real-valued in the rest of this paper, leading to $V_t = U_s$ and $T_t = E_s$. However, the algorithm can be extended to nonsymmetric or complex-valued matrices, as in, for instance, (Sushnikova *et al.*, 2023).

### 1.2 *Literature review*

A number of works have addressed the direct factorization of $\mathscr{H}^2$ matrices with strong admissibility conditions, motivated by the needed to reduce rank growth that is encountered when weak admissibility

TABLE 1 *Definitions of symbols*

| Symbol | Description |
|---|---|
| $H$ | admissible part of a matrix |
| $H_{ij}^l$ | admissible block at level $l$ defined by clusters $i$ and $j$ |
| $D$ | inadmissible part of a matrix |
| $F$ | working fill-in matrix |
| $U_i$, $V_i$ | bases of cluster $i$ in the hierarchical matrix |
| $E_i$, $T_i$ | transfer matrices for bases of cluster $i$ to its parent cluster |
| $S_{ts}$ | coupling matrix for the admissible block defined by clusters $t$ and $s$ |
| $A_\tau^l$ | matrix $A$ after skeletonization of cluster $\tau$ at level $l$ |
| $D_{i*}$ | all nonzero blocks in block row $i$ of a block sparse matrix $D$ |
| $D_{*j}$ | all nonzero blocks in block column $j$ of a block sparse matrix $D$ |
| $V^\perp$ | orthogonal complement of a matrix $V$ |
| $i_R$ | redundant indices of cluster $i$ |
| $i_S$ | skeletonized indices of cluster $i$ |
| $\epsilon_{\text{lu}}$ | compression threshold for the fill-in |

structures are used, particularly in three-dimensional problems. A major line of work has been the use of skeletonization-based factorizations adapted to strong admissibility.

Minden *et al.* (2017) introduced the strong recursive skeletonization (RS-S) factorization, generalizing the recursive skeletonization (RS) process of Martinsson & Rokhlin (2005). RS-S compresses only far-field interactions and leaves near-field interactions uncompressed and results in a hierarchical LU-like factorization expressed as a product of many block unit-triangular factors. It can achieve a slower rank growth than is possible with weak admissibility conditions, which compress all neighboring nodes. Under mild assumptions on numerical ranks, the complexity of RS-S scales linearly in the matrix size.

The strong-admissibility skeletonization idea has since been extended and applied in various contexts. Sushnikova *et al.* (2023) developed the FMM-LU solver which builds on the RS-S framework for three-dimensional boundary IEs. FMM-LU constructs an LU-like hierarchical factorization for dense matrices arising from elliptic and low-frequency wave equations on surfaces. By using a level-restricted octree and high-order quadrature, it was demonstrated that even complex three-dimensional geometry problems can be factored in linear time with strong-admissibility compression, permitting efficient solution of large boundary-value problems. Yesypenko & Martinsson (2026) generalize RS-S via randomized sampling to make it blackbox and applicable to matrices available via matrix–vector products only. The method uses randomized SVDs, draws test vectors once up front, and injects structure dynamically as the factorization proceeds, yielding a sample-efficient direct factorization even when individual matrix entries cannot be evaluated efficiently. Ma *et al.* (2022); Ma & Yokota (2024) also use RS-S but introduce a phase that pre-computes all possible fill-in contributions that may arise during the factorization and incorporates them into the basis matrices ahead of time, allowing the dependency on trailing submatrices to be removed. Ma & Jiao (2019) propose an RS-S variant that dynamically augments the original basis with new ones to account for the fill-in produced at every level, and a related version (Ma & Yokota, 2024) that recomputes a new basis to account for the Schur complement updates to the original blocks during factorization.

Other approaches to direct factorization have also been proposed. Coulier *et al.* (2017) describe a strategy that uses a telescoping additive decomposition for applying the approximate inverse operator. The methods is termed an 'inverse fast multipole' as it rewrites the dense system as an extended sparse system by introducing FMM multipole/local variables, then performs elimination while compressing and redirecting fill-ins that occur between well-separated clusters. Börm & Reimer (2013) propose methods based on matrix multiplication and local LRUs that can be performed in linear complexity. These updates can be combined with a recursive procedure to approximate the product of two $\mathscr{H}^2$ matrices, and these products can be used to approximate the matrix inverse and the LU factorization. Although the asymptotic rates are favorable for these methods, their hidden constants are nonnegligible in practice.

### 1.3 *Summary of contributions*

We present a parallel fine-grained RS-S-style algorithm suitable for execution on GPUs and other massively parallel high-throughput architectures. Our algorithm is purely algebraic in its operations and runs as a black box starting only with an $\mathscr{H}^2$ matrix, facilitating its incorporation in libraries and computational workflows. The algorithm builds on the ideas in Ma & Jiao (2019), and introduces a number of innovations to allow for scalable performance on modern hardware. It augments the $\mathscr{H}^2$ basis incrementally and adaptively to accommodate the necessary 'fill-in'. Our contributions may be summarized as follows:

- *GPU-centric pipeline.* The computationally demanding operations of the algorithm are all marshaled into batches and expressed in terms of batched linear algebra that can then be executed efficiently. In addition, dynamic memory allocations are completely avoided through the use of prefix-sum memory management operations that can also be efficiently executed on GPU architectures.
- *Multi-level matrix graph coloring.* For every level of the matrix tree, a connectivity graph of its inadmissible blocks is partitioned into colors that allow for independent execution of the orthogonal projections and partial factorizations involved in the skeletonization operations of matrix blocks at that level. This is an effective and scalable strategy as the number of colors does not grow with problem size.
- *QR-based basis augmentation.* We propose and demonstrate the use of QR decompositions for building a basis for the fill-in induced by the Schur complement operations of the factorization. This is designed as a compromise between the potential loss of accuracy of Gram matrix approaches that square the condition number and the less-than-optimal ranks that may be produced by randomized SVDs. Increasingly on GPUs, maximum performance is focused on single (and even lower) precision operations, putting a premium on handling linear algebra operations with large condition numbers in a stable manner.
- *Parallel solves.* In many settings, the primary reason for a direct factorization is to allow for fast solves. This involves the application of the inverse factorization transformations in forward and backward substitution phases, which we cast in a computational structure similar to that of a hierarchical matrix–vector product, known to have effective parallel GPU execution (Boukaram *et al.*, 2019; Zampini *et al.*, 2022). As with the factorization, the solve phase involves splitting the application operations into smaller parallel conflict-free sub-batches whose effective execution is demonstrated.

We present a battery of tests originating from two- and three-dimensional contexts to evaluate the performance of the algorithm. We are releasing the code and test cases as open-source for reproducibility

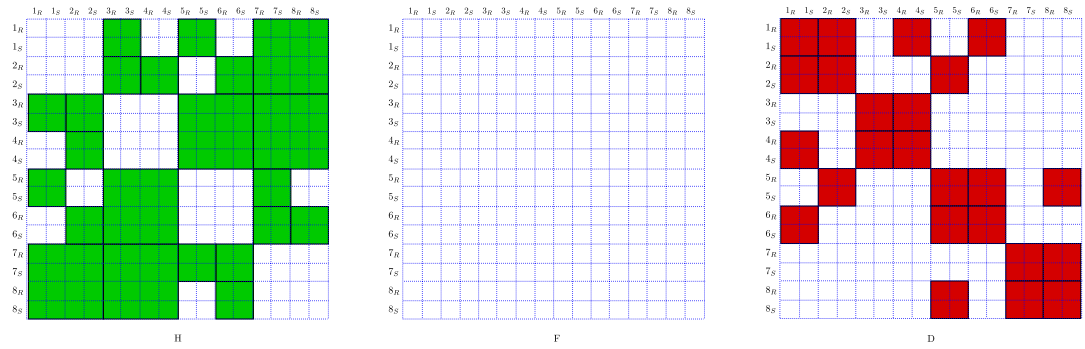



FIG. 4. $A$ split into the admissible part $H$ (left), the inadmissible part $D$ (right) and a fill-in matrix $F$ which is initially empty (middle).

and to address the practical need in the community for wider availability of readily usable source codes for factorizations of general $\mathscr{H}^2$ matrices with strong-admissibility structures.

## 2. Algorithms

### 2.1 *Factorization algorithm*

We use a skeletonization process to generate approximations of blocks in the matrix by generating a basis that approximates each block by eliminating redundant components [i.e., the degrees of freedom that lie (approximately) in the span of others], and retaining skeletonized components, i.e., a minimal subset of representative degrees of freedom from which the admissible blocks can be accurately interpolated. In sketching based approaches, this is typically achieved using interpolative decomposition (Cheng *et al.*, 2005) to determine the redundant and skeleton indices. This usually involves computing some permutation of the original matrix that can be well approximated using a subset $S$ of the original blocks, called the skeleton indices, an interpolation matrix $T$ and eliminating the redundant columns whose indices $R$ we call the redundant indices: $A\Pi = \begin{bmatrix} A_S & A_R \end{bmatrix} \approx A_S \begin{bmatrix} I & T \end{bmatrix}$. If we construct an orthogonal basis for our hierarchical matrix, we can use the complementary basis vectors to do the elimination instead.

Let us split the hierarchical matrix $A$ in Fig. 1 into three parts: the low rank part $H$, the dense part $D$ and a fill-in matrix $F$ which is initially the zero matrix: $A = H + D + F$ as shown in Fig. 4. Any transformation applied to $A$ will be applied to all three of its parts and after each transformation, we are free to shuffle around the blocks between the three parts as needed (for example, moving the sum of certain blocks from $F$ and $H$ to $D$). Let $m$ be the leaf size and $k^l$ be a representative rank of the blocks at level $l$.

Factorization using skeletonization follows two major phases: orthogonal projections to eliminate the redundant rows and columns of a cluster within $F$ and $H$ and partial factorization in $D$ to eliminate the same rows and columns without allowing additional fill-in from previous steps to affect $F$ and $H$. We refer to this process as a partial factorization since only the redundant rows and columns of the cluster are eliminated in contrast to standard factorization algorithms that eliminate the entire cluster at once. Once the redundant rows and columns of a cluster are eliminated, we say that a cluster has been skeletonized (for example, see Fig. 7 where cluster 1 has been skeletonized). Let us call the transformed matrix at level $l$ after cluster $\tau$ has been skeletonized $A^l_\tau$. The first phase will often involve computing a basis that

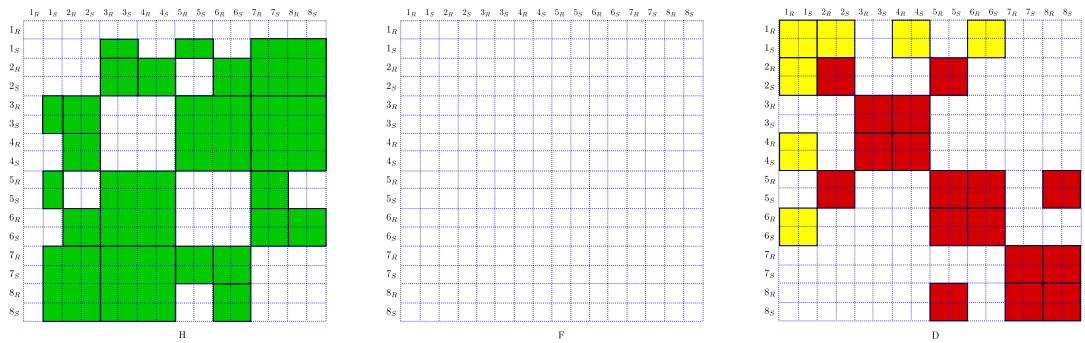


FIG. 5. The effect of the orthogonal projection generated from cluster 1 on the three parts of $A$, where the rows and columns $1_R$ of $H$ are zeroed out and the first block row and column of $D$, rows and columns $1_\mathrm{R}$ and $1_\mathrm{S}$ (right), are scaled by $\tilde{Q}_1$.

will include the original cluster basis as well as a basis that approximates the blocks in $F$. Starting from the leaf level, cluster 1 of the matrix is skeletonized as follows:

1. We first complete the cluster basis $V_1$ with its complement $V_1^\perp$ to form the matrix $\tilde{Q}_1 = [V_1^\perp, V_1]$.
2. The block row and column corresponding to the first cluster are then scaled by $\tilde{Q}_1$ by forming the product $A_1^1 = Q_1^\mathrm{T} A Q_1 = Q_1^\mathrm{T}(H + D + F)Q_1$, where $Q_1 = \mathtt{diag}(\tilde{Q}_1, I, \cdots, I)$. This zeros out the rows and columns of $H$ corresponding to the redundant portion of the cluster which we call $1_R$, since $\tilde{Q}_1^\mathrm{T} V_1 = [0; I]$. Note that since the basis is nested, this zeros out the block rows and columns not just on the current level, but on the higher levels as well. The fill-in matrix is still the zero matrix at this point. The block row and column $D_{1*}$ and $D_{*1}$, highlighted in yellow in Fig. 5, are scaled by $\tilde{Q}_1^\mathrm{T}$ and $\tilde{Q}_1$, respectively, i.e., $\tilde{Q}_1^\mathrm{T} D_1^1 \tilde{Q}_1$.
3. The off-diagonal blocks in the block row and column $1_R$ of $D$ are eliminated using partial LU factorization. The diagonal block $D_{1_R 1_R}$ is factorized using dense LU decomposition and elementary lower and upper block triangular matrices are constructed to eliminate the redundant off-diagonal blocks $D_{1_R *}$ and $D_{*1_R}$:

$$L_1 = \begin{bmatrix} I & & \\ -D_{1_S 1_R} D_{1_R 1_R}^{-1} & I & \\ -D_{*1_R} D_{1_R 1_R}^{-1} & & I \end{bmatrix} \qquad U_1 = \begin{bmatrix} I & -D_{1_R 1_R}^{-1} D_{1_R 1_S} & -D_{1_R 1_R}^{-1} D_{1_R *} \\ & I & \\ & & I \end{bmatrix} \tag{2.1}$$

   The matrix can then be transformed as $A_1^1 \rightarrow L_1 A_1^1 U_1$, leaving only the diagonal block $D_{1_R 1_R}$ in the redundant part of the inadmissible matrix. Since the redundant block rows and columns are zero in $H$ and $F$, they are unaffected by the row and column operations performed during this step, whereas fill-in is introduced in $D$ due to the Schur complement updates. This is shown in Fig. 6 with fill-in in black and updated original blocks are in blue.
4. The fill-in introduced in the previous step is moved from $D$ to $F$. Though they are formally dense, they are low rank by construction as the outer product of the cluster's redundant off-diagonal blocks. If we can ensure that the redundant block rows and columns in $F$ are eliminated at the same time as the elimination in $H$ during the projection phase, this will prevent the fill-in produced by the partial factorizations from creating more fill-in as we proceed to other clusters. The next step will explore

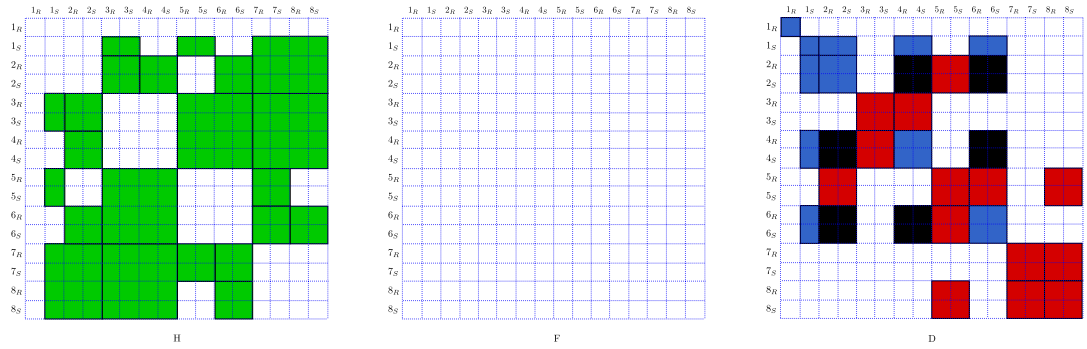



FIG. 6. Partial factorization of $D$ to eliminate the off-diagonal blocks in the 1R row and columns. The fill-in due to the updates are the dark-highlighted blocks in the intersections of rows and columns $(2_R, 2_S)$, $(4_R, 4_S)$, and $(6_R, 6_S)$ and updated blocks of the original matrix are in the light-highlighted blocks in the intersections of rows and columns $(1_R, 1_R)$, $(1_S, 2_R, 2_S)$, $(4_R, 4_S)$, and $(6_R, 6_S)$.

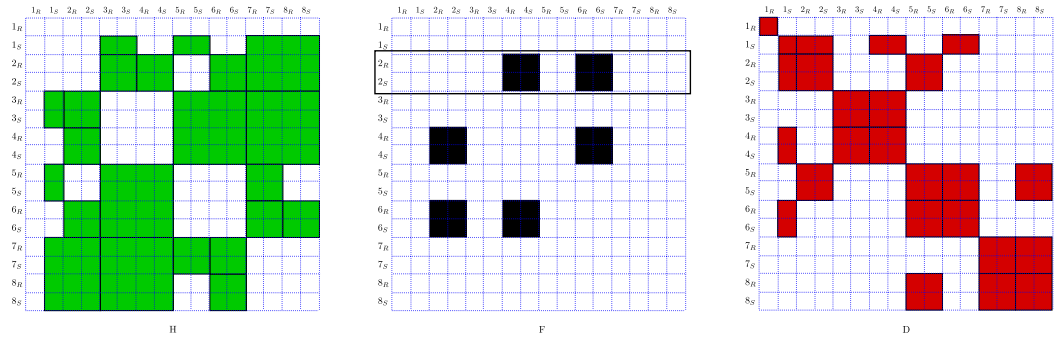



FIG. 7. Moving the black fill-in blocks generated by the partial factorization of $D$ to the fill-in matrix $F$. The block row of $F$ for the second cluster is highlighted and will be used to construct an augmented basis for that cluster.

how this can be done while maintaining the nested basis property. Figure 7 shows all three parts of the updated matrix at the end of processing cluster 1.

Moving on to the second cluster, we must determine additional orthogonal basis vectors that allow us to eliminate the redundant portions of the fill-in blocks in the rows and columns corresponding to that cluster. Consider all the row blocks of the second cluster in the fill-in matrix $F_{2*} = [F_{24}, F_{26}]$, highlighted in Fig. 7. We would like to find an orthogonal basis $\tilde{V}_2 = [V_2, \bar{V}_2]$ that includes the original basis $V_2$ and produces an approximation of $F_{2*} \approx \tilde{V}_2\tilde{V}_2^{\mathrm{T}}F_{2*}$, minimizing the error $||\tilde{V}_2\tilde{V}_2^{\mathrm{T}}F_{2*} - F_{2*}||$. This can be achieved by computing the SVD of the matrix $Y = (I - V_2V_2^{\mathrm{T}})F_{2*} = \bar{V}_2SW_2^{\mathrm{T}}$, since the computed singular vectors $\bar{V}_2$ minimize the error:

$$\begin{aligned} ||\bar{V}_2\bar{V}_2^{\mathrm{T}}(I - V_2V_2^{\mathrm{T}})F_{2*} - (I - V_2V_2^{\mathrm{T}})F_{2*}|| &= ||\bar{V}_2\bar{V}_2^{\mathrm{T}}F_{2*} - \bar{V}_2\bar{V}_2^{\mathrm{T}}\overbrace{V_2V_2^{\mathrm{T}}}^{0}F_{2*} - F_{2*} + V_2V_2^{\mathrm{T}}F_{2*}|| \\ &= ||[V_2, \bar{V}_2][V_2, \bar{V}_2]^{\mathrm{T}}F_{2*} - F_{2*}|| = ||\tilde{V}_2\tilde{V}_2^{\mathrm{T}}F_{2*} - F_{2*}|| \end{aligned} \tag{2.2}$$

We then truncate the singular vectors of $\bar{V}_2$ that are unnecessary to satisfy a compression threshold $\epsilon_{fill} = \epsilon_{\mathrm{lu}}||A||$. As for computing the SVD of $Y$, there are at least three algorithm choices: (1) one can form

the Gram matrix $G = YY^{\mathrm{T}}$ and find its SVD and truncate to $\epsilon^2_{\text{fill}}$; (2) perform the QR decomposition of $F^{\mathrm{T}}_{2*} = QR$ and then the SVD of $R$; or (3) use a randomized SVD. In our tests, we use the second approach since it will provide the best accuracy, but the Gram matrix approach can trade accuracy for speed, while the randomized SVD can lead to slightly larger ranks but is more stable, providing a middle ground between the two. We note that this truncation is the primary source of errors in the algorithm, but we leave the aggregate error analysis as future work. Regardless of the choice of compression algorithm, we end up with additional basis vectors $\bar{V}_2$ and an augmented cluster basis $\tilde{V}_2 = [V_2, \bar{V}_2]$. To account for the additional basis vectors, we simply need to pad the original coupling matrices with zeros, increasing the rank $k^1$:

$$V_i S_{ij} V_j^{\mathrm{T}} = [V_i, \bar{V}_i] \begin{bmatrix} S_{ij} & 0 \\ 0 & 0 \end{bmatrix} [V_j, \bar{V}_j]^{\mathrm{T}} = \tilde{V}_i \tilde{S}_{ij} \tilde{V}_j^{\mathrm{T}} \tag{2.3}$$

Although the fill-in, as the outer product of two low rank matrices, and the original basis are both low rank, the augmented rank is not guaranteed to be remain low; however, in practice this seems to hold true. Indeed, we know that the inverse of a hierarchical matrix will have a low rank structure (Bebendorf & Hackbusch, 2003; Faustmann *et al.*, 2015), but we will leave a more thorough analysis of the way the ranks of the skeletonization behave as future work. To maintain the nested basis property, the rows of the corresponding transfer matrices $T_i$ are padded with zeros as well, transforming into $\tilde{T}_i = [T_i; 0]$.

The fill-in in $F$ for the second cluster can now be skeletonized at the same time as the second block row and columns in $H$. This step of the skeletonization is crucial to preventing fill-in from a previously skeletonized cluster from propagating any further during the Schur complement updates of the partial factorization. We again form the projection matrix by finding the complement $\tilde{V}_2^{\perp}$ to form the matrix $\tilde{Q}_2 = [\tilde{V}_2^{\perp}, \tilde{V}_2]$. As with the first cluster, scaling the matrix as $A_2^1 = Q_2^{\mathrm{T}} A_1^1 Q_2$ where $Q_2 = \texttt{diag}(I, \tilde{Q}_2, \cdots, I)$ eliminates the redundant portion $2_R$ of the cluster rows and columns since $\tilde{Q}_2^{\mathrm{T}} \tilde{V}_2 = [0; I]$. It also eliminates the same rows and columns of the fill-in matrix, since we approximate each block in that cluster as $F_{2j} \approx \tilde{V}_2 \tilde{V}_2^{\mathrm{T}} F_{2j}$:

$$\tilde{Q}_2^{\mathrm{T}} F_{2j} \approx [\tilde{V}_2^{\perp}, \tilde{V}_2]^{\mathrm{T}} \tilde{V}_2 \tilde{V}_2^{\mathrm{T}} F_{2j} = \begin{bmatrix} 0 \\ I \end{bmatrix} \tilde{V}_2^{\mathrm{T}} F_{2j} = \begin{bmatrix} 0 \\ \tilde{V}_2^{\mathrm{T}} F_{2j} \end{bmatrix} \tag{2.4}$$

Figure 8 shows the skeletonization of $H$ and $F$ as well as the scaled blocks of $D$ in yellow. The partial LU factorization then proceeds as before with fill-in generated in $D$ as shown in Fig. 9. Elementary triangular matrices $L_2$ and $U_2$ are computed and used to transform the matrix as $A_2^1 \rightarrow L_2 A_2^1 U_2$, as with the first cluster. Note that rectangular fill-in can be produced due to the elimination of previously skeletonized clusters and can simply be moved to $F$ after the updates are complete.

Once all eight clusters have been skeletonized, we end up with the matrix $A_8^1$ as the sum of the three matrices in Fig. 10. At this point, all blocks $H_{ij}^1$ at the leaf level of $H$ have been transformed as:

$$\begin{aligned} \tilde{Q}_i^{\mathrm{T}} H_{ij}^1 \tilde{Q}_j = \begin{bmatrix} \tilde{V}_i^{\perp} & \tilde{V}_i \end{bmatrix}^{\mathrm{T}} \tilde{V}_i \tilde{S}_{ij} \tilde{V}_j^{\mathrm{T}} \begin{bmatrix} \tilde{V}_j^{\perp} & \tilde{V}_j \end{bmatrix} &= \begin{bmatrix} 0_{(m-k^l)\times k^l} \\ I_{k^l \times k^l} \end{bmatrix} \tilde{S}_{ij} \begin{bmatrix} 0_{k^l \times (m-k^l)} & I_{k^l \times k^l} \end{bmatrix} \\ &= \begin{bmatrix} 0_{(m-k^l)\times(m-k^l)} & 0_{(m-k^l)\times k^l} \\ 0_{k^l \times (m-k^l)} & \tilde{S}_{ij} \end{bmatrix} \end{aligned} \tag{2.5}$$

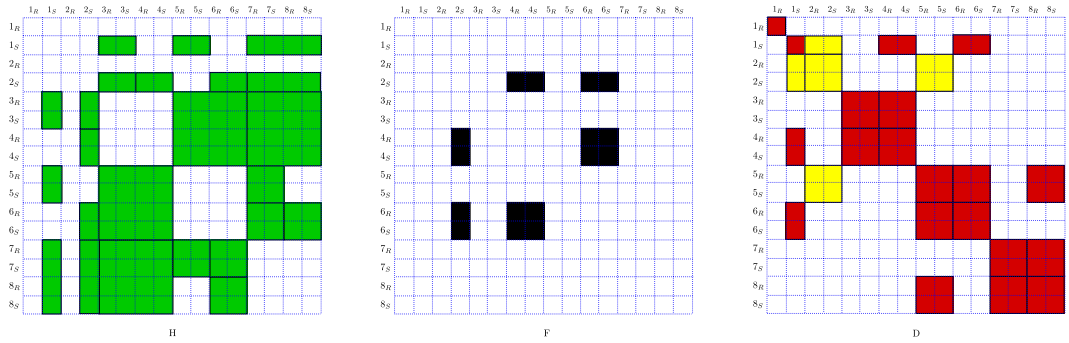



FIG. 8. Skeletonizing the second cluster. The projection matrix for this step includes additional basis vectors for the fill-in blocks in block row 2 of $F$. This allows the rows and column $2_R$ in both $H$ and $F$ to be skeletonized at the same time.

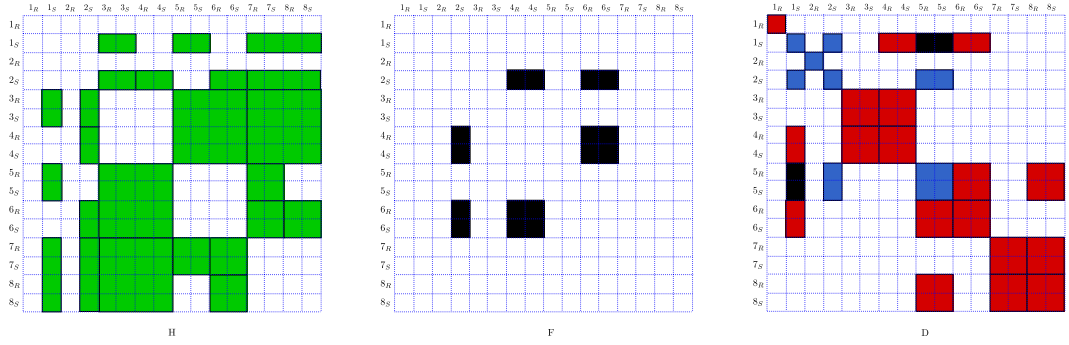



FIG. 9. The partial factorization of the second cluster in $D$ generates rectangular fill-in when eliminating the blocks $D_{2_R 1_S}$ and $D_{1_S 2_R}$.

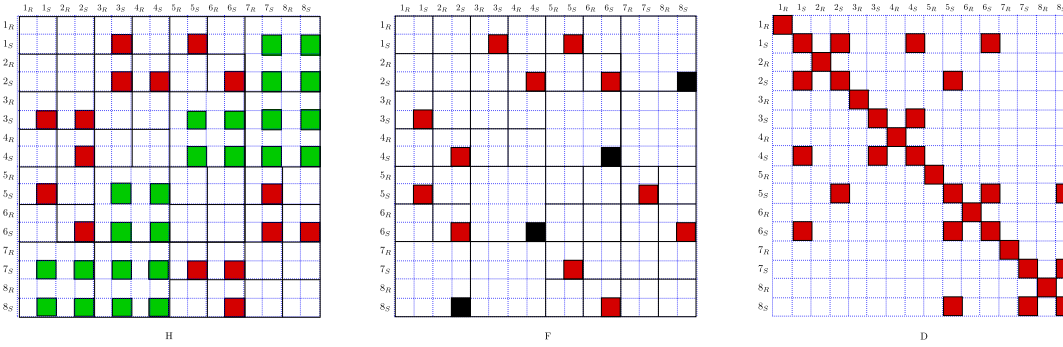



FIG. 10. All clusters have been skeletonized and updated. The blocks at the same level of $H$ and $F$ can now be treated as dense blocks.

Now each admissible block $H^1_{ij}$ can be expanded as a dense block containing only $k^l \times k^l$ nonzeros in the skeletonized rows and columns of the matrix $i_S$ and $j_S$. The same applies for the fill-in matrix $F$. Since we can freely move blocks between the three matrices, we select all blocks $H^1_{ij}$ from $H$ and the

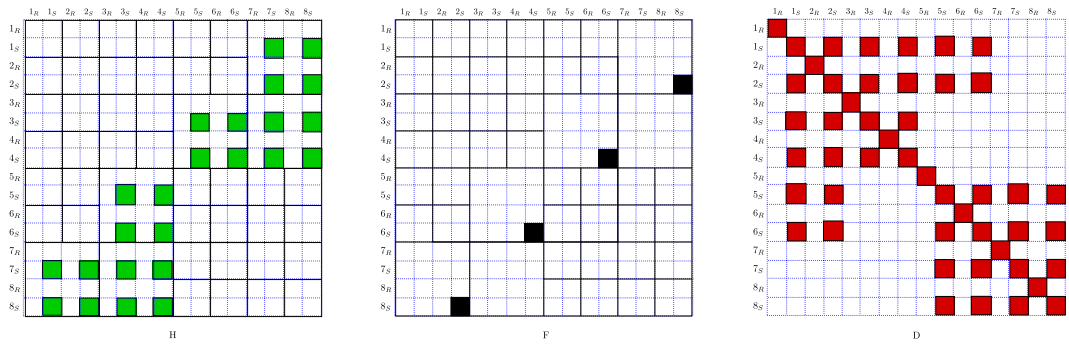



FIG. 11. Adding the converted blocks from $H$ and $F$ and moving them to $D$.

corresponding blocks $F_{ij}$ from $F$ as shown in red in Fig. 10, add them together and move them to $D$ as shown in Fig. 11. Since we have removed all blocks $H^1_{ij}$ from $H$ and the basis clusters $U_i$ and $U_j$ have all taken the form $[0; I]$, we can completely remove the first level of the basis from $H$. Let $\tau_1$ and $\tau_2$ be the children of a node $\tau$ at the next level of the matrix. The basis node $V_\tau$, originally expressed in terms its children $V_{\tau_1}$ and $V_{\tau_2}$ and two transfer matrices $T_{\tau_1}$ and $T_{\tau_2}$, is transformed as:

$$\begin{bmatrix} \tilde{Q}^{\mathrm{T}}_{\tau_1} \\ \tilde{Q}^{\mathrm{T}}_{\tau_2} \end{bmatrix} V_\tau = \begin{bmatrix} \tilde{V}^{\perp}_{\tau_1} & \tilde{V}_{\tau_1} \\ \tilde{V}^{\perp}_{\tau_2} & \tilde{V}_{\tau_2} \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} \tilde{V}_{\tau_1} & 0 \\ 0 & \tilde{V}_{\tau_2} \end{bmatrix} \begin{bmatrix} \tilde{T}_{\tau_1} \\ \tilde{T}_{\tau_2} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} 0 \\ I \end{bmatrix} \tilde{T}_{\tau_1} \\ \begin{bmatrix} 0 \\ I \end{bmatrix} \tilde{T}_{\tau_2} \end{bmatrix} = \begin{bmatrix} 0 \\ \tilde{T}_{\tau_1} \\ 0 \\ \tilde{T}_{\tau_2} \end{bmatrix} \tag{2.6}$$

The zero blocks correspond to the redundant indices in the matrix. When we separate the redundant indices from the skeletonized ones by applying a permutation $P^l$ at level $l$, the resulting matrices are shown in Fig. 12. All matrices are now $2 \times 2$ block diagonal matrices, with the lower right blocks of $H$ and $F$ as zero blocks. $H$ maintains its properties as a hierarchical matrix, with the basis at its leaf level now consisting only of the transfer matrices $V_t = \begin{bmatrix} \tilde{T}_{\tau_1} \\ \tilde{T}_{\tau_2} \end{bmatrix}$ and its coupling matrices untouched. We observe that the inadmissible leaves of this hierarchical matrix are now $2k^l \times 2k^l$, where $k^l$ is the representative rank of the augmented basis at level $l$. Once all clusters in a level have been skeletonized, we end up with the matrix $A^1_8 = \left(L_8 Q_8^{\mathrm{T}} \dots L_1 Q_1^{\mathrm{T}}\right) A \left(Q_1 U_1 \dots Q_8 U_8\right)$. Note that each factor in the transformation is readily invertible, either as the transpose of the orthogonal matrices $Q$ or by flipping the sign of the off-diagonal blocks in the elementary triangular matrices. The lower right block of $D$ is block diagonal, so we can invert it by applying a block diagonal factorization of the redundant blocks as $L^l_r U^l_r$, whereas the upper left blocks can then be processed in the next level. The computation continues level by level until we reach the highest level $t$ of the tree that contained admissible blocks. At that point, only inadmissible blocks remain and the matrix $D_t$ is a relatively small dense matrix that can be factorized directly into its factors $D_t = L_t U_t$, ending the factorization. The final factorization can therefore be expressed as a sequence of transformations shown in Equation 2.7.

$$A = \mathscr{L}\mathscr{U} = \left( \prod_{l=1}^{t} \left( \prod_{\tau \in I^l} Q_\tau L_\tau^{-1} \right) P^{l^{\mathrm{T}}} L^l_r \right) L_t U_t \left( \prod_{l=t}^{1} \left( U^l_r P^l \prod_{\tau \in I^l} U_\tau^{-1} Q_\tau^{\mathrm{T}} \right) \right). \tag{2.7}$$

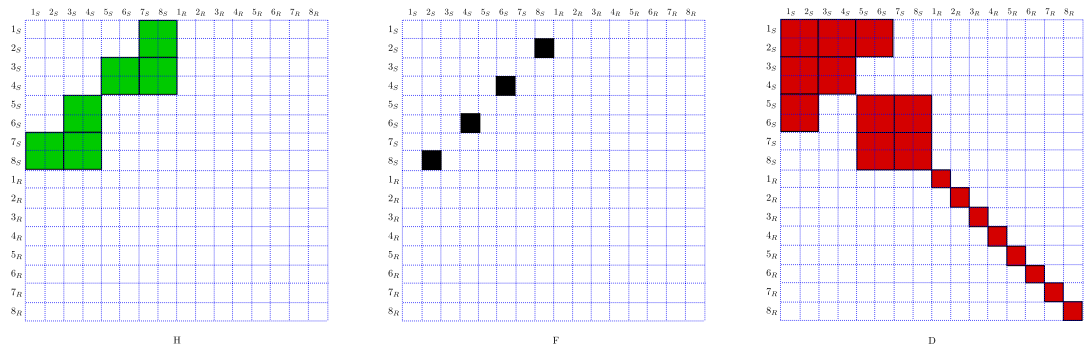



FIG. 12. Permuting all three matrices to split the $S$ and $R$ indices. The resulting matrix is a $2 \times 2$ block diagonal. The lower right block is also block diagonal, so inverting it is trivial. The upper left block can then be processed using the same method recursively.

Algorithm 1 describes the entire factorization algorithm starting at the leaf level up to the top level where the final dense factorization takes place, while Algorithm 2 describes the parallel skeletonization of all clusters which we describe in the following section.

**Algorithm 1** FACTORIZE($A$, $\epsilon_{\text{lu}}$): Factorize a hierarchical matrix using skeletonization

**Require:** $\mathscr{H}^2$-matrix $A$, truncation threshold $\epsilon_{\text{lu}}$

$d = \texttt{depth}(A)$
$t = \texttt{topLevel}(A)$ ▷ First level that has admissible nodes
$F = 0$ ▷ Initially zero fill-in matrix
$Z = []$ ▷ Empty factorization
**for** $l \leftarrow d$ to $t$ **do**
  $D = \texttt{inadmissible}\,(A, F, l)$ ▷ Extract the inadmissible nodes at level $l$
  $F = \texttt{fillIn}\,(A, F, l)$ ▷ Extract the fill-in to higher levels
  $C^l = \texttt{coloring}\,(A, l)$ ▷ Color nodes to determine independent skeletonization indices
  **for** $C \in C^l$ **do**
    $D, F, L, U, V_D = \texttt{skeletonizeColor}(A, D, F, C, \epsilon_{\text{lu}})$
    $Z = \texttt{addFactor}(Z, C, L, U, V_D)$
  **end for**
**end for**
$D_t = \texttt{inadmissible}\,(A, F, t)$ ▷ Extract the dense top level remnants
$L_t, U_t = \texttt{LU}(D_t)$
$Z = \texttt{addTopLevel}(Z, L_t, U_t)$
**return** $Z$

## 2.2 *Parallelization*

An important property of the factorization algorithm is that, since we skeletonize the fill-in at the same time as the admissible matrix, fill-in due to the partial elimination of one cluster will affect only the rows and columns of the cluster's direct nonzero neighbors in the block sparse matrix. Any two clusters that are not directly connected to each other by a nonzero block can therefore be skeletonized independently.

**Algorithm 2** SKELETONIZECOLOR($A$, $D$, $F$, $C$, $\epsilon_{\text{lu}}$): Skeletonize the colored nodes

**Require:** $\mathscr{H}^2$-matrix $A$, inadmissible blocks $D$, fill-in matrix $F$
**Require:** Color node indices $C$, truncation threshold $\epsilon_{\text{lu}}$
1: $Q = I$
2: **parallel for** $i \in C$ **do**
3: $\tilde{V}_i = \texttt{truncSVD}\left((I - V_i V_i^{\mathrm{T}})F_{i*}, \epsilon_{\text{lu}}\right)$ ▷ Compute truncated left singular vectors
4: $\tilde{V}_i = \begin{bmatrix} V_i & \tilde{V}_i \end{bmatrix}$ ▷ Augment the basis to accommodate fill-in
5: $\tilde{V}_i^{\perp} = \texttt{complement}\left(\tilde{V}_i\right)$ ▷ Get the vectors orthogonal to the augmented basis
6: $\tilde{Q}_i = \begin{bmatrix} \tilde{V}_i^{\perp} & \tilde{V}_i \end{bmatrix}$ ▷ Complete the basis
7: $Q(i,i) = \tilde{Q}_i$ ▷ Set the diagonal block for node $i$
8: **end for**
9: $D = Q^{\mathrm{T}} D Q$
10: $F = Q^{\mathrm{T}} F Q$
11: $(D, L, U, F^a) = \texttt{partialLU}(D, C)$ ▷ Generate fill-in and elementary triangular matrices
12: $F = F + F^a$
13: **return** $(D, F, L, U, Q)$

Forming the connectivity graph of the inadmissible block sparse matrix at each level, we can therefore apply a graph coloring algorithm to determine partitions of clusters that can be skeletonized in parallel. While each color can be processed in parallel, the number of colors will dictate the number of serial steps for each level of the tree. Since the maximum number of colors is determined by the maximum degree of the connectivity graph and the degree is the sparsity constant, the number of serial steps does not grow with the problem size. This should allow the algorithm to scale well with the problem size.

While the current results are obtained on a CPU in a development phase, we designed the algorithm with efficient GPU execution (and fine-grained parallelism) in mind. Since all of the blocks involved in the computation are quite small, either as the leaf size of the cluster or as the rank of the blocks, we first marshal the operations for all the clusters of the same color into batches that can be executed efficiently using batched linear algebra routines. Memory allocations are also kept to a minimum by computing matrix memory requirements for all clusters involved in a color's skeletonization, allocating a single large buffer and then using prefix sums on the products of the block dimensions to determine offsets into the buffer. Since workspace requirements are not known beforehand, we avoid costly allocations and de-allocations by using a custom memory pool allocator for all temporary allocations.

Though some operations are available on the GPU in a batched form, the GPU implementation of the missing operations (such as a nonuniform batched SVD for basis augmentation) remain as future work. On the CPU these batched operations are simple parallel OpenMP loops around single-threaded linear algebra kernels, while the marshaling is handled by Thrust routines with the OpenMP execution backend. A complication that must be addressed is that although the scaling of the rows and columns by the completed orthogonal basis is trivial, the partial factorization potentially involves multiple clusters of the same color trying to update the same block at the same time. To overcome this, we split the batched update into multiple smaller sub-batches that can be completed in parallel without any race conditions. The sub-batches are then executed in serial order. It is worth noting that since many of the operations are potentially quite small (for example, due to low ranks), they may be limited by the available memory bandwidth rather than by compute power.

### 2.3 *Forward and backward solves*

As noted in the previous section, all transformations applied to the matrix $A$ in Equation (2.7) to skeletonize and factorize it are easily invertible. To solve the system of equations represented by $A$ with a right-hand side $b$, we simply need to apply the inverse of those transformations in the forward and backward substitution phases akin to the regular dense case, but with a computational structure similar to that of a hierarchical matrix–vector product. In the forward solve, we compute a vector tree $\hat{b}$ that shares the same structure as our basis tree. One can think of each level $\hat{b}^l$ as a vector of the same dimension as the skeletonized matrix $A^l$, with the leaf level of the tree initialized with our input vector $b$. The vector $\hat{b}^l$ is scaled by the orthogonal factor $Q_\tau$ followed by the elementary factor $L_\tau^{-1}$ which is simply $L_\tau$ with the sign of the off-diagonal blocks flipped. After all level transformations are completed, the block diagonal solves are carried out and the skeletonized portions can be swept up to the next level of $\hat{b}$. At the top level $t$ a regular dense forward solve with $L_t$ and the tree vector $\hat{b}^t$ ends the forward solve phase. This is followed by a regular backwards solve with $U_t$, leading us to the start of the downwards sweep of the tree in the backwards solve phase. This is similar to the previous phase, starting at level $t$, applying the inverse of the transformations and sweeping down the tree at the end of the level. Once we reach the leaf level, we have our final solution vector $x$ as the leaf level of the downswept $\hat{b}$. The hierarchical solve routine is described in Algorithm 3. The upsweep and downsweep at each level encapsulate the permutation process shown in Fig. 12 and its inverse, respectively, while the diagonal solve steps apply the inverses of the lower and upper diagonal redundant blocks. Finally, the `applyForward` and `applyBackward` routines apply the orthogonal factors and the elementary triangular factors for all the nodes of a specified color in parallel. Like the factorization, this operation involves splitting the application of the inverse of the elementary triangular operations into smaller parallel conflict-free sub-batches that are executed one after the other.

### 2.4 *Algorithmic complexity*

We consider the sparsity constant $C_{\text{sp}}$ for the inadmissible block sparse matrices at each level of the matrix tree and a representative rank $k$ for the basis of the clusters. At the leaf level, the cluster size is equal to the leaf size which we will assume to be a constant on the same order as $k$ and at higher levels they are $2k \times 2k$ blocks, so we will assume that all blocks are of size $O(k)$. The computational effort for the skeletonization of a cluster $\tau$ can be split into three major steps: finding the augmented basis, applying the complete orthogonal factor to the inadmissible and fill-in matrices and the partial LU factorization. The first step involves the compression of the $C_{\text{sp}}$ fill-in blocks in the cluster's block row $F_{\tau *}$ at a cost $O(C_{\text{sp}}k^3)$ with a similar cost for applying the orthogonal projection to the matrix blocks. On the other hand, the partial LU factorization involves $O(C_{\text{sp}}^2)$ blocks during the Schur complement updates, dominating the cost at $O(C_{\text{sp}}^2k^3)$ operations. Since there are $O(n)$ clusters in total, the cost of the factorization algorithm is therefore $O(C_{\text{sp}}^2k^3n)$ operations. The solve phase is similarly bound by the cost of applying the elementary triangular factors at the cost of $O(C_{\text{sp}}k^2)$ operations for each cluster, bringing it to a total cost of $O(C_{\text{sp}}k^2n)$. Since the sparsity constant does not grow with problem size, the complexity of both methods grows linearly with $n$, if we assume a bound $k$ on the rank.

In terms of memory, each cluster stores the orthogonal factor, the redundant diagonal blocks, and the $O(C_{\text{sp}})$ blocks of the elementary triangular factor. This yields a total cost of storing the factors to $O(C_{\text{sp}}k^2n)$, indicating that, under the same assumptions, memory requirements also grow linearly with problem size.

**Algorithm 3** SOLVE($Z$, $b$): Solve $Ax = b$ using the factorization $Z$ of $A$

$d = \texttt{depth}(Z)$
$t = \texttt{topLevel}(Z)$ ▷ First level that has admissible nodes
$\hat{b}^d = b$ ▷ Initialize leaf level with RHS
**for** $l \leftarrow d$ to $t+1$ **do** ▷ Forward upsweep solve
  $C^l = \texttt{coloring}\,(Z, l)$ ▷ Color nodes
  **for** $C \in C^l$ **do**
    $\hat{b}^l = \texttt{applyForward}(Z, C, \hat{b}^l)$
  **end for**
  $\hat{b}^l = \texttt{diagonalForward}(Z, \hat{b}^l)$ ▷ Forward solve with redundant diagonal blocks
  $\hat{b}^{l-1} = \texttt{upsweep}(Z, \hat{b}^l)$ ▷ Sweep skeletonized vector to parent level
**end for**
$\hat{b}^t = \texttt{solve}(Z.L_t, Z.U_t, \hat{b}^t)$ ▷ Dense top level solve
**for** $l \leftarrow t+1$ to $d$ **do** ▷ Backwards downsweep solve
  $\hat{b}^l = \texttt{downsweep}(Z, \hat{b}^{l-1})$ ▷ Sweep skeletonized vector to child level
  $\hat{b}^l = \texttt{diagonalBackward}(Z, \hat{b}^l)$ ▷ Backward solve with redundant diagonal blocks
  $C^l = \texttt{coloring}\,(Z, l)$ ▷ Color nodes
  **for** $C \in C^l$ **do**
    $\hat{b}^l = \texttt{applyBackward}(Z, C, \hat{b}^l)$
  **end for**
**end for**
**return** $\hat{b}^d$ ▷ The solution is the leaf level

## 3. Numerical experiments

We illustrate the algorithm developed herein on four families of symmetric positive definite systems. To construct the initial hierarchical matrices in the following examples, we first generate a uniform grid of points in a $d$-dimensional space and build a cluster tree using a KD-tree with a leaf size $m$. The mesh is refined anisotropically as the problem size increases by cycling through the dimensions and doubling the subdivisions along that dimension. A dual tree traversal on this tree with an admissibility parameter $\eta$ is used to determine the structure of the $\mathcal{H}^2$-matrix. Then the coupling, basis and transfer matrices are generated using Chebyshev interpolation (Börm & Garcke, 2007) of order $p$, where a $p^d$ tensor product grids of Chebyshev interpolation points are overlaid on the bounding boxes of each cluster. The order is increased from an initial order $p_0$ at the leaf level with every other level to compensate for the coarser matrix structures. Algebraic compression is carried out to a specified tolerance $\epsilon$ to reduce the original ranks $k = p^d$ and orthogonalize the basis of the matrix. To help moderate the condition number of the matrices in these applications, we add a small diagonal regularization term $\alpha_r I$. Finally, factorization is carried out to the threshold $\epsilon_{\text{lu}}$. The problem sizes vary from $2^{14}$ to $2^{20}$ and we list the parameters for the construction and factorization of the applications in Table 2. The table also shows the maximum sparsity constant $C_{\text{sp}}$ of the inadmissible nodes for each problem across all levels for each problem as well as the maximum rank after algebraic compression $k_{\max}$, before factorization commences.

### 3.1 *Test problems*

We consider four applications. The first application involves spatial statistics covariance matrices from the Gaussian Process with the exponential kernel $K(x, y) = e^{-\frac{|x-y|}{l}}$. We test both two-dimensional

TABLE 2 *Parameters for construction and factorization of the problems (leaf size m, leaf-level Chebyshev interpolation order $p_0$, physical domain dimension d, admissibility constant η[Equation (1.1)], diagonal regularization term $\alpha_r$, algebraic compression tolerance $\epsilon$, and LU factorization tolerance $\epsilon_{lu}$) and maximum ranks of the initial matrix $k_{max}$ and sparsity constant $C_{sp}$*

| Problem | $m$ | $p_0$ | $d$ | $\eta$ | $\alpha_r$ | $\epsilon$ | $\epsilon_{\text{lu}}$ | $k_{\max}$ | $C_{\text{sp}}$ |
|---|---|---|---|---|---|---|---|---|---|
| Two-dimensional covariance | 64 | 8 | 2 | 0.9 | $10^{-2}$ | $10^{-7}$ | $10^{-6}$ | 39 | 11 |
| Three-dimensional covariance | 64 | 4 | 3 | 0.7 | $10^{-2}$ | $10^{-7}$ | $10^{-6}$ | 111 | 77 |
| Two-dimensional laplace IE | 64 | 8 | 2 | 0.9 | $10^{-5}$ | $10^{-7}$ | $10^{-6}$ | 23 | 11 |
| Three-dimensional Helmholtz IE | 64 | 4 | 3 | 0.7 | $10^{-2}$ | $10^{-7}$ | $10^{-6}$ | 86 | 77 |
| LRU Three-dimensional covariance | 128 | 4 | 3 | 0.9 | $10^{-2}$ | $10^{-8}$ | $10^{-7}$ | 190 | 39 |

and three-dimensional problems using a uniform distribution of points in a unit square or cube with correlations lengths $l$ set to 0.1 and 0.2, respectively. For the second application, we consider the hierarchical matrix constructed in the same way for the free-space Green's function associated with the two-dimensional Laplace equation, solving the volume IE whose operator is

$$K(x, y) = -\frac{1}{2\pi} \log(||x - y||),\ x \neq y. \tag{3.1}$$

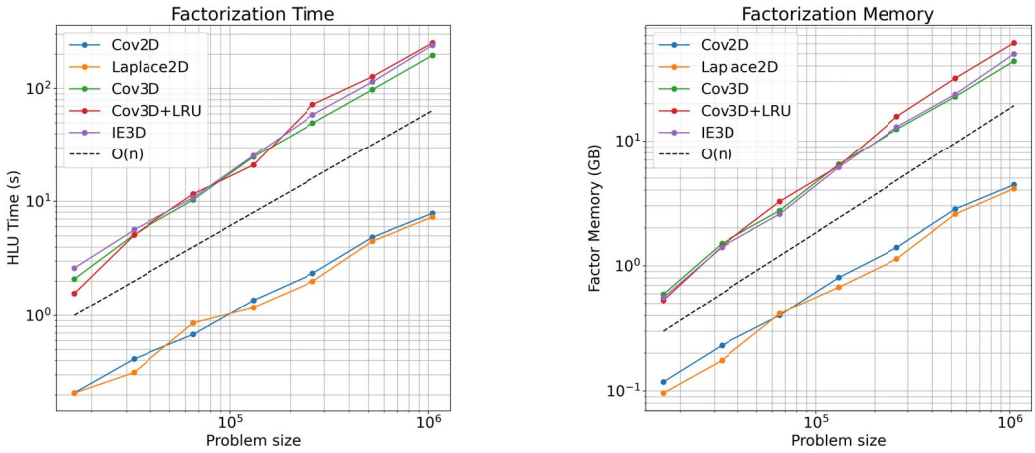
The next two applications construct the hierarchical matrix using adaptive sketching-based construction (Boukaram *et al.*, 2025), where matrix entry evaluation and fast matrix-vector products are all that is needed to construct the hierarchical matrix to a specified error threshold. First, we consider the discretization of a three-dimensional volume IE operator for the Helmholtz equation also on a uniform cubic lattice. The oscillatory IE kernel is

$$K(x, y) = \frac{\cos(\kappa |x - y|)}{|x - y|},\ x \neq y \tag{3.2}$$

with wavenumber $\kappa$ fixed at be 3 as the mesh resolution is varied. The final application involves updating a hierarchical matrix with a global LRU. This is commonly encountered during the LU decomposition of hierarchical matrices or in the multifrontal factorization of sparse matrices. For that, we take a previously generated three-dimensional covariance matrix and apply a random rank 32 update. We also increase the accuracy for this problem and increased the leaf size to 128 due to the larger ranks at the leaves making it inefficient to use the same leaf size of 64 as the other problems. These examples, typical of many applications, occupy a 'sweet spot' in which the block ranks do not grow substantially during the factorization. In more general cases, for truly 'blackbox' factorization software, some intermediate recompression may be required.

### 3.2 *Testing setup*

All tests were run on an AMD Ryzen Threadripper PRO 5995WX 64 core processor system with 512 GB of memory. The OpenMP number of threads is set to 16, which is the point where memory bandwidth




(a) Log-log plot of the factorization time for various problem sizes showing linear growth for factorization.

(b) Log-log plot of the memory consumed by the factors of various problems showing linear growth in memory.

FIG. 13. The factorization algorithm exhibits linear complexity in time and memory.

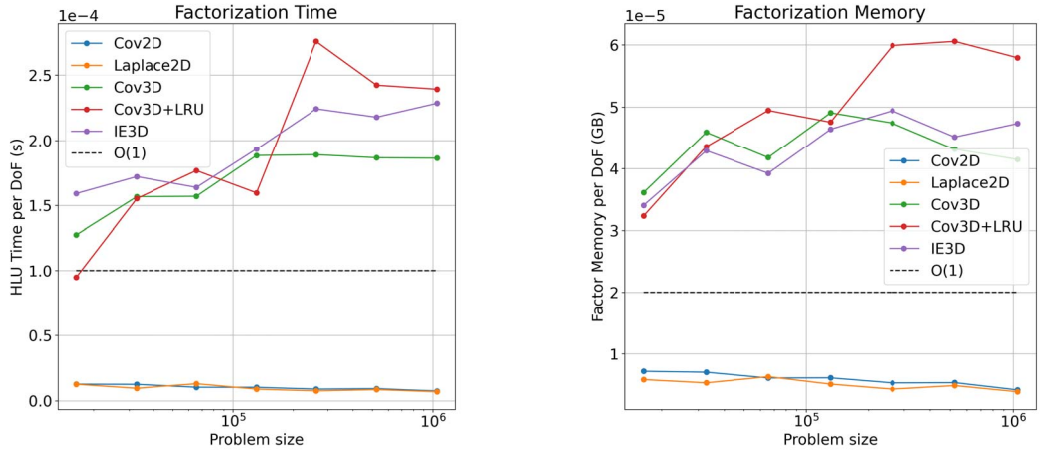
becomes the performance limiter for the batched linear algebra. We used the OpenBLAS linear algebra library and set the number of threads used to 1, since most operations are sufficiently small.

### 3.3 *Results*

*Factorization.* Figure 13(a) shows the factorization times for our applications along with $O(n)$ reference lines, clearly demonstrating the linear complexity of the algorithm. The three-dimensional problems are considerably slower than the two-dimensional ones due not only to higher ranks, but also because the sparsity constant tends to be higher for three-dimensional problems, as can be seen in Table 2, due to the $O(C_{\text{sp}}^2 k^3 n)$ complexity of the algorithm. Figure 14 shows the time and memory normalized by the problem size, demonstrating their expected constant asymptotic growth. The example with the LRU matrix exhibits some oscillation in per-dof costs caused by an increase in the sparsity constant due to the anisotropic refinement cycling across the three dimensions as the problem size increases. This is more apparent in this problem since the LRUs are random and the admissibility parameter $\eta = 0.9$ leads to a relatively coarse matrix structure with higher ranks on the upper levels. Figure 15 breaks down the runtime of the factorization into its major phases, with the partial LU dominating runtime as expected.

The basis augmentation step, where the QR decomposition of the fill-in block row followed by an SVD, is a close second despite being $O(C_{\text{sp}})$ fewer operations. This is partially due to the fact that the matrix multiplications in the Schur complement updates are far more efficient operations, and partially due to the seemingly poor scaling of the QR decompositions of the basis augmentation with the number of threads. This seems to be the case for both the two- and three-dimensional problems despite the significantly larger sparsity constants of the three-dimensional problem.

To more clearly illustrate this, Table 3 shows how increasing the number of OpenMP threads changes the factorization and solve times (in seconds) for the $n = 2^{20}$ two-dimensional and three-dimensional covariance matrices and the time (in milliseconds) for 1000 operations for GEMM and QR batches for small (S) and large (L) operands. For the GEMMs, small operands are $30 \times 30$ and large operand dimensions are $100 \times 100$, while the QR small operands are $300 \times 30$ and large operands are $1000 \times 100$.




(a) Factorization time per degree of freedom.

(b) Memory per degree of freedom.

FIG. 14. Per degree of freedom plots for the factorization time and memory demonstrating almost constant growth.

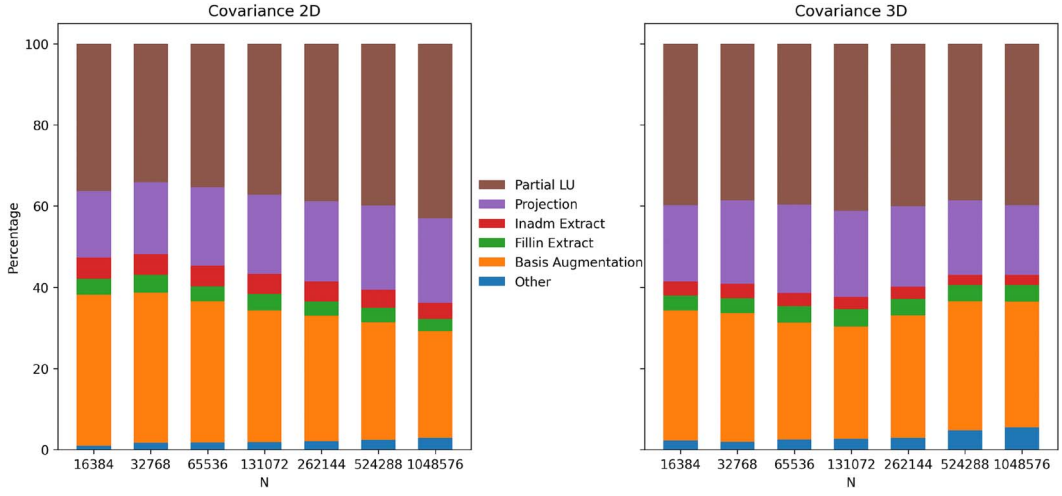



FIG. 15. The percentage of time spent in each of the major computational kernels of the factorization for the two- (left) and three-dimensional (right) covariance matrices.

Clearly, adjusting the number of threads used in each batched routine based on operand sizes would lead to more efficient batched routines like those specialized on the GPU and could yield some performance benefits, which is future work.

Replacing the Householder QR decompositions with Cholesky QR could trade accuracy for greater efficiency in the basis augmentation. Alternately, the Gram matrix approach followed by an eigenvalue decomposition could be preferable. The performance of the two-dimensional problems is mostly bandwidth limited due to the smaller ranks, limiting the benefits of increasing the number of threads

TABLE 3 *Thread scaling of the factorization and solve (in seconds) for the two- and three-dimensional covariance matrices, and the OpenMP parallelized batched GEMM and QR operations for a batch of 1000 small and large matrix operands (in milliseconds)*

| Threads | HLU two dimensions | Solve two dimensions | HLU three dimensions | Solve three dimensions | GEMM (S) | GEMM (L) | QR (S) | QR (L) |
|---|---|---|---|---|---|---|---|---|
| 1 | 28.280 | 0.376 | 848.640 | 2.389 | 1.850 | 46.470 | 37.580 | 1141.540 |
| 2 | 15.020 | 0.222 | 447.573 | 1.448 | 1.220 | 23.950 | 23.370 | 582.340 |
| 4 | 8.812 | 0.173 | 258.186 | 1.214 | 0.970 | 12.440 | 12.560 | 299.510 |
| 8 | 7.099 | 0.172 | 204.928 | 1.202 | 1.140 | 7.330 | 17.570 | 179.650 |
| 16 | 7.881 | 0.133 | 142.617 | 0.768 | 1.170 | 4.400 | 36.490 | 198.180 |

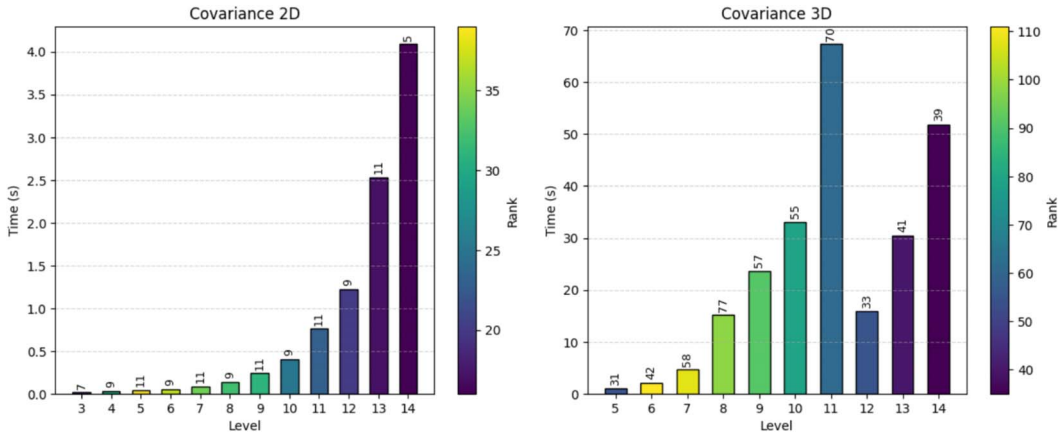
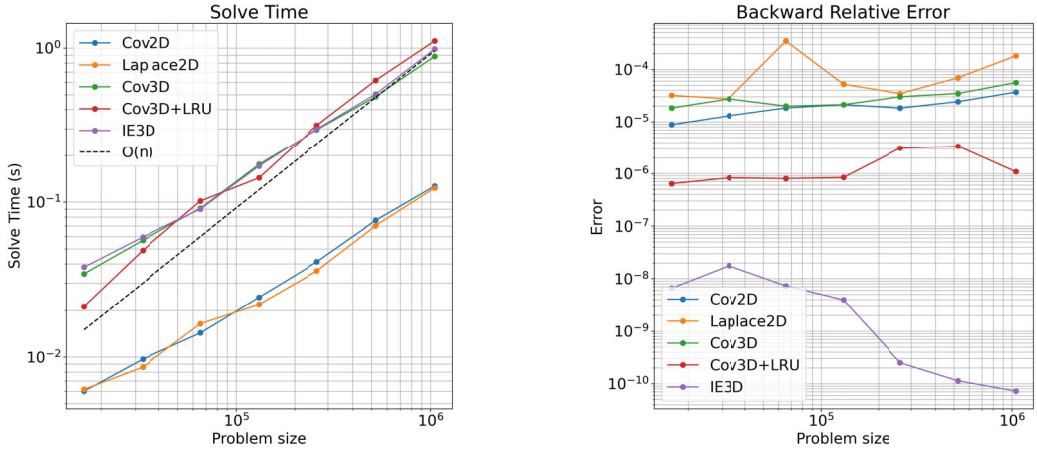



FIG. 16. The time spent in each level of the factorization for $N = 2^{20}$ for the two-dimensional (left) and three-dimensional (right) covariance matrices. The sparsity constant of the level is labeled on top of each bar and the maximum rank of each level is color coded into the bars.

used in the batched routines. The smaller batch sizes generated in the smaller problems also prevent threads from being fully utilized. Figure 16 shows the time taken at each level of the factorization for the two- and three-dimensional covariance matrices. The sudden jump in the sparsity constant for the three-dimensional problem at level 11 is due to the nonisotropic distribution of the points in the $128 \times 128 \times 64$ ($= 2^{20}$ discrete size) three-dimensional cube. This increase along with the larger ranks at that level makes the upper level take more time than the leaf level despite having only $1/8$ of the clusters, showing that the leaf level is not necessarily the most expensive part of the computation. The expected linear growth of the memory consumed by the factors of the matrix is also clearly shown in Fig. 13(b). The three-dimensional problems again consume more memory than the two-dimensional problems due to the aforementioned differences in rank and sparsity.

*Solve.* Figure 17(a) shows the solve times for our applications for a randomly generated right-hand side vector $b$ along with the $O(n)$ reference lines, showing its overall linear growth. Unlike the factorization, all operations are bandwidth limited in the solve phase and the overhead of the smaller batch sizes have a clearer impact on runtime for all applications. To compute the error in the factorization,




(a) Log-log plot of the solve time for various problem sizes showing linear growth for solves.

(b) Log-log plot of the backward error of the factorization.

FIG. 17. The factorization algorithm exhibits linear complexity in solve time and controls the error well.

we generate a random vector $x$, multiply it by the matrix $A$ to generate the right-hand side $b = Ax$, and solve for $\tilde{x} = A^{-1}b$. The relative backward error is computed as $e_b = ||A\tilde{x} - b||/||b||$ and is shown in Fig. 17(b), demonstrating the high accuracy achieved by the factorization for the given thresholds.

## 4. Conclusion

Building on prior work for $\mathscr{H}^2$ matrix construction on GPUs for classes of matrices possessing data sparsity, which arise broadly in contemporary large-scale applications, we provide new 'blackbox' factorization and solution routines exploiting the same recursive skeletonization structure and the similar modular composition. Demonstrations herein include covariance matrices in two- and three dimensions, a two-dimensional IE with a monotonic kernel, a three-dimensional IE with an oscillatory kernel, and a global low-rank update to a covariance matrix. We demonstrate the expected linear complexity in execution time and memory for systems up to one million in size, using a multi-threaded CPU implementation.

To promote a culture of reproducibility, full parameter tunings for the computational experiments are documented herein and we are releasing code for the solver in open-source. The constants of the linear scaling are described theoretically and coefficient values documented in the experiments. Algorithmic decisions for various modules are described and alternatives listed. Memory-bound phases of the algorithm are noted. At the time of submission, a GPU implementation is in progress.

We anticipate a steady appetite for exploiting data sparsity in practical large-scale applications in modeling and simulation, data analytics, and machine learning on which hierarchically low-rank solvers such as the one described herein will be deployed.

## Acknowledgments

The authors acknowledge their deep and several debts to Nicholas Higham in many aspects of their careers and work: research, pedagogy and personal inspiration. They dedicate this particular body of work to his profound legacy in computational linear algebra and far, far beyond.

## Funding

This work was supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research's Applied Mathematics Competitive Portfolios program under Contract No. AC02-05CH11231.

## References

Ambikasaran, S. & Darve, E. (2013) An $\mathcal{O}(N \log N)$ fast direct solver for partial hierarchically semi-separable matrices: With application to radial basis function interpolation. *J. Sci. Comput.*, **57**, 477–501.

Ambikasaran, S., Foreman-Mackey, D., Greengard, L., Hogg, D. W. & O'Neil, M. (2015) Fast direct methods for gaussian processes. *IEEE Trans. Pattern Anal. Mach. Intell.*, **38**, 252–265.

Bebendorf, M. & Hackbusch, W. (2003) Existence of $\mathscr{H}$-matrix approximants to the inverse FE-matrix of elliptic operators with $L^{\infty}$-coefficients. *Numer. Math.*, **95**, 1–28.

Börm, S. & Garcke, J. (2007) Approximating gaussian processes with $\mathscr{H}^2$-matrices. *European Conference on Machine Learning* (J. N. Kok et al., eds). Berlin: Springer, pp. 42–53.

Börm, S. & Reimer, K. (2013) Efficient arithmetic operations for rank-structured matrices based on hierarchical low-rank updates. *Comput. Vis. Sci.*, **16**, 247–258.

Börm, S., Grasedyck, L. & Hackbusch, W. (2003) Introduction to hierarchical matrices with applications. *Eng. Anal. Bound. Elem.*, **27**, 405–422.

Boukaram, W. H., Turkiyyah, G. & Keyes, D. (2019) Hierarchical matrix operations on GPUs: Matrix-vector multiplication and compression. *ACM Trans. Math. Softw.*, **45**, 1–28.

Boukaram, W., Liu, Y., Ghysels, P. & Li, X. S. (2025) Adaptive sketching based construction of H2 matrices on GPUs. *The 26th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2025)*. IEEE Xplore. IEEE Best Paper Award. https://doi.org/10.1109/IPDPSW66978.2025.00069.

Chandrasekaran, S., Dewilde, P., Gu, M., Lyons, W. & Pals, T. (2007) A fast solver for HSS representations via sparse matrices. *SIAM J. Matrix Anal. Appl.*, **29**, 67–81.

Chandrasekaran, S., Gu, M., Sun, X., Xia, J. & Zhu, J. (2008) A superfast algorithm for Toeplitz systems of linear equations. *SIAM J. Matrix Anal. Appl.*, **29**, 1247–1266.

Chávez, G., Liu, Y., Ghysels, P., Li, X. S. & Rebrova, E. (2020) Scalable and memory-efficient kernel ridge regression. *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Xplore, pp. 956–965. https://doi.org/10.1109/IPDPS47924.2020.00102.

Cheng, H., Gimbutas, Z., Martinsson, P.-G. & Rokhlin, V. (2005) On the compression of low rank matrices. *SIAM J. Sci. Comput.*, **26**, 1389–1404.

Coulier, P., Pouransari, H. & Darve, E. (2017) The inverse fast multipole method: Using a fast approximate direct solver as a preconditioner for dense linear systems. *SIAM J. Sci. Comput.*, **39**, A761–A796.

Faustmann, M., Melenk, J. M. & Praetorius, D. (2015) $\mathscr{H}$-matrix approximability of the inverses of FEM matrices. *Numer. Math.*, **131**, 615–642.

Ghysels, P., Li, X. S., Gorman, C. & Rouet, F.-H. (2017) A robust parallel preconditioner for indefinite systems using hierarchical matrices and randomized sampling. *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Xplore, pp. 897–906. https://doi.org/10.1109/IPDPS.2017.21.

Gillman, A., Young, P. M. & Martinsson, P.-G. (2012) A direct solver with $O(N)$ complexity for integral equations on one-dimensional domains. *Frontiers of Mathematics in China*, **7**, 217–247.

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

Ho, K. L. & Ying, L. (2016) Hierarchical interpolative factorization for elliptic operators: Integral equations. *Comm. Pure Appl. Math.*, **69**, 1314–1353.

Keyes, D., Ltaief, H. & Turkiyyah, G. (2020) Hierarchical algorithms on hierarchical architectures. *Philos. Trans. A Math. Phys. Eng. Sci.*, **378**, 20190055.

Liang, T., Chen, C., Martinsson, P.-G. & Biros, G. (2024) An $O(N)$ distributed-memory parallel direct solver for planar integral equations. *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Xplore, pp. 440–452. https://doi.org/10.1109/IPDPS57955.2024.00046.

Ma, M. & Jiao, D. (2019) Direct solution of general $\mathscr{H}^2$ -matrices with controlled accuracy and concurrent change of cluster bases for electromagnetic analysis. *IEEE Trans. Microw. Theory Techn.*, **67**, 2114–2127. IEEE Xplore. https://doi.org/10.1109/TMTT.2019.2914391.

Ma, Q. & Yokota, R. (2024) An inherently parallel $\mathscr{H}^2$-ULV factorization for solving dense linear systems on GPUs. *Int. J. High Perform. Comput. Appl.*, **38**, 314–336. https://doi.org/10.1177/10943420241242021.

Ma, Q., Deshmukh, S. & Yokota, R. (2022) Scalable linear time dense direct solver for 3-d problems without trailing sub-matrix dependencies. *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Xplore, pp. 1–12. https://doi.org/10.1109/SC41404.2022.00088.

Martinsson, P.-G. & Rokhlin, V. (2005) A fast direct solver for boundary integral equations in two dimensions. *J. Comput. Phys.*, **205**, 1–23.

Minden, V., Ho, K. L., Damle, A. & Ying, L. (2017) A recursive skeletonization factorization based on strong admissibility. *Multiscale Model. Simul.*, **15**, 768–796.

Sushnikova, D., Greengard, L., O'Neil, M. & Rachh, M. (2023) FMM-LU: A fast direct solver for multiscale boundary integral equations in three dimensions. *Multiscale Model. Simul.*, **21**, 1570–1601.

Xia, J. (2013) Randomized sparse direct solvers. *SIAM J. Matrix Anal. Appl.*, **34**, 197–227.

Yesypenko, A. & Martinsson, P.-G. (2026) Randomized Strong Recursive Skeletonization: Simultaneous Compression and LU Factorization of Hierarchical Matrices Using Matrix-Vector Products. *J. Sci. Comput.*, **106**, 3.

Zampini, S., Boukaram, W., Turkiyyah, G., Knio, O. & Keyes, D. (2022) H2Opus: A distributed-memory multi-GPU software package for non-local operators. *Adv. Comput. Math.*, **48**, 31. https://doi.org/10.1007/s10444-022-09942-6.