Title

A Distributed-Memory Package for Dense Hierarchically Semi-Separable Matrix Computations Using Randomization

Journal

ACM Transactions on Mathematical Software, 42(4)

ISSN

0098-3500

Authors

Rouet, François-Henry
Li, Xiaoye S
Ghysels, Pieter
et al.

Publication Date

2016-07-26

DOI

10.1145/2930660

Peer reviewed

A distributed-memory package for dense Hierarchically Semi-Separable matrix computations using randomization

François-Henry Rouet* Xiaoye S. Li* Pieter Ghysels* Artem Napov†

June 30, 2015

Abstract

We present a distributed-memory library for computations with dense structured matrices. A matrix is considered structured if its off-diagonal blocks can be approximated by a rank-deficient matrix with low numerical rank. Here, we use Hierarchically Semi-Separable representations (HSS). Such matrices appear in many applications, e.g., finite element methods, boundary element methods, etc. Exploiting this structure allows for fast solution of linear systems and/or fast computation of matrix-vector products, which are the two main building blocks of matrix computations. The *compression* algorithm that we use, that computes the HSS form of an input dense matrix, relies on randomized sampling with a novel adaptive sampling mechanism. We discuss the parallelization of this algorithm and also present the parallelization of structured matrix-vector product, structured factorization and solution routines. The efficiency of the approach is demonstrated on large problems from different academic and industrial applications, on up to 8,000 cores.

This work is part of a more global effort, the STRUMPACK (STRUctured Matrices PACKage) software package for computations with sparse and dense structured matrices. Hence, although useful on their own right, the routines also represent a step in the direction of a distributed-memory sparse solver.

1 Introduction

1.1 Background

Many applications involve dense matrix computations with *structured* (or *low-rank*, or *data-sparse*) matrices, i.e., matrices that are *compressible* in some sense. In some applications, these matrices are rank-deficient or nearly so and can be readily compressed exactly or approximately using such algorithms as SVD, CUR [22], or a rank-revealing factorization. In many applications, the matrix is not (nearly) singular, but contains low-rank blocks, typically the blocks away from the main diagonal. Such matrices appear in the boundary element methods and finite element methods [9, 18] for solving partial differential equations (PDEs). In the discretized matrices, the low-rank off-diagonal blocks arise because the associated Green's functions are smooth. The low-rank structured matrices also arise in applications that involve Toeplitz matrices (e.g., quantum chemistry, time-series analysis, queuing theory...), etc. Identifying and compressing these low-rank blocks is the key to reducing the storage and computational costs of many matrix operations, such as solving linear systems, performing matrix-vector products, and computing eigenvalues.

Different algebraic *low-rank representations* have been proposed in the literature. In particular, $\mathcal{H}$-matrices, $\mathcal{H}^2$-matrices, and Hierarchically Semi-Separable (HSS) matrices have been widely studied. It is not our goal to review these techniques and we recommend the references listed in [36] for an overview. Some of these low-rank representations have been successfully implemented in software packages, but we are not aware of many publicly-available parallel libraries. In previous works, two codes based on the multifrontal

*Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA. ({fhrouet,xsli,pghysels}@lbl.gov)
†Université Libre de Bruxelles, B-1050 Brussels, Belgium.

method for solving sparse linear systems embedded HSS algorithms: Hsolver, a distributed-memory geometric code for finite-difference discretizations on regular meshes [31], and StruMF, a sequential algebraic code [25]. The other software packages that use low-rank approximation techniques include: Hlib (for $\mathcal{H}$- and $\mathcal{H}^2$-matrices) [7], and MUMPS (sparse direct solver with Block Low-Rank approximation techniques) [2, 1].

1.2 Contributions of this work

Despite a large number of papers on the asymptotically low complexity of HSS-based representation and operations, the methods are mostly inaccessible to the high-performance computing community due to the lack of parallel software. Our work aims at providing a scalable package that can be used in large-scale applications. We have developed STRUMPACK - STRUctured Matrices PACKage - a package for computations with sparse and dense matrices. It combines HSS representations with a randomized sampling technique, which was not the case in our previous contributions. STRUMPACK has presently two main components: a distributed-memory dense matrix computations package and a shared-memory sparse direct solver. In this paper, we present the distributed-memory package. It is implemented using MPI and contains the following features:

- Compression into HSS form using randomized sampling.
- Solving linear systems using ULV-like factorization and solution.
- Computing HSS matrix-vector products.

STRUMPACK is a general package that does not make any assumption on the input matrix. It is algebraic (as opposed to geometric) and can work on any number of MPI processes. Our previously-developed geometric solver Hsolver also employs HSS compression and factorization kernels that can be used in a standalone way for dense matrices [32]. However, Hsolver is limited in usability – it is a simplified code that works only with power-of-two number of processes and in single precision complex arithmetic. STRUMPACK does not have these limitations and, as presented here, it employs more recent algorithm advances (e.g., HSS combined with randomized sampling). It typically outperforms Hsolver, as we demonstrate in Section 4.6.

In summary, the contributions of this work are the following:

- The library we present here (part of the STRUMPACK package) is the first randomized, distributed-memory, general purpose package for HSS matrix operations. It can use any number of MPI processes, not restricted to power-of-two as in Hsolver. It is up to 6x faster than the dense kernels used in Hsolver [32].
- We developed a flexible task-to-process mapping algorithm to accommodate non-uniform hierarchical matrix partitionings and unbalanced HSS trees. Therefore, the algorithms herein are fast for a wide range of applications (see Sections 3.1 and 4.2).
- We developed an efficient parallel adaptive sampling method that is essential for problems with rank structures that cannot be estimated *a priori*. This permits the solver to be used in a black-box fashion and increases its usability (see Section 3.3).
- We evaluated our algorithms for large-scale problems from a wide range of different academic and industrial applications, using large number of cores.

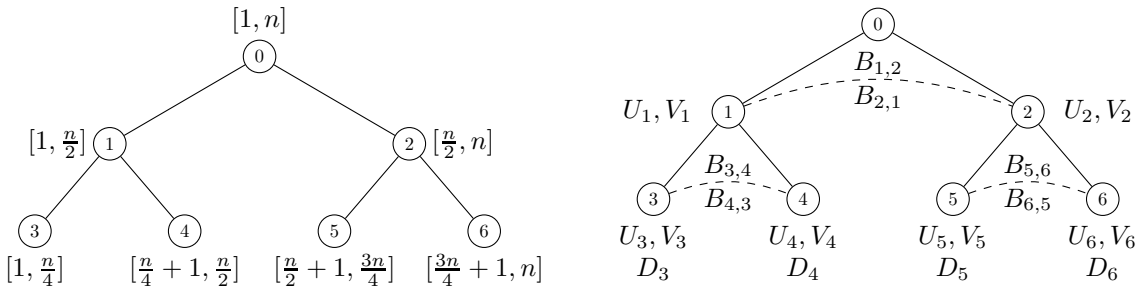
The rest of the paper is organized as follows. In Section 2, we review HSS techniques and the different ingredients of the HSS framework (HSS compression, ULV factorization and solution). In Section 3, we present our parallelization approach. We show how tasks are mapped and parallelized, we present our adaptive sampling mechanism, and we describe the communication features of our compression algorithm (number of messages and volume of communication). In Section 4, we report on results using matrices from different applications. We show how HSS algorithms behave for different applications, and we present weak and strong scaling experiment to assess the performance of our code.

2 Background on Hierarchically Semi-Separable matrices

We briefly introduce Hierarchically Semi-Separable (HSS) matrices. We mostly follow the notation used by Martinsson [23]. We recommend [36] for more theoretical aspects, and [37] for the use of HSS techniques for solving Toeplitz problems. The following references are works related to solving sparse linear systems using HSS techniques: [35] (geometric setting, serial code), [33] (algebraic setting, serial code), [34] (algebraic setting, HSS techniques combined with randomized sampling, serial code), and [31] (geometric setting, distributed-memory code).

2.1 Representation

HSS representations rely on a *cluster tree* that defines a hierarchical clustering (or partitioning) of the index set $[1, n]$, where n is the number of rows and columns of the matrix we consider. A cluster tree is such that every node τ is associated with an interval I_τ. The root node is associated with the interval $[1, n]$, and, for every node τ of the tree with children ν_1 and ν_2, we have $I_\tau = I_{\nu_1} \cup I_{\nu_2}$, and $I_{\nu_1} \cap I_{\nu_2} = \emptyset$ (for simplicity we only consider binary trees, but the generalization is straightforward). The numbering of the nodes is done top-down; the root node is 0, and a node numbered i has children numbered $2i+1$ and $2i+2$. In Figure 1(a), we show a possible cluster tree of $[1, n]$. In this example, the children of 2 are 5 and 6.



(a) Cluster tree.

(b) Three-level HSS tree.

Figure 1: Cluster tree and HSS tree associated with the example in Section 2.1.

Any $n \times n$ matrix A can be written into HSS form as follows:

1. Considering a 2×2 partitioning of A, i.e., a two-level cluster tree (one root node and two leaves), the off-diagonal blocks of A are decomposed into an "SVD-like" UBV form:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} = \begin{bmatrix} D_1 & U_1^{\text{big}} B_{1,2} V_2^{\text{big}*} \\ U_2^{\text{big}} B_{2,1} V_1^{\text{big}*} & D_2 \end{bmatrix} \tag{1}$$

   The D matrices are simply the diagonal blocks of A and the U, B, V matrices are called *generators*. We explain the reason of the "big" superscript in the third point. *This decomposition holds for any matrix A, but it is useful in practice* (i.e., it reduces storage requirements and can be used for fast operations with A), *only if the off-diagonal blocks of A are low-rank.* As mentioned in the introduction, this happens in many applications such as boundary element and finite element methods. When the off-diagonal blocks are low-rank, the U matrices are "tall and skinny", the B matrices are small and square or nearly square, and the V^* matrices are "short and wide", the aspect ratio depending on the ranks. The U, B, V matrices are computed using a rank-revealing factorization; we elaborate on this in the next sections.

   Note that in situations where the off-diagonal blocks have large ranks, we may wish to *approximate* them instead of computing their exact UBV decomposition; we show how in Section 2.2. In this case, Equation (1) provides us with an approximation of A that can be used, e.g., for preconditioning.

Note that this partitioning of the matrix corresponds to partitioning $[1, n]$ as $[1, n] = I_1 \cup I_2$.

2. Recursively, i.e., considering a three-level cluster tree, the off-diagonal blocks of the diagonal blocks of A are also decomposed into U, B, V form, and so on. After another stage of recursion:

$$A = \begin{bmatrix} \begin{bmatrix} D_3 & U_3^{\text{big}} B_{3,4} V_4^{\text{big}^*} \\ U_4^{\text{big}} B_{4,3} V_3^{\text{big}^*} & D_4 \end{bmatrix} & U_1^{\text{big}} B_{1,2} V_2^{\text{big}^*} \\ U_2^{\text{big}} B_{2,1} V_1^{\text{big}^*} & \begin{bmatrix} D_5 & U_5^{\text{big}} B_{5,6} V_6^{\text{big}^*} \\ U_6^{\text{big}} B_{6,5} V_5^{\text{big}^*} & D_6 \end{bmatrix} \end{bmatrix} \tag{2}$$

   This partitioning corresponds to $I_1 = I_3 \cup I_4$ and $I_2 = I_5 \cup I_6$.

3. There is a recursive relation between the generators appearing at different stages of recursions (which is the specificity of HSS and $\mathcal{H}^2$-matrices over the other classes of $\mathcal{H}$-matrices, and explains the use of the "big" superscript):

$$U_1^{\text{big}} = \begin{bmatrix} U_3^{\text{big}} & 0 \\ 0 & U_4^{\text{big}} \end{bmatrix} U_1 \ , \quad V_1^{\text{big}} = \begin{bmatrix} V_3^{\text{big}} & 0 \\ 0 & V_4^{\text{big}} \end{bmatrix} V_1 \tag{3}$$

   Thus,

$$A = \begin{bmatrix} \begin{bmatrix} D_3 & U_3^{\text{big}} B_{3,4} V_4^{\text{big}^*} \\ U_4^{\text{big}} B_{4,3} V_3^{\text{big}^*} & D_4 \end{bmatrix} & \begin{bmatrix} U_3^{\text{big}} & 0 \\ 0 & U_4^{\text{big}} \end{bmatrix} U_1 B_{1,2} V_2^* \begin{bmatrix} V_5^{\text{big}^*} & 0 \\ 0 & V_6^{\text{big}^*} \end{bmatrix} \\ \begin{bmatrix} U_5^{\text{big}} & 0 \\ 0 & U_6^{\text{big}} \end{bmatrix} U_2 B_{2,1} V_1^* \begin{bmatrix} V_3^{\text{big}^*} & 0 \\ 0 & V_4^{\text{big}^*} \end{bmatrix} & \begin{bmatrix} D_5 & U_5^{\text{big}} B_{5,6} V_6^{\text{big}^*} \\ U_6^{\text{big}} B_{6,5} V_5^{\text{big}^*} & D_6 \end{bmatrix} \end{bmatrix} \tag{4}$$

   This property is called the *nested basis* property.

In general, the HSS representation of A follows the structure of the cluster tree:

- For each leaf node τ, the corresponding diagonal block $D_\tau = A(I_\tau, I_\tau)$ is left untouched (uncompressed, or "full-rank").
- For each non-leaf node τ with children ν_1 and ν_2, the corresponding off-diagonal blocks $A_{\nu_1,\nu_2} = A(I_{\nu_1}, I_{\nu_2})$ and $A_{\nu_2,\nu_1} = A(I_{\nu_2}, I_{\nu_1})$ are represented (exactly or approximately) by:[1]

$$A_{\nu_1,\nu_2} \approx U_{\nu_1}^{\text{big}} B_{\nu_1,\nu_2} V_{\nu_2}^{\text{big}^*} \tag{5}$$

  Furthermore, the hierarchical relation holds, i.e., basis are *nested*:

$$U_\tau^{\text{big}} = \begin{bmatrix} U_{\nu_1}^{\text{big}} & 0 \\ 0 & U_{\nu_2}^{\text{big}} \end{bmatrix} U_\tau \ , \quad V_\tau^{\text{big}} = \begin{bmatrix} V_{\nu_1}^{\text{big}} & 0 \\ 0 & V_{\nu_2}^{\text{big}} \end{bmatrix} V_\tau \tag{6}$$

Note that we never have to store or form explicitly the "big" matrices at non-leaf nodes. Indeed, U at node τ is given by U_τ and the U_{big} matrices at its children ν_1 and ν_2, which are themselves given by looking at the grand-children of τ, and so on. At leaf nodes, $U^{\text{big}} = U$. In Figure 1(b), we show the tree corresponding to the previous example.

It is important to mention that the order of the rows and columns of matrix A matters. If A is shuffled randomly, the low-rank property is lost. In practice, matrices from real-life applications are often generated following an order that preserves the low-rank property. This was the case with all the matrices that we use in Section 4. This point is developed in the literature [25, 31, 1].

In the rest of this section, we show how to obtain the HSS form of a matrix using randomized sampling. Then, we describe the different operations that can be performed with an HSS representation: matrix-vector product, ULV factorization (a specialized LU factorization), and triangular solution.

[1] In the subsequent sections, when the context is clear, we will use equal sign instead of approximately equal.

2.2 Compression with randomized sampling

Compression, i.e., construction of the HSS form of a matrix, is the most important algorithm of the HSS framework. Once the matrix is compressed, fast operations, such as a specialized factorization or specialized matrix-vectors products can be performed. We provide algorithmic details in the following sections.

The HSS compression algorithm we use is based on randomized sampling, which is essentially done by multiplying the input matrix with a set of random vectors. It was introduced by Martinsson [23] and was also used by Xia et al. in a sparse multifrontal solver [34] and for algorithms for Toeplitz matrices [37]. The main advantage of this approach is that it does not require explicit access to all the entries of A; it only requires a matrix-vector product routine and access to selected elements of A. Therefore, A does not need to be explicitly formed, which saves memory, and the algorithm can benefit from an application-specific matrix-vector product. Furthermore, using randomized sampling simplifies the embedding of HSS kernels within a sparse solver [34]. This is the other component of the STRUMPACK project and is described in [16].

Using a classical $\mathcal{O}(n^2)$ matrix-vector product, the complexity of the compression operation is $\mathcal{O}(rn^2)$ with r the maximum rank found during the compression, that we refer to as the *HSS rank* of A. In many applications, r is much smaller than n. For example, it can be a small constant (e.g., 2D Poisson problems), or grow slowly with n (e.g., $\log n$ for 2D Helmholtz or $n^{1/3}$ for 3D Helmholtz problems) [33]. If a fast (typically $\mathcal{O}(n)$) matrix-vector product is available, the complexity drops to $\mathcal{O}(r^2n)$. Most of the floating-point operations happen when computing the samples, i.e., in the matrix-vector product. In a parallel setting, this helps load balancing in situations where very different ranks appear in different branches of the HSS tree.

We briefly recall how HSS compression *without* randomized sampling works, as described in [36]. The main property that we use is that, at each node τ, the off-diagonal row blocks and column blocks $A(I_\tau, I_0 \setminus I_\tau)$ and $A(I_0 \setminus I_\tau, I_\tau)$ are low-rank, denoting $I_0 = [1, n]$. These blocks are referred to as the *strip row Hankel blocks* and *strip column Hankel blocks* of A in [9]. Consider row blocks. We traverse the tree following a postorder, from the leaf nodes up to the root node. At a leaf node l, $A(I_l, I_0 \setminus I_l)$ is low rank and we can find a basis U_l for the rows, by using a rank revealing factorization: $A(I_l, I_0 \setminus I_l) = U_l X_l$. At the parent node p, we wish to compress $A(I_p, I_0 \setminus I_p)$. However compressing this block directly is potentially expensive and does not make use of the nested basis property. Instead, we use:

$$A(I_p, I_0 \setminus I_p) = \begin{bmatrix} A(I_{\nu_1}, I_0 \setminus I_p) \\ A(I_{\nu_2}, I_0 \setminus I_p) \end{bmatrix} = \begin{bmatrix} U_{\nu_1} X_{\nu_1}(:, I_0 \setminus I_p) \\ U_{\nu_2} X_{\nu_2}(:, I_0 \setminus I_p) \end{bmatrix} = \begin{bmatrix} U_{\nu_1} & 0 \\ 0 & U_{\nu_2} \end{bmatrix} \begin{bmatrix} X_{\nu_1}(:, I_0 \setminus I_p) \\ X_{\nu_2}(:, I_0 \setminus I_p) \end{bmatrix} \tag{7}$$

Our objective is to compress $A(I_p, I_0 \setminus I_p)$ as $A(I_p, I_0 \setminus I_p) = U_p^{\text{big}} X_p$; using the above equation, we get U_p^{big} and X_p by computing a rank revealing factorization of $\begin{bmatrix} X_{\nu_1}(:, I_0 \setminus I_p) \\ X_{\nu_2}(:, I_0 \setminus I_p) \end{bmatrix}$, instead of compressing $A(I_p, I_0 \setminus I_p)$ directly. This process is illustrated in Figure 2. Column blocks are compressed in a similar way to obtain the V generators, using the X obtained during the compression of row blocks.

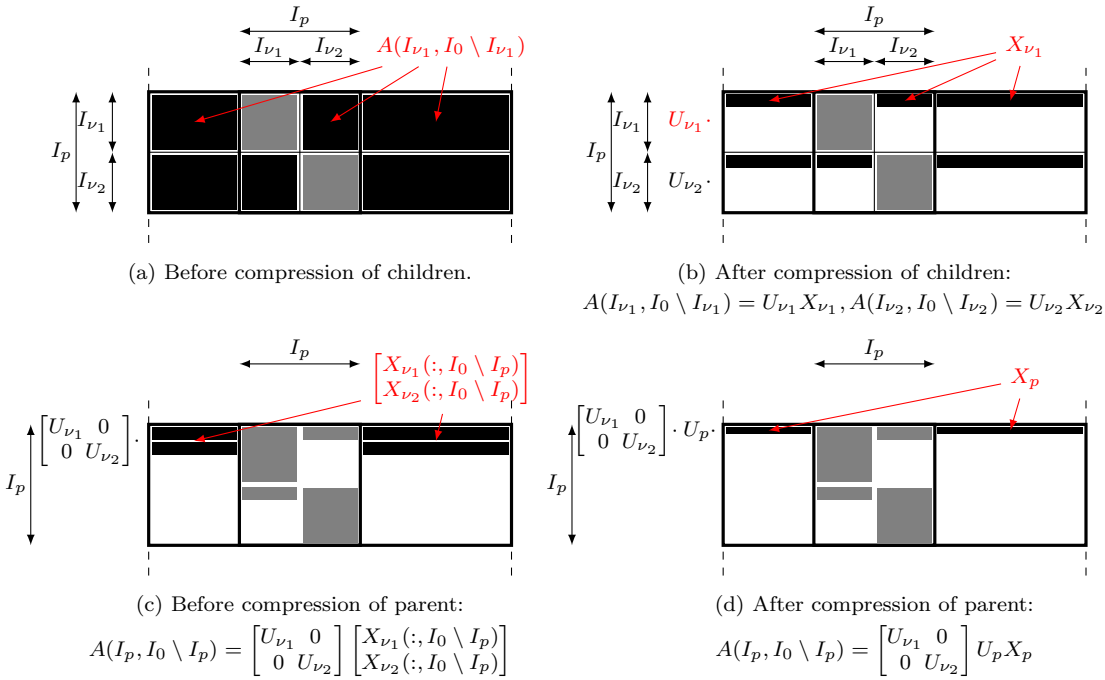
The randomized compression algorithm follows a similar process, except that it relies on samples of the input matrix instead of accessing the matrix directly. For now we suppose that the maximum rank r is known a priori. We relax this assumption in Section 3.3. Let R^r and R^c be $n \times d$ tall and skinny random matrices with $d = r + p$ columns, where p is a small oversampling parameter (Martinsson recommends $p = 10$). Let $S^r = AR^r$ and $S^c = A^* R^c$ be samples for the row and column bases of A respectively. For a non-leaf node τ with children ν_1 and ν_2, let D_τ be defined as

$$D_\tau = \begin{bmatrix} D_{\nu_1} & A_{\nu_1,\nu_2} \\ A_{\nu_2,\nu_1} & D_{\nu_2} \end{bmatrix}$$

If $\{\tau_1, \tau_2, \ldots, \tau_q\}$ are the nodes at level ℓ of the HSS tree, then

$$D^{(\ell)} = \text{diag}(D_{\tau_1}, D_{\tau_2}, \ldots, D_{\tau_q})$$

is an $n \times n$ block diagonal matrix. The main idea of the randomized sampling algorithm is to construct a



(a) Before compression of children.

(b) After compression of children:
$A(I_{\nu_1}, I_0 \setminus I_{\nu_1}) = U_{\nu_1} X_{\nu_1}, A(I_{\nu_2}, I_0 \setminus I_{\nu_2}) = U_{\nu_2} X_{\nu_2}$

(c) Before compression of parent:
$A(I_p, I_0 \setminus I_p) = \begin{bmatrix} U_{\nu_1} & 0 \\ 0 & U_{\nu_2} \end{bmatrix} \begin{bmatrix} X_{\nu_1}(:, I_0 \setminus I_p) \\ X_{\nu_2}(:, I_0 \setminus I_p) \end{bmatrix}$

(d) After compression of parent:
$A(I_p, I_0 \setminus I_p) = \begin{bmatrix} U_{\nu_1} & 0 \\ 0 & U_{\nu_2} \end{bmatrix} U_p X_p$

Figure 2: Compression process without randomized sampling, at two child nodes ν_1 and ν_2 and their parent τ. Full blocks are off-diagonal blocks to be compressed. Shaded blocks are that are left untouched.

row sample matrix $S^{(\ell)}$ for each level of the tree as

$$S^{(\ell)} = \left(A - D^{(\ell)}\right) R^r = S^r - D^{(\ell)} R^r$$

This row sample matrix $S^{(\ell)}$ captures the action of a product of the block off-diagonal part of A with a set of random vectors R^r. It is exactly this block off-diagonal part that needs to be compressed using low-rank approximation to obtain the HSS generators. Similarly, we compute a sample matrix using S^c and R^c to capture the column space of the off-diagonal blocks.

A central component of the randomized sampling algorithm is the Interpolative Decomposition (ID) [11]. The ID computes a factorization of a rank-k $m \times n$ matrix Y by expressing the columns of Y as linear combinations of a subset of columns of Y:

$$[X, J] = \mathrm{ID}(Y), \text{ s.t. } Y = Y(:, J)X \text{ where } Y \text{ is } m \times k \text{ and } X \text{ is } k \times n$$

A compression tolerance ε can be added as a parameter:

$$[X, J] = \mathrm{ID}(Y, \varepsilon), \text{ s.t. } Y \simeq Y(:, J)X \text{ where } Y \text{ is } m \times k' \text{ and } X \text{ is } k' \times n$$

where the numerical rank $k' \leq k$. The ID can be computed using, for example, a QR factorization with column pivoting [8, 28]

$$\begin{aligned}
Y &= Q\, R\, \Pi^{-1} && (\Pi\text{: permutation matrix representing column pivoting}) \\
&= Q\, \begin{bmatrix} R_1 & R_2 \end{bmatrix} \Pi^{-1} && (R_1 : k \times k) \\
&= (Q\, R_1) \left(\begin{bmatrix} I & R_1^{-1} R_2 \end{bmatrix} \Pi^{-1} \right) \\
&= Y(:, J)\, X && (Q\, R_1\text{: first columns of pivoted } Y)
\end{aligned}$$

A consequence of using Interpolative Decomposition is that $B_{\nu_1,\nu_2} = A(I^r_{\nu_1}, I^c_{\nu_2})$ is a submatrix of the original matrix A. Furthermore, it also leads to a special structure for the U_τ and V_τ generators:

$$U_\tau = \Pi^r_\tau \begin{bmatrix} I \\ E^r_\tau \end{bmatrix} \quad \text{and} \quad V_\tau = \Pi^c_\tau \begin{bmatrix} I \\ E^c_\tau \end{bmatrix}$$

where U_τ and V_τ have respective column ranks r^r_τ and r^c_τ, Π^r_τ and Π^c_τ are permutation matrices and the I's are the Identity matrices, one is of order r^r_τ, the other of order r^c_τ. This structure is exploited in the factorization, as shown in Section 2.4, and it allows for faster operations with the generators. From a memory viewpoint, we only need to store the E matrices, and the permutation matrices Π are represented by a single vector. A remarkable consequence is that when the block we want to compress is full-rank, the generators have the degenerate form $U_\tau = \Pi^r_\tau I$, and therefore they can be stored at very low cost (only the permutation information needs to be stored).

The compression algorithm works as follows:

1. Generate R^r and R^c random $n \times d$ matrices.
2. Compute the samples $S^r = AR^r$ and $S^c = A^* R^c$.
3. Traverse the tree in topological order (i.e., children before parents): at each node,
   (a) Construct local samples.
   (b) Compute generators using Interpolative Decomposition.
   (c) Update samples and random vectors to make the construction of local samples faster at subsequent nodes.

The detailed algorithm is presented in Algorithm 1. Note that in the serial case, the topological order that we follow is simply a postordering of the HSS tree. However, in the parallel case, we follow a more general topological order, as described in Section 3.1. Note that the Interpolative Decomposition (step (3)(b), line 10 in the algorithm) is the step where the user-given threshold ε is used. The QR factorization with column pivoting stops when $\frac{R_{ii}}{R_{11}} \leq \varepsilon$.

2.3 Matrix-vector product

Once a matrix is compressed into an HSS form, matrix-vector products can be computed in $\mathcal{O}(rn)$, thus typically faster than using a classical $\mathcal{O}(n^2)$ product. However, the compression cost is $\mathcal{O}(rn^2)$ using a standard non-randomized algorithm, or using a randomized algorithm based on samples computed with standard matrix-vector products; therefore, it is amortized only when multiple products are computed, either successively or with blocks of vectors. This is the case for example in iterative linear solvers or eigensolvers. The HSS matrix-vector algorithm consists of two traversals of the HSS tree, as shown in Algorithm 2. The first traversal accumulates the actions of the V generators, while the other traversal uses the U generators as well as the B_{ν_1,ν_2}, B_{ν_2,ν_1} and D_τ matrices.

2.4 ULV-like factorization

A matrix in HSS form can be factored using a special form of factorization called ULV factorization [10]. Then, the factored form can be used to obtain the solution to the linear system. We now describe the factorization algorithm, using a two-stage HSS example (i.e., a three-level tree) to aid exposition.

In the original ULV factorization, fast orthogonal transformations are used to eliminate $\mathcal{O}(n-r)$ unknowns; the remaining $\mathcal{O}(r)$ unknowns are eliminated using a standard LU factorization. The factorization we use does not use orthogonal transformations but instead it exploits the special structure of the HSS generators that comes from the Interpolative Decomposition. Algorithm 3 shows the complete ULV factorization procedure. In the following we explain how it works, starting from the one-stage HSS form (1), i.e., a two-level tree.

Algorithm 1: Computing the HSS representation of an unsymmetric matrix.

Data: $d = r + 10$ with r an upper bound for the rank of $A \in \mathbb{R}^{n\times n}$
$S^r = AR^r$ and $S^c = A^*R^c$ with $\{S^r, S^c, R^r, R^c\} \in \mathbb{R}^{n\times d}$
A tree on the index vector $[1, n]$ with an index set I_τ at each node τ

Result: Basis matrices defining the HSS matrix:
D_τ at the leaves, U_τ, V_τ at all nodes except the root
B_{ν_1,ν_2} at non-leaves for all children combinations

foreach *node τ in topological order (bottom-up traversal)* **do**
  if *node τ is a leaf* **then**
    $D_\tau = A(I_\tau, I_\tau)$
    $S^r_{\text{loc}} = S^r(I_\tau, :) - D_\tau R^r(I_\tau, :)$ $\qquad$ $S^c_{\text{loc}} = S^c(I_\tau, :) - D^*_\tau R^c(I_\tau, :)$
  else
    Let ν_1 and ν_2 be the two children of node τ
    $B_{\nu_1,\nu_2} = A(I^r_{\nu_1}, I^c_{\nu_2})$ $\qquad$ $B_{\nu_2,\nu_1} = A(I^r_{\nu_2}, I^c_{\nu_1})$
    $S^r_{\text{loc}} = \begin{bmatrix} S^r_{\nu_1} - B_{\nu_1,\nu_2} R^r_{\nu_2} \\ S^r_{\nu_2} - B_{\nu_2,\nu_1} R^r_{\nu_1} \end{bmatrix}$ $\qquad$ $S^c_{\text{loc}} = \begin{bmatrix} S^c_{\nu_1} - B^*_{\nu_2,\nu_1} R^c_{\nu_2} \\ S^c_{\nu_2} - B^*_{\nu_1,\nu_2} R^c_{\nu_1} \end{bmatrix}$
  end
  $\left[(U_\tau)^*, J^r_\tau\right] = \mathbf{ID}\left((S^r_{\text{loc}})^*\right)$ $\qquad$ $\left[(V_\tau)^*, J^c_\tau\right] = \mathbf{ID}\left((S^c_{\text{loc}})^*\right)$
  $S^r_\tau = S^r_{\text{loc}}(J^r_\tau, :)$ $\qquad$ $S^c_\tau = S^c_{\text{loc}}(J^c_\tau, :)$
  if *node τ is a leaf* **then**
    $R^r_\tau = (V_\tau)^* R^r(I_\tau, :)$ $\qquad$ $R^c_\tau = (U_\tau)^* R^c(I_\tau, :)$
    $I^r_\tau = I_\tau(J^r_\tau)$ $\qquad$ $I^c_\tau = I_\tau(J^c_\tau)$
  else
    $R^r_\tau = (V_\tau)^* \begin{bmatrix} R^r_{\nu_1} \\ R^r_{\nu_2} \end{bmatrix}$ $\qquad$ $R^c_\tau = (U_\tau)^* \begin{bmatrix} R^c_{\nu_1} \\ R^c_{\nu_2} \end{bmatrix}$
    $I^r_\tau = [I^r_{\nu_1}\ I^r_{\nu_2}](J^r_\tau)$ $\qquad$ $I^c_\tau = [I^c_{\nu_1}\ I^c_{\nu_2}](J^c_\tau)$
  end
end

Recall that each U generator has the special structure $U_\tau = \Pi^r_\tau \begin{bmatrix} I \\ E^r_\tau \end{bmatrix}$. Define $\Omega_\tau = \begin{bmatrix} -E^r_\tau & I \\ I & 0 \end{bmatrix} \Pi^{r\,T}_\tau$. Then the transformation $\Omega_\tau U_\tau = \begin{bmatrix} 0 \\ I \end{bmatrix}$ introduces a zero block on the top, where I is of order r^r_τ. Now consider the one-stage HSS decomposition (as in Equation (1)):

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} = \begin{bmatrix} D_1 & U_1 B_{1,2} V_2^* \\ U_2 B_{2,1} V_1^* & D_2 \end{bmatrix}$$

Applying Ω_1 and Ω_2, we get:

$$\begin{bmatrix} \Omega_1 & 0 \\ 0 & \Omega_2 \end{bmatrix} A = \begin{bmatrix} \Omega_1 D_1 & \begin{bmatrix} 0 \\ B_{1,2} V_2^* \end{bmatrix} \\ \begin{bmatrix} 0 \\ B_{2,1} V_1^* \end{bmatrix} & \Omega_2 D_2 \end{bmatrix}$$

At each node τ, we partition $W_\tau = \Omega_\tau D_\tau$ into the top (t) and bottom (b) parts, $W_\tau = \begin{bmatrix} W_{\tau;t} \\ W_{\tau;b} \end{bmatrix}$ where $W_{\tau;b}$

Algorithm 2: HSS matrix-vector product, for a non-symmetric matrix.

Data: HSS form: D_τ (leaves), U_τ, V_τ (all nodes except root), B_{ν_1,ν_2} and B_{ν_2,ν_1} (non-leaves). Right-hand side x (one or more columns).

Result: $b = Ax$.

foreach *node* τ *in topological order (bottom-up traversal)* **do**
  if *node* τ *is a leaf* **then**
    $y_\tau = V_\tau^* x(I_\tau, :)$
  else
    $y_\tau = V_\tau^* \begin{bmatrix} y_{\nu_1} \\ y_{\nu_2} \end{bmatrix}$
  end
end
$z_\tau = 0$ for root node
foreach *node* τ *in reverse topological order (top-down traversal)* **do**
  if *node* τ *is a leaf* **then**
    $b(I_\tau, :) = U_\tau z_\tau + D_\tau x(I_\tau, :)$
  else
    $\begin{bmatrix} z_{\nu_1} \\ z_{\nu_2} \end{bmatrix} = \begin{bmatrix} 0 & B_{\nu_1,\nu_2} \\ B_{\nu_2,\nu_1} & 0 \end{bmatrix} \begin{bmatrix} y_{\nu_1} \\ y_{\nu_2} \end{bmatrix} + U_\tau z_\tau$
  end
end

has r_τ^r rows, and we perform an LQ decomposition of $W_{\tau;t}$, $W_{\tau;t} = [L_\tau \ 0]\, Q_\tau$. Then,

$$\begin{aligned}\begin{bmatrix} \Omega_1 & \\ & \Omega_2 \end{bmatrix} A \begin{bmatrix} Q_1^* & \\ & Q_2^* \end{bmatrix} &= \begin{bmatrix} \begin{bmatrix} L_1 & 0 \end{bmatrix} & 0 \\ W_{1;b} Q_1^* & B_{1,2} V_2^* Q_2^* \\ 0 & \begin{bmatrix} L_2 & 0 \end{bmatrix} \\ B_{2,1} V_1^* Q_1^* & W_{2;b} Q_2^* \end{bmatrix} \\ &= \begin{bmatrix} L_1 & 0 & 0 & 0 \\ W_{1;b} Q_{1;t}^* & \underline{W_{1;b} Q_{1;b}^*} & B_{1,2} V_2^* Q_{2;t}^* & \underline{B_{1,2} V_2^* Q_{2;b}^*} \\ 0 & 0 & L_2 & 0 \\ B_{2,1} V_1^* Q_{1;t}^* & \underline{B_{2,1} V_1^* Q_{1;b}^*} & W_{2;b} Q_{2;t}^* & \underline{W_{2;b} Q_{2;b}^*} \end{bmatrix} \end{aligned} \tag{8}$$

Implicitly, if we swap block rows (and columns) corresponding to the {1;b} and {2;t} parts, denoted by a permutation matrix $\Gamma_{1;b\leftrightarrow 2;t} = \begin{bmatrix} I & & & \\ & 0 & I & \\ & I & 0 & \\ & & & I \end{bmatrix}$ the above transformation can be written in the ULV factored form:

$$A = \underbrace{\begin{bmatrix} \Omega_1^{-1} & \\ & \Omega_2^{-1} \end{bmatrix} \Gamma_{1;b\leftrightarrow 2;t}}_{U} \cdot \underbrace{\begin{bmatrix} L_1 & & \\ 0 & L_2 & \\ L_{2,1} & L_{1,2} & D_0 \end{bmatrix}}_{L} \cdot \underbrace{\Gamma_{1;b\leftrightarrow 2;t}^T \begin{bmatrix} Q_1 & \\ & Q_2 \end{bmatrix}}_{V} \tag{9}$$

where $L_{2,1} = \begin{bmatrix} W_{1;b} Q_{1;t}^* \\ B_{2,1} V_1^* Q_{1;t}^* \end{bmatrix}$ and $L_{1,2} = \begin{bmatrix} B_{1,2} V_2^* Q_{2;t}^* \\ W_{2;b} Q_{2;t}^* \end{bmatrix}$, and D_0 is the reduced submatrix

$$D_0 \stackrel{\text{def}}{=} \begin{bmatrix} W_{1;b} Q_{1;b}^* & B_{1,2} V_2^* Q_{2;b}^* \\ B_{2,1} V_1^* Q_{1;b}^* & W_{2;b} Q_{2;b}^* \end{bmatrix} \stackrel{\text{def}}{=} \begin{bmatrix} \tilde{D}_1 & B_{1,2} V_2^* Q_{2;b}^* \\ B_{2,1} V_1^* Q_{1;b}^* & \tilde{D}_2 \end{bmatrix}$$

This is how the name "ULV-factorization" came from [10]. In the original form, both "U" and "V" transformations are orthogonal. But here, since Ω_τ has the special structure stemming from the Interpolative

Decomposition, it may not be orthogonal. Therefore, we refer to it as a "ULV-like" factorization. Note that factorization (9) is not used in the solution procedure; instead, it is the transformation (8) that is actually used, as we show in the next section. This transformation is also used in the sparse factorization by Xia [34].

With the "L" form above, the unknowns corresponding to L_1 and L_2 can be eliminated using a regular forward substitution. The reduced submatrix D_0 corresponds to the $\mathcal{O}(r)$ remaining unknowns, and is formed at the parent node, which is also a root node, where $LU(D_0)$ is performed. This is illustrated in Figure 3, with D_0 colored red.

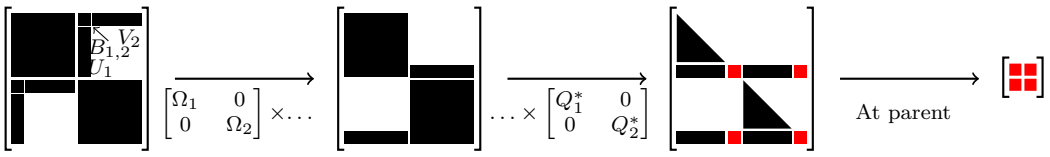


Figure 3: Illustration of the one-stage ULV factorization process.

Note that in the one-stage case presented above, we have to perform LQ factorizations of two matrices with order n rows and columns, assuming r is small; therefore the cost is $\mathcal{O}(n^3)$. To bring the asymptotic cost down, we need more levels in the HSS tree. In the next step, we consider the two-stage HSS decomposition in Equation (4), i.e., a three-level tree. We assume that the two diagonal blocks (children 1 and 2) are already transformed into ULV form (9), via $U = \mathrm{diag}(\Omega_3^{-1}, \Omega_4^{-1}, \Omega_5^{-1}, \Omega_6^{-1})$ and $V = \mathrm{diag}(Q_3, Q_4, Q_5, Q_6)$. The remaining uneliminated blocks are

$$D_1 = \begin{bmatrix} W_{3;b}Q_{3;b}^* & B_{3,4}V_4^*Q_{4;b}^* \\ B_{4,3}V_3^*Q_{3;b}^* & W_{4;b}Q_{4;b}^* \end{bmatrix},\ D_2 = \begin{bmatrix} W_{5;b}Q_{5;b}^* & B_{5,6}V_6^*Q_{6;b}^* \\ B_{6,5}V_5^*Q_{5;b}^* & W_{6;b}Q_{6;b}^* \end{bmatrix}. \tag{10}$$

The two transformations $\mathrm{diag}(\Omega_3, \Omega_4)$ and $\mathrm{diag}(\Omega_5, \Omega_6)$ can be respectively applied to the off-diagonal blocks (1,2) and (2,1) of the matrix A. Due to the nested basis property (see (3)), U_3^{big} and U_4^{big} are already annihilated to $\begin{bmatrix}0\\ I\end{bmatrix}$. Therefore, the only nonzero part of the (1,2) block of A is $U_1 B_{1,2} V_2^* \begin{bmatrix} V_5^*Q_{5;t}^* & V_5^*Q_{5;b}^* \\ V_6^*Q_{6;t}^* & V_6^*Q_{6;b}^* \end{bmatrix}$; similarly for the (2,1) block of A.

At the parent nodes 1 and 2, which are non-root nodes and have U_1 and U_2 bases associated with them, we use the transformations $\Omega_1 U_1 = \begin{bmatrix}0\\ I\end{bmatrix}$ and $\Omega_2 U_2 = \begin{bmatrix}0\\ I\end{bmatrix}$ to introduce the zero blocks. Then, we apply the above annihilation and transformation to the diagonal blocks D_1 and D_2 (see (10)), followed by the LQ decomposition of each top part. Eventually, at the root node, only $\mathcal{O}(r)$ unknowns are left and a regular LU factorization (with pivoting if needed) is used.

The two-stage transformation process can be written as follows:

$$\Gamma_{1;b\leftrightarrow 2;t} \begin{bmatrix} I & & \\ & \Omega_1 & \\ & & I \\ & & & \Omega_2 \end{bmatrix} \begin{bmatrix} & & \Gamma_{3;b\leftrightarrow 4;t} & \\ & & & \Gamma_{5;b\leftrightarrow 6;t} \\ \Gamma_{3;b\leftrightarrow 4;t} & & & \\ & \Gamma_{5;b\leftrightarrow 6;t} & & \end{bmatrix} \begin{bmatrix} \Omega_3 & & & \\ & \Omega_4 & & \\ & & \Omega_5 & \\ & & & \Omega_6 \end{bmatrix} A \begin{bmatrix} Q_3^* & & & \\ & Q_4^* & & \\ & & Q_5^* & \\ & & & Q_6^* \end{bmatrix} \begin{bmatrix} & & \Gamma_{3;b\leftrightarrow 4;t}^T & \\ & & & \Gamma_{5;b\leftrightarrow 6;t}^T \\ \Gamma_{3;b\leftrightarrow 4;t}^T & & & \\ & \Gamma_{5;b\leftrightarrow 6;t}^T & & \end{bmatrix} \begin{bmatrix} I & & & \\ & Q_1^* & & \\ & & I & \\ & & & Q_2^* \end{bmatrix} \Gamma_{1;b\leftrightarrow 2;t}^T$$

$$= \left[\begin{array}{ccc|ccc} L_3 & & & & & \\ 0 & L_4 & & & & \\ (\Omega_1 L_{4,3})_t & (\Omega_1 L_{3,4})_t & L_1 & & & \\ \hline & 0 & & L_5 & & \\ & & & 0 & L_6 & \\ & & & (\Omega_2 L_{6,5})_t & (\Omega_2 L_{5,6})_t & L_2 \\ \hline (\Omega_1 L_{4,3})_b & (\Omega_1 L_{3,4})_b & W_{1;b}Q_{1;t}^* & B_{1,2}V_2^* \begin{bmatrix} V_5^*Q_{5;t}^* & V_5^*Q_{5;b}^* \\ V_6^*Q_{6;t}^* & V_6^*Q_{6;b}^* \end{bmatrix} \begin{bmatrix} I & \\ & Q_2^* \end{bmatrix} & & D_0 \\ B_{2,1}V_1^* \begin{bmatrix} V_3^*Q_{3;t}^* & V_3^*Q_{3;b}^* \\ V_4^*Q_{4;t}^* & V_4^*Q_{4;b}^* \end{bmatrix} \begin{bmatrix} I & \\ & Q_1^* \end{bmatrix} & & & (\Omega_2 L_{6,5})_b & (\Omega_2 L_{5,6})_b & W_{2;b}Q_{2;t}^* \end{array}\right] \tag{11}$$

The algorithm is presented in Algorithm 3. The complexity is $\mathcal{O}(r^2 n)$ [10, 34]. Notice that the output of the algorithm is, at each non-root node τ, the Q_τ and L_τ matrices that represent the ULV factors, but also the matrix $W_\tau = \Omega_\tau D_\tau$ and the matrix $\tilde{V}_\tau$, that accumulates the actions of V bases and the Q transformations, as shown in lines 5 and 15 of the algorithm. $\tilde{V}_\tau$ is conceptually similar to V_τ^{big}, except it has only $\mathcal{O}(r)$ rows, corresponding to the uneliminated variables. The matrices W_τ and $\tilde{V}_\tau$ are useful for the solution phase, as shown in the next section.

Algorithm 3: ULV-like factorization of a non-symmetric matrix in HSS form.

Data: HSS form: D_τ (leaves), U_τ, V_τ (all nodes except root), B_{ν_1,ν_2} and B_{ν_2,ν_1} (non-leaves).
Result: ULV factors: Q_τ orthonormal, L_τ lower triangular (all nodes except root). LU at root. W_τ and $\tilde{V}_\tau$ to be used in solution step.

1 **foreach** *node τ in topological order (fine to coarse)* **do**
2 **if** *node τ is a non-leaf* **then**
3 $D_\tau = \begin{bmatrix} \tilde{D}_{\nu_1} & B_{\nu_1,\nu_2}\tilde{V}^*_{\nu_2;b} \\ B_{\nu_2,\nu_1}\tilde{V}^*_{\nu_1;b} & \tilde{D}_{\nu_2} \end{bmatrix}$
4 **if** *node τ is not the root node* **then**
5 $\hat{V}_\tau = \begin{bmatrix} \tilde{V}_{\nu_1;b} & 0 \\ 0 & \tilde{V}_{\nu_2;b} \end{bmatrix} V_\tau$
6 **end**
7 **else**
8 $\hat{V}_\tau = V_\tau$
9 **end**
10 **if** *node τ is the root node* **then**
11 $\left[P_\tau, L_\tau, U_\tau\right] = \mathrm{LU}\left(D_\tau\right)$
12 **else**
13 $W_\tau = \Omega_\tau D_\tau = \begin{bmatrix} -E_\tau^r & I \\ I & 0 \end{bmatrix} {\Pi_\tau^r}^T D_\tau = \begin{bmatrix} W_{\tau;t} \\ W_{\tau;b} \end{bmatrix}$
14 $\mathrm{LQ}\left(W_{\tau;t}\right) = \begin{bmatrix} L_\tau & 0 \end{bmatrix} \begin{bmatrix} Q_{\tau;t} \\ Q_{\tau;b} \end{bmatrix}$
15 $\tilde{V}_\tau = Q_\tau \hat{V}_\tau = \begin{bmatrix} \tilde{V}_{\tau;t} \\ \tilde{V}_{\tau;b} \end{bmatrix}$
16 $\tilde{D}_\tau = W_{\tau;b} Q^*_{\tau;b}$
17 **end**
18 **end**

2.5 Solution using ULV factorization

The ULV-factored form (11) can be used to solve a linear system $Ax = b$. We still use the two-stage HSS example (three-level tree) to explain the solution procedure.

Consider a partitioning of the right-hand side b and the solution vector x along the cluster tree: $b = \begin{bmatrix} b(I_1,:) \\ b(I_2,:) \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$ and similarly, $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ and the one-stage ULV factorization given in Equation (8). The solution x can be obtained by the following five steps:

1. Transform the right-hand side: $\tilde{b}_1 = \Omega_1 b_1$, and $\tilde{b}_2 = \Omega_2 b_2$;
2. Forward substitution: $y_1 = L_1^{-1}\tilde{b}_{1;t}$, $y_2 = L_2^{-1}\tilde{b}_{2;t}$;

3. Update right-hand side:
   $b_{1;b} = \tilde{b}_{1;b} - W_{1;b}Q_{1;t}^* \, y_1 - B_{1,2}V_2^*Q_{2;t}^* \, y_2$,
   $b_{2;b} = \tilde{b}_{2;b} - B_{2,1}V_1^*Q_{1;t}^* \, y_1 - W_{2;b}Q_{2;t}^* \, y_2$;

4. Triangular solution at root: $x_0 = U_0^{-1}L_0^{-1}P_0 \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$.

5. Orthogonally transform back to the original solution: $x_1 = Q_1^* \begin{bmatrix} y_1 \\ x_{0;t} \end{bmatrix}$, $x_2 = Q_2^* \begin{bmatrix} y_2 \\ x_{0;b} \end{bmatrix}$.

Next, consider the two-stage ULV transformation given in Equation (11). Algorithm 4 shows the complete procedure, which follows a bottom-up traversal of the HSS tree. We first apply all the transformations involving Ω's to the right-hand side b, to obtain $\tilde{b}$ (line 10 in the Algorithm.) Then we obtain all the intermediate variable y_τ for the non-root node τ via forward substitution (line 11 in the Algorithm). Now looking at the last block row of (11) involving D_0, we need the contributions from the children of the root node (nodes 1 and 2). For example, the intermediate solution y_1 coming from node 1 contributes to the terms $W_{1;b}Q_{1;t}^*y_1$ and $B_{2,1}V_1^*Q_{1;t}^*y_1$. Furthermore, there are contibutions coming from the grand children of the root node, i.e., nodes 3, 4, 5, and 6. For example, nodes 3 and 4 contribute via the term $B_{2,1}V_1^* \begin{bmatrix} V_3^*Q_{3;t}^* & & V_3^*Q_{3;b}^* & \\ & V_4^*Q_{4;t}^* & & V_4^*Q_{4;b}^* \end{bmatrix} \begin{bmatrix} I & \\ & Q_1^* \end{bmatrix} \begin{bmatrix} y_3 \\ y_4 \\ y_1 \end{bmatrix}$. In the general case (arbitrary number of levels), b_0 (updated right-hand side at root node) receives contributions from all the nodes in the tree, because the last block row of the L is full. In the algorithm, we accumulate these updates when going up the tree, as shown in lines 10 and 13 of the algorithm. We illustrate this in more detail in Appendix A.

Finally, the intermediate solution involving y needs to be transformed back to the original solution x (line 21 in the Algorithm). The complexity of Algorithm 4 is $\mathcal{O}(rn)$ [10, 34].

3 Distributed-memory parallelism

In this section, we present our distributed-memory algorithms. We mostly focus on the implementation of the HSS compression algorithm, as this is the most complicated of all HSS operations but also the most critical for performance. In Section 3.3, we present a novel parallel adaptive sampling mechanism.

3.1 Task mapping

The HSS tree presented in Section 2.1 is a task graph and data-dependency graph for all the different operations: compression, factorization, solution, and product. The tree structure allows for two levels of parallelism. *Tree parallelism* comes from the fact that nodes lying on different branches of the tree can be processed in parallel, independently of one another. *Node parallelism* consists in assigning a node of the tree to multiple processes. We enforce node parallelism by using parallel kernels from PBLAS [12] and ScaLAPACK [6].

We rely on a static mapping technique to assign tasks to different processes. We use the idea of the *proportional mapping* by Pothen and Sun [26], which is popular for mapping tasks along the elimination tree of sparse factorizations. The mapping process consists in a top-down traversal of the tree. All the processes are assigned to work on the root node, because this is the last task to be executed during a bottom-up traversal (e.g., compression, factorization) and the first task to be executed during a top-down traversal (e.g., matrix-vector product and triangular solution). Then, for every node in the tree, the list of processes working at that node is split among its children, proportionally to the weights (determined according to a given metric) of the subtrees rooted at these children. Consider a parent node f in the tree with nc_f children. Let p_f be the number of processes working at that node and W_i be the load of the subtree rooted

Algorithm 4: Solution of a linear system $Ax = b$ after ULV-like factorization, for a non-symmetric matrix.

Data: ULV factors: Q_τ orthonormal, L_τ lower triangular (all nodes except root). LU at root.
Result: x, solution of $Ax = b$.

foreach *node* τ *in topological order (bottom-up traversal)* **do**
  if *node* τ *is a non-leaf* **then**
    $b_\tau = \begin{bmatrix} \tilde{b}_{\nu_1;b} - W_{\nu_1;b} Q^*_{\nu_1;t}\, y_{\nu_1} - B_{\nu_1,\nu_2}\, z_{\nu_2} \\ \tilde{b}_{\nu_2;b} - B_{\nu_2,\nu_1}\, z_{\nu_1} - W_{\nu_2;b} Q^*_{\nu_2;t}\, y_{\nu_2} \end{bmatrix}$
  else
    $b_\tau = b(I_\tau, :)$
  end
  if *node* τ *is the root node* **then**
    $x_\tau = U_\tau^{-1} L_\tau^{-1} P_\tau b_\tau = \begin{bmatrix} x_{\tau;t} \\ x_{\tau;b} \end{bmatrix}$
  else
    $\tilde{b}_\tau = \Omega_\tau b_\tau = \begin{bmatrix} -E_\tau^r & I \\ I & 0 \end{bmatrix} {\Pi_\tau^r}^T b_\tau = \begin{bmatrix} \tilde{b}_{\tau;t} \\ \tilde{b}_{\tau;b} \end{bmatrix}$
    $y_\tau = L_\tau^{-1} \tilde{b}_{\tau;t}$
    if *node* τ *is a non-leaf* **then**
      $z_\tau = V_\tau^* \begin{bmatrix} z_{\nu_1} \\ z_{\nu_2} \end{bmatrix} + \tilde{V}^*_{\tau;t}\, y_\tau$
    else
      $z_\tau = \tilde{V}^*_{\tau;t}\, y_\tau$
    end
  end
end
foreach *node* τ *in reverse topological order (top-down traversal)* **do**
  if *node* τ *is a non-leaf* **then**
    $x_{\nu_1} = Q^*_{\nu_1} \begin{bmatrix} y_{\nu_1} \\ x_{\tau;t} \end{bmatrix}, \quad x_{\nu_2} = Q^*_{\nu_2} \begin{bmatrix} y_{\nu_2} \\ x_{\tau;b} \end{bmatrix}$
  else
    $x(I_\tau, :) = x_\tau$
  end
end

at a child i. The number of processes given to node i is

$$p_i = \frac{W_i}{\sum_{j=1}^{nc_f} W_j} \cdot p_f$$

This procedure is applied in a recursive fashion to all the children of f; the recursion stops when leaf nodes are reached or entire subtrees are mapped onto single processes, which happens because the number of nodes in the tree is commonly much larger than the number of processes.

The usual metric used at each step of the mapping is the workload of each subtree. However, in our case, we cannot use this because we do not know in advance the cost of processing a node since it depends on the ranks found at that node. Instead, we use the size of the interval I_τ associated with each node τ. The idea is that, at leaf nodes, the compression cost (computing local samples and performing Interpolative Decomposition) is proportional to the size of the interval. If the ranks found at different branches of the tree are balanced, workloads will be balanced. Otherwise, workloads might be unbalanced, leading to

poorer performance of the compression process. However, after the compression is done, the tree can be remapped using the rank information, which can be useful for subsequent operations (factorization, etc.) or for improving compression times for different problems from the same application. We illustrate this in Section 4.2.

An interesting property of the proportional mapping is that the traversal of every process (i.e., the set of tasks that this process executes and the order in which those are processed) is fully known in advance. Indeed, every process is in charge of a sequential subtree and takes part in the computation of the parallel nodes in the path between that subtree and the root of the elimination tree; this defines a single possible traversal. Denoting by i the root of the sequential subtree mapped on a given process, the traversal followed by that process consists of a postorder traversal of the subtree rooted at i followed by the path from i to the root node. This makes the code easier to write.

As we just saw, a node of the HSS tree can be mapped onto several processes. Within a node, our choice is to perform all the arithmetic operations with PBLAS and ScaLAPACK. All the matrices that we handle are distributed following a 2D block-cyclic scheme and each node of the HSS tree is associated with a 2D grid of processes that handle the computations. We typically try to make the grid as square as possible, as advised in the ScaLAPACK documentation [6], but the code can accommodate any kind for grid. For example, if 32 processes work at a node, our grid has $\lfloor\sqrt{32}\rfloor = 5$ rows and $\lfloor 32/5 \rfloor = 6$ columns. In this example, $32 - 6 \times 5 = 2$ processes stay idle at that node. However, it does not mean these processes are idle throughout the whole computation; they are idle at that node, but can be active at ancestors or descendants of that node. We illustrate this situation in Figure 4. In this example, node 7 is mapped on processes P_0 to P_4, but P_4 is out of the 2D grid associated with node 7 and is thus idle at that node. However it is active at nodes 4 and 6 (descendants of 7) and 15 (ancestor of 7). At a given node mapped on P processes, the associated grid is $P_r \times P_c$ and there are at most $\lfloor\sqrt{P}\rfloor - 1$ idle processes.

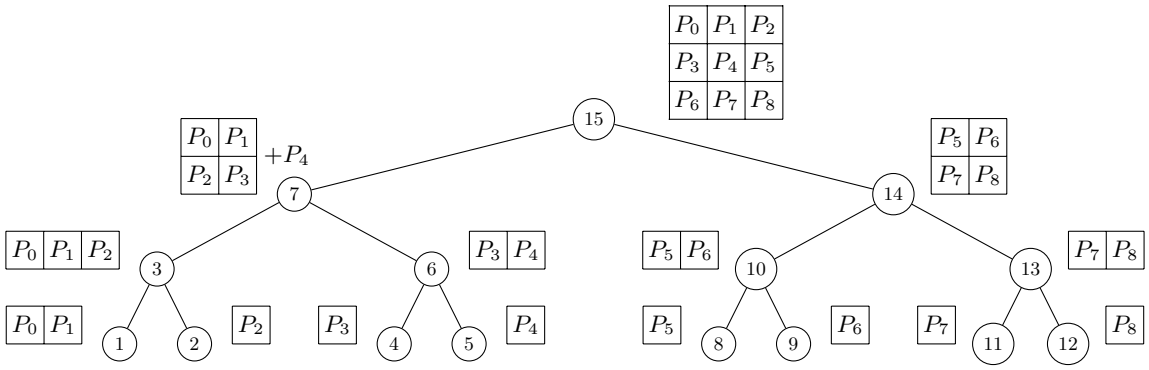


Figure 4: Proportional mapping of an HSS tree with 9 processes and uniform weights. Every node is associated with a 2D grid of processes and, sometimes, a few idle processes.

3.2 Parallel compression

We provide some details about our implementation of the parallel HSS construction (compression) algorithm. The first stage of the compression algorithm is to generate random vectors. In STRUMPACK, different generators can be used: the legacy `rand` C function, or advanced generators from the `C++11` standard, like the Mersenne Twister [24]. They can be combined with a postprocessing that enforces certain distribution of random numbers, e.g., uniform or normal.

The second stage is to generate the samples $S^r = AR^r$ and $S^c = A^*R^c$. For this, the compression algorithm needs either access to a user-given matrix-vector product or explicit access to the whole matrix A. We require the input matrix to be distributed in 2D block-cyclic form and we use the PBLAS matrix-matrix product `PxGEMM` to compute the product.

The third stage is a topological traversal of the tree, where at each node, a local sample is formed then compressed and updated. To form the sample we need access to some selected elements of the matrix. For this, the compression algorithm needs either access to a user-given routine that provides selected elements or explicit access to the whole matrix A. If the input matrix A is explicitly given (in 2D block-cyclic form), we distribute it so that at each stage of the compression, a process can extract selected elements without communicating with processes working at other nodes of the HSS tree. This is done by traversing the tree following a serialized postorder (i.e., all the processes traverse the whole HSS tree synchronously). At each node, a piece of the original matrix (shared by all the processes) is redistributed to the subset of processes that work at that node. The diagonal blocks of A correspond to leaves of the HSS tree and are redistributed to the processes working at these nodes, so that they can extract a diagonal block D_τ without communicating with processes mapped at other nodes. Similarly, the off-diagonal blocks are also distributed along the mapping of the tree, so that the B_{ν_1,ν_2} and B_{ν_2,ν_1} matrices at non-leaf nodes can be extracted without communication. For each block, we rely on a 2D block-cyclic distribution using the process grid associated with the corresponding node.

We provide an example in Figure 5, corresponding to the mapping in Figure 4. Consider node 1. The first step in the compression at node 1 is to extract $D_1 = A(I_1, I_1)$ from the input matrix. These entries are distributed on P_0 and P_1 and readily available. Then, at node 3, which is mapped on P_0, P_1 and P_2, matrix $B_{1,2} = A(I_1^r, I_2^c)$ is extracted by selecting some rows and columns of $A(I_1, I_2)$. $A(I_1, I_2)$ is distributed on P_0, P_1 and P_2, therefore the extraction can be done without communicating with processes working at other nodes.

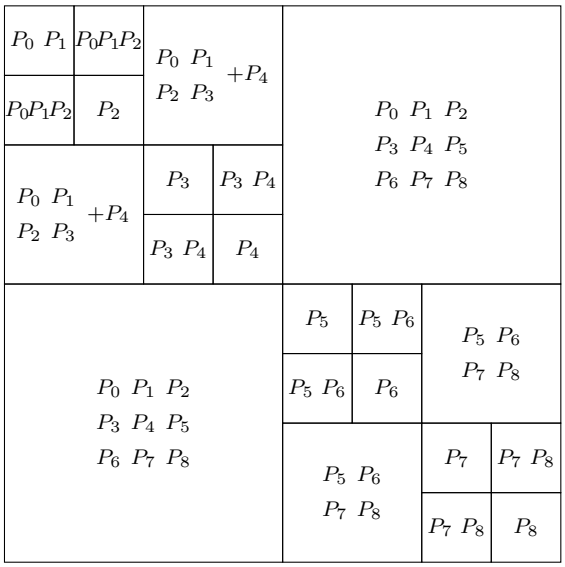


Figure 5: Distribution of the input matrix conforming to the mapping in Figure 4.

After building random vectors, computing samples, and distributing the input matrix, the postorder traversal starts. Serial subtrees (subtrees mapped on one process) are processed by a sequential compression routine that relies on BLAS and LAPACK kernels (which is usually better than using PBLAS or ScaLAPACK kernels serially). Then, parallel nodes are processed using PBLAS and ScaLAPACK operations. The main computational kernels are matrix-matrix products (performed with PBLAS `PxGEMM`) and the Interpolative Decomposition procedure described previously. For the latter, we explored two options:

1. Modifying the `xGEQP3` and `PxGEQPF` from LAPACK and ScaLAPACK respectively. These routines perform a QR factorization with column pivoting but they compute the full factorization. We modified them to embed our compression threshold ε. The factorization stops when the norm of the pivot

column becomes too small, i.e., $\frac{R_{ii}}{R_{11}} \leq \varepsilon$, with R the partial R factor. The number of columns actually eliminated is the ε-rank of the block to be compressed.

2. Implementing a Modified Gram-Schmidt (MGS) algorithm with column pivoting. The parallel implementation uses 2D block-cyclic operations. A similar version was used in Hsolver [32].

In a parallel setting, we have not observed much difference in performance between the two options. In a serial setting, the modified xGEQP3 routine, which uses a BLAS3 implementation, is typically two to three times faster than our BLAS2 MGS implementation.

3.3 Adaptive sampling mechanism

The algorithm in Section 2.2 assumes that the HSS rank r of the input matrix is known, so that the number of sample vectors d (number of columns of R^r and R^c) is chosen to be a tight upper bound of r. Indeed, d needs to be larger than r to get a stable compression, but it also needs to be not too large, because the sampling process requires $\mathcal{O}(dn^2)$ operations and can dominate the other parts of the compression stage.

In practice, r is rarely known. For some specific applications, we have a rough idea of its value, as described in Section 2.2. In order to get a more black-box compression process, it is important to design an *adaptive sampling* mechanism. This is mentioned in [23, 37] but neither an algorithm nor an implementation is described in detail. Here we explain our parallel adaptive sampling algorithm and implementation. The idea is to start with a low number of random vectors d, and whenever the rank found during Interpolative Decomposition is too large, d is increased. Instead of restarting the compression from scratch, we keep the generators that have been computed and the computation restarts at the node(s) where the rank was too large.

In a serial setting, the sketch of the algorithm is the following. When the rank at a given node τ_{fail} is too large, add new columns to R^r and R^c, compute the new columns of S^r and S^c with a product, and restart the postorder traversal:

1. At nodes preceding τ_{fail}, keep the generators (D, U, etc.) that were previously computed. Update S^r_{loc} and S^c_{loc} with new columns.

2. At τ_{fail}, update S^r_{loc} and S^c_{loc} with new columns, and recompute the Interpolative Decomposition. If the rank is again too large, restart again, otherwise proceed to the next node.

3. At nodes following τ_{fail}, proceed as before.

In this serial mechanism, a node can have three states: it can be `UNTOUCHED` if it has never been traversed before, `PARTIALLY_COMPRESSED` if the local samples have been computed but the rank obtained by Interpolative Decomposition was found too high (i.e., the traversal restarts because of this node), or `COMPRESSED` if the generators have been successfully computed. There can be at most one `PARTIALLY_COMPRESSED` node in the tree. All the nodes that precede that node in the postorder are necessarily `COMPRESSED`, and all the nodes that follow that node in the postorder are necessarily `UNTOUCHED`.

In a parallel execution, since we follow a parallel topological ordering of the tree instead of a serial postorder, we have different options. The choice we made is to implement a "late notification" mechanism. Whenever a process finds that the number of random vectors is not sufficient, it does not immediately notify the other processes. Instead, it simply invalidates the current node by leaving it `UNTOUCHED`. Then, whenever a parent node is activated, we check the state of its two children. If they are not both `COMPRESSED`, the parent node is left `UNTOUCHED`. Therefore, all the ancestors of the node that failed are left untouched. All the processes meet at the root node and can generate new random vectors, recompute samples, and restart the traversal. The difference with the serial case is that the tree can contain several `PARTIALLY_COMPRESSED` nodes. All the descendants of these `PARTIALLY_COMPRESSED` (i.e., failed) nodes are compressed, and all their ancestors have been left `UNTOUCHED`. This adaptive sampling mechanism is shown in Algorithm 5.

The main idea of this approach is that the different branches make as much progress as possible as long as the number of random vectors is sufficient. In the serial case, whenever a node fails, the traversal restarts

with more random vectors, meaning that the subsequent branches will be processed with more – and maybe unnecessary – random vectors. As a consequence, different executions on different numbers of processes will lead to slightly different ranks and HSS representations.

Another choice, that we have not implemented, would be an "early notification" mechanism where processes are notified as early as possible that a node has failed somewhere in the tree. This is more complicated to implement and requires asynchronous communications to avoid barriers at each level or node of the tree. It is not clear that it would be significantly faster.

Algorithm 5: Processing a node τ.

if *myid is in the 2D grid of* τ **then**
  if τ *non-leaf and not all children are* `COMPRESSED` **then**
    state stays `UNTOUCHED`
    return
  end
  `// Sampling`
  if *node is* `UNTOUCHED` **then**
    Extract D or B_{12}, B_{21}, compute local samples S^r_{loc} and S^c_{loc}
  else
    Compute updates to the samples, e.g., $S^r_{upd} - DR^r_{upd}$ or $\begin{bmatrix} S^r_{\nu_1\, upd} - B_{12}R^r_{\nu_2\, upd} \\ S^r_{\nu_2\, upd} - B_{21}R^r_{\nu_1\, upd} \end{bmatrix}$
  end
  `// Interpolative Decomposition`
  if *node is* `PARTIALLY_COMPRESSED` **then**
    `// Merge updates into the samples and random vectors`
    $R^r \leftarrow \left[R^r\, R^r_{upd}\right],\ S^r \leftarrow \left[S^r\, S^r_{upd}\right],\ R^c \leftarrow \left[R^c\, R^c_{upd}\right],\ S^c \leftarrow \left[S^c\, S^c_{upd}\right]$
  end
  if *node is not* `COMPRESSED` **then**
    Try Interpolative Decomposition of S^r **if** *rank too small* **then**
      Throw away U and I_r
      Mark node as `PARTIALLY_COMPRESSED`
      return
    end
    Same for S^c $S^r \leftarrow S^r(I_r, :),\ S^c \leftarrow S^c(I_r, :)$
  else
    $S^r_{upd} \leftarrow S^r_{upd}(I_r, :),\ S^c_{upd} \leftarrow S^c_{upd}(I_r, :)$
  end
  `// Update`
  if *node is* `UNTOUCHED` **then**
    $R^r = V^* \times \ldots,\ R^c = V^* \times \ldots$
  else
    $R^r_{upd} = V^* \times \ldots,\ R^c_{upd} = V^* \times \ldots$
  end
  if *node is* `COMPRESSED` *and parent is* `UNTOUCHED` **then**
    `// Merge updates into the samples and random vectors`
    $R^r \leftarrow \left[R^r\, R^r_{upd}\right],\ S^r \leftarrow \left[S^r\, S^r_{upd}\right],\ R^c \leftarrow \left[R^c\, R^c_{upd}\right],\ S^c \leftarrow \left[S^c\, S^c_{upd}\right]$
  end
  Mark node as `COMPRESSED`
else
  `// myid is out of the 2D grid`
  Receive state from P_τ **if** *state==*`COMPRESSED` **then**
    Receive ranks and indices from P_τ.
  else
    restart()
  end
end

3.4 Communication analysis

We briefly analyze the amount of communication of our parallel compression algorithm. The analysis is similar to the one we derived previously on non-randomized algorithms [32]. We consider that each node of the HSS tree has the same rank r for its U and V generators; for some applications, a specific rank pattern can be used instead, as it is sometimes done in the literature [33, 34], but this is not our goal here. We also consider that, at the leaf nodes, the diagonal blocks have size $\mathcal{O}(r)$. Finally, we assume that the number of processes is a power of 2, and the HSS tree is a complete binary tree. The pair [#messages, #words] is used to count the number of messages and the number of words transferred during a given operation, typically along the critical path. For example, a broadcast of w words among p processes is modeled as $[\log p, w \log p]$. This assumes that the broadcast follows a tree-based implementation; there are $\log p$ steps on the critical path (any branch of the tree) and w words are transferred at each step, yielding $\log p$ messages and $w \log p$ words.

We denote n the size of the matrix and p the total number of processes. The parallel compression algorithm has three main steps:

1. Matrix-matrix product to compute the samples. We use the `PxGEMM` routine from PBLAS that relies on the SUMMA algorithm [30] and can be modeled, asymptotically, as $[r \log p, \frac{rn}{\sqrt{p}}]$. This relies on the fact that, when computing a product $S = AR$, the `PxGEMM` routine selects an algorithm that reduces communication based on the size of the operands A, S, R. In our case, matrix A is the largest operand, so `PxGEMM` chooses an algorithm that communicates only S and R. The selection strategy is described in [17].

2. Initial distribution of the matrix along the HSS tree, as described in Section 3.2. This is a serialized postorder traversal of the parallel part of the tree, where, at each node τ, we use the `PxGEMR2D` routine from ScaLAPACK to redistribute a block of the matrix with size $n_\tau \times n_\tau$ from the p processes to the p_τ processes that work at τ. The cost for one such redistribution is $[p, \frac{n_\tau^2}{p_\tau}]$ for the receiving processes and $[p_\tau, \frac{n_\tau^2}{p}]$ for the sending processes [27]. To get the total cost, we sum over the $\mathcal{O}(p)$ nodes of the parallel part of the tree, and we use the fact that, at level i (0 being the root node), a node τ is associated with two blocks of the original matrix with $n_\tau = \frac{n}{2^i}$ rows and columns and is mapped on $p_\tau = \frac{p}{2^i}$ processes. Each level has 2^i nodes; at a given level, each process is receiver at one node (the node mapped on that process) and sender at $2^i - 1$ nodes. Therefore, the number of messages is

$$\sum_{\text{level } i=1}^{\log p} \left(1 \cdot p + (2^i - 1)\frac{p}{2^i}\right) = 2p \log p - p \sum_{\text{level } i=1}^{\log p} \frac{1}{2^i} = p \log p - p \cdot \mathcal{O}(1) = \mathcal{O}(p \log p)$$

   Similarly, the number of words to be transferred is:

$$\sum_{\text{level } i=1}^{\log p} \left(1 \cdot \frac{(n/2^i)^2}{p/2^i} + (2^i - 1)\frac{(n/2^i)^2}{p}\right) = \frac{n^2}{p} \sum_{\text{level } i=1}^{\log p} \frac{2^{i+1} - 1}{2^{2i}} = \frac{n^2}{p} \cdot \mathcal{O}(1) = \mathcal{O}(\frac{n^2}{p})$$

   Therefore, the cost for the initial distribution is, asymptotically, $[p \log p, \frac{n^2}{p}]$.

3. Postorder traversal of the tree to compute the generators. At a given node, there are three main ingredients:

   (a) Matrix-matrix products to compute the samples and updates. Using the above assumptions, all the blocks have size $\mathcal{O}(r) \times \mathcal{O}(r)$ (e.g., $2r \times r$). The cost is thus $[r \log p_\tau, \frac{r^2}{\sqrt{p_\tau}}]$.

   (b) Interpolative decomposition of a block of size $\mathcal{O}(r) \times \mathcal{O}(r)$ with rank $\mathcal{O}(r)$; the cost is $[r \log p_\tau, r^2 \frac{\log p_\tau}{\sqrt{p_\tau}}]$ (using Equation (4.1) from [32] with $M = N = r$).

   (c) Redistribution of blocks of size $\mathcal{O}(r) \times \mathcal{O}(r)$ to the parent; the cost is $[1, \frac{r^2}{p_\tau}]$ [32].

The term corresponding to the redistribution (c) is negligible compared to the two other terms, and the term corresponding to Interpolative Decompositions (b) dominates the term corresponding to local matrix-matrix products (a). We sum (b) over the critical path (a branch of the tree). This time we number the levels so that the leaves of the parallel tree are at level 0, and the root is at level $\log p$. At level i, a node is mapped on $p_i = 2^i$ processes. The number of messages is

$$\sum_{i=1}^{\log p-1} r \log p_i = r \sum_{i=1}^{\log p-1} i = \mathcal{O}(r \log^2 p)$$

The number of words is, similarly,

$$\sum_{i=1}^{\log p-1} r^2 \frac{\log p_i}{\sqrt{p_i}} = r^2 \sum_{i=1}^{\log p-1} \frac{i}{2^{i/2}} = \mathcal{O}(r^2)$$

We summarize the results in the following table:

Algorithm	Messages	Words
ScaLAPACK *LU*	$\mathcal{O}(n \log p)$	$\mathcal{O}\left(n^2 \frac{\log p}{\sqrt{p}}\right)$
Non-randomized HSS compression	$\mathcal{O}(p + r \log^2 p)$	$\mathcal{O}\left(\frac{n^2}{p} + rn + r^2 \log p\right)$
Randomized HSS compression	$\mathcal{O}(p \log p + r \log p + r \log^2 p)$ dist GEMM tree	$\mathcal{O}\left(\frac{n^2}{p} + \frac{rn}{\sqrt{p}} + r^2\right)$ dist GEMM tree

Table 1: Summary of communication costs.

Now we take a closer look at the various communication costs in the randomized algorithm (last row of Table 1).

- In terms of latency, the initial distribution dominates for problems with small maximum rank, while the traversal of the tree dominates for problems with large rank.
- In terms of bandwidth, when the rank is large, i.e., $r > \mathcal{O}(\frac{n}{\sqrt{p}})$, the traversal of the tree dominates the matrix-matrix product, and the matrix-matrix product dominates the initial distribution. When r is small, i.e., $r < \mathcal{O}(\frac{n}{\sqrt{p}})$, the initial distribution dominates the matrix-matrix product, and the matrix-matrix product dominates the traversal of the tree.

Comparing our randomized compression algorithm to ScaLAPACK LU, one can observe that, for problems with small rank r, our algorithm communicates fewer messages and less communication volume than ScaLAPACK does. However, for a large rank, it can be the opposite. We illustrate this in Section 4.6.

Comparing our randomized compression algorithm to the non-randomized one previously developed, we observe the following:

- In terms of latency, our algorithm has a slightly larger complexity due to the $\log p$ in the distribution term and the latency of the matrix-matrix product. We are investigating a way to reduce the number of messages to $\mathcal{O}(p)$ in the initial distribution phase. For the matrix-matrix product, we could benefit from advances in communication-avoiding algorithms, such as the 2.5D matrix multiplication [29].
- In terms of bandwidth, for both compression algorithms, the first term corresponds to the initial distribution of the input matrix (Step (2) above). Afterwards, in the non-randomized HSS compression, there is a term for the *row compression* (rn) and a term for the *column compression* ($r^2 \log p$). In

the randomized algorithm, we have a term corresponding to the matrix-matrix product used for the sampling phase, and a term corresponding to the tree traversals. These terms are smaller than what appears in the communication cost of the non-randomized algorithm. Therefore, our randomized algorithm communicates fewer words, and we expect better performance in practice. We illustrate this in Section 4.6.

3.5 Parallel factorization, solution, and product

The parallelization strategy for the factorization, triangular solution, and matrix-vector product are similar to the one we use for the compression. We exploit both tree parallelism, using a proportional mapping of the tasks, and node parallelism, by using PBLAS and ScaLAPACK operations. Serial subtrees are processed using sequential routines written using BLAS and LAPACK.

4 Experimental results

4.1 Applications

We report experimental results using the following matrices:

- **Toeplitz matrix:** a matrix $A = [a_{i,j}]$ is a *Toeplitz matrix* (or diagonal-constant matrix) if $\forall(i,j), a_{i,j} = a_{i+1,j+1}$. We experimented with two Toeplitz matrices. The first one is a simple matrix with $a_{i,i} = n^2$ and $a_{i,j} = i - j$. It is diagonally dominant and yields very low HSS rank (a small constant). The second one is a kinetic energy matrix from quantum chemistry [19]; $a_{i,i} = \frac{\pi^2}{6}$ and $a_{i,j} = \frac{(-1)^{i-j}}{(i-j)^2 d^2}$ where d is a discretization parameter (grid spacing). This matrix yields slightly larger maximum rank (that grows slowly with n) and is fairly ill-conditioned. This is a collaboration with D. J. Haxton and J. Jones at Lawrence Berkeley National Laboratory.

- **Matrices from boundary element methods:** these matrices are known to be structured [18, 4]. We obtained matrices from G. Sylvand (Airbus), and B. Notaros and A. Manic (Colorado State University). The matrices represent electromagnetic spheres (or collections of spheres). These matrices are known to be structured although the maximum rank is often large.

- **Matrices from finite differences:** it is known that the *inverse* of a sparse matrix arising from finite differences is dense and structured [18, 9]. More specifically, the dense matrices that appear during sparse Gaussian Elimination are structured. Different approaches have been used to exploit this fact, especially in the context of the *multifrontal method* [14]: using Block Low-Rank representations [1], Hierarchically Off-Diagonal Low-Rank matrices [3] and HSS techniques [33, 34, 31]. In our experiments, we use dense matrices coming from the sparse factorization of the discretized Helmholtz equation; these matrices are generated by our code Hsolver [31].

- **Covariance matrices:** spatially correlated Gaussian random fields are useful in many modeling applications. They can be generated by solving an eigenvalue problem with a *covariance matrix* [21]. These matrices are dense and are generally very large as they have as many degrees of freedom as the computational (physical) domain. However they are often compressible. We experimented with a covariance matrix generator provided by Panayot Vassilevski and Umberto Villa at the Lawrence Livermore National Lab, that relies on the MFEM code [20].

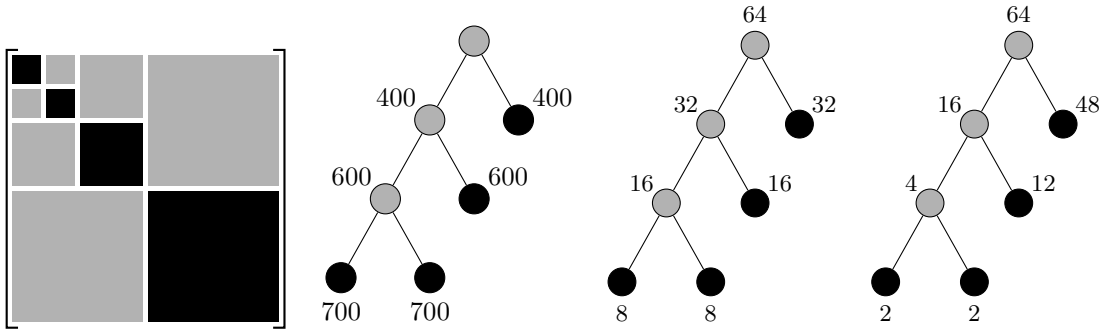
- **$\mathcal{H}-$matrices:** we use an in-house matrix generator that produces $\mathcal{H}-$matrices (i.e., they have low-rank off-diagonal blocks but there is no recursive relation between the different blocks) with prescribed size. Such matrices can be compressed using HSS techniques; even though the maximum HSS rank will be larger than the maximum $\mathcal{H}-$rank, the compression can sometimes be done faster, depending on the problem.

We use two parallel machines at the National Energy Scientific Computing Center (NERSC). Hopper is a Cray XE6 system with 6384 nodes; each node has two twelve-core 2.1 GHz AMD Opteron 6172 processors and 32 GB of main memory. Edison is a Cray XC30 system; each node has two twelve-core 2.4 GHz Intel Xeon E5-2695 processors and 64 GB of main memory.

4.2 General trees

In most of our examples and experiments, we use complete binary trees. Here, we briefly illustrate that our code can handle more general trees. This is important, as, in some applications, the clustering of the variables might not be a straightforward recursive bisection, thus the tree might not be balanced. Here, we use an example where the tree is a binary "comb", i.e., for each pair siblings, only one of the siblings has children. Hilbert matrices exhibit such a structure [4].

The matrix we use is an $\mathcal{H}-$matrix with size $40,000 \times 40,000$. It has the structure illustrated in Figure 6(a), corresponding to the comb tree in Figure 6(b).



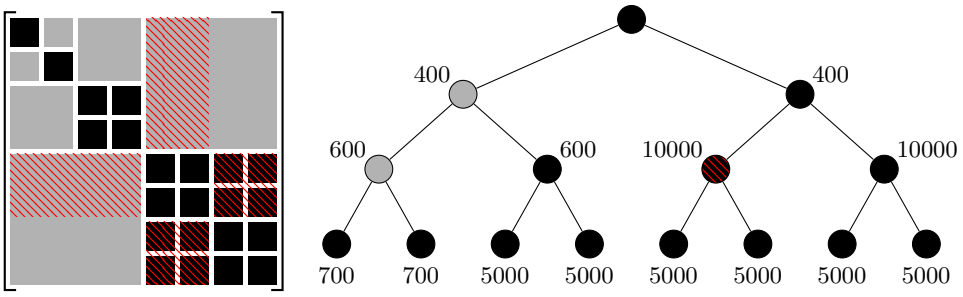
(a) Matrix structure. (b) Ranks. (c) Uniform mapping. (d) Weighted mapping.

Figure 6: Structured matrix with a comb-shaped clustering tree (a). HSS compression with a comb-shaped HSS tree yields low maximum rank (b). The tree can be mapped to processes using a uniform mapping where pairs of siblings are mapped on the same number of processes (c) or a mapping that assigns more processes to right nodes (d). For example, in (c), the children of the root node are both mapped on 32 processes, but in (d) and they are mapped on 16 and 48 processes.

In Table 2, we report some experiments with this matrix. We compare the effect of using a comb-shaped tree instead of a binary tree for the HSS compression, and we illustrate that we can modify the weights used in the proportional mapping to improve performance. In the first experiment, the HSS compression is based on a binary tree with 4 levels shown in Figure 7(b). One can easily understand why the maximum rank is $\frac{40000}{4} = 10000$; it comes from the fact that the (2,2) block of the matrix, of size $20,000 \times 20,000$ is not structured, i.e., not HSS compressible. Its off-diagonal blocks, of size $10,000 \times 10,000$ are full-rank. This yields the maximum rank because, at the striped node in Figure 7(b), the blocks to be compressed are the striped blocks in Figure 7(a) and they have rank 10,000 because they contain full-rank blocks (black in the figure).

In the second experiment, the HSS compression is based on a comb tree with the same structure as in Figure 6(b). The compression is much faster and the maximum rank is only 700 (a parameter of our example).

Finally, the last experiment consists in remapping the HSS tree by modifying the weights used in the proportional mapping. Instead of using even weights (which yield an even splitting of processes between the left and right branches of the tree), we choose to attribute more processes to right nodes. Whenever a pair of siblings is mapped, the right child inherits from 75% of the processes working at its parent. This is motivated by the fact that, in the factorization, we have to perform the LQ factorization of a $20,000 \times 20,000$ block (corresponding to the (2,2) block of the original matrix), which is the most costly operation in the



(a) Matrix structure. (b) Tree structure and ranks.

Figure 7: Matrix from Figure 6 compressed using a complete binary tree.

factorization and corresponds to the right child of the root node. By simply changing the weights in the mapping procedure using some knowledge of the input matrix, we significantly speed-up the factorization (65.8 seconds instead of 101.4 seconds), at the price of a small increase in the compression time (19.5 seconds instead of 14.1).

	Binary tree	Comb tree	Comb tree, remapped
HSS compression time (s)	704.1	14.1	19.5
Maximum rank	10000	700	700
ULV factorization time (s)	122.0	101.4	65.8

Table 2: Experiments with the structured matrix in Figure 6 using 64 MPI tasks.

This simple example illustrates that our code is flexible. It can handle different tree structures, and different task-to-process mappings, using any number of processes. In Hsolver, this was not the case. The code was restricted to some specific problems, relied on complete binary trees, and could only work with power-of-two number of processes.

4.3 Solving linear systems

In this section, we illustrate the performance of our code for seven different matrices from the abovementioned applications. The results are reported in Table 3. For each matrix, we experiment with four different compression thresholds ($\varepsilon = 10^{-8}, 10^{-6}, 10^{-4}, 10^{-2}$) and we report statistics for the HSS compression, the ULV factorization and the triangular solution with iterative refinement. We provide run time, number of floating-point operations, size of the HSS/ULV factors, and we compare with the run time for solving a system with ScaLAPACK. In terms of memory, ignoring small temporary storage and communication buffers, the memory footprint for ScaLAPACK is simply the storage of the matrix A. In our new code, the memory usage consists not only of the original matrix, but also storage for the random vectors and the samples, and storage for the HSS and ULV factors. We report a *memory overhead*, which represents the extra memory usage of STRUMPACK relative to that of ScaLAPACK.

Among this collection of matrices, the data type for the two matrices BEMMultiSphere and Schur100 is single precision. The other matrices are of double precision. For the single precision input, the compression threshold 10^{-8} is very small, leading to almost no compression. For example, for matrix Schur100, the (1,2) block is of size 5,000, whereas the maximum rank is 4933 with $\varepsilon = 10^{-8}$, which is essentially full rank. The memory overhead is much larger with HSS representation. This is mostly due to the ULV factors. Indeed, the special structure of the U and V generators keep memory usage low when blocks are full-rank, and the HSS form has roughly the same memory footprint as the input matrix. However, in this situation, ULV factors are usually much larger than the HSS form. Therefore, the practical use of HSS algorithms is not with very small compression threshold ε.

Matrix	Size	ε	STRUMPACK: solution with HSS compression and factorization									Solution with ScaLAPACK		Comparison: HSS vs ScaLAPACK	
			HSS compression				ULV factorization			Solution+IR					
			Max rank	Factors (MB)	Flops ($\times 10^9$)	Time (s)	Factors (MB)	Flops ($\times 10^9$)	Time (s)	Flops ($\times 10^9$)	Time (s)	Flops ($\times 10^{12}$)	Time (s)	Memory overhead	Time speedup
Simple Toeplitz	80,000	10^{-8}	2	14.6	307.3	11.4	37.2	0.10	0.008	0.008	0.2	341.3	856.6	0.1%	**75.2**
		10^{-6}	2	14.2	307.3	11.3	37.0	0.10	0.007	0.025	1.2			0.1%	68.1
		10^{-4}	3	13.3	307.3	11.3	36.3	0.09	0.006	0.049	2.8			0.1%	60.5
		10^{-2}	3	13.2	307.3	11.3	36.2	0.09	0.006	0.088	5.4			0.1%	51.0
QChem Toeplitz	80,000	10^{-8}	169	55.1	6411.2	19.0	152.7	1.40	0.04	1.0	1.5	341.3	894.1	1.5%	43.5
		10^{-6}	147	42.1	5126.6	17.6	110.0	0.86	0.03	0.9	2.0			1.4%	**45.5**
		10^{-4}	120	33.3	3843.7	16.7	83.6	0.58	0.02	1.7	5.6			1.0%	40.0
		10^{-2}	30	18.1	1280.6	13.4	42.7	0.12	0.01	N/A	N/A			0.5%	N/A
HMatrix	80,000	10^{-8}	787	2235.3	24707.5	52.5	7897.7	1865.0	3.5	2.1	0.22	341.3	862.1	10.3%	15.3
		10^{-6}	785	2263.4	24723.3	53.8	7966.9	1897.9	3.5	4.3	0.77			10.3%	14.9
		10^{-4}	4	14.3	409.7	11.1	37.1	0.1	0.006	0.02	0.72			0.2%	**72.9**
		10^{-2}	2	13.2	409.7	11.1	36.2	0.1	0.006	0.02	1.26			0.2%	69.7
BEM Acoustics	10,002	10^{-8}	1433	722.0	972.4	9.6	2029.2	401.8	3.1	0.5	0.07	0.7	3.3	120.1%	0.3
		10^{-6}	1016	507.6	624.3	4.7	1265.5	172.6	1.7	0.5	0.1			82.4%	0.5
		10^{-4}	793	420.0	453.6	3.3	988.8	109.2	1.1	0.7	0.3			66.6%	**0.7**
		10^{-2}	379	288.4	243.1	1.7	667.4	54.4	0.5	N/A	N/A			44.9%	N/A
BEM Multi Sphere	27,648 (Single prec.)	10^{-8}	5995	4489.0	38980.1	354.5	15897.3	26406.2	60.4	8.9	0.9	14.1	30.7	403.2%	0.07
		10^{-6}	2145	811.7	8499.1	29.1	2568.9	1015.6	3.4	1.5	0.5			53.5%	0.9
		10^{-4}	1488	425.6	5396.0	15.0	1237.9	322.8	1.2	1.0	0.7			38.4%	1.8
		10^{-2}	800	184.0	3179.7	8.5	495.3	52.2	0.3	1.0	1.8			25.2%	**2.9**
Schur100	10,000 (Single prec.)	10^{-8}	4933	763.0	4520.4	98.3	2957.1	2928.8	19.8	16.3	3.6	0.7	4.2	722.6%	0.03
		10^{-6}	840	136.2	601.0	6.3	388.3	61.0	0.9	2.2	1.7			76.6%	0.5
		10^{-4}	501	90.3	278.3	3.2	235.2	23.0	0.5	1.3	1.4			40.5%	0.8
		10^{-2}	282	56.6	153.0	2.0	134.0	7.4	0.2	1.3	2.5			25.6%	**0.9**
Covar30	27,000	10^{-8}	2247	1221.6	8976.3	34.3	3157.2	1515.2	4.6	0.4	0.1	13.1	36.8	61.1%	0.9
		10^{-6}	1609	948.2	6363.7	18.6	2301.7	815.8	2.7	3.2	0.8			47.7%	**1.7**
		10^{-4}	215	380.8	1093.2	3.4	1054.7	212.6	0.6	N/A	N/A			14.3%	N/A
		10^{-2}	3	348.0	49.6	1.9	1042.8	205.4	0.5	N/A	N/A			6.7%	N/A

Table 3: Solving linear systems from different applications using 64 MPI tasks.

All the problems exhibit the same – and expected – behavior. When the compression threshold ε is higher (e.g., 10^{-2}), the compression and the factorization are faster and the HSS and ULV factors are smaller than when ε is closer to machine precision. The gains in compression and factorization come at the price of accuracy; for some problems, the solution is inaccurate when ε is too large. This is the case for matrices QChemToeplitz, BEMAcoustics and Covar30. For some other problems, accuracy is satisfying with the largest value of ε, but the best choice of ε is not 10^{-2}. For example, for problem Hmatrix, the best choice is $\varepsilon = 10^{-4}$.

We now compare the behavior of our dense solver with ScaLAPACK. The last column in Table 3 is the speed-up of STRUMPACK with respect to ScaLAPACK. For synthetic problems (SimpleToeplitz, HMatrix) and problems with a very simple structure (QChemToeplitz), using HSS techniques yields very large gains. For example, for the HMatrix problem and $\varepsilon = 10^{-4}$, our solution process is 72.9 times faster than a traditional dense *LU* factorization. For problems BEMMultiSphere and Covar30, the gains are less impressive but still significant; STRUMPACK exhibits a 2.9x speed-up for BEMMultiSphere and

a 1.7x speed-up in run time for COVAR30. For the last two problems, SCHUR100 and BEMACOUSTICS, STRUMPACK is slower than ScaLAPACK regardless of the parameters. These two matrices exhibit some low-rank property and the number of floating-point operations performed by STRUMPACK is lower than that of ScaLAPACK, but, however, the total run time is larger with STRUMPACK. This highlights a drawback of our approach. Traditional dense *LU* factorization is an algorithm with a very regular computational pattern, than can be written with BLAS3 kernels, and good implementations (e.g., vendor-tuned) usually exhibit very good flop-rate and can reach a very large fraction of the peak performance. On the other hand, algorithms that takes advantage of low-rank structures (e.g., HSS, but also $\mathcal{H}$-matrices or Block Low-Rank representations) have to deal with more irregular and imbalanced task flows, and manipulate a collection of small matrices instead of one large matrix. Therefore, these algorithms cannot be expected to reach the same flop rate as traditional algorithms. This is visible in Table 3.

We want to highlight that for a given class of applications, using low-rank approximation techniques usually pay off past a certain size. This is because, although HSS techniques allow to solve linear systems with a lower asymptotic complexity, but with a larger constant prefactor. Also, as we mentioned previously, the flop rate with HSS is often lower than with traditional algorithms. These effects are visible in Section 4.6 where we experiment with matrices of growing size from a particular application; in this framework, gains increase with problem size.

The last point that we elaborate on is memory. As stated previously, the *memory overhead* is the extra memory usage of STRUMPACK relative to that of ScaLAPACK; calling mem_{sca} the memory usage of ScaLAPACK and mem_{str} the memory usage of STRUMPACK, this is simply $\frac{mem_{str}-mem_{sca}}{mem_{str}}$. It is important to understand that this memory overhead also represents the amount of memory that would be used if we were to use a matrix-free implementation. We recall that our algorithm is amenable to a matrix-free framework since it only requires access to a matrix-vector routine and *selected* elements, more specifically $\mathcal{O}(r^2 n)$, elements of the matrix (with r the maximum rank and assuming the tree has $\log n$ levels). For example, for matrix BEMMULTISPHERE, the memory overhead is 25.2%, which means:

1. The memory consumption of STRUMPACK is 1.252 times that of ScaLAPACK.
2. If we were to use a matrix-free version the memory consumption would be 25.2% that of ScaLAPACK, i.e., a 4-fold reduction.

4.4 Fast matrix-vector product

In this section, we briefly illustrate the use of HSS techniques for fast matrix-vector products. Here, the matrix is not factored with ULV but is simply kept in HSS form to perform matrix-vector multiplication. We use the power method (that relies mainly on matrix-vector products) to compute the largest eigenvalue of the QCHEMTOEPLITZ matrix.

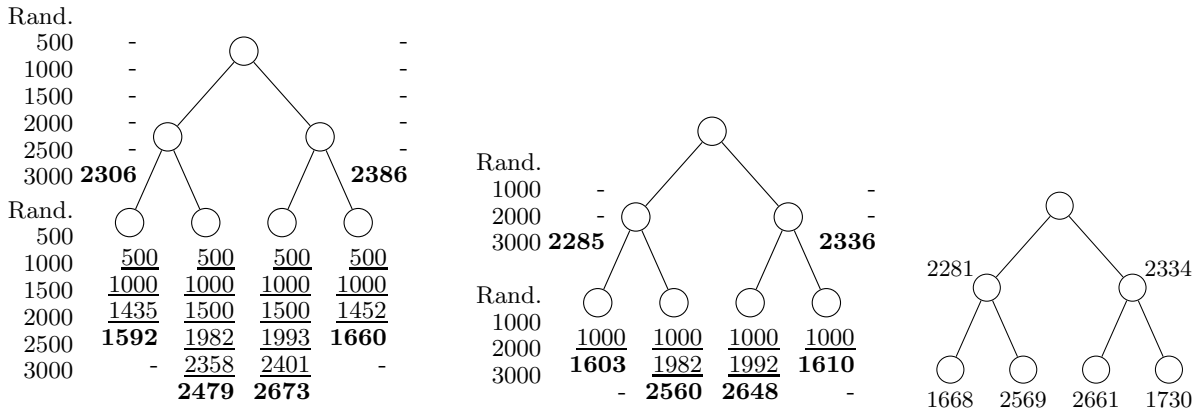
HSS							Traditional GEMV			Speed-up
Compression				Iterations			Iterations			with HSS
Max rank	Factors (MB)	Flops ($\times 10^9$)	Time	#It	Flops ($\times 10^9$)	Time	#It	Flops	Time	
147	42.1	5126.6	17.6	318	1053.0	21.8	318	4070.4	69.7	1.8

Table 4: Power method for QChemToeplitz using 64 MPI tasks.

4.5 Adaptive-rank mechanism

In this section, we illustrate the behavior of the adaptive sampling mechanism described in Section 3.3. The matrix we use corresponds to an electromagnetic sphere discretized with the boundary element method and has size 58,800.

In Figure 8, we examine three configurations. In Figure 8(c), we use 3,000 random vectors, which is enough to guarantee that we reveal the "true" rank of each node (in this discussion we only consider the



(a) In steps of 500. (b) In steps of 1,000. (c) No adaptive sampling.

Figure 8: Ranks of the U generators using adaptive sampling in steps of 500 (a), in steps of 1,000 (b), and without adaptive sampling (c). An underlined rank means that the corresponding node triggers a restart. A rank marked in bold is final.

rank of the U generator, for simplicity). In Figure 8(a), we start the compression with 500 random vectors. Every time a block is compressed, we look at the difference between its rank and the number of random vectors; if it was less than 200, we discarded the generators at the node we consider, we add 500 new random vectors, and the compression restarts. In this example, the four leaves of the tree are compressed in parallel; they all have rank 500, and the compression restarts with $500 + 500 = 1,000$ random vectors. Again, this is not enough (rank 1000 is found at the leaves), and we add 500 new random vectors. 1,000 random vectors is not enough and the compression restarts with $1,000 + 500 = 1,500$ random vectors; this is is enough, and the compression restarts with $1500 + 500 = 2000$ random vectors. This time, the leaves exhibit four different ranks: 1,592, 1,982, 1,993, and 1,660. At the leaves with 1,592 and 1,660, the generators are kept because the difference between their rank and the number of random vectors is more than the limit that we picked. However, the compression needs to restart because of the two other leaves. We add 500 random vectors and the traversal restarts. At the leaves that have rank 1,592 and 1,660, we simply update the samples S^r_τ and S^c_τ; their generators have been obtained at the previous iteration and are not recomputed. At the two other leaves, we recompute the generators. This fails again and the compression restarts with 3,000 random vectors. This time the ranks are small enough and the generators are kept. The compression proceeds to the next level then terminates.

In Figure 8(b), we start with 1,000 random vectors and we add 1,000 random vectors whenever a step of compression fails; this time, the compression restarts only twice and successfully terminates with 3,000 random vectors. One can observe that the ranks that we obtain using different numbers of random vectors vary slightly. This is an effect of the sampling mechanism, but it does not have any major effect on accuracy, or the size of the HSS and ULV representations. However, the adaptive sampling mechanism influences performance. In Table 5, we report on the run time for the HSS compression when the tree has 3 levels (as in Figure 8) and when the tree has 8 levels. One can observe that when the HSS tree has 3 levels, using the adaptive sampling mechanism induces a penalty in run time. This is due to the fact that processes need to synchronize to restart the computations, and the HSS tree has to be traversed multiple times instead of once. Also, instead of being computed in one shot, the samples $S^r = AR^r$ and $S^c = A^*R^c$ are computed in multiples stages, which mitigates the benefits of BLAS3 kernels. However, when the tree has more levels, we can see that the adaptive strategy can be faster than using directly the correct number of random vectors (3,000 here). This is due to the fact that, at the bottom of the tree, nodes have ranks much lower than 3,000. Their generators can be computed with less random vectors (e.g., 500 or 1,000). Using less random vectors makes the Interpolative Decomposition faster, and it can potentially make the whole compression stage faster.

Levels in HSS tree	Strategy		
	Steps of 500	Steps of 1,000	No adaptive sampling
3	259.9	223.9	**186.2**
8	147.0	**125.1**	137.3

Table 5: Time in seconds for the HSS compression as a function of the number of levels in the HSS tree and the sampling strategy (as in Figure 8). The matrix arises from the discretization of an electromagnetic sphere using BEM and has size 58,800.

In a practical setting, it is impossible to predict what the fastest strategy is. However, in many applications, practitioners have a rough idea of the compressibility of their matrices and can predict the order of magnitude of the maximum rank. In that case, we advise to set the sampling parameters so that, at worst, the compression routine needs to restart a limited number of times. For example, if the rank is expected to be between 1,000 and 10,000, we would start with 1,000 random vectors and increase the number by 1,000 every time a step of compression fails, guaranteeing no more than 10 steps.

4.6 Scalability

In this section, we evaluate the scalability of our structured code using three experiments.

In the first experiment, we process dense matrices with increasing size from the same application, using an increasing number of MPI processes. The experimental setting is the same as in [32]; we use the same system (Hopper at NERSC) and the same settings. The matrices we consider correspond to the root node of the multifrontal factorization of the discretized Helmholtz equations, with a fixed number of points per wavelength. They correspond to five different cubic meshes, ranging from $100 \times 100 \times 100$ to $500 \times 500 \times 500$. The topmost separators, i.e., the dense matrices we consider here, have between $10,000$ and $250,000$ rows and columns. In Table 6, we compare the performance of ScaLAPACK, Hsolver (more precisely, the dense kernel used within Hsolver), and STRUMPACK when solving a linear system with these matrices. Under this setup, the maximum HSS rank grows linearly with the mesh size k. Note that this is not strictly a weak scaling experiment since the number of processes does not increase as fast as the number of operations. The next experiment in this section is a strict weak scaling experiment.

One can observe that STRUMPACK and Hsolver find similar maximum ranks for all the problems. The size of HSS factors is smaller with STRUMPACK, which is due to the special structure of U and V generators, as described in Section 2.2. In terms of performance, STRUMPACK is 2 to 6 times faster than Hsolver, and 1.8 to 5.4 faster than ScaLAPACK. It is interesting to notice that STRUMPACK spends less time in communication. The percentage of wall time spent in communications, as reported by the IPM tool [15], is similar to that of Hsolver, but the overall wall time is shorter, implying less time is spent doing communication (assuming computations and communications do not overlap, which is fair since our algorithm is mostly synchronous).

The second experiment in this section is a strict weak-scaling experiment. We consider the root node of the multifrontal factorization of the discretized Poisson equation on a 2D mesh. The mesh is a $k \times k$ regular grid, therefore the dense matrix that we consider (last frontal matrix) is $k \times k$. The multifrontal factorization yields frontal matrices that can be compressed using HSS techniques with a very low maximum rank [33], that is almost constant with respect to the size of the grid (in practice, it increases very slowly – logarithmically). In the experiment, we use a fixed number of random vectors (slightly larger than the rank of the largest problem). Therefore, the complexity of the HSS compression grows as $O(k^2)$. In the experiment, the number of processes also grows as k^2, yielding a constant number of operations per process for the different grid sizes, as shown in Table 7. One can observe that the run time increases as k increases. This is due to the overhead of communications. In particular, for the last problem, the redistribution of the input matrix represents over 80% of the compression time. This is what is expected for problems with very small maximum rank, as shown in Section 3.4, Table 1. If we ignore the redistribution time, then

k (mesh: $k \times k \times k$)		100	200	300	400	500
Matrix size ($=k^2$)		10,000	40,000	90,000	160,000	250,000
MPI tasks		64	256	1,024	4,096	8,192
ScaLAPACK	Flops ($\times 10^{12}$)	2.7	170.7	1944.0	10922.7	41666.7
	Time (s)	4.2	57.7	176.1	313.6	541.6
	Communication time	30.5%	20.5%	24.9%	40.4%	36.1%
Hsolver	Maximum rank	335	618	894	1226	1497
	HSS factors (GB)	0.1	0.8	2.0	4.6	6.8
	Compression flops ($\times 10^{12}$)	0.8	19.7	115.2	424.0	1051.0
	Compression time (s)	8.3	51.5	193.4	207.8	259.5
	Factorization flops ($\times 10^{12}$)	0.1	0.7	2.3	7.2	10.6
	Factorization time (s)	0.4	1.4	1.8	2.5	4.2
	Solution flops ($\times 10^{9}$)	0.1	0.3	0.8	2.1	2.9
	Solution time (s)	0.1	0.2	0.6	2.3	9.5
	Communication time	12.4%	19.4%	27.7%	26.4%	31.3%
	Speed-up over ScaLAPACK	0.5	1.1	0.9	1.5	2.0
STRUMPACK	Maximum rank	313	638	903	1289	1625
	HSS factors (GB)	0.1	0.5	1.1	3.0	3.3
	Compression flops ($\times 10^{12}$)	0.6	18.8	132.7	626.1	1716.7
	Compression time (s)	2.0	13.0	30.6	60.8	133.6
	Factorization flops ($\times 10^{12}$)	0.04	0.5	1.7	5.9	7.7
	Factorization time (s)	0.3	1.0	1.5	2.7	5.0
	Solution flops ($\times 10^{9}$)	0.2	1.1	2.9	6.9	9.5
	Solution time (s)	0.04	0.3	0.4	0.6	0.8
	Communication time	24.2%	24.0%	27.0%	28.5%	28.9%
	Speed-up over ScaLAPACK	**1.8**	**4.0**	**5.4**	**4.8**	**3.9**
	Speed-up over Hsolver	**3.8**	**3.7**	**6.0**	**3.3**	**2.0**

Table 6: Comparison between ScaLAPACK, Hsolver, and STRUMPACK for dense matrices arising from the multifrontal factorization of the discretized Helmholtz equations.

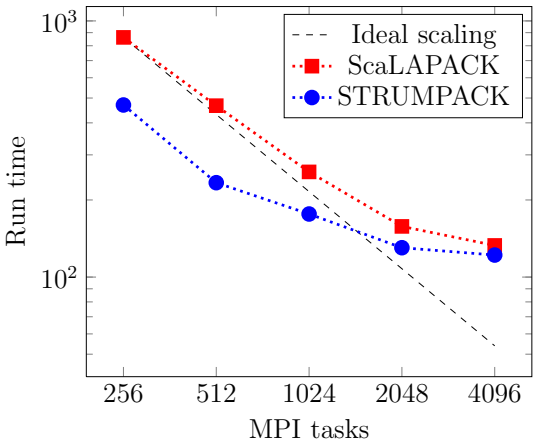
compression time is reasonably constant, as shown in the last row of the table.

The last experiment in this section is a strong scaling benchmark. We use one test problem, a matrix arising from the discretization of an electromagnetic sphere using BEM, with size 130,000. We compare the run time for solving a linear system with ScaLAPACK and STRUMPACK, using a number of MPI processes ranging from 256 to 4,096. For this problem, the maximum rank is 5,500.

We observe that although the scalability of STRUMPACK is quite good, the gap between ScaLAPACK and STRUMPACK reduces when the number of processes increases. As explained in Section 3.4, this is because when the HSS rank is large (which is the case in this problem), communication volume for the traversal of the tree becomes larger than that with ScaLAPACK. The breakdown of the run time for the parallel HSS compression with 4,096 MPI tasks is the following: 15% of the time is spent in the initial matrix distribution, and 25% of the time is spent in the two products $S^r = AR^r$ and $S^c = A^*R^c$. The rest (60%) is spent traversing the HSS tree to compute the local samples and generators. The major part is spent

k (matrix size: $k \times k$)	2,000	4,000	8,000	16,000	32,000
MPI tasks	1	4	16	64	256
Maximum rank	21	26	31	38	44
Compression flops per process ($\times 10^9$)	1.08	1.06	1.06	1.06	1.05
Compression time (s)	0.10	0.21	0.37	0.54	2.51
Compression time w/o redistribution (s)	0.10	0.11	0.13	0.20	0.42

Table 7: Weak-scaling experiment for dense matrices arising from the multifrontal factorization of the discretized 2D Poisson equation.



(a) Run time.

MPI tasks		256	512	1,024	2,048	4,096
LU	Time (s)	862.4	466.8	257.4	157.7	132.9
	% comm.	30.5%	37.9%	45.0%	55.9%	74.3%
HSS	Time (s)	469.7	233.6	176.3	130.2	121.9
	% comm.	34.7%	31.2%	42.1%	47.3%	68.0%

(b) Statistics.

Figure 9: Strong scaling experiment: run time for solving a linear system for a matrix with size 130,000, arising from the discretization of an electromagnetic sphere using BEM, with maximum rank 5,500.

computing Interpolative Decompositions, which represent 50% of total compression time. We observed that in most cases the flop-rate of the Interpolative Decomposition (modified version of `PxGEQPF`) is much lower than that of `PxGEMM` or `PxGETRF`. This is due to the fact that it relies on a BLAS2 algorithm. A BLAS3 implementation appears in the literature but the code is not publicly available [5]. Implementing a BLAS3 version is left for future work.

We are investigating different techniques to accelerate the distribution of the input matrix A. Furthermore, some recent works investigate communication optimal matrix-matrix multiplication algorithms [29] and improvements for rectangular matrix multiplications [13]. Our implementation would directly benefit from any improvement resulting from this research.

5 Conclusion

We presented the dense matrix computation package STRUMPACK that uses Hierarchically Semi-Separable representations to compress an input matrix and performs operations with this compressed form, such as solving linear systems or performing matrix-vector products. For matrices from certain classes of applications, such as finite element or boundary element methods, or applications that involve Toeplitz matrices, using HSS techniques allows to perform these operations asymptotically faster than when traditional algorithms (e.g., LU factorization) are used. The compression algorithm, which is the cornerstone of the framework, is parametrized by a compression threshold that allows the package to be used as a direct solver with full

accuracy or as a robust preconditioner. Our compression algorithm employs randomized sampling and is the first distributed-memory implementation that we know of. Furthermore, we introduced an adaptive sampling mechanism that allows the code to be used in a black-box fashion.

The STRUMPACK package is very general; it can be used with any number of MPI processes and can accommodate different hierarchical partitionings of the input matrix. Furthermore, it is an open source package made available to the community. The code is released under the BSD-LBNL license and the version presented here is currently available at `http://portal.nersc.gov/project/sparse/strumpack/STRUMPACK-Dense-0.9.0.tar`

Work is in progress to use this dense package within a sparse solver. We have also developed a shared-memory sparse solver [16] and our goal is to combine these two codes in order to obtain a hybrid (MPI+OpenMP) sparse solver. Another aspect that we wish to explore is using HSS techniques in matrix-free frameworks. As mentioned here, our algorithm is amenable to a matrix-free implementation where the user only provides a matrix-vector product and a routine to access selected elements of the matrix on the fly. This feature will be included in a future version of STRUMPACK.

Acknowledgments Partial support for this work was provided through Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (and Basic Energy Sciences/Biological and Environmental Research/High Energy Physics/Fusion Energy Sciences/Nuclear Physics). This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

We wish to thank people who provided us with test problems and helped us: Ana Manic, Guillaume Sylvand, Umberto Villa.

References

[1] Patrick R Amestoy, Cleve Ashcraft, Olivier Boiteau, Alfredo Buttari, Jean-Yves L'Excellent, and Clément Weisbecker. Improving multifrontal methods by means of block low-rank representations. To appear in SISC, 2014.

[2] Patrick R Amestoy, Abdou Guermouche, Jean-Yves L'Excellent, and Stéphane Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel computing*, 32(2):136–156, 2006.

[3] Amirhossein Aminfar, Sivaram Ambikasaran, and Eric Darve. A fast block low-rank dense solver with applications to finite-element matrices. arXiv preprint arXiv:1403.5337, 2014.

[4] Mario Bebendorf. *Hierarchical matrices*. Springer, Berlin, 2008.

[5] Peter Benner, Enrique S Quintana-Ortí, and Gregorio Quintana-Ortí. Parallel model reduction of large linear descriptor systems via balanced truncation. In *High Performance Computing for Computational Science-VECPAR 2004*, pages 340–353. Springer, Valencia, 2005.

[6] L Susan Blackford, Jaeyoung Choi, Andy Cleary, Eduardo D'Azevedo, James W Demmel, Inderjit Dhillon, Jack J Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, Ken Stanley, David Walker, and R Clinton Whaley. *ScaLAPACK users' guide*, volume 4. SIAM, Philadelphia, 1997.

[7] Steffen Börm and Lars Grasedyck. H-Lib – a library for h-and h2-matrices, 1999.

[8] Tony F Chan. Rank revealing QR factorizations. *Linear Algebra and Its Applications*, 88:67–82, 1987.

[9] Shivkumar Chandrasekaran, Patrick Dewilde, Ming Gu, and N Somasunderam. On the numerical rank of the off-diagonal blocks of schur complements of discretized elliptic pdes. *SIAM Journal on Matrix Analysis and Applications*, 31(5):2261–2290, 2010.

[10] Shivkumar Chandrasekaran, Ming Gu, and Timothy Pals. A fast ULV decomposition solver for hierarchically semiseparable representations. *SIAM Journal on Matrix Analysis and Applications*, 28(3):603–622, 2006.

[11] Hongwei Cheng, Zydrunas Gimbutas, Per-Gunnar Martinsson, and Vladimir Rokhlin. On the compression of low rank matrices. *SIAM Journal on Scientific Computing*, 26(4):1389–1404, 2005.

[12] Jaeyoung Choi, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, David Walker, and R Clinton Whaley. A proposal for a set of parallel basic linear algebra subprograms. In *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*, pages 107–114. Springer, Berlin, 1996.

[13] James W. Demmel, David Eliahu, Armando Fox, Shoaib Kamil, Benjamin Lipshitz, Oded Schwartz, and Omer Spillinger. Communication-optimal parallel recursive rectangular matrix multiplication. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 261–272, Boston, 2013. IEEE.

[14] Iain S Duff and John K Reid. The multifrontal solution of indefinite sparse symmetric linear. *ACM Transactions on Mathematical Software (TOMS)*, 9(3):302–325, 1983.

[15] Karl Fuerlinger, Nicholas J. Wright, and David Skinner. Effective performance measurement at petascale using ipm. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 373–380, Shanghai, 2010. IEEE.

[16] Pieter Ghysels, Xiaoye S. Li, François-Henry Rouet, Samuel Williams, and Artem Napov. An efficient multi-core implementation of a novel hss-structured multifrontal solver using randomized sampling. Submitted to SIAM Journal on Scientific Computing, 2014.

[17] John Gunnels, Calvin Lin, Greg Morrow, and Robert Van De Geijn. A flexible class of parallel matrix multiplication algorithms. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 110–116, Orlando, 1998. IEEE.

[18] Wolfgang Hackbusch and Boris N Khoromskij. A sparse -matrix arithmetic. *Computing*, 64(1):21–47, 2000.

[19] Jeremiah Jones and Dan Haxton. Development of a cartesian sinc dvr basis for single and double ionization, 2014. Bulletin of the American Physical Society.

[20] Tzanio Kolev and Veslin Dobrev. MFEM: Finite element discretization library, 2010.

[21] Scott Ladenheim, Panayot S. Vassilevski, and Umberto Villa. A multilevel, hierarchical sampling technique for spatially correlated random fields. In preparation, 2014.

[22] Michael W Mahoney and Petros Drineas. Cur matrix decompositions for improved data analysis. *Proceedings of the National Academy of Sciences*, 106(3):697–702, 2009.

[23] Per-Gunnar Martinsson. A fast randomized algorithm for computing a hierarchically semiseparable representation of a matrix. *SIAM Journal on Matrix Analysis and Applications*, 32(4):1251–1274, 2011.

[24] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[25] Artem Napov and Xiaoye S Li. An algebraic multifrontal preconditioner that exploits the low-rank property. Numerical Linear Algebra with Applications. (accepted), 2015.

[26] Alex Pothen and Chunguang Sun. A mapping algorithm for parallel sparse Cholesky factorization. *SIAM Journal on Scientific Computing*, 14:1253–1253, 1993.

[27] Loïc Prylli and Bernard Tourancheau. Fast runtime block cyclic data redistribution on multiprocessors. *Journal of Parallel and Distributed Computing*, 45(1):63–72, 1997.

[28] Gregorio Quintana-Ortí, Xiaobai Sun, and Christian H Bischof. A BLAS-3 version of the QR factorization with column pivoting. *SIAM Journal on Scientific Computing*, 19(5):1486–1494, 1998.

[29] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms. In *Euro-Par 2011 Parallel Processing*, pages 90–109. Springer, Bordeaux, 2011.

[30] Robert A Van De Geijn and Jerrell Watts. Summa: Scalable universal matrix multiplication algorithm. *Concurrency-Practice and Experience*, 9(4):255–274, 1997.

[31] Shen Wang, Xiaoye S Li, François-Henry Rouet, Jianlin Xia, and Maarten V De Hoop. A parallel geometric multifrontal solver using hierarchically semiseparable structure. Submitted to ACM Transactions on Mathematical Software, 2014.

[32] Shen Wang, Xiaoye S Li, Jianlin Xia, Yingchong Situ, and Maarten V De Hoop. Efficient scalable algorithms for solving dense linear systems with hierarchically semiseparable structures. *SIAM Journal on Scientific Computing*, 35(6):C519–C544, 2013.

[33] Jianlin Xia. Efficient structured multifrontal factorization for general large sparse matrices. *SIAM Journal on Scientific Computing*, 35(2):A832–A860, 2013.

[34] Jianlin Xia. Randomized sparse direct solvers. *SIAM Journal on Matrix Analysis and Applications*, 34(1):197–227, 2013.

[35] Jianlin Xia, Shivkumar Chandrasekaran, Ming Gu, and Xiaoye S Li. Superfast multifrontal method for large structured linear systems of equations. *SIAM Journal on Matrix Analysis and Applications*, 31(3):1382–1411, 2009.

[36] Jianlin Xia, Shivkumar Chandrasekaran, Ming Gu, and Xiaoye S Li. Fast algorithms for hierarchically semiseparable matrices. *Numerical Linear Algebra with Applications*, 17(6):953–976, 2010.

[37] Jianlin Xia, Yuanzhe Xi, and Ming Gu. A superfast structured solver for Toeplitz linear systems via randomized sampling. *SIAM Journal on Matrix Analysis and Applications*, 33(3):837–858, 2012.

A Two-stage triangular solution process

This appendix illustrates the ULV solve algorithm 4 starting from expression (11), giving the explicit ULV factorization of A for a 3 level HSS matrix. After ULV factorization, the solution of $Ax = b$ can be obtained as $x = V^{-1}L^{-1}U^{-1}b$. In (11), the transformations applied to A from the left form U^{-1}, the transformations applied to A from the right form V^{-1} and the big matrix in the right-hand side of Equation (11) forms L. Define

$$\tilde{V}_\tau = Q_\tau \hat{V}_\tau \quad \text{with} \quad \hat{V}_\tau = \begin{cases} V_\tau, & \text{if } \tau \text{ is a leaf} \\ \begin{bmatrix} \tilde{V}_{\nu_1;b} & \\ & \tilde{V}_{\nu_2;b} \end{bmatrix} V_\tau, & \text{if } \tau \text{ is a non-leaf} \end{cases} \tag{12}$$

Let $b_\tau = b(I_\tau)$ for leaves τ and $\tilde{b}_\tau = \Omega_\tau b_\tau$. Now, we first compute $U^{-1}b$ by applying the Ω_τ transformations and the permutations $\Gamma_{\nu_1;b\leftrightarrow\nu_2;t}$ to the right-hand side b

$$U^{-1}b = \Gamma_{1;b\leftrightarrow 2;t} \begin{bmatrix} I & & & \\ & \Omega_1 & & \\ & & I & \\ & & & \Omega_2 \end{bmatrix} \begin{bmatrix} \Gamma_{3;b\leftrightarrow 4;t} & \\ & \Gamma_{5;b\leftrightarrow 6;t} \end{bmatrix} \begin{bmatrix} \Omega_3 b_3 \\ \Omega_4 b_4 \\ \Omega_5 b_5 \\ \Omega_6 b_6 \end{bmatrix} \tag{13}$$

$$= \Gamma_{1;b\leftrightarrow 2;t} \begin{bmatrix} I & & & \\ & \Omega_1 & & \\ & & I & \\ & & & \Omega_2 \end{bmatrix} \begin{bmatrix} \tilde{b}_{3;t} \\ \tilde{b}_{4;t} \\ \tilde{b}_{3;b} \\ \tilde{b}_{4;b} \\ \tilde{b}_{5;t} \\ \tilde{b}_{6;t} \\ \tilde{b}_{5;b} \\ \tilde{b}_{6;b} \end{bmatrix} = \Gamma_{1;b\leftrightarrow 2;t} \begin{bmatrix} \tilde{b}_{3;t} \\ \tilde{b}_{4;t} \\ \Omega_1 \begin{bmatrix} \tilde{b}_{3;b} \\ \tilde{b}_{4;b} \end{bmatrix} \\ \tilde{b}_{5;t} \\ \tilde{b}_{6;t} \\ \Omega_2 \begin{bmatrix} \tilde{b}_{5;b} \\ \tilde{b}_{6;b} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \tilde{b}_{3;t} \\ \tilde{b}_{4;t} \\ \Omega_{1;t} \begin{bmatrix} \tilde{b}_{3;b} \\ \tilde{b}_{4;b} \end{bmatrix} \\ \tilde{b}_{5;t} \\ \tilde{b}_{6;t} \\ \Omega_{2;t} \begin{bmatrix} \tilde{b}_{5;b} \\ \tilde{b}_{6;b} \end{bmatrix} \\ \Omega_{1;b} \begin{bmatrix} \tilde{b}_{3;b} \\ \tilde{b}_{4;b} \end{bmatrix} \\ \Omega_{2;b} \begin{bmatrix} \tilde{b}_{5;b} \\ \tilde{b}_{6;b} \end{bmatrix} \end{bmatrix} \tag{14}$$

We write down explicitly the forward triangular substitution $y = L^{-1}U^{-1}b$

$$y = \begin{bmatrix} y_3 = L_3^{-1}\tilde{b}_{3;t} \\ y_4 = L_4^{-1}\tilde{b}_{4;t} \\ y_1 = L_1^{-1}\Omega_{1;t}\left(\begin{bmatrix} \tilde{b}_{3;b} \\ \tilde{b}_{4;b} \end{bmatrix} - L_{4,3}y_3 - L_{3,4}y_4\right) \\ y_5 = L_5^{-1}\tilde{b}_{5;t} \\ y_6 = L_6^{-1}\tilde{b}_{6;t} \\ y_2 = L_2^{-1}\Omega_{2;t}\left(\begin{bmatrix} \tilde{b}_{5;b} \\ \tilde{b}_{6;b} \end{bmatrix} - L_{6,5}y_5 - L_{5,6}y_6\right) \\ y_0 = D_0^{-1}\begin{bmatrix} \Omega_{1;b}\left(\begin{bmatrix} \tilde{b}_{3;b} \\ \tilde{b}_{4;b} \end{bmatrix} - L_{4,3}y_3 - L_{3,4}y_4\right) - W_{1;b}Q_{1;t}^* y_1 - B_{1,2}V_2^* \begin{bmatrix} \tilde{V}_{5;t}^* & & \tilde{V}_{5;b}^* & \\ & \tilde{V}_{6;t}^* & & \tilde{V}_{6;b}^* \end{bmatrix} \begin{bmatrix} y_5 \\ y_6 \\ Q_2^* y_2 \end{bmatrix} \\ \Omega_{2;b}\left(\begin{bmatrix} \tilde{b}_{5;b} \\ \tilde{b}_{6;b} \end{bmatrix} - L_{6,5}y_5 - L_{5,6}y_6\right) - W_{2;b}Q_{2;t}^* y_2 - B_{2,1}V_1^* \begin{bmatrix} \tilde{V}_{3;t}^* & & \tilde{V}_{3;b}^* & \\ & \tilde{V}_{4;t}^* & & \tilde{V}_{4;b}^* \end{bmatrix} \begin{bmatrix} y_3 \\ y_4 \\ Q_1^* y_1 \end{bmatrix} \end{bmatrix} \end{bmatrix} \tag{15}$$

Clearly, this substitution should be performed bottom-up, i.e., first compute the leaves y_3, y_4 y_5 and y_6, then y_1 and y_2 and finally y_0. Now we introduce the intermediate variable z_τ, defined as

$$z_\tau = \begin{cases} \tilde{V}^*_{\tau;t} y_\tau, & \text{if } \tau \text{ is a leaf} \\ V^*_\tau \begin{bmatrix} z_{\nu_1} \\ z_{\nu_2} \end{bmatrix} + \tilde{V}^*_{\tau;t} y_\tau, & \text{if } \tau \text{ is a non-leaf} \end{cases} \tag{16}$$

Then for a non-leaf node τ (f.i., nodes 1 and 2), with two children ν_1 and ν_2 *which are both leaves*, we have $y_\tau = L_\tau^{-1} \Omega_{\tau;t} b_\tau$ with

$$b_\tau = \left(\begin{bmatrix} \tilde{b}_{\nu_1;b} \\ \tilde{b}_{\nu_2;b} \end{bmatrix} - L_{\nu_2,\nu_1} y_{\nu_1} - L_{\nu_1,\nu_2} y_{\nu_2} \right) = \begin{bmatrix} \tilde{b}_{\nu_1;b} - W_{\nu_1;b} Q^*_{\nu_1;t} y_{\nu_1} - B_{\nu_1,\nu_2} V^*_{\nu_2} Q^*_{\nu_2;t} y_{\nu_2} \\ \tilde{b}_{\nu_2;b} - B_{\nu_2,\nu_1} V^*_{\nu_1} Q^*_{\nu_1;t} y_{\nu_1} - W_{\nu_2;b} Q^*_{\nu_2;t} y_{\nu_2} \end{bmatrix} \tag{17}$$

$$= \begin{bmatrix} \tilde{b}_{\nu_1;b} - W_{\nu_1;b} Q^*_{\nu_1;t} y_{\nu_1} - B_{\nu_1,\nu_2} z_{\nu_2} \\ \tilde{b}_{\nu_2;b} - B_{\nu_2,\nu_1} z_{\nu_1} - W_{\nu_2;b} Q^*_{\nu_2;t} y_{\nu_2} \end{bmatrix} \tag{18}$$

Due to the definition of z_τ as given in (16), the definition of b_τ for non-leaf nodes, Equation (18), is also valid for nodes higher up in the hierarchy, for which the situation is slightly more complicated. Consider node 0, for which $y_0 = D_0^{-1} b_0$, with

$$b_0 = \begin{bmatrix} \tilde{b}_{1;b} - W_{1;b} Q^*_{1;t} y_1 - B_{1,2} \left(V^*_2 \begin{bmatrix} \tilde{V}^*_{5;t} y_5 & \\ & \tilde{V}^*_{6;t} y_6 \end{bmatrix} + V^*_2 \begin{bmatrix} \tilde{V}^*_{5;b} & \\ & \tilde{V}^*_{6;b} \end{bmatrix} Q^*_2 y_2 \right) \\ \tilde{b}_{2;b} - W_{2;b} Q^*_{2;t} y_2 - B_{2,1} \left(V^*_1 \begin{bmatrix} \tilde{V}^*_{3;t} y_3 & \\ & \tilde{V}^*_{4;t} y_4 \end{bmatrix} + V^*_1 \begin{bmatrix} \tilde{V}^*_{3;b} & \\ & \tilde{V}^*_{4;b} \end{bmatrix} Q^*_1 y_1 \right) \end{bmatrix} \tag{19}$$

$$= \begin{bmatrix} \tilde{b}_{1;b} - W_{1;b} Q^*_{1;t} y_1 - B_{1,2} \left(V^*_2 \begin{bmatrix} z_5 \\ z_6 \end{bmatrix} + \tilde{V}^*_{2;t} y_2 \right) \\ \tilde{b}_{2;b} - W_{2;b} Q^*_{2;t} y_2 - B_{2,1} \left(V^*_1 \begin{bmatrix} z_3 \\ z_4 \end{bmatrix} + \tilde{V}^*_{1;t} y_1 \right) \end{bmatrix} = \begin{bmatrix} \tilde{b}_{1;b} - W_{1;b} Q^*_{1;t} y_1 - B_{1,2} z_2 \\ \tilde{b}_{2;b} - W_{2;b} Q^*_{2;t} y_2 - B_{2,1} z_1 \end{bmatrix} \tag{20}$$

Hence, the z_τ variables accumulate the contributions to the right-hand side from the already eliminated HSS nodes. We can compute y_τ as $y_\tau = L_\tau^{-1} \tilde{b}_\tau$, except at the root where $y_0 = D_0^{-1} b_0$, which is computed using standard LU decomposition of D_0. Finally, the orthogonal transformation V^{-1} involving the Q_τ matrices should be applied to y to obtain the solution vector x.