Sparse Approximate Multifrontal Factorization with Composite Compression Methods

LISA CLAUS, National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory, USA
PIETER GHYSELS and YANG LIU, Applied Mathematics and Computational Research Division, Lawrence Berkeley National Laboratory, USA
THÁI ANH NHAN, Department of Mathematics and Computer Science, Santa Clara University, USA
RAMAKRISHNAN THIRUMALAISAMY and AMNEET PAL SINGH BHALLA, Department of Mechanical Engineering, San Diego State University, USA
SHERRY LI, Applied Mathematics and Computational Research Division, Lawrence Berkeley National Laboratory, USA

This article presents a fast and approximate multifrontal solver for large sparse linear systems. In a recent work by Liu et al., we showed the efficiency of a multifrontal solver leveraging the butterfly algorithm and its hierarchical matrix extension, HODBF (hierarchical off-diagonal butterfly) compression to compress large frontal matrices. The resulting multifrontal solver can attain quasi-linear computation and memory complexity when applied to sparse linear systems arising from spatial discretization of high-frequency wave equations. To further reduce the overall number of operations and especially the factorization memory usage to scale to larger problem sizes, in this article we develop a composite multifrontal solver that employs the HODBF format for large-sized fronts, a reduced-memory version of the nonhierarchical block low-rank format for medium-sized fronts, and a lossy compression format for small-sized fronts. This allows us to solve sparse linear systems of dimension up to 2.7× larger than before and leads to a memory consumption that is reduced by 70% while ensuring the same execution time. The code is made publicly available in GitHub.

CCS Concepts: • **Mathematics of computing** → **Solvers**; *Numerical analysis*; *Partial differential equations*; • **Theory of computation** → Design and analysis of algorithms; • **Computing methodologies** → Parallel algorithms;

Additional Key Words and Phrases: Sparse direct solver, multifrontal method, butterfly algorithm, block low-rank compression;

This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, and in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program through the FASTMath Institute under contract DE-AC02-05CH11231 at Lawrence Berkeley National Laboratory. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under contract DE-AC02-05CH11231.
Authors' addresses: L. Claus, National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory, One Cyclotron road Berkeley, CA 94720; email: lclaus@lbl.gov; P. Ghysels, Y. Liu, and S. Li, Applied Mathematics and Computational Research Division, Lawrence Berkeley National Laboratory, One Cyclotron road Berkeley, CA 94720; emails: {pghysels, liuyang, xsli}@lbl.gov; T. A. Nhan, Department of Mathematics and Computer Science, Santa Clara University, 500 El Camino Real Santa Clara, CA 95053; email: anhan@scu.edu; R. Thirumalaisamy and A. P. S. Bhalla, Department of Mechanical Engineering, San Diego State University, 5500 Campanile Drive San Diego, CA 92182; emails: {rthirumalaisam1857, asbhalla}@sdsu.edu.

0098-3500/2023/09-ART24 $15.00
https://doi.org/10.1145/3611662

ACM Reference format:
Lisa Claus, Pieter Ghysels, Yang Liu, Thái Anh Nhan, Ramakrishnan Thirumalaisamy, Amneet Pal Singh Bhalla, and Sherry Li. 2023. Sparse Approximate Multifrontal Factorization with Composite Compression Methods. *ACM Trans. Math. Softw.* 49, 3, Article 24 (September 2023), 28 pages.
https://doi.org/10.1145/3611662

1 INTRODUCTION

Efficiently computing the solution of large sparse linear systems arising from finite element, finite difference, or finite volume discretizations of **Partial Differential Equations (PDEs)** is an important requirement for many scientific and engineering applications. The multifrontal method is a fast solution method that can be implemented efficiently on modern hardware, as it arranges the computations in such a way that most of the computational work is done on smaller dense submatrices, so-called frontal matrices. Unfortunately, the overall amount of dense linear algebra operations needed to complete the multifrontal method sums up to $O(N^2)$ for typical 3D PDEs, where N is the matrix dimension corresponding to the sparse linear system.

For many applications arising from wide classes of PDEs, this complexity can be reduced up to $O(N \log^{\alpha} N)$ for some α by leveraging algebraic compression formats to exploit rank structures in off-diagonal blocks of the matrix. Examples of low-rank-based compression methods include $\mathcal{H}$ matrices [12, 15], HODLER (hierarchically off-diagonal low-rank) formats [1], hierarchically semiseparable (HSS) formats [44], and **Block Low-Rank (BLR)** formats [2, 4, 43]. Available software packages that couple these rank-structured formats with multifrontal methods include STRUMPACK [10] and MUMPS [4]. PaStiX [17, 35] is an additional software package that couples low-rank compression methods with supernodal methods instead of multifrontal methods.

In addition to the algorithms mentioned earlier, we consider another low-rank-based compression tool called *butterfly* [21, 22, 29, 32, 38], a multilevel matrix decomposition algorithm well suited for representing highly oscillatory operators such as Fourier transforms and integral operators and special function transforms. When combined with hierarchical matrix techniques, butterfly can also serve as the building block for accelerating iterative methods, direct solvers, and preconditioners for boundary element methods for high-frequency wave equations. These techniques essentially replace low-rank products in the $\mathcal{H}$ and HODLR formats with butterflies and leverage fast and randomized butterfly algebra to compute the matrix inverse (for direct solvers and preconditioners). An open source software package that provides an implementation of the butterfly algorithm is available at https://github.com/liuyangzhuan/ButterflyPACK.

In a related work [27], we presented a fast multifrontal sparse solver for high-frequency wave equations. The solver leverages the butterfly algorithm and its hierarchical matrix extension, HODBF (hierarchically off-diagonal butterfly) compression, to compress large frontal matrices. The resulting solver can attain quasi-linear computation and memory complexity when applied to high-frequency Helmholtz and Maxwell equations. Similar complexities have been analyzed and observed for Poisson equations as well. Nevertheless, to further reduce the overall number of operations and to enable solving larger problem sizes, in this article we present a composite multifrontal solver that employs the HODBF format for compressing the large frontal matrices in the multifrontal method and leverages additional compression methods for the remaining fronts. To be more specific, HODBF serves as a good compression method as shown in the related work [27]. However, due to its more complex data structures, it only pays off to use HODBF for larger fronts to yield a high compression ratio. For medium-sized fronts, we employ BLR compression. We combine **Left-Looking (LL)** and **Right-Looking (RL)**

versions of BLR to a hybrid method that decreases the memory consumption significantly. For the remaining small-sized fronts, we make use of lossy compression enabled through the zfp software [24] to further decrease the memory consumption while maintaining accuracy of the preconditioner.

Our contributions in this work are the development of a composite multifrontal solver that employs three compression methods: HODBF, BLR and floating point compression, and an implementation of BLR with a reduced memory footprint that makes use of a column-wise construction of matrix tiles. The sparse approximate multifrontal solver is used as a preconditioner for restarted GMRES(30) with modified Gram-Schmidt and a zero-vector initial guess. These updates lead to a significant decrease of memory consumption, and it allows us to solve sparse linear systems of dimension up to 2.7× larger compared to a multifrontal solver with HODBF compression only [27]. The code is made publicly available in the sparse solver package STRUMPACK [9].

The rest of the article is organized as follows. The multifrontal factorization method is presented in Section 2. The zfp, hierarchically off-diagonal butterfly compression, and BLR algorithms are described in Section 3, including the classical LL and RL BLR versions as well as our proposed hybrid and memory optimized BLR algorithm. The proposed composite rank-structured multifrontal method is detailed in Section 4. Numerical results demonstrating the efficiency and applicability of the proposed solver for the 3D Helmholtz, reaction-diffusion, and Navier-Stokes equations are presented in Section 5, followed by our conclusion in Section 6.

2 MULTIFRONTAL FACTORIZATION

This section briefly recalls the main ingredients of the multifrontal method for general invertible sparse matrices. For a more detailed discussion of multifrontal methods, see other works [20, 25]. The method separates the factorization of a sparse matrix ($A = LU$) into a series of partial factorizations of many smaller dense matrices, which correspond to the separators from a nested dissection ordering. After each factorization step, a Schur complement is formed and carried along temporarily, and its scattering to the global Schur complement is delayed until that part of panel factorization is about to start.

As a preprocessing step, the system matrix A is first scaled and permuted for numerical stability: $A \leftarrow D_r A D_c Q_c$, where D_r and D_c are diagonal matrices that scale the rows and columns of A and Q_c is a column permutation that places large entries on the diagonal. We use the MC64 code by Duff and Koster [19] or the parallel method—without the diagonal scaling—described by Azad et al. [6] to perform the scaling and column permutation. After that, a fill-reducing permutation $A \leftarrow PAP^T$ is applied—that is, the number of nonzero entries in the sparse factors L and U is minimized. The permutation matrix P is typically computed using nested dissection applied to the adjacency graph of $A + A^T$, as implemented in Scotch [39] or METIS [11].

The multifrontal method relies on a structure called the *assembly tree*. Each node τ of the assembly tree is represented by a dense frontal matrix F_τ, with the following 2×2 block structure: $F_\tau = \begin{bmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{bmatrix}$. The rows and columns corresponding to the F_{11} block are called the *fully summed variables*. After the fully summed variables have received all of their Schur complement updates, the front is constructed. We denote the dimension of F_{11} by $\#I_\tau^s$ and the dimension of F_{22} by $\#I_\tau^u$. The I_τ^u index sets define the temporary Schur complement update blocks. Let $n_\tau = \#I_\tau^s + \#I_\tau^u$ denote the dimension of F_τ. Note that the frontal matrices tend to get bigger toward the root of the assembly tree. Furthermore, if v is a child of τ in the assembly tree, then $I_v^u \subset \{I_\tau^s \cup I_\tau^u\}$. For the root node t, $I_t^u = \emptyset$. When considering a single front, we will omit the τ subscript.

The multifrontal method consists of a bottom-up traversal of the assembly tree following a topological ordering. Processing a node consists of four steps making up the numerical factorization of that node:

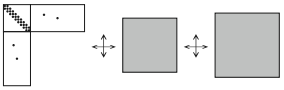
ALGORITHM 1: Sparse multifrontal factorization and solve

Input: $A \in \mathbb{R}^{N\times N}$, $b \in \mathbb{R}^N$
Output: $x \approx A^{-1}b$

1: $A \leftarrow D_r A D_c Q_c$ ▷ (optional) col perm & scaling
2: $A \leftarrow PAP^\top$ ▷ symm fill-reducing reordering
3: Build assembly tree: define I_τ^{s} and I_τ^{u} for every frontal matrix F_τ
4: **for** nodes τ in assembly tree in topological order **do**
5: ▷ sparse with the children updates extended and added
6: $F_\tau \leftarrow \begin{bmatrix} A(I_\tau^{\mathrm{s}}, I_\tau^{\mathrm{s}}) & A(I_\tau^{\mathrm{s}}, I_\tau^{\mathrm{u}}) \\ A(I_\tau^{\mathrm{u}}, I_\tau^{\mathrm{s}}) & 0 \end{bmatrix} \overset{\updownarrow}{+} F_{22;v_1} \overset{\updownarrow}{+} F_{22;v_2}$
7: $P_\tau L_\tau U_\tau \leftarrow F_{11}$ ▷ LU with partial pivoting
8: $F_{12} \leftarrow L_\tau^{-1} P_\tau^\top F_{12}$
9: $F_{21} \leftarrow F_{21} U_\tau^{-1}$
10: $F_{22} \leftarrow F_{22} - F_{21}F_{12}$ ▷ Schur update
11: **end for**
12: $x \leftarrow D_c Q_c P^\top$ bwd-solve (fwd-solve $(PD_r b)$)

(1) Assembling the frontal matrix F_τ (i.e., combining elements from the sparse matrix A with the children's (v_1 and v_2) contribution blocks). This involves a scatter operation and is called *extend-add*, denoted by $\overset{\updownarrow}{+}$.

$$F_\tau = \begin{bmatrix} A(I_\tau^s, I_\tau^s) & A(I_\tau^s, I_\tau^u) \\ A(I_\tau^u, I_\tau^s) & \end{bmatrix} \overset{\updownarrow}{+} F_{22;v_1} \overset{\updownarrow}{+} F_{22;v_2} =$$



(2) Elimination of the fully summed variables in the F_{11} block (i.e., dense LU factorization with partial pivoting of F_{11}).
(3) Updating the off-diagonal blocks F_{12} and F_{21}.
(4) Computing the contribution block from the Schur complement update of $F_{22} \leftarrow F_{22} - F_{21}F_{11}^{-1}F_{12}$. F_{22} is temporary storage and can be released as soon as it has been used in the front assembly (step 1) of the parent node.

After the numerical factorization, the lower triangular sparse factor is available in the F_{21} and F_{11} blocks and the upper triangular factor in the F_{11} and F_{12} blocks. These can then be used to efficiently solve linear systems, using forward and backward substitution. A high-level overview is given in Algorithm 1.

Figure 1 illustrates the multifrontal algorithm for a sparse matrix resulting from the discretization of a PDE using a 5-point finite difference stencil on a regular 2D 11×11 mesh. Figure 1(a) shows the mesh and the top three levels of the nested dissection ordering. Nested dissection is a heuristic algorithm for the ordering of a sparse matrix to reduce the fill-in in the sparse factors. It is based on recursively finding vertex separators. The vertical line marked S^0 is the root separator, and this separator corresponds to the root of the assembly tree (see Figure 1(b)). The next level separators, S_0^1 and S_1^1, correspond to the F_{11} blocks of the next lower level in the assembly tree. Typically, the larger frontal matrices are found near the root of the assembly tree since the separators tend to get smaller further in the nested dissection recursion.

2.1 Parallel Traversal of the Assembly Tree

Following the assembly tree, the distributed algorithm creates nested **Message Passing Interface (MPI)** subcommunicators to facilitate the computation at each node and its subtree. At the

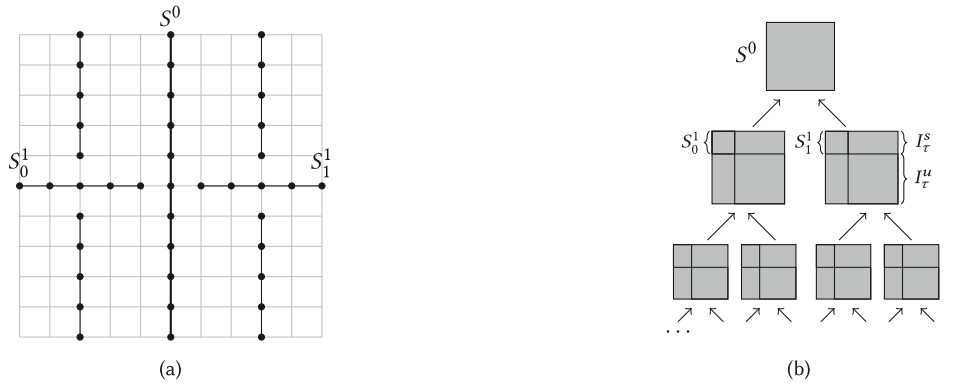


Fig. 1. (1(a)) The top three levels of nested dissection for an 11^2 mesh. The root separator S^0 is a vertical 11-point line. The next level separators are S_0^1 and S_1^1. The root separator corresponds to the top-level front in (b), and similarly for the next level down in the assembly/frontal tree. Note that the fronts in (b) typically get smaller lower in the tree.

root of the tree, we create a 2D process grid using the available processes in the root MPI communicator and distribute the frontal matrix over this grid using the ScaLAPACK 2D block-cyclic data layout. Next, the root MPI communicator is split in two communicators proportionally to the memory required by the subtrees rooted at the children of the root node. Each child constructs a 2D process grid and distributes the child's frontal matrix over this subgrid. This is repeated recursively until the MPI communicator has only one process in it, at which point the local subtree is traversed within a single OpenMP region using OpenMP task parallelism. While moving up the distributed part of the assembly tree, communication between fronts is required for the extend-add operation. This is implemented using an MPI_Alltoallv on the MPI subcommunicator of the parent node. Note that this paragraph discusses the parallel traversal of the assembly tree with a focus on dense frontal matrices. In Section 4, we will discuss details of the procedure for compressed frontal matrices, where a 2D block-cyclic layout is used only for dense, BLR, and lossy compressed fronts, whereas the HODBF compression is based on a 1D block layout (see Section 3.3.1).

We implemented the multifrontal method in the STRUMPACK library [9], using C++, MPI, and OpenMP. STRUMPACK supports real/complex arithmetic, single/double precision, and 32/64-bit integers. STRUMPACK has recently ported the sparse direct multifrontal solver to GPU, targeting both NVIDIA and AMD hardware. In this article, we focus on the CPU implementation only.

In what follows, we leverage multiple compression methods, namely lossy compression enabled through the zfp software [24] (see Section 3.1) BLR compression (Section 3.2), and the hierarchical matrix extension of the butterfly algorithm (HODBF) (see Section 3.3). These methods are used to represent frontal matrices and to construct fast sparse direct solvers, particularly for large matrix systems resulting, for example, from high-frequency wave equations.

3 COMPRESSION METHODS

In this article, we make use of compression methods within the multifrontal solver, which maintains the solver's robustness and reliability and reduces the computational complexity. The three compression methods of interest are the HODBF format, the BLR format, and lossy compression. The HODBF format can be described as a hierarchical compression format with a multilevel matrix decomposition algorithm. HODBF is available as an effort to integrate the dense solver package ButterflyPACK [26] into the sparse solver package STRUMPACK and can be used to compress

frontal matrices within the multifrontal solver. BLR is based on a flat low-rank based compression format, which exploits rank structures in off-diagonal blocks of the frontal matrix. BLR is implemented in STRUMPACK and can also be used to compress fronts. The lossy floating point compression method is available through the zfp package [24] and is integrated into STRUMPACK. Since the multifrontal method relies on dense factorizations, all three approximations can be easily incorporated into the multifrontal factorization by representing the frontal matrices as zfp, BLR, and HODBF matrices, respectively, as will be described in Section 4. All three compression formats are described in detail in the following subsections.

3.1 Lossy Compression with zfp

As an open source library for compressed floating point data, zfp was designed to achieve high compression ratios and therefore uses lossy compression. zfp is often more accurate and faster than other lossy compressors. For more details on the zfp software library, see the work of Lindstrom [24].

In contrast to low-rank formats, lossy compression is a near-lossless compression scheme that maps small blocks of 4^d values with dimension d to a fixed number of bits per block, called *bitplanes* in Section 5. This compressor is based on an orthogonal block transform. For more details on the zfp algorithm, please refer to the work of Lindstrom [23].

In Section 4, we make use of the zfp software library integrated into STRUMPACK to compress small frontal matrices within a multifrontal solver.

3.2 BLR Compression

Among the possible low-rank formats, BLR is the simplest. The format partitions the matrix with a flat, nonhierarchical blocking of the matrix that is defined by conveniently clustering the associated unknowns and approximates its off-diagonal blocks by low-rank submatrices. A BLR representation $\tilde{B}$ of a dense matrix B is shown in (1), with $p \times p$ blocks.

$$\tilde{B} = \begin{bmatrix} \tilde{B}_{11} & \tilde{B}_{12} & \dots & \tilde{B}_{1p} \\ \vdots & \ddots & \ddots & \vdots \\ \tilde{B}_{p1} & \dots & \dots & \tilde{B}_{pp} \end{bmatrix} \tag{1}$$

Assuming $I = \{1, \dots, n\}$ is the set of row and column indices of B, we can define the blocking of the matrix as follows. We call a set of indices $\sigma \subseteq I$ a cluster. Then, a clustering of I is a disjoint union of clusters that equals I. $b = \sigma \times \tau \in I \times I$ is called a *block cluster* based on clusters σ and τ. A block clustering of $I \times I$ is then defined as a disjoint union of block clusters that equals $I \times I$. A block $B_{\sigma\tau}$ corresponds to an interaction between two subdomains σ and τ, where σ contains the row indices of $B_{\sigma\tau}$ while τ contains its column indices. The rank of a given block $B_{\sigma\tau}$ depends on the interaction it represents. Indeed, if $B_{\sigma\tau}$ is a diagonal block (i.e., $\sigma = \tau$), it represents a self-interaction and is thus full-rank. However, if $B_{\sigma\tau}$ is an off-diagonal block, it may be either full rank or low rank depending on the interaction it represents: the weaker the interaction, the lower the rank. The admissibility condition determines whether a block $\sigma \times \tau$ is admissible for low-rank compression. We support both weak admissibility, where every off-diagonal block is compressed, and strong admissibility, where only matrix blocks corresponding to well-separated clusters are compressed. The block clustering, and the admissibility condition, are typically formulated in terms of the geometry of the physical system being modeled. However, in this article, we discuss BLR matrices and the corresponding block clustering in the context of an algebraic multifrontal solver, where no geometry information is available. The block clustering and the admissibility condition in the context of the multifrontal solver are discussed in Section 4. Based on the choice of the admissibility condition,

the off-diagonal blocks $B_{\sigma\tau}(\sigma \neq \tau)$ of size $m_\sigma \times m_\tau$ and numerical rank $r_{\sigma\tau}$ are approximated by a low-rank matrix $\tilde{B}_{\sigma\tau} = X_{\sigma\tau} Y_{\sigma\tau}^T$ at accuracy ε. $X_{\sigma\tau}$ is a $m_\sigma \times r_{\sigma\tau}$ matrix, and $Y_{\sigma\tau}$ is a $m_\tau \times r_{\sigma\tau}$ matrix.

As part of the numerical factorization within the multifrontal solver, as discussed in Section 2, we make use of a LU factorization of each F_{11} part of a frontal matrix. To compute the blocked factorization of the front, four fundamental tasks must be performed:

— *Factor (F)*: LU decomposition $B_{\sigma\sigma} = L_{\sigma\sigma} U_{\sigma\sigma}$ for diagonal blocks $\sigma = 1, \ldots, p$ with partial pivoting
— *Solve (S)*: Solve triangular linear system $B_{\sigma\tau} = L_{\sigma\sigma}^{-1} B_{\sigma\tau}$, $B_{\tau\sigma} = B_{\tau\sigma} U_{\sigma\sigma}^{-1}$, with $\sigma = 1, \ldots, p$, $\tau = \sigma + 1, \ldots, p$
— *Update (U)*: Matrix-matrix multiplication $B_{\kappa\tau} = B_{\kappa\tau} - B_{\kappa\sigma} B_{\sigma\tau}$, with $\sigma = 1, \ldots, p$, $\tau = \sigma + 1, \ldots, p, \kappa = \sigma + 1, \ldots, p$
— *Compression (C)*: Compress off-diagonal blocks $B_{\sigma\tau} \approx \tilde{B}_{\sigma\tau} = X_{\sigma\tau} Y_{\sigma\tau}$.

We refer to the computation of the low-rank approximation $\tilde{B}_{\sigma\tau}$ of each block as the compression step, which can be performed in different ways. We chose a QR factorization with column pivoting (i.e., LAPACK's Anderson et al. [45] geqp3 routine), which is modified to stop the factorization when the diagonal coefficient of R, $r_{i,i}$, falls below a prescribed threshold ε. We use a relative tolerance (i.e., we stop the factorization after $|r_{i,i}|/|r_{0,0}| < \varepsilon$).

Depending on when the compression step is performed within the numerical factorization, several algorithm variants can be defined and implemented based on the execution order of the four tasks defined previously: FSUC, FSCU, FCSU, CFSU. These acronyms indicate the order in which the tasks are performed [31].

ALGORITHM 2: RL BLR algorithm: FCSU

Input: a $p \times p$ block matrix B of size n,
$B = [B_{i,j}]_{i=1:p, j=1:p}$

```
1: Construct tiles B_{i,j}, ∀i, j
2: for i = 1 to p do
3:     Factor: B_{i,i} = L_{i,i} U_{i,i}
4:     for j = i + 1 to p do
5:         Compress: B_{i,j} ≈ X_{i,j} Y_{i,j}^T
6:         Solve: B_{i,j} ← L_{i,i}^{-1} B_{i,j}
7:         Compress: B_{j,i} ≈ X_{j,i} Y_{j,i}^T
8:         Solve: B_{j,i} ← B_{j,i} U_{i,i}^{-1}
9:     end for
10:    for j = i + 1 to p do
11:        for k = i + 1 to p do
12:            Update: B_{k,j} ← B_{k,j} − X_{k,i} (Y_{k,i}^T X_{i,j}) Y_{i,j}^T
13:        end for
14:    end for
15: end for
```

ALGORITHM 3: LL BLR algorithm: UFCS

Input: a $p \times p$ block matrix B of size n,
$B = [B_{i,j}]_{i=1:p, j=1:p}$

```
1: Construct tiles B_{i,j}, ∀i, j
2: for i = 1 to p do
3:     for j = i to p do
4:         for k = 1 to i − 1 do
5:             Update: B_{i,j} ← B_{i,j} − X_{i,k} (Y_{i,k}^T X_{k,j}) Y_{k,j}^T
6:             if j ≠ i then
7:                 B_{j,i} ← B_{j,i} − X_{j,k} (Y_{j,k}^T X_{k,i}) Y_{k,i}^T
8:             end if
9:         end for
10:    end for
11:    Factor: B_{i,i} = L_{i,i} U_{i,i}
12:    for j = i + 1 to p do
13:        Compress: B_{i,j} ≈ X_{i,j} Y_{i,j}^T
14:        Solve: B_{i,j} ← L_{i,i}^{-1} B_{i,j}
15:        Compress: B_{j,i} ≈ X_{j,i} Y_{j,i}^T
16:        Solve: B_{j,i} ← B_{j,i} U_{i,i}^{-1}
17:    end for
18: end for
```

These variants are so-called RL versions, in the sense that at each step k, as soon as the factor and solve tasks for all blocks in row k and column k have been performed, the entire trailing submatrix (column blocks to its "right") is updated (Figure 2(a)). We make use of the FCSU (standing for Factor, Compress, Solve, and Update) variant of the BLR factorization algorithm. All low-rank updates of a given block $\tilde{B}_{\sigma\tau}$ are compressed before the triangular solve of the *LU* factorization of a dense BLR

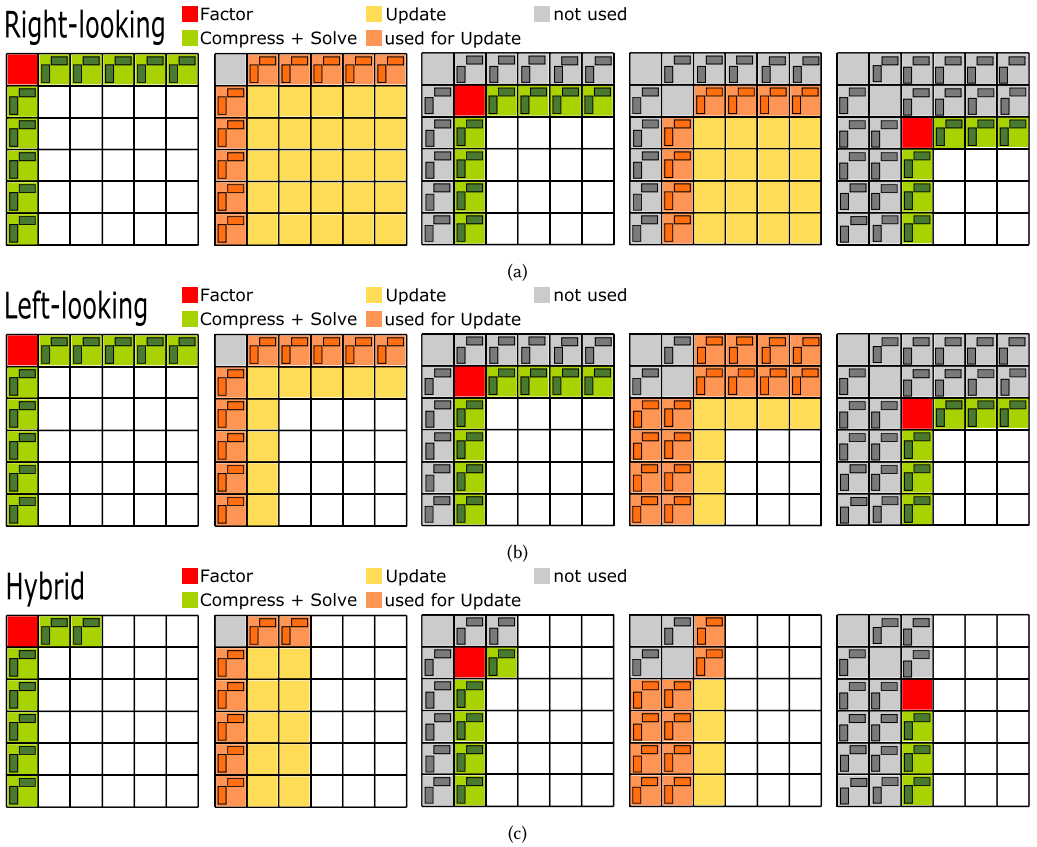


Fig. 2. First steps of a BLR compression with the FCSU/UFCS version. (a) Right-looking. (b) Left-looking. (c) Hybrid with $col_{\max} = 3$.

matrix. Based on the comparative study by Mary [31], the FCSU variant seems the most promising out of the RL versions, as it provides a good balance between efficiency and accuracy. Compressing earlier influences the accuracy negatively, an effect that is quantified in the work of Higham and Mary [18].

The RL algorithms can be rewritten in an LL[1] form, where at each step k, blocks in row k as well as column k are updated using all blocks already computed (those at its "left"): UFS, UFSC, UFCS, UCFS, CUFS.

In this article, we focus on the UFCS version, as this version is most promising in terms of timing due to a different memory access pattern [8] as described in other works [4, 31]. See Algorithm 2 for the RL implementation and Algorithm 3 for the LL implementation, and Figures 2(a) and 2(b) for a comparison between RL and LL, respectively.

The RL and LL variants perform the same number of operations but in a different order, which results in a different memory access pattern. The impact of using RL or LL factorization is mainly observed on the update step. In particular, for the RL variant, at each step k, the full-rank blocks of the trailing submatrix are written, and therefore they are loaded many times (at each step of the

[1]This algorithmic variant is also called *left-up looking* in the literature; however, for brevity, we use *left-looking* (or LL) throughout the article.

ALGORITHM 4: Hybrid BLR algorithm (column-wise constructed)

Input: a $p \times p$ block matrix B of size n, $B = [B_{i,j}]_{i=1:p, j=1:p}$

for $b = 0$ to $\lceil p/col_{\max} \rceil - 1$ **do** ▷ Advance $col_{\max}$ block columns each time
  for $i = col_{\max} b + 1$ to $col_{\max}(b+1)$ **do** ▷ Construct $col_{\max}$ block columns among P_c process columns
    for $j = 1$ to p **do**
      Construct tiles $B_{j,i}$
    end for
  end for
  if $b > 0$ **then** ▷ See Figure 3, subfigures 4–6
    for $j = 1$ to $col_{\max} b$ **do**
      for $i = col_{\max} b + 1$ to $col_{\max}(b+1)$ **do**
        Compress: $B_{j,i} \approx X_{j,i} Y_{j,i}^T$
        Solve: $B_{j,i} \approx B_{j,i} U_{i,i}^{-1}$
        for $k = j + 1$ to p **do**
          Update: $B_{k,i} \approx B_{k,i} - X_{k,j}(Y_{k,j}^T X_{j,i}) Y_{j,i}^T$
        end for
      end for
    end for
  end if
  for $i = col_{\max} b + 1$ to $col_{\max}(b+1)$ **do** ▷ See Figure 3, subfigures 1–3 and 7–9
    Factor: $P_i B_{i,i} = L_{i,i} U_{i,i}$ ▷ LU with partial pivoting for the diagonal block
    for $j = i{+}1$ to $col_{\max}(b+1)$ **do**
      Compress: $B_{i,j} \approx X_{i,j} Y_{i,j}^T$
      Solve: $B_{i,j} \leftarrow L_{i,i}^{-1} B_{i,j}$
    end for
    for $j = i{+}1$ to p **do**
      Compress: $B_{j,i} \approx X_{j,i} Y_{j,i}^T$
      Solve: $B_{j,i} \approx B_{j,i} U_{i,i}^{-1}$
    end for
    for $j = i{+}1$ to $col_{\max}(b+1)$ **do**
      for $k = i{+}1$ to p **do**
        Update: $B_{k,j} \approx B_{k,j} - X_{k,i}(Y_{k,i}^T X_{i,j}) Y_{i,j}^T$
      end for
    end for
  end for
end for
▷ We assume that p is a multiple of $col_{\max}$ for simplicity, but our code can easily handle any value of p.

factorization), whereas the low-rank blocks of the current panel are read once and never loaded again. In the LL variant, at each step k, the full-rank blocks of the current panel are written for the first and last time of the factorization, whereas the low-rank blocks of all previous panels are read, and therefore they are loaded many times during the entire factorization.

As visualized in Figure 2, all BLR tiles of the entire frontal matrix for the LL and RL versions are constructed at once, as dense tiles, which results in a high peak memory consumption. This is followed by the factorization of the i-th diagonal tile, then the tiles of the i-th row and i-th column are compressed and solved. For the RL version, the update task is executed on all remaining tiles of the trailing submatrix. For the LL version, the update task is executed on the immediate neighboring row and column block columns of the trailing submatrix.

Based on the observation that both the LL and RL versions result in a memory bottleneck due to an initial construction of the entire frontal matrix as dense tiles, the problem sizes that can be executed within a multifrontal solver are limited (see Section 5). Therefore, we decided to implement a memory-efficient version where we combine aspects of the LL and RL versions, which results in a column-wise construction of the frontal matrices (see Algorithm 4 and Figure 2(c)). We call this variant the *hybrid BLR* version. For the hybrid version, only the neighboring $\#col_{\max}$ block

columns are created, and subsequently the compression and solve steps are executed only on the neighboring $\#col_{\max}$ row tiles (see Algorithm 4 for more details). Algorithm 4 is based on the assumption that p is an integer multiple of b for simplicity, but our code can easily handle any value of p in general. The experiments in Section 5 are based on $\#col_{\max}$ = number of columns in the 2D MPI process grid such that each MPI process is involved in updating one local column.

The benefit of using the hybrid version over the RL or the LL version is the reduced memory consumption that eventually enables solving of larger problem sizes (see Section 5). In future work, we intend to investigate other memory-efficient strategies, comparing, for example, the "Minimal-Memory" strategy used in the supernodal solver Pastix [40].

3.2.1 Parallel Layout of the Hybrid BLR Matrix. In our implementation, the hybrid BLR matrix is parallelized with a 2D process grid using the available processes in the MPI communicator, and the matrix is distributed over this grid using a 2D block-cyclic data layout, similar to the ScaLAPACK layout but with nonuniform block sizes. The process grid is $P_r \times P_c$ with $P_r = \lfloor \sqrt{P} \rfloor$ and $P_c = P/P_r$, with at most $\lfloor \sqrt{P} \rfloor - 1$ idle processes, where P represents the number of available MPI processes. In a parallel setting, the block columns for the hybrid BLR algorithm are distributed among all available MPI processes in the grid, based on the 2D block-cyclic data layout with $col_{\max} = P_c$. After the block columns are created locally, the factor, compression, and solve steps are executed on the neighboring $\#col_{\max}$ row tiles. In between each of these four tasks, the MPI processes communicate their updates to the processes in the same row communicator and in the same column communicator.

Figure 3 presents the algorithmic steps for a parallel hybrid BLR method with 12 MPI processes arranged as four processes for each column times three processes for each row. In particular, each process constructs its local tiles within the first $col_{\max} = 3$ columns. The process that owns the tile in the first row and column executes the factorization step; this is followed by two broadcast operations to share the updated tile with all processes of row 1 and all processes of column 1. Afterward, the compression and solve steps are executed for all tiles in the first column as well as the two tiles in row 1. The update operation as described in Algorithm 4 is executed for all tiles in columns 2 and 3. These four tasks are repeated over and over such that all tiles that have been constructed already are updated. Then, these steps are executed again for the next $col_{\max} = 3$ columns.

3.2.2 Communication Cost Comparison for BLR Factorization. As described in Section 3.2.1, the parallel layout of the BLR matrix follows a 2D block-cyclic data layout, similar to the ScaLAPACK layout but with varying block sizes. As visualized in Figures 2 and 3, the BLR(RL), BLR(LL), and BLR(Hybrid) algorithms consist of various different execution steps that lead to a different communication pattern in a parallel setting.

The BLR(RL) algorithm consists of a repetition of the following steps:

(1) LU factorization followed by a broadcast of the LU factored tile along the row processes and the column processes. In addition, the pivot elements are shared along the row processes with an additional broadcast.
(2) After the compression and solve steps, the row of updated tiles as well as the column of updated tiles are broadcast along all row and column processes using two broadcast operations times the number of column/row processes each.

Step 1 consists of three broadcast operations, whereas step 2 consists of two broadcast operations times the number of column/row processes each. These steps are repeated for each row of tiles—that is:

$$(\lceil n/b \rceil - 1) \times 3 \text{ bcast\_ops},$$
$$(\lceil n/b \rceil - 1) \times (2 \times (P_r + P_c)) \text{ bcast\_ops},$$

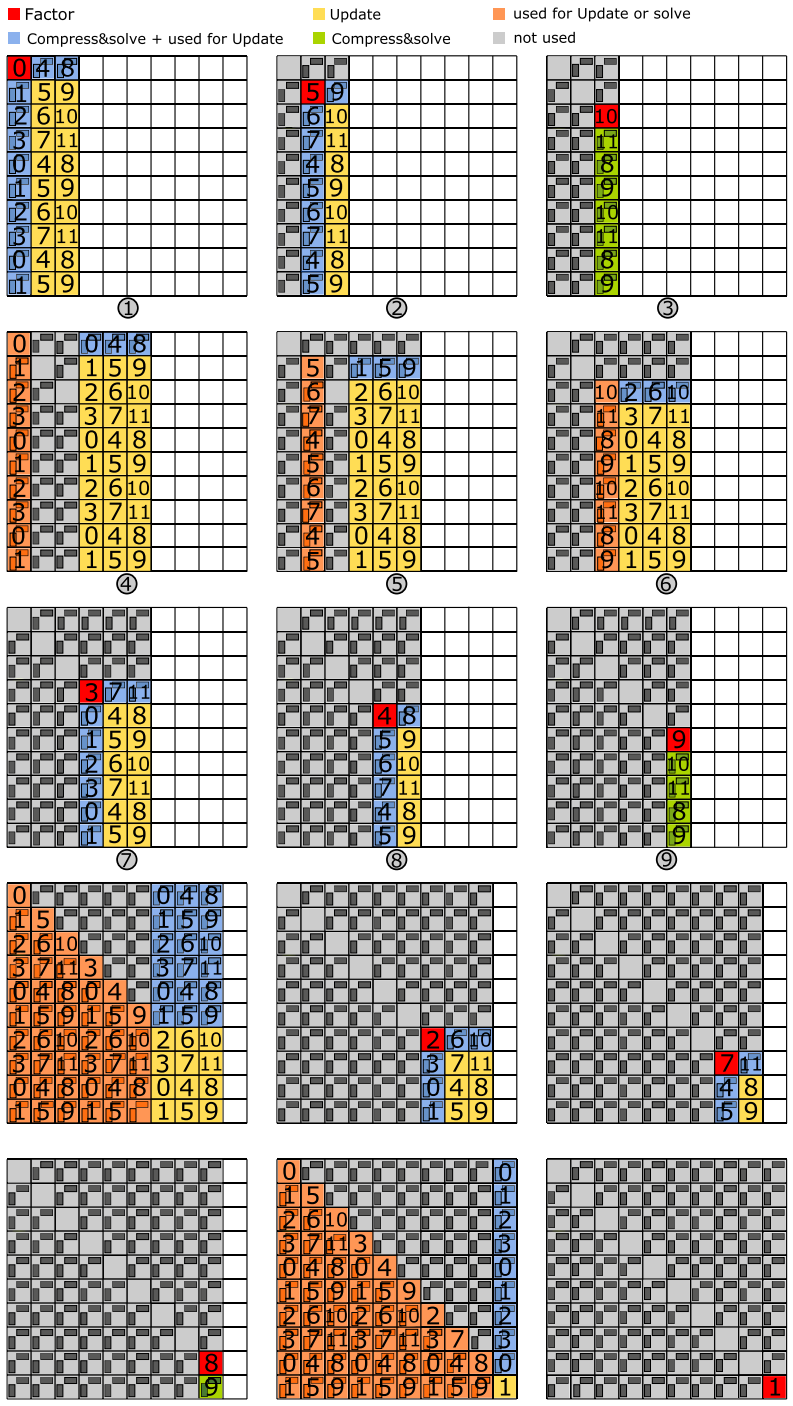


Fig. 3. Hybrid BLR algorithm with 12 processes arranged as a 4 x 3 process grid and $col_{\max} = 3$.

with n = size of F_{11}, b = size of tile, P_r = number of row processes, and P_c = number of column processes.

The BLR(LL) algorithm consists of a repetition of the following steps:

(1) LU factorization followed by a broadcast of the LU factored tile along the row processes and the column processes. In addition, the pivot elements are shared along the row processes with an additional broadcast.
(2) After the compression and solve steps, the row and column of updated tiles as well as all previous updated rows and columns are communicated to the necessary MPI processes using four broadcast and two send operations times the number of column/row processes each. The broadcast operations are used for the communication of one tile along columns and one tile along rows, whereas the send operations are used for the remaining tiles of the rows and columns, which results in two send operations times the number of column/row processes each.

Step 1 consists of three broadcast operations, whereas step 2 consists of four broadcast operations times the number of previous updated columns/rows and two send operations times the number of column/row processes each times the number of previous updated columns/rows, leading to a total of $3 + (i \times 4)$ broadcast and $i \times (P_r \times 2 + P_c \times 2)$ send operations, with i the current phase of the factorization. These steps are repeated for each row of tiles $i = 1 : \lceil n/b \rceil - 1$—that is:

$$(\lceil n/b \rceil - 1) \times 3 \text{ bcast\_ops},$$
$$\sum_{i=1}^{\lceil n/b \rceil - 1} (i \times 4) \text{ bcast\_ops},$$
$$\sum_{i=1}^{\lceil n/b \rceil - 1} (i \times (2 \times (P_r + P_c))) \text{ send\_ops}.$$

The formula indicates that BLR(RL) uses fewer broadcast operations than BLR(RL). However, BLR(RL) consists of broadcast operations only, whereas BLR(LL) needs additional send operations, which leads to higher communication cost overall for BLR(LL) compared to the BLR(RL) variant.

The BLR(Hybrid) algorithm consists of a repetition of the following two stages:

(1) Construct columns, and if columns to the left already exist, execute the following step:
   (a) After compression and solve steps of the new partial row, the updated tiles are communicated to the necessary MPI processes using two broadcast operations times the number of column processes. In addition, the previously updated block columns are communicated to the necessary MPI processes using two broadcast operations times the number of row processes.

This first stage is executed for all previously constructed columns—that is, four broadcast operations (times the number of row/column processes) times the number of previously constructed columns. The second stage consists of the following operations:

(2)(a) LU factorization followed by a broadcast of the LU factored tile along the row processes and the column processes. In addition, the pivot elements are shared along the row processes with an additional broadcast. Even though only a part of a row is constructed in each step, all processes are involved in the update and communication steps.
   (b) After the compression and solve steps of one partial row and one column, the updated tiles are communicated to the necessary MPI processes using two broadcast operations times the number of row/column processes each.

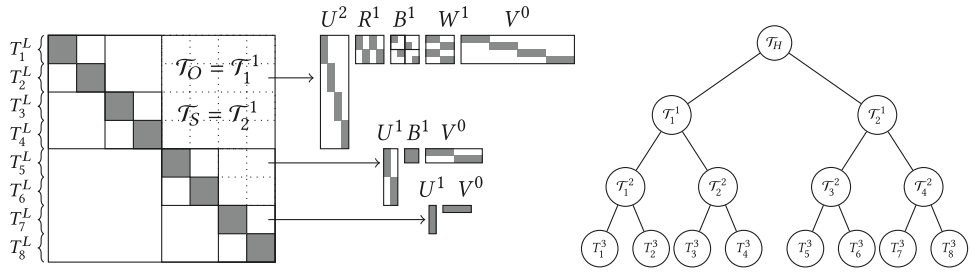


Fig. 4. Illustration of a four-level hierarchically off-diagonal butterfly matrix. The root node is at level $l = 0$, and all leaf nodes are at level $L = 3$. The two largest off-diagonal blocks are approximated using two-level butterfly matrices. The four off-diagonal blocks one level down in the hierarchy are approximated using a one-level butterfly ($U^1B^1V^0$). Finally, the smallest off-diagonal blocks are approximated as zero-level butterfly matrices.

Step (a) of stage 2 consists of three broadcast operations, whereas step (b) consists of two broadcast operations. For both stages, the number of communication operations sums up to

$$\lceil I/col_{\max} \rceil \times col_{\max} \times 3 \text{ bcast\_ops},$$

$$\sum_{i=1}^{I} (col_{\max} \times i \times (2 \times P_r + 2 \times P_c)) \text{ bcast\_ops},$$

with $I = \lceil \lceil n/b \rceil / col_{\max} \rceil$.

In summary, we notice additional communication cost for the two BLR variants (LL) and (Hybrid) compared to the BLR(RL) variant. BLR(RL) and BLR(Hybrid) consist of broadcast operations only, whereas BLR(LL) needs additional send operations. The formula indicates that BLR(RL) uses the least amount of broadcast operations and BLR(Hybrid) uses significantly more than the other two variants, which is due to additional broadcast operations needed for each newly constructed set of columns.

Incorporating BLR fronts in the sparse multifrontal solver adds some additional challenges that we discuss in Section 4.

3.3 HODBF Compression

The HODBF matrix representation [28] is the butterfly extension of the HODLR matrix—that is, $\mathcal{H}$-matrix with weak admissibility condition [16]—which means that every off-diagonal block is compressed. The clustering into blocks as well as the weak admissibility condition in the context of the multifrontal solver are discussed in detail in Section 4. In what follows, we briefly describe the HODBF format, which is used in Section 4 to construct the quasi-linear complexity multifrontal solver.

HODBF starts with a hierarchical clustering of the row and column indices of an $n \times n$ matrix A into $L = O(\log n)$ levels. At the leaf level L, we have a partitioning $T_1^L, T_2^L, \ldots, T_{2^L}^L$, the same as the BLR matrix; at the next level, we have a new partitioning $T_1^{L-1}, \ldots, T_{2^{L-1}}^{L-1}$ by combining every two adjacent clusters/nodes at level L into one. For a given cluster at level $l < L$, T_k^l, we use $\mathcal{T}_k^l$ to denote the subtree rooted at T_k^l and $\mathcal{T}_H = \mathcal{T}_0^0$ (Figure 4).

It follows that off-diagonal blocks of the HODBF matrix, representing interactions between two distinct level-l clusters with subtrees $\mathcal{T}_S$ and $\mathcal{T}_O$, are compressed as butterfly representations, whereas the diagonal blocks at the leaf level $D_\tau = A(T_\tau^L, T_\tau^L)$ for cluster τ are stored as regular

dense matrices (see Figure 4). The butterfly representation of $L_b = L - l$ levels for the block $K = A(T_O, T_S)$ reads as follows:

$$K \approx U^{L_b} R^{L_b-1} \cdots R^h B^h (W^h)^* \cdots (W^1)^* (V^0)^*, \tag{2}$$

where $h = L_b/2$, U^{L_b} and V^0 are block diagonal matrices, and $R^{L_b-1}, \cdots R^h, B^h, W^1, \cdots, W^h$ are all block diagonal matrices after certain predefined permutations. Typically, butterfly representation of an $m \times n$ matrix K contains at most $O(n \log n)$ nonzeros. In practice, one can rapidly constructs such a representation by evaluating $O(n \log n)$ matrix entries from K. Once constructed, the HODBF matrix can be inverted efficiently with randomized algorithms [28]. For more details on butterfly decomposition and its components, please refer to other works [27, 29].

3.3.1 Parallel HODBF Layout. In our implementation, the HODBF matrix is distributed using a given MPI communicator. The overall 1D block layout can be summarized as follows: starting with $D_\tau = A$ at the root τ of $\mathcal{T}_H$, the given communicator sharing D_τ is split into two subcommunicators of similar number of processes. The two subcommunicators store and compute D_{τ_1}, B_{τ_1} and D_{τ_2}, B_{τ_2}, respectively. Layouts of B_{τ_1} and B_{τ_2} follow those described in related work [27]. Each of the two subcommunicators is further recursively split unless the communicator has only one process that stores and computes D_τ. Once constructed following this layout, the HODBF representation can also be inverted following this layout. See more detail on parallelization in the work of Sayed et al. [42].

4 RANK-STRUCTURED MULTIFRONTAL FACTORIZATION

Algorithm 5 outlines the rank-structured multifrontal factorization using BLR compression for medium-sized fronts and HODBF compression for large-sized fronts. Since the more complicated HODBF matrix format has larger overhead for smaller matrices compared to the BLR compression, HODBF compression is only used for fronts corresponding to a separator size larger than a certain threshold n_{Hmin}. In addition, BLR compression is only used for fronts corresponding to a separator size smaller than n_{Hmin} and larger than a threshold n_{Bmin}. We have the option to add zfp compression [24] for small fronts, below threshold n_{Bmin} and larger than or equal to a separator size of 8. All fronts corresponding to a separator size smaller than 8 will not be compressed. The advantage of adding zfp compression is an additional reduction in memory consumption; however, the solve time increases due to the decompression of zfp fronts during the solve step (e.g., see Figure 8(c) and (e), presented later).

Typically, the larger fronts are found closer to the root of the multifrontal assembly tree. This is illustrated in Figure 5 for a small regular $5 \times 5 \times 4$ mesh (Figures 5(a)), and 5(c) shows the corresponding multifrontal assembly tree, where only the top three fronts are compressed using HODBF, the next two levels down in the assembly tree consist of BLR compressed fronts, and the remaining fronts are compressed with zfp.

We now discuss the construction as well as the partial factorization of frontal matrices within a multifrontal solver with a focus on BLR and HODBF compressed fronts. Recall from Section 2 that a front F_τ is built up from elements of the reordered sparse input matrix A and the contribution blocks of the children of the front in the assembly tree: $F_{22;\nu_1}$ and $F_{22;\nu_2}$, where ν_1 and ν_2 are the two children of τ. Since the multifrontal factorization traverses the assembly tree from the leaves to the root, these children contribution blocks might already be compressed using the BLR or HODBF format. Hence, extracting frontal matrix elements requires getting them from fronts compressed previously. There are four different options that we describe in detail in Sections 4.1 through 4.4: extracting from HODBF to construct HODBF, extracting from BLR to construct HODBF, extracting from LL/RL BLR to construct LL/RL BLR, and extracting from hybrid BLR to construct hybrid

ALGORITHM 5: Sparse rank-structured multifrontal factorization using zfp, BLR, and HODBF compressions, followed by a GMRES iterative solve using the multifrontal factorization as an efficient preconditioner

Input: $A \in \mathbb{R}^{N \times N}$, $b \in \mathbb{R}^N$
Output: $x \approx A^{-1}b$

$A \leftarrow P(D_r A D_c Q_c)P^\top$ ▷ scaling, and permutation for stability and fill reduction
$A \leftarrow \hat{P}A\hat{P}^\top$ ▷ rank-reducing separator reordering
Build assembly tree: define I_τ^{s} and I_τ^{u} for every frontal matrix F_τ
for nodes τ in assembly tree in topological order **do**
  if dimension$(I_\tau^{\mathrm{s}}) < 8$ **then**
    construct F_τ as a dense matrix ▷ Algorithm 1
  else if dimension$(I_\tau^{\mathrm{s}}) < n_{\mathrm{Bmin}}$ **then**
    construct F_τ as a dense matrix ▷ Algorithm 1
    if zfp enabled **then**
      compress as zfp matrix
    end if
  else if dimension$(I_\tau^{\mathrm{s}}) < n_{\mathrm{Hmin}}$ && LL/RL BLR **then**
    $F_\tau \leftarrow \begin{bmatrix} A(I_\tau^{\mathrm{s}}, I_\tau^{\mathrm{s}}) & A(I_\tau^{\mathrm{s}}, I_\tau^{\mathrm{u}}) \\ A(I_\tau^{\mathrm{u}}, I_\tau^{\mathrm{s}}) & 0 \end{bmatrix} \dashv\vdash F_{22;\nu_1} \dashv\vdash F_{22;\nu_2}$
    $P_\tau L_\tau U_\tau \leftarrow F_{11}$ ▷ LU with partial pivoting
    $F_{11}, F_{12}, F_{21} \leftarrow$ BLR compress(F_{11}, F_{12}, F_{21})
    $F_{12} \leftarrow L_\tau^{-1} P_\tau^\top F_{12}$
    $F_{21} \leftarrow F_{21} U_\tau^{-1}$
    $F_{22} \leftarrow F_{22} - F_{21}F_{12}$ ▷ Schur update
  else if dimension$(I_\tau^{\mathrm{s}}) < n_{\mathrm{Hmin}}$ && BLR(Hybrid) **then**
    for columns C in frontal matrix F_τ corresponding to I_τ^{s} **do**
      $F_{11;C} \leftarrow$ BLR compress $\left(A(I_\tau^{\mathrm{spartial}}, I_\tau^{\mathrm{spartial}}) \dashv\vdash F_{22;\nu_1;\mathrm{partial}} \dashv\vdash F_{22;\nu_2;\mathrm{partial}}\right)$
      $P_\tau L_\tau U_\tau \leftarrow F_{11;C}$ ▷ LU with partial pivoting
      $F_{11;C}, F_{21;C} \leftarrow$ BLR compress$(F_{11;C}, F_{21;C})$
      $F_{21;C} \leftarrow F_{21;C} U_\tau^{-1}$
    end for
    for columns C in frontal matrix F_τ corresponding to I_τ^{u} **do**
      $F_{12;C} \leftarrow$ BLR compress$(F_{12;C})$
      $F_{12;C} \leftarrow L_\tau^{-1} P_\tau^\top F_{12;C}$
      $F_{22;C} \leftarrow F_{22;C} - F_{21}F_{12}$ ▷ Schur update
    end for
  else
    $F_{11} \leftarrow$ HODBF compress $\left(A(I_\tau^{\mathrm{s}}, I_\tau^{\mathrm{s}}) \dashv\vdash F_{22;\nu_1} \dashv\vdash F_{22;\nu_2}\right)$
    $F_{11}^{-1} \leftarrow$ HODBF invert (F_{11})
    $F_{12} \leftarrow$ butterfly compress $\left(A(I_\tau^{\mathrm{s}}, I_\tau^{\mathrm{u}}) \dashv\vdash F_{22;\nu_1} \dashv\vdash F_{22;\nu_2}\right)$
    $F_{21} \leftarrow$ butterfly compress $\left(A(I_\tau^{\mathrm{u}}, I_\tau^{\mathrm{s}}) \dashv\vdash F_{22;\nu_1} \dashv\vdash F_{22;\nu_2}\right)$
    $S \leftarrow$ butterfly compress $\left(F_{21}F_{11}^{-1}F_{12}\right)$
    $F_{22} \leftarrow$ HODBF compress $\left(F_{22;\nu_1} \dashv\vdash F_{22;\nu_2} - S\right)$ ▷ Schur update
  end if
end for
$x \leftarrow$ GMRES$(A, b, M : u \leftarrow D_c Q_c P^\top \hat{P}^\top$ bwd-solve(fwd-solve$(\hat{P}PD_r v)))$

BLR. In addition to the preceding extracting operations, also called *extend-add operations*, there are extracting operations that we do not discuss here either because they are straightforward operations, like extracting from dense matrix to construct a matrix in BLR form, or because they are similar to the other operations that we explain in detail in the following sections, like extracting from dense or zfp matrix to construct HODBF.

The block clustering for the BLR representation of F_{11} and the HODBF cluster tree for the F_{11} part of a front are defined by performing a recursive bisection (not to be confused with nested

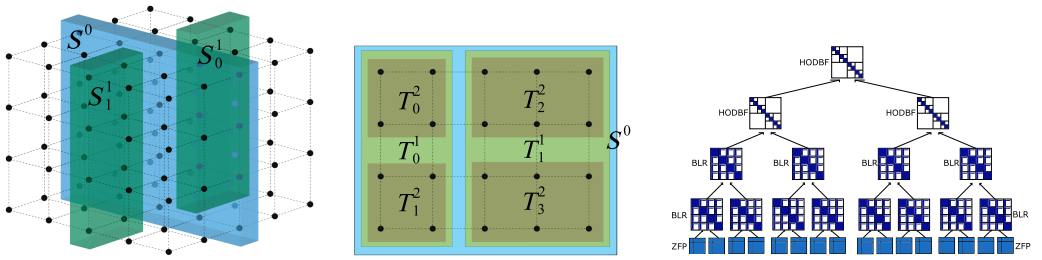


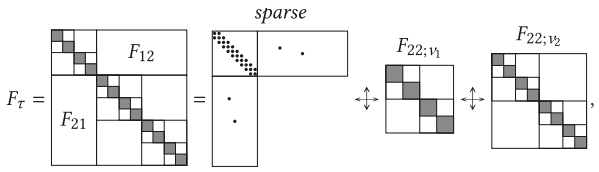
Fig. 5. (a) The top two levels of nested dissection for a $5 \times 5 \times 4$ mesh. The top three separators are S^0, S_0^1, and S_1^1. (b) The root separator S^0 is a vertical plane of 20 points, which is recursively bisected to define level 1 (T_0^1 to T_1^1) and level 2 (T_0^2 to T_3^2) of the hierarchical matrix partitioning. (c) The root separator corresponds to the top-level front, and its HODBF partitioning is defined by the recursive bisection of the root separator, as shown in (b), similar to the next level down; the following two levels in the assembly/frontal tree consist of BLR fronts, and the remaining fronts are compressed with zfp.

dissection), using METIS, of the graph corresponding to $A(I_\tau^s, I_\tau^s)$, where I_τ^s denotes the index sets of F_{11}. Recursive bisection leads to a tree structure that can be used to define the HODBF cluster tree and a corresponding permutation of the rows/columns of F_{11}. For BLR, only the leaves of this tree are considered for the definition of the blocks.

4.0.1 BLR Admissibility Condition. The admissibility condition determines which blocks in the BLR matrix should be considered as compressible. We implement two different admissibility conditions. As the default strategy, each off-diagonal block is compressed, but if the rank of an off-diagonal block is too large, such that the compression does not decrease memory consumption, then that block is stored in its dense format. The diagonal blocks B_{ii} are always stored as full-rank matrices ($\tilde{B}_{ii} = B_{ii}$). We also provide an alternative strategy, where we use the graph of $A(I_\tau^s, I_\tau^s)$ to determine whether a block is admissible. For BLR with this alternative strategy, we say that an interaction $\sigma \times \tau$ is inadmissible if $\sigma \equiv \tau$, or if the matrix block $B_{\sigma\tau}$ contains a nonzero entry coming from sparse matrix A. We found that trying to compress each block performed slightly better in terms of both compression ratio and runtime.

4.1 HODBF with HODBF Children Nodes

Extracting an HODBF frontal matrix to construct an HODBF matrix looks like this:



(3)

with F_{11} and F_{22} compressed as HODBF, and F_{12} and F_{21} compressed as butterfly. For the construction of an HODBF front with HODBF children, the following tasks need to be executed:

(1) Since fronts are constructed as a combination (extend-add) of other smaller fronts, a list of submatrices needs to be extracted from other fronts that are already compressed using HODBF. Therefore, it is critical for performance to use an efficient algorithm to extract a list of submatrices from a butterfly matrix. This is presented as extract_BF in related work [27]. The F_{11} block of $F \equiv F_\tau$ is compressed as an HODBF matrix (see the related

work [27] for details), which uses the extract_BF routine to extract elements from $F_{11} = A(I^{\mathrm{s}}_{\tau}, I^{\mathrm{s}}_{\tau}) \leftarrow\!\updownarrow\!\rightarrow F_{22;\nu_1} \leftarrow\!\updownarrow\!\rightarrow F_{22;\nu_2}$ (see line 32 in Algorithm 5). Note that in this case, the extend-add operation just requires checking whether the required matrix entries appear in the sparse matrix, or in the child contribution blocks, and then adding those different contributions together.

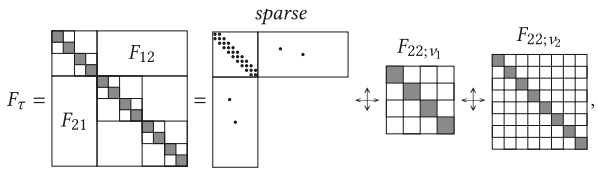
(2) Line 33 approximates F_{11}^{-1} from the butterfly representation of F_{11} (see the related work [27] for details).

(3) In lines 34 and 35, the off-diagonal blocks/fronts F_{12} and F_{21} are each approximated as a single butterfly matrix, using routines to extract elements from $A(I^{\mathrm{s}}_{\tau}, I^{\mathrm{u}}_{\tau}) \leftarrow\!\updownarrow\!\rightarrow F_{22;\nu_1} \leftarrow\!\updownarrow\!\rightarrow F_{22;\nu_2}$ and $A(I^{\mathrm{u}}_{\tau}, I^{\mathrm{s}}_{\tau}) \leftarrow\!\updownarrow\!\rightarrow F_{22;\nu_1} \leftarrow\!\updownarrow\!\rightarrow F_{22;\nu_2}$, respectively. For F_{12}, the tree $\mathcal{T}_H$ corresponding to F_{11} is used as $\mathcal{T}_O$, and the tree corresponding to F_{22} is used for $\mathcal{T}_S$, and vice versa for F_{21}.

(4) The final step for this front is to construct the contribution block of τ, F_{22}, as an HODBF matrix, again using element extraction, now from $F_{22;\nu_1} \leftarrow\!\updownarrow\!\rightarrow F_{22;\nu_2} - S$, where $F_{22;\nu_1}$ and $F_{22;\nu_2}$ are in HODBF form and $S = F_{21}F_{11}^{-1}F_{12}$ is a single butterfly matrix compressed via the randomized algorithm in related work [27]. S can be released as soon as the contribution block has been assembled, and the contribution block is kept in memory until it has been used to assemble the parent front.

4.2 HODBF with BLR Children Nodes

Extracting a BLR(Hybrid) frontal matrix to construct an HODBF matrix is similar to an extraction from HODBF and looks like this:



(4)

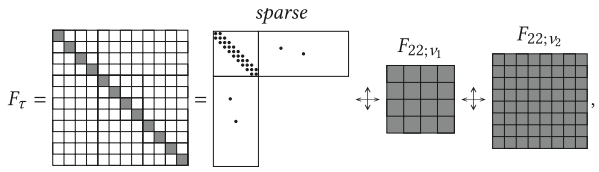
For the construction of an HODBF front with BLR children, the following tasks need to be changed compared to those shown in Section 4.1:

- For line 32 in Algorithm 5, a different routine to extract elements from $F_{11} = A(I^{\mathrm{s}}_{\tau}, I^{\mathrm{s}}_{\tau}) \leftarrow\!\updownarrow\!\rightarrow F_{22;\nu_1} \leftarrow\!\updownarrow\!\rightarrow F_{22;\nu_2}$ needs to be executed. For LL/RL BLR, the extract is straightforward since $F_{22;\nu_1}$ and $F_{22;\nu_2}$ remain dense at the time of extraction. Hence, we simply extract the elements from a dense submatrix using a scatter operation. In case of the BLR(Hybrid), children $F_{22;\nu_1}$ and $F_{22;\nu_2}$ are compressed. Hence, more computational effort is needed to extract from compressed BLR tiles.
- In lines 34 and 35, the F_{12} and F_{21} front off-diagonal blocks are each approximated as a single butterfly matrix, using routines to extract elements from $A(I^{\mathrm{s}}_{\tau}, I^{\mathrm{u}}_{\tau}) \leftarrow\!\updownarrow\!\rightarrow F_{22;\nu_1} \leftarrow\!\updownarrow\!\rightarrow F_{22;\nu_2}$ and $A(I^{\mathrm{u}}_{\tau}, I^{\mathrm{s}}_{\tau}) \leftarrow\!\updownarrow\!\rightarrow F_{22;\nu_1} \leftarrow\!\updownarrow\!\rightarrow F_{22;\nu_2}$, respectively, where $F_{22;\nu_1}$ and $F_{22;\nu_2}$ are in compressed form as in BLR(Hybrid) or dense as in LL/RL BLR.
- The final step to construct the contribution block of τ, F_{22}, as an HODBF matrix, uses element extraction, now from $F_{22;\nu_1} \leftarrow\!\updownarrow\!\rightarrow F_{22;\nu_2} - S$, where $F_{22;\nu_1}$ and $F_{22;\nu_2}$ are in BLR form and S is a single butterfly matrix.

Note that for BLR fronts, the extend-add operation just requires checking whether the required matrix entries appear in the sparse matrix, or in the child contribution blocks, and then adding those different contributions together.

4.3 BLR with BLR Children Nodes with Immediate Construction of All Tiles (LL/RL BLR)

Extracting a BLR frontal matrix to construct a BLR matrix looks like this:



(5)

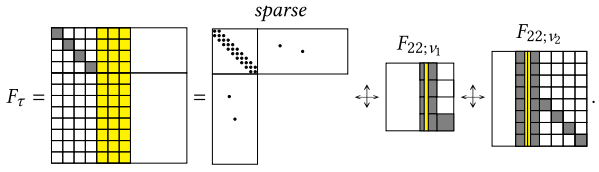
and the following tasks need to be executed for the RL and LL BLR versions:

(1) The $F \equiv F_\tau$ frontal matrix is constructed as a BLR matrix, which implies that it is separated into smaller dense tiles. First, we apply extend-add operations to update $F_\tau \leftarrow \begin{bmatrix} A(I^s_\tau, I^s_\tau) & A(I^s_\tau, I^u_\tau) \\ A(I^u_\tau, I^s_\tau) & 0 \end{bmatrix} \dotplus F_{22;\nu_1} \dotplus F_{22;\nu_2}$ (see line 13 in Algorithm 5). Since $F_{22;\nu_1}$ and $F_{22;\nu_2}$ are not compressed, the extract is straightforward, and we simply extract the elements from a dense submatrix using a scatter operation.
(2) Line 14 computes the LU decomposition of F_{11} tile by tile.
(3) Line 15 compresses the off-diagonal tiles of F_{11} and the tiles of F_{12}, F_{21} via QR factorization with column pivoting.
(4) In lines 16 and 17, the off-diagonal blocks F_{12} and F_{21} are updated.
(5) The final step for this front is to construct the contribution block F_{22} as a BLR matrix with dense tiles (see line 18). The contribution block is kept in memory until it has been used to assemble the parent front.

4.3.1 Construction of a Parallel BLR Parent Front with Immediate Construction of All Tiles. In our implementation, the BLR matrix is distributed among MPI communicators (i.e., a certain amount of BLR tiles is assigned to each MPI communicator). The parallel construction is executed for each MPI communicator. First, each MPI communicator collects their contribution from children fronts $F_{22;\nu_1}$ and $F_{22;\nu_2}$. This is followed by an all-to-all exchange of BLR tiles such that each MPI process has access to the necessary update tiles. Afterward, each MPI communicator updates their tiles within the parent front. A more detailed description and a visualization can be found for the similar hybrid BLR algorithm with fragmentary construction of the BLR matrix (see Section 4.4)).

4.4 BLR with BLR Children Nodes with Fragmentary Construction of the BLR Parent Matrix (BLR(Hybrid))

Extracting a BLR frontal matrix to construct a parent BLR matrix in fragments (a few columns at a time) is similar to a BLR construction where we construct all tiles immediately (see Section 4.3). However, it requires some algorithmic updates as represented in the following:



(6)

The tasks to be executed need to be updated as described next:

(1) The extend-add operations to update $F_\tau \leftarrow \begin{bmatrix} A(I^s_\tau, I^s_\tau) & A(I^s_\tau, I^u_\tau) \\ A(I^u_\tau, I^s_\tau) & 0 \end{bmatrix} \dotplus F_{22;\nu_1} \dotplus F_{22;\nu_2}$ require a different routine to only update the columns that have been constructed. Additional

extend-add steps need to be executed every time block columns are added to the front. Equation (6) visualizes the extend-add operation involving three newly constructed columns of the parent front F_τ and columns of the child fronts $F_{22;\nu_1}$ and $F_{22;\nu_2}$, highlighted in yellow. $F_{22;\nu_1}$ and $F_{22;\nu_2}$ are in BLR matrix form, fully constructed and compressed. Note that only the columns of the contribution blocks corresponding to the data points represented in the columns of the parent front need to be extracted for the extend-add operation. The extract operation for the required block columns can be executed as follows:

- If the tiles of $F_{22;\nu}$ needed for the extend-add operation are compressed, we first decompress the block columns and then apply a scatter operation to construct F_τ.
- If the tiles of $F_{22;\nu}$ needed for the extend-add operation have been decompressed already, we simply apply a scatter operation to construct F_τ.

These steps need to be repeated for each column that is constructed.

(2) The remaining steps are adjusted such that only the constructed block columns are considered. A loop is added for partial construction and update of block columns (see lines 20 and 26 in Algorithm 5).

4.4.1 F_{22} Compression. For the final step, the so-called update step of the contribution block F_{22} of a frontal matrix (see line 29 in Algorithm 5), we include an additional compression step for the hybrid BLR algorithm. In contrast to the BLR matrices with immediate construction of all tiles (compare with Section 4.3), we do a simple compression step after the General Matrix Multiply (GeMM) operation is executed.

4.4.2 Construction of a Parallel Hybrid BLR Parent Front. In our implementation, the BLR matrix is distributed among MPI communicators—that is, a certain amount of BLR tiles is assigned to each MPI process. The parallel construction is executed for each MPI communicator. First, each MPI process collects their contribution from children fronts $F_{22;\nu_1}$ and $F_{22;\nu_2}$ needed for the update of the block columns in the parent front. This is followed by an all-to-all exchange of BLR tiles such that each MPI process has access to the necessary update tiles. Afterward, each MPI communicator updates their tiles within the columns of the parent front.

As discussed in Section 3.2.2, the communication cost of the BLR matrix incorporated in the sparse multifrontal solver vary based on the BLR variant. We concluded that there was a higher communication cost for a BLR(Hybrid) matrix compared to BLR(RL) and BLR(LL) in Section 3.2.2. In addition, when BLR is used within a multifrontal solver, BLR(Hybrid) requires additional communication steps for the extend-add operation sending updates from children fronts to parent frontal matrices. These observations lead to an increased factorization time, as can be seen in Section 5.

5 EXPERIMENTAL RESULTS

Experiments reported here are all performed on the Haswell nodes of the Cori machine, a Cray XC40, at the National Energy Research Scientific Computing Center in Berkeley. Each of the 2,388 Haswell nodes has two 16-core Intel Xeon E5-2698v3 processors and 128 GB of 2.133-MHz DDR4 memory. The approximate multifrontal solver is used as a preconditioner for restarted GMRES(30) with modified Gram-Schmidt and a zero initial guess. Unless noted otherwise, all experiments are performed in double precision with absolute or relative stopping criteria $\|u_i\| \leq 10^{-10}$ or $\|u_i\|/\|u_0\| \leq 10^{-6}$, where $u_i = M^{-1}(Ax_i - b)$ is the residual at Krylov iteration i, with M the approximate multifrontal factorization of A. We use iterative refinement instead of GMRES for the exact multifrontal solver, which is also called *multifrontal solver with no compression*. For simplicity, all experiments are in double precision. For a discussion on mixed-precision iterative refinement for approximate sparse solvers, see the work of Amestoy et al. [3].

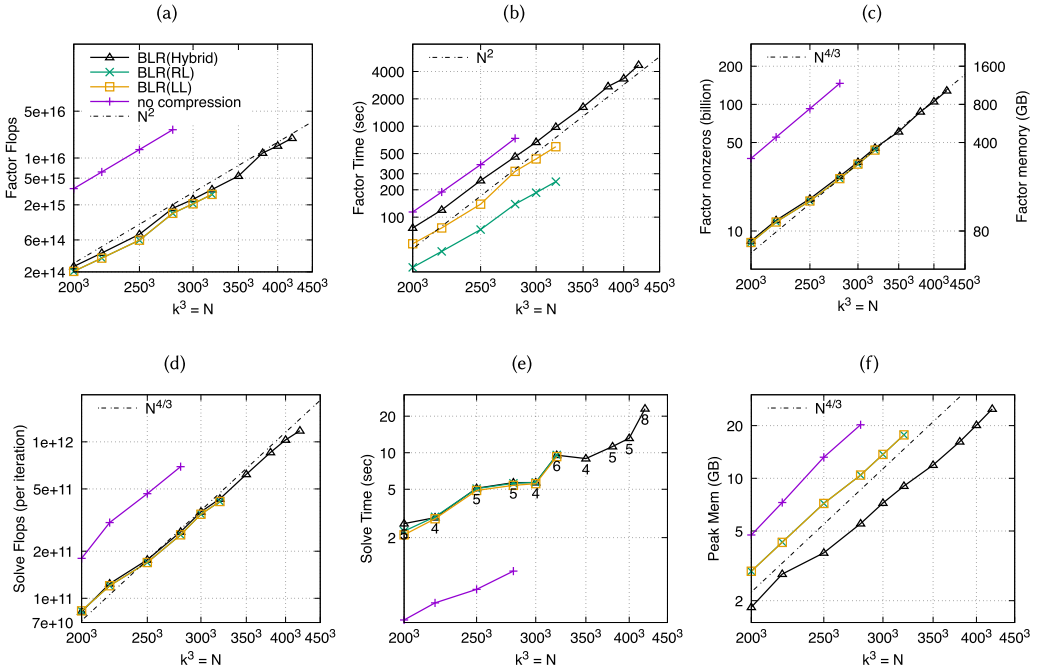


Fig. 6. Results for high-frequency 3D Helmholtz using the BLR(10^{-3}) multifrontal solver with GMRES. (a) Flop counts for factorization. (b) CPU time for factorization. (c) Memory usage for factorization (i.e., factor nonzeros/factor memory). (d) Flop counts for solve per iteration in GMRES. (e) CPU time for solve. The number of GMRES iterations are shown at every data point (additional data in Table 1). (f) Peak working memory. We use 64 compute nodes, with four MPI ranks per node and eight OpenMP threads per MPI process.

5.1 Visco-Acoustic Wave Propagation

We first consider the 3D visco-acoustic wave propagation governed by the Helmholtz equation:

$$\left(\sum_i \rho(\mathbf{x})\frac{\partial}{\partial x_i}\frac{1}{\rho(\mathbf{x})}\frac{\partial}{\partial x_i}\right)p(\mathbf{x}) + \frac{\omega^2}{\kappa^2(\mathbf{x})}p(\mathbf{x}) = -f(\mathbf{x}). \tag{7}$$

Here, $\mathbf{x} = (x_1, x_2, x_3)$, $\rho(\mathbf{x})$ is the mass density, $f(\mathbf{x})$ is the acoustic excitation, $p(\mathbf{x})$ is the pressure wave field, ω is the angular frequency, and $\kappa(\mathbf{x}) = v(\mathbf{x})(1 - i/(2q(\mathbf{x})))$ is the complex bulk modulus with the velocity $v(\mathbf{x})$ and quality factor $q(\mathbf{x})$. We solve Equation (7) by a finite-difference discretization on staggered grids using a 27-point stencil and eight PML absorbing boundary layers [36]. This requires direct solution of a sparse linear system where each matrix row contains 27 nonzeros, whose values depend on the coefficients and frequency in Equation (7).

We consider a cubed domain with $v(\mathbf{x}) = 4{,}000$ m/s, $\rho(\mathbf{x}) = 1\text{kg/m}^3$, and $q(\mathbf{x}) = 10^4$. The frequency is set to $\omega = 8\pi$ Hz, and the grid spacing is set such that there are 15 grid points per wavelength.

First, we vary N from 200^3 to 420^3 and compare four types of multifrontal solvers: "no compression" (exact solver), "BLR(RL)" (BLR compression with RL variant), "BLR(LL)" (BLR compression with LL variant), and "BLR(Hybrid)" (BLR compression with hybrid variant) (Figure 6). All fronts corresponding to separators with size $n_{\text{Bmin}} \geq 200$ are compressed with tolerance $\varepsilon = 10^{-3}$. For all experiments, the relative error was $\|x - \tilde{x}\|_2/\|\tilde{x}\|_2 < 10^{-5}$. We observed that all variants of

Table 1. Iteration Counts Corresponding to Results in Figure 6

problem size	200^3	220^3	250^3	280^3	300^3	320^3	350^3	380^3	400^3	420^3
BLR(Hybrid)	5	4	5	5	4	6	4	5	5	8
BLR(RL)	4	4	5	5	4	6				
BLR(LL)	4	4	5	5	4	6				

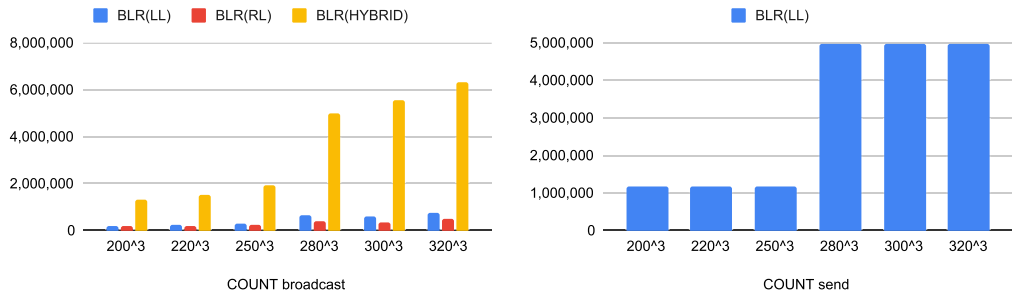


Fig. 7. Amount of communication for BLR(LL), BLR(RL), and BLR(Hybrid). (a) Number of broadcast operations. (b) Number of send operations. BLR(RL) and BLR(Hybrid) require zero send operations; see Section 3.2.2.

the BLR multifrontal solver outperform the exact solver in terms of factor flops, factor time, factor nonzeros, and peak memory (see Figure 6). The iteration counts are shown in Figure 6(e) for BLR(Hybrid) (see Table 1 for iteration counts of other variants). The BLR(RL) version outperforms the other two BLR variants in terms of factor time. This can be explained with additional communication cost for the other two BLR variants. As explained in Section 3.2.2, BLR(RL) and BLR(Hybrid) consist of broadcast operations only, whereas BLR(LL) needs additional send operations. The formula in Section 3.2.2 indicates that BLR(RL) uses the least amount of broadcast operations and BLR(Hybrid) uses significantly more than the other two variants (Figure 7(a)), which is due to additional broadcast operations needed for each newly constructed set of columns. Figure 7(b) shows that the BLR(LL) variant dominates the amount of send operations since they are not used within fronts in the other two variants. The BLR(Hybrid) algorithm has lower peak memory consumption, which allows solving of larger problem sizes. BLR(RL) and BLR(LL) run out of memory when solving Equation (7) with sizes larger than 320^3. Among all three variants, BLR(RL) requires the least amount of communication and is the fastest, but it requires the largest amount of memory.

Next, we consider a problem with size $N = k^3$, with k ranging from 200 to 420, and compare the performance of four multifrontal solver, with BLR, HODBF, HODBF_BLR and HODBF_BLR_ZFP (Figure 8). For the BLR multifrontal solver, we use the BLR(Hybrid) variant, which allows solving of problems of size up to 420^3. We set the tolerance $\varepsilon = 10^{-2}$ for HODBF and $\varepsilon = 10^{-3}$ for BLR, respectively. These different tolerances are important settings to ensure good butterfly compression ($\varepsilon = 10^{-2}$) and accurate matrix entries in BLR ($\varepsilon = 10^{-3}$) that are extracted and yield the desired butterfly ranks. The zfp compressed fronts use 16 bitplanes in HODBF_BLR_ZFP.

For HODBF compression, all fronts corresponding to separators with sizes $n_{\text{Hmin}} \geq 7\text{K}$ are compressed. For HODBF_BLR_ZFP and HODBF_BLR compression, all fronts corresponding to separators with sizes $n_{\text{Hmin}} \geq 15\text{K}$ are compressed with HODBF, all fronts corresponding to separators $200 \leq n_{Bmin} \leq 15\text{K}$ are compressed with BLR, and all fronts corresponding to separators smaller than 200 are either not compressed or compressed with zfp compression. For all experiments, the

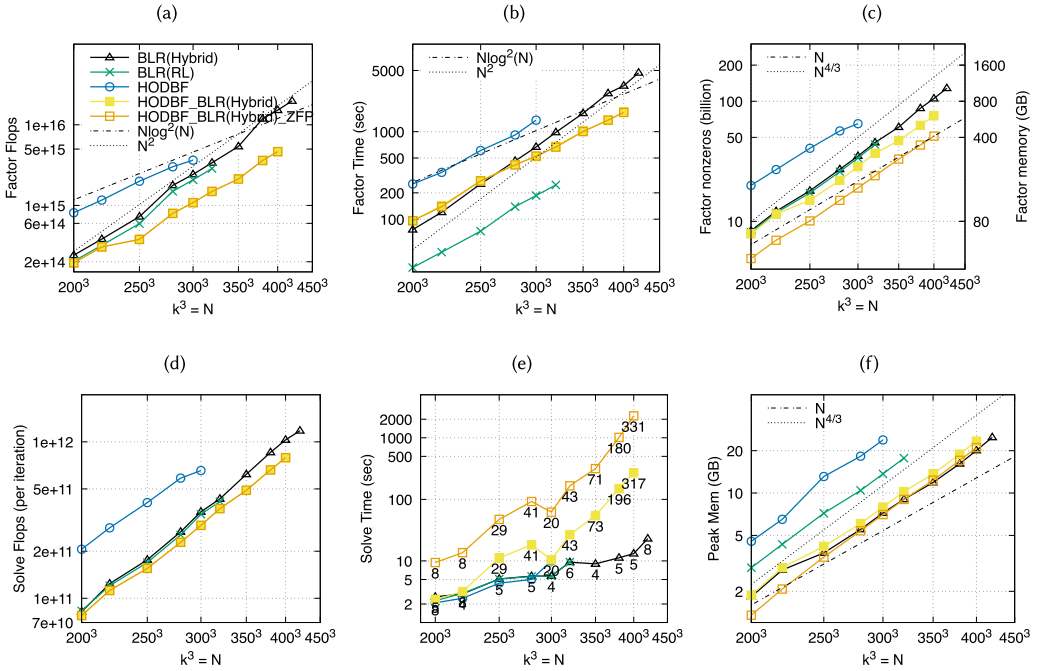


Fig. 8. Results for high-frequency 3D Helmholtz using the four different multifrontal solver with GMRES. (a) Flop counts for factorization. (b) CPU time for factorization. (c) Memory usage for the factorization (i.e., factor nonzeros/factor memory). (d) Flop counts for solve per iteration in GMRES. (e) CPU time for solve. The number of GMRES iterations are shown at every data point (additional data in Table 2). (f) Peak working memory. We use 64 compute nodes, with four MPI ranks per node and eight OpenMP threads per MPI process.

Table 2. Iteration Counts Corresponding to Results in Figure 8

problem size	200^3	220^3	250^3	280^3	300^3	320^3	350^3	380^3	400^3	420^3
BLR(Hybrid)	5	4	5	5	4	6	4	5	5	8
BLR(RL)	4	4	5	5	4	6				
HODBF	6	6	9	8	15					
HODBF_BLR(Hybrid)	8	8	29	41	20	43	73	196	317	
HODBF_BLR(Hybrid)_ZFP	8	8	29	41	20	43	72	179	354	

relative error was $\|x - \tilde{x}\|_2 / \|\tilde{x}\|_2 < 10^{-5}$. Compared to the $O(N^2)$ computation and $O(N^{4/3})$ memory complexities using the exact multifrontal solver and the BLR multifrontal solver, we observe the predicted $O(N \log^2 N)$ computation and $O(N)$ memory complexities (Figure 8(a)–(e)) for the HODBF multifrontal solver variants. The iteration counts are shown in Figure 8(e) for BLR(Hybrid) and HODBF_BLR_ZFP (see Table 2 for iteration counts of the other multifrontal solvers).

Note that HODBF_BLR_ZFP and HODBF_BLR outperform the other solvers in terms of factor time, factor flops, and solve flops. HODBF_BLR_ZFP outperforms all solvers in terms of factor nonzeros. Due to its lower peak memory consumption, the BLR(Hybrid) multifrontal solver allows solving of larger problem sizes up to 420^3 with a low solve time. From Figure 8, notice that the peak memory is higher than the factor memory, which is due to the need for temporary construction of the F_{22} blocks.

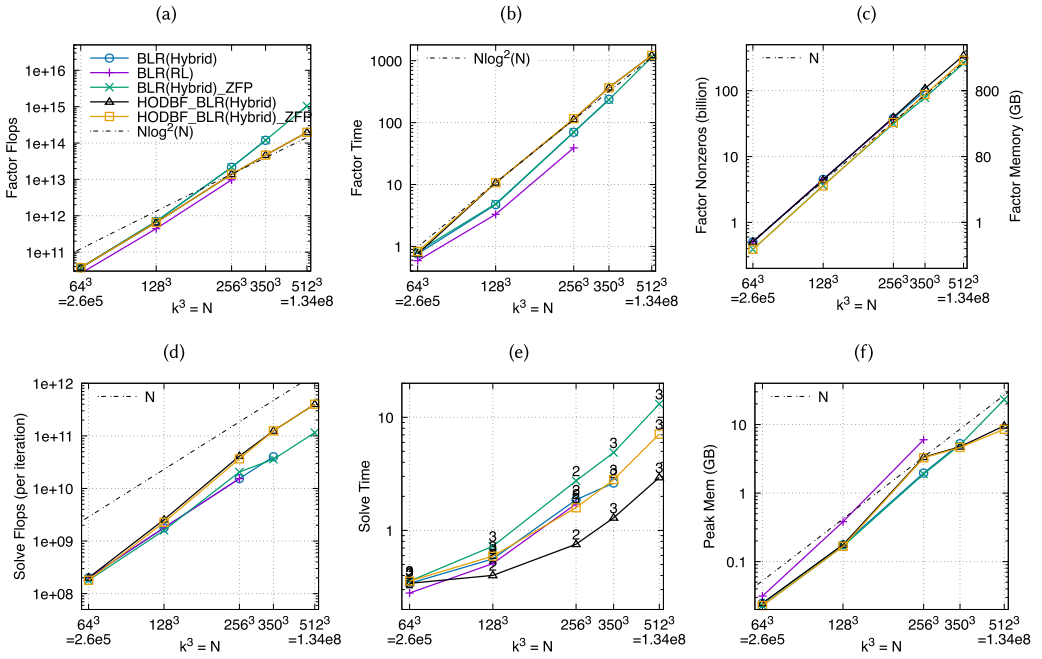


Fig. 9. Results for 3D singularly perturbed differential equations using the BLR, BLR-ZFP, BLR-HODBF, and BLR-HODBF-ZFP (10^{-2}) multifrontal solver with GMRES. (a) Flop counts for factorization. (b) CPU time for factorization. (c) Memory usage for factorization (i.e., factor nonzeros/factor memory). (d) Flop counts for solve per iteration in GMRES. (e) CPU time for solve. (f) Peak working memory per MPI. We use 64 compute nodes, with two MPI ranks per node and 16 OpenMP threads per MPI process.

5.2 Singularly Perturbed Differential Equations

Next, we consider the following 3D singularly perturbed reaction-diffusion differential equation that arises in fluid dynamics, computational chemistry, and biological applications (e.g., see [41]):

$$-\delta^2 \Delta u + u = f, \quad \text{on } \Omega = (0,1)^3, \quad \text{and } u(\partial\Omega) = g, \tag{8}$$

where the *perturbation parameter* δ is small and positive, and g and f are some given functions. Solving large sparse systems arising from finite difference discretizations, even for the 2D analogue of (8), is a challenging task. For example, MacLachlan and Madden [30] showed that standard Cholesky-based solvers exhibit poor performance when δ is small. The underlying reason for this is that the fill-in entries in the Cholesky factors are so small as to fall into the range of *subnormal* numbers, which are expensive to compute [37]. A thorough investigation of how the subnormal numbers propagate in the Cholesky factors for the 2D problems can be found in the work of Nhan and Madden [34].

Here, we set $\delta = 10^{-4}$ and solve Equation (8) by a 7-point finite difference discretization. We vary N from 64^3 to 512^3 and compare the performance of four multifrontal solvers, with BLR, BLR_ZFP, HODBF_BLR, and HODBF_BLR_ZFP (Figure 9). We set the tolerance $\varepsilon = 10^{-2}$ for both the BLR and the HODBF compression tolerance and use 16 bitplanes for zfp compression. For the multifrontal solver with BLR or BLR_ZFP compression, all fronts with separator sizes $n_{\text{Bmin}} \geq 200$ are compressed with BLR. For the multifrontal solver with HODBF_BLR_ZFP or HODBF_BLR compression, all fronts with separator sizes $n_{\text{Hmin}} \geq 5\text{K}$ are compressed with HODBF, all fronts corresponding to separator sizes $200 \leq n_{\text{Bmin}} \leq 5\text{K}$ are compressed with BLR, and all fronts

corresponding to separator sizes below 200 are either not compressed or compressed with zfp compression. For all experiments, the relative error was $\|x - \tilde{x}\|_2 / \|\tilde{x}\|_2 < 10^{-5}$. We observed that when adding the HODBF compression, one can attain the $O(N \log^2 N)$ computation and $O(N)$ memory complexities. Adding zfp on top of BLR and HODBF can slightly improve the total factor memory, but it can increase the solve time per iteration.

5.3 Incompressible Navier-Stokes Flow

We solve linear systems with the Stokes operator, modeling incompressible flow described by the Navier-Stokes equations. The system is discretized on a regular 3D mesh using a staggered grid with the velocity components approximated at the cell faces and the pressure at the cell centers. Velocity boundary conditions are used on all (six) faces of the cube. For a 3D cube with k mesh points in each direction, there are $3(k+1)k^2$ velocity degrees of freedom and k^3 pressure degrees of freedom. The discretization is performed using IBAMR [13, 14, 33] (Immersed Boundary Method Adaptive Mesh Refinement Software Infrastructure), which is built on top of SAMRAI [5] (Structured Adaptive Mesh Refinement Application Infrastructure). Matrix assembly is done through PETSc [7].

Discretization of the governing equations leads to a linear system $Mx = b$, or

$$\begin{bmatrix} A & G \\ -D & 0 \end{bmatrix} \begin{bmatrix} x_u \\ x_p \end{bmatrix} = \begin{bmatrix} b_u \\ b_p \end{bmatrix}, \tag{9}$$

where A corresponds to the temporal and viscous terms, G to the (pressure) gradient, and D to the diverge (of the velocity). x_u and x_p are the velocity (three spatial components) and pressure, respectively.

Since the pressure is only defined up to a constant, the matrix M is singular. The nullspace is $Z = k^{-3/2}\,[0\ \ldots\ 0\,1\ \ldots\ 1]$, with the last k^3 elements of Z corresponding to the pressure degrees of freedom. We construct an exact solution $\tilde{x} = x_\mathrm{r} - ZZ^T x_\mathrm{r}$ with x_r a vector with elements in $\mathcal{N}(0,1)$. From the solution vector x obtained using BiCGStab with the approximate multifrontal preconditioner, we compute the final solution $x \leftarrow x - ZZ^T x$ and compare that to $\tilde{x}$. The BiCGStab stopping criterion is a 10^{-10} relative residual decrease. Results of the BLR_ZFP multifrontal solver and the BLR multifrontal solver are shown in Figure 10 with k varying from 100 to 250. For BLR, fronts corresponding to separator sizes n_{Bmin} above 1,000 are compressed with a relative compression tolerance 10^{-6} and all fronts corresponding to separators sizes below 1,000 are either not compressed or compressed with zfp compression using 32 bitplanes. M has a large zero block on the diagonal. Our solver can apply a static permutation, before the start of the numerical factorization, to make the main diagonal of the matrix nonzero. This permutation is implemented using the MC64 matching code. However, doing so completely destroys the symmetry of the pattern of M. Note that our solver computes (nonsymmetric) LU factorization, but using a symmetric nonzero pattern. Furthermore, we observed numerical difficulties when trying to solve the linear system with the matrix M permuted with the MC64 matching. Our solver also implements a small pivot replacement option, in which, during numerical factorization, small pivots, which would cause overflow during triangular solution, are replaced with a slightly larger value of $\sqrt{\varepsilon_{\mathrm{mach}}}\|M\|_1$. However, since the operator M is highly ill conditioned, instead of relying on this small pivot replacement, we replace the zero diagonal elements of M with $\tau\sqrt{\varepsilon_{\mathrm{mach}}}\|M\|_1$ before starting the numerical factorization. Since we also apply a matrix equilibration similar to LAPACK's dgeequ/dlaqge, $\|M\|_1 = 1$. The factor τ needs to be chosen carefully. A larger τ reduces the condition number of the preconditioner and allows for better compression. However, a larger τ leads to a worse preconditioner, resulting in more BiCGStab iterations. We pick $\tau = 10^4$. The diagonal shift is also applied for the multifrontal solver without compression, which is

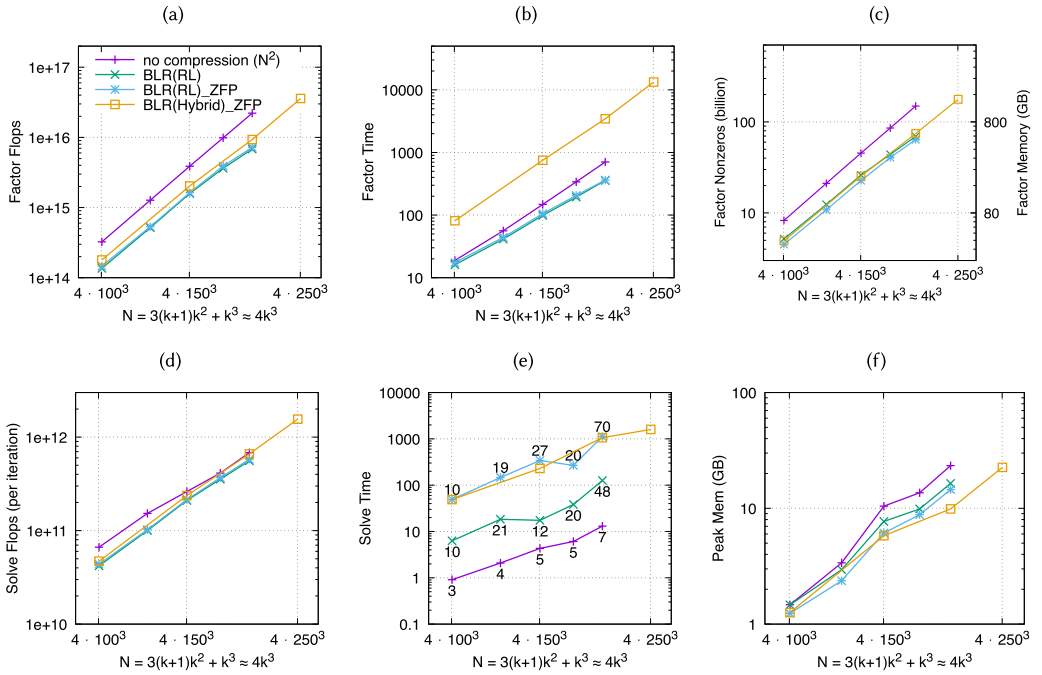


Fig. 10. Results for linear systems modeling incompressible Navier-Stokes flow using the BLR and BLR-ZFP multifrontal solver with BiCGStab. (a) Flop counts for factorization. (b) CPU time for factorization. (c) Memory usage for factorization (i.e., factor nonzeros/factor memory). (d) Total solve flop counts in BiCGStab and the multifrontal preconditioner. (e) CPU time for solve. (f) Peak working memory per MPI rank. We use 64 compute nodes, with four MPI ranks per node and eight OpenMP threads per MPI process.

then also used with BiCGStab (instead of iterative refinement). For all experiments, the relative error was $\|x - \tilde{x}\|_2/\|\tilde{x}\|_2 < 10^{-5}$. Figure 10 shows a lower peak memory consumption when using BLR(Hybrid)_ZFP, which allows solving of larger problem sizes. However, BLR(RL) and BLR(RL)_ZFP outperform BLR(Hybrid)_ZFP in terms of factor and solve time. We did not include HODBF results for these highly ill conditioned matrices, since HODBF with high accuracy will not compress sufficiently, and using HODBF with low accuracy will cause nonconvergence in GMRES.

5.4 SuiteSparse Matrix Collection

The problems shown so far were all defined on a regular 3D domain. Table 3 shows results for a number of matrices from the SuiteSparse matrix collection. These are some of the larger problems in this collection, and they correspond to a range of different applications, with varying numerical and structural properties. Table 3 shows a comparison between the exact solver (no compression) and the approximate multifrontal solver with either BLR compression (RL variant with relative compression tolerance 10^{-2}) or the BLR_ZFP compression (lossy compression with 32 bitplanes). For all other parameters, and for each problem, the default values are used (e.g., $n_{\text{Bmin}} = 512$, $n_{\text{Lmin}} = 8$).

In Table 3, we notice the speedup obtained for the factorization when using the BLR compression. When using BLR_ZFP compression, there is a small overhead in factorization time compared to using BLR compression only. Likewise, when enabling zfp compression, the solve becomes significantly slower. However, enabling zfp leads to better compression ratios. For instance, for the

Table 3. Results for the Numerical Factorization and Solve for a Number of Matrices from the SuiteSparse Matrix Collection

			no compr.			BLR($RL,10^{-2}$) compression								BLR($RL,10^{-2}$)_ZFP(32) compression								
	N	nnz	fact	sol		#fronts		fact	sol		comp	peak	rel.	#fronts			fact	sol		comp	peak	rel.
matrix	$\times 10^3$	$\times 10^3$	(s)	(s)	its	dense	BLR	(s)	(s)	its	(%)	(%)	err.	dense	ZFP	BLR	(s)	(s)	its	(%)	(%)	err.
boneS01	127	6,715	0.45	0.02	1	15K	4	**0.38**	0.20	14	87.3	101.0	8e-06	11K	4K	4	0.43	4.92	14	**46.7**	74.0	8e-06
xenon2	157	3,866	0.49	0.02	1	19K	8	**0.47**	0.23	12	87.4	102.0	9e-07	13K	5K	8	0.56	4.49	12	**50.2**	76.0	9e-07
scircuit	170	959	0.12	0.01	1	21K	0	**0.12**	0.01	1	100.0	102.0	1e-10	12K	8K	0	0.12	0.11	4	**53.7**	47.0	2e-08
pwtk	217	11,634	0.27	0.02	1	27K	1	**0.27**	0.06	3	99.7	99.0	6e-08	20K	6K	1	0.33	1.06	3	**50.8**	70.0	1e-06
torso3	259	4,632	0.81	0.03	1	32K	14	**0.59**	0.10	2	72.1	100.0	2e-07	25K	7K	14	0.70	1.66	2	**37.0**	70.0	2e-07
cage13	445	7,479	75.11	0.25	1	55K	56	**28.74**	0.53	2	24.1	54.0	6e-07	46K	8K	56	31.69	8.50	2	**17.8**	50.0	6e-07
audikw_1	943	77,651	12.54	0.42	1	117K	124	**6.60**	4.90	48	47.7	47.0	5e-04	94K	22K	124	7.04	141.04	58	**29.2**	37.0	1e-03
atmosmodd	1,270	8,814	9.82	0.12	1	158K	63	**5.60**	1.03	8	44.5	64.0	5e-07	106K	52K	63	5.94	15.32	8	**29.3**	51.0	5e-07
Serena	1,391	64,531	42.39	0.36	1	173K	134	**18.66**	1.87	10	34.8	52.0	2e-05	137K	36K	134	19.01	55.42	10	**22.3**	39.0	2e-02
Geo_1438	1,437	63,156	26.76	0.23	1	179K	177	**12.76**	2.01	13	45.4	47.0	8e-05	142K	36K	177	13.39	41.93	12	**30.1**	36.0	1e-03
atmosmodl	1,489	10,319	9.18	0.13	1	186K	64	**5.40**	0.60	4	45.8	64.0	9e-07	126K	59K	64	5.85	10.36	4	**27.4**	52.0	9e-07
Hook_1498	1,498	60,917	15.00	0.22	1	187K	135	**7.81**	4.81	34	46.9	48.0	4e-05	137K	49K	135	8.37	139.68	34	**27.3**	31.0	4e-05
ML_Geer	1,504	110,879	4.79	0.14	1	187K	115	**2.97**	2.32	24	65.4	40.0	2e-05	144K	43K	115	3.79	84.81	23	**33.5**	24.0	2e-05
Transport	1,602	23,500	8.76	0.18	1	200K	128	**4.69**	3.01	26	51.7	60.0	1e-05	152K	47K	128	5.41	81.27	26	**30.5**	42.0	7e-06
memchip	2,707	15,950	0.35	0.13	1	338K	0	**0.35**	0.17	1	100.0	100.0	7e-15	204K	134K	0	0.42	0.68	1	**48.3**	51.0	2e-06

Compression ratio (comp %) refers to the size of the final LU factors relative to the exact solver without compression, whereas peak (%) refers to peak memory usage during factorization, also relative to the exact solver. The factorization is always fastest for the solver with BLR compression. The hybrid BLR_ZFP solver uses the least memory.

scircuit and memchip systems, using BLR only does not give any compression, whereas zfp compresses the factors by at least 2×. Unlike most of the other problems, the scircuit and memchip matrices are not derived from PDE discretizations, which explains why their graphs do not have separators larger than $n_{\text{Bmin}} = 512$. The fact that there are no large separators means that the amount of fill-in will be relatively small, and hence the exact sparse direct solver should be an efficient solver.

6 CONCLUSION

This article presented a fast and approximate multifrontal solver for large sparse linear systems. The solver leverages HODBF, a reduced-memory version of the nonhierarchical BLR format, BLR, and lossy compression. Depending on the application as well as problem sizes, we made use of different combinations of the three compression methods. In general, HODBF is used to compress large frontal matrices, BLR for medium-sized frontal matrices, and lossy compression for small frontal matrices. The reduced-memory version of the BLR format, BLR(Hybrid), leads to a reduction in peak memory consumption that allows solving of larger problem sizes. The resulting solver can attain the $O(N \log^2 N)$ computation and $O(N)$ memory complexities when adding HODBF compression. Some of the presented results for smaller problem sizes do not make use of HODBF compression because HODBF is beneficial for really large fronts only. Adding zfp on top of BLR and/or HODBF can improve the compression ratios and the total factor memory, but it can increase the solve time per iteration.

The code is made publicly available through the sparse solver package STRUMPACK.[2] The HODBF implementation is integrated using the dense solver package ButterflyPACK,[3] and lossy compression is provided with the software package zfp.[4]

REFERENCES

[1] Sivaram Ambikasaran and Eric Darve. 2013. An $O(N \log N)$ fast direct solver for partial hierarchically semi-separable matrices. *SIAM J. Sci. Comput.* 57, 3 (Dec. 2013), 477–501.

[2] https://github.com/pghysels/STRUMPACK
[3] https://github.com/liuyangzhuan/ButterflyPACK
[4] https://github.com/LLNL/zfp

[2] Patrick Amestoy, Cleve Ashcraft, Olivier Boiteau, Alfredo Buttari, Jean-Yves L'Excellent, and Clément Weisbecker. 2015. Improving multifrontal methods by means of block low-rank representations. *SIAM J. Sci. Comput.* 37, 3 (2015), A1451–A1474.

[3] Patrick Amestoy, Alfredo Buttari, Nicholas J. Higham, Jean-Yves L'Excellent, Théo Mary, and Bastien Vieuble. 2023. Combining sparse approximate factorizations with mixed-precision iterative refinement. *ACM Trans. Math. Softw.* 49, 1 (2023), 1–29.

[4] Patrick R. Amestoy, Alfredo Buttari, Jean-Yves L'Excellent, and Theo Mary. 2019. Performance and scalability of the block low-rank multifrontal factorization on multicore architectures. *ACM Trans. Math. Softw.* 45, 1 (Feb. 2019), Article 2, 26 pages. https://doi.org/10.1145/3242094

[5] R. Anderson, W. Arrighi, N. Elliott, B. Gunney, and R. Hornung. 2013. *SAMRAI Concepts and Software Design*. Technical Report. LLNL.

[6] Ariful Azad, Aydin Buluç, Xiaoye S. Li, Xinliang Wang, and Johannes Langguth. 2020. A distributed-memory algorithm for computing a heavy-weight perfect matching on bipartite graphs. *SIAM J. Sci. Comput.* 42, 4 (2020), C143–C168. https://doi.org/10.1137/18M1189348

[7] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. Gropp, D. Karpeyev, D. Kaushik, M. Knepley, D. May, L. Curfman McInnes, R. Mills, T. Munson, K. Rupp, P. Sanan, B. Smith, S. Zampini, H. Zhang, and H. Zhang. 2018. *PETSc Users Manual*. Argonne National Laboratory.

[8] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. 1998. *Numerical Linear Algebra for High-Performance Computers*. Society for Industrial and Applied Mathematics, Philadelphia, PA. https://doi.org/10.1137/1.9780898719611

[9] Pieter Ghysels. 2014. STRUMPACK: STRUctured Matrix PACKage. Retrieved August 7, 2023 from http://portal.nersc.gov/project/sparse/strumpack/

[10] Pieter Ghysels, Xiaoye Sherry Li, Christopher Gorman, and François-Henry Rouet. 2017. A robust parallel preconditioner for indefinite systems using hierarchical matrices and randomized sampling. In *Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS'17)*. IEEE, Los Alamitos, CA, 897–906.

[11] G. Karypis and V. Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.* 20, 1 (1998), 359–392.

[12] Lars Grasedyck and Wolfgang Hackbusch. 2003. Construction and arithmetics of h-matrices. *Computing* 70, 4 (2003), 295–334.

[13] Boyce Eugene Griffith. 2005. *Simulating the Blood-Muscle-Valve Mechanics of the Heart by an Adaptive and Parallel Version of the Immersed Boundary Method*. Ph. D. dissertation. New York University.

[14] Boyce E. Griffith, Richard D. Hornung, David M. McQueen, and Charles S. Peskin. 2007. An adaptive, formally second order accurate version of the immersed boundary method. *J. Comput. Phys.* 223, 1 (2007), 10–49.

[15] Wolfgang Hackbusch. 1999. A sparse matrix arithmetic based on $\mathcal{H}$-matrices. Part I: Introduction to $\mathcal{H}$-matrices. *Computing* 62, 2 (April 1999), 89–108.

[16] Wolfgang Hackbusch, Boris N. Khoromskij, and Ronald Kriemann. 2004. Hierarchical matrices based on a weak admissibility criterion. *Computing* 73, 3 (2004), 207–243.

[17] Pascal Hénon, Pierre Ramet, and Jean Roman. 2002. PaStiX: A high-performance parallel direct solver for sparse symmetric positive definite systems. *Parallel Comput.* 28, 2 (2002), 301–321.

[18] Nicholas J. Higham and Theo Mary. 2022. Solving block low-rank linear systems by LU factorization is numerically stable. *IMA J. Numer. Anal.* 42, 2 (2022), 951–980.

[19] I. S. Duff and J. Koster. 1999. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM J. Matrix Anal. A.* 20, 4 (1999), 889901.

[20] I. S. Duff and J. K. Reid. 1983. The multifrontal solution of indefinite sparse symmetric linear. *ACM Trans. Math. Softw.* 9, 3 (1983), 302–325.

[21] Yingzhou Li and Haizhao Yang. 2017. Interpolative butterfly factorization. *SIAM J. Sci. Comput.* 39, 2 (2017), A503–A531. https://doi.org/10.1137/16M1074941

[22] Yingzhou Li, Haizhao Yang, Eileen R. Martin, Kenneth L. Ho, and Lexing Ying. 2015. Butterfly factorization. *Multiscale Model. Sim.* 13, 2 (2015), 714–732.

[23] Peter Lindstrom. 2014. Fixed-rate compressed floating-point arrays. *IEEE Trans. Vis. Comput. Graph.* 20 (2014), 2674–2683. https://doi.org/10.1109/TVCG.2014.2346458

[24] Peter Lindstrom. 2014. zfp: Compressed Floating-Point and Integer Arrays. Retrieved August 7, 2023 from https://computing.llnl.gov/projects/zfp

[25] J. W. H. Liu. 1992. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Rev.* 34 (1992), 82–109. https://doi.org/10.1137/1034004

[26] Yang Liu. 2018. ButterflyPACK. Retrieved August 7, 2023 from https://portal.nersc.gov/project/sparse/butterflypack/

[27] Yang Liu, Pieter Ghysels, Lisa Claus, and Xiaoye Sherry Li. 2021. Sparse approximate multifrontal factorization with butterfly compression for high-frequency wave equations. *SIAM J. Sci. Comput.* 43, 5 (2021), S367–S391. https://doi.org/10.1137/20M1349667

[28] Yang Liu, Han Guo, and Eric Michielssen. 2017. An HSS matrix-inspired butterfly-based direct solver for analyzing scattering from two-dimensional objects. *IEEE Antennas Wirel. Propag. Lett.* 16 (2017), 1179–1183.

[29] Yang Liu, Xin Xing, Han Guo, Eric Michielssen, Pieter Ghysels, and Xiaoye Sherry Li. 2021. Butterfly factorization via randomized matrix-vector multiplications. *SIAM J. Sci. Comput.* 43, 2 (2021), A883–A907. https://doi.org/10.1137/20M1315853

[30] S. MacLachlan and N. Madden. 2013. Robust solution of singularly perturbed problems using multigrid methods. *SIAM J. Sci. Comput.* 35, 5 (2013), A2225–A2254. https://doi.org/10.1137/120889770

[31] Théo Mary. 2017. *Block Low-Rank Multifrontal Solvers: Complexity, Performance, and Scalability*. Ph. D. dissertation. l'Université de Toulouse.

[32] Eric Michielssen and Amir Boag. 1994. Multilevel evaluation of electromagnetic fields for the rapid solution of scattering problems. *Microw. Opt. Technol. Lett.* 7, 17 (1994), 790–795.

[33] Nishant Nangia, Boyce E. Griffith, Neelesh A. Patankar, and Amneet Pal Singh Bhalla. 2019. A robust incompressible Navier-Stokes solver for high density ratio multiphase flows. *J. Comput. Phys.* 390 (2019), 548–594.

[34] T. A. Nhan and N. Madden. 2015. Cholesky factorisation of linear systems coming from finite difference approximations of singularly perturbed problems. In *Boundary and Interior Layers, Computational and Asymptotic Methods—BAIL 2014*, Petr Knobloch (Ed.). Springer International Publishing, Cham, Switzerland, 209–220.

[35] Richard Nies and Matthias Hoelzl. 2019. Testing performance with and without block low rank compression in MUMPS and the new PaStiX 6.0 for JOREK nonlinear MHD simulations. *arXiv e-prints arXiv:1907.13442 (2019).*

[36] Stéphane Operto, Jean Virieux, Patrick Amestoy, Jean-Yves L'Excellent, Luc Giraud, and Hafedh Ben Hadj Ali. 2007. 3D finite-difference frequency-domain modeling of visco-acoustic wave propagation using a massively parallel direct solver: A feasibility study. *Geophysics* 72, 5 (2007), SM195–SM211.

[37] Michael L. Overton. 2001. *Numerical Computing with IEEE Floating Point Arithmetic*. Society for Industrial and Applied Mathematics, Philadelphia, PA. https://doi.org/10.1137/1.9780898718072

[38] Qiyuan Pang, Kenneth L. Ho, and Haizhao Yang. 2020. Interpolative decomposition butterfly factorization. *SIAM J. Sci. Comput.* 42, 2 (2020), A1097–A1115. https://doi.org/10.1137/19M1294873

[39] François Pellegrini and Jean Roman. 1997. Sparse matrix ordering with scotch. In *High-Performance Computing and Networking*, Bob Hertzberger and Peter Sloot (Eds.). Springer, Berlin, Germany, 370–378. https://doi.org/10.1007/BFb0031609

[40] Grégoire Pichon, Eric Darve, Mathieu Faverge, Pierre Ramet, and Jean Roman. 2018. Sparse supernodal solver using block low-rank compression: Design, performance and analysis. *Int. J. Comput. Sci. Eng.* 27 (2018), 255–270.

[41] H.-G. Roos, M. Stynes, and L. Tobiska. 2008. *Robust Numerical Methods for Singularly Perturbed Differential Equations* (2nd ed.). Springer Series in Computational Mathematics, Vol. 24. Springer-Verlag, Berlin, Germany.

[42] Sadeed Bin Sayed, Yang Liu, Luis J. Gomez, and Abdulkadir C. Yucel. 2022. A butterfly-accelerated volume integral equation solver for broad permittivity and large-scale electromagnetic analysis. *IEEE Trans. Antennas Propag.* 70, 5 (2022), 3549–3559. https://doi.org/10.1109/TAP.2021.3137193

[43] John Shaeffer. 2008. Direct solve of electrically large integral equations for problem sizes to 1 M unknowns. *IEEE Trans. Antennas Propag.* 56, 8 (2008), 2306–2313.

[44] Raf Vandebril, Marc Van Barel, Gene Golub, and Nicola Mastronardi. 2005. A bibliography on semiseparable matrices. *CALCOLO* 42, 3-4 (2005), 249–270.

[45] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide*. Vol. 9.

Received 18 November 2022; revised 17 May 2023; accepted 17 July 2023