# Sparse approximate multifrontal factorization with composite compression methods 

LISA CLAUS, National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory, USA

PIETER GHYSELS, Applied Mathematics and Computational Research Division, Lawrence Berkeley National Laboratory, USA
YANG LIU, Applied Mathematics and Computational Research Division, Lawrence Berkeley National Laboratory, USA
THÁl ANH NHAN, Department of Mathematics and Computer Science, Santa Clara University, USA
RAMAKRISHNAN THIRUMALAISAMY, Department of Mechanical Engineering, San Diego State University, USA

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 paper presents a fast and approximate multifrontal solver for large sparse linear systems. In a recent paper by Liu et al. we showed the efficiency of a multifrontal solver leveraging the butterfly algorithm and its hierarchical matrix extension, Hierarchically OffDiagonal Butterfly compression (HODBF), 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 in order to scale to larger problem sizes, in this paper, we develop a composite multifrontal solver that employs the HODBF format for large sized fronts, a reduced-memory version of the non-hierarchical 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 $2.7 \times$ larger than before and leads to a memory consumption that is reduced by 70 percent while ensuring the same execution time. The code is made publicly available in github.

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

## ACM Reference Format:

Lisa Claus, Pieter Ghysels, Yang Liu, Thái Anh Nhan, Ramakrishnan Thirumalaisamy, Amneet Pal Singh Bhalla, and Sherry Li. 2018. Sparse approximate multifrontal factorization with composite compression methods. 1, 1 (November 2018), 29 pages. https: //doi.org/XXXXXXX.XXXXXXX

Authors' addresses: Lisa Claus, lclaus@lbl.gov, National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory, Berkeley, CA, USA; Pieter Ghysels, pghysels@lbl.gov, Applied Mathematics and Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA, USA; Yang Liu, liuyang@lbl.gov, Applied Mathematics and Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA, USA; Thái Anh Nhan, anhan@scu.edu, Department of Mathematics and Computer Science, Santa Clara University, Santa Clara, CA, USA; Ramakrishnan Thirumalaisamy, rthirumalaisam1857@sdsu.edu, Department of Mechanical Engineering, San Diego State University, San Diego, CA, USA; Amneet Pal Singh Bhalla, asbhalla@sdsu.edu, Department of Mechanical Engineering, San Diego State University, San Diego, CA, USA; Sherry Li, xsli@lbl.gov, Applied Mathematics and Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 Association for Computing Machinery.
Manuscript submitted to ACM

## 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 very efficiently on modern hardware, since 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 $O\left(N^{2}\right)$ for typical three dimensional 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 $O\left(N \log ^{\alpha} N\right)$ for some $\alpha$ 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], hierarchically off-diagonal low-rank (HODLR) formats [1], hierarchically semiseparable (HSS) formats [43], and block low-rank (BLR) formats [2, 4, 42]. 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 above mentioned algorithms, 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, Hierarchically Off-Diagonal Butterfly compression (HODBF), 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 paper, we presents a composite multifrontal solver which employs the HODBF format for compressing the large frontal matrices in the multifrontal method and leverage additional compression methods for the remaining fronts. To be more specific, HODBF serves as a good compression method as shown in [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 block low-rank (BLR) compression. We combine a left-looking and a right-looking versions of BLR to a hybrid method which 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 paper 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 which makes use of a column-wise construction of matrix tiles. The sparse approximate multifrontal solver is used as a

Manuscript submitted to ACM
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 $2.7 \times$ 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 paper is organized as follows. The multifrontal factorization method is presented in Section 2. The zfp, hierarchically off-diagonal butterfly compression and block low-rank algorithms are described in Section 3, including the classical left- and right-looking 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 conclusions 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 [20,25]. The method separates the factorization of a sparse matrix $(A=L U)$ 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 in [6] to perform the scaling and column permutation. After that, a fill-reducing permutation $A \leftarrow P A P^{T}$ is applied, i.e. 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 $\tau$ of the assembly tree is represented by a dense frontal matrix $F_{\tau}$, with the following $2 \times 2$ block structure: $F_{\tau}=\left[\begin{array}{ll}F_{11} & F_{12} \\ F_{21} & F_{22}\end{array}\right]$. The rows and columns corresponding to the $F_{11}$ block are called the fully summed variables, these variables have received all their Schur complement updates, when 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 $I_{\tau}^{u}$ index sets define the temporary Schur complement update blocks. Let $n_{\tau}=\# I_{\tau}^{s}+\# I_{\tau}^{u}$ denote the dimension of $F_{\tau}$. Note that the frontal matrices tend to get bigger toward the root of the assembly tree. Furthermore, if $v$ is a child of $\tau$ in the assembly tree, then $I_{v}^{u} \subset\left\{I_{\tau}^{s} \cup I_{\tau}^{u}\right\}$. For the root node $t, I_{t}^{u}=\emptyset$. When considering a single front, we will omit the $\tau$ 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:
(1) Assembling the frontal matrix $F_{\tau}$, 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 $\hat{\downarrow}$ :
(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.

```
Algorithm 1 Sparse multifrontal factorization and solve.
Input: \(A \in \mathbb{R}^{N \times N}, b \in \mathbb{R}^{N}\)
Output: \(x \approx A^{-1} b\)
    \(A \leftarrow D_{r} A D_{c} Q_{c} \quad \triangleright\) (optional) col perm \& scaling
    \(A \leftarrow P A P^{\top} \quad \triangleright\) symm fill-reducing reordering
    Build assembly tree: define \(I_{\tau}^{\mathrm{s}}\) and \(I_{\tau}^{\mathrm{u}}\) for every frontal matrix \(F_{\tau}\)
    for nodes \(\tau\) in assembly tree in topological order do
                                    \(\triangleright\) sparse with the children updates extended and added
        \(F_{\tau} \leftarrow\left[\begin{array}{cc}A\left(I\left(I_{\tau}^{\mathrm{I}}, I_{\tau}^{\mathrm{s}}\right)\right. \\ A\left(I_{\tau}^{\mathrm{T}}, I_{\tau}^{\mathrm{T}}\right)\end{array} \quad A\left(I_{\tau}^{\mathrm{s}}, I_{\tau}^{\mathrm{u}}\right)\right]+F_{22 ; v_{1}}+F_{22 ; v_{2}}\)
        \(P_{\tau} L_{\tau} U_{\tau} \leftarrow F_{11} \quad \triangleright\) LU with partial pivoting
        \(F_{12} \leftarrow L_{\tau}^{-1} P_{\tau}^{\top} F_{12}\)
        \(F_{21} \leftarrow F_{21} U_{\tau}^{-1}\)
        \(F_{22} \leftarrow F_{22}-F_{21} F_{12} \quad \triangleright\) Schur update
    end for
    \(x \leftarrow D_{c} Q_{c} P^{\top}\) bwd-solve (fwd-solve \(\left(P D_{r} b\right)\) )
```


(a)

(b)

Fig. 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.

Figure 1 illustrates the multifrontal algorithm for a sparse matrix resulting from the discretization of a partial differential equation using a 5-point finite difference stencil on a regular two-dimensional $11 \times 11$ mesh. Figure 1a Manuscript submitted to ACM
shows the mesh and the top 3 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 Fig. 1b. 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 MPI subcommunicators to facilitate the computation at each node and its subtree. At the root of the tree, we create a two-dimensional 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, Block Low-Rank and lossy compressed fronts while the Hierarchically Off-Diagonal Butterfly 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++, message passing interface (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 paper, 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, Block Low-Rank (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 from for example high-frequency wave equations.

## 3 COMPRESSION METHODS

In this paper, 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 $\mathrm{zfp}, \mathrm{BLR}$, 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

zfp is 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 [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 [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 Block Low-Rank Compression

Among the possible low-rank formats, Block Low-Rank (BLR) is the simplest. The format partitions the matrix with a flat, non-hierarchical blocking of the matrix which 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.

$$
\tilde{B}=\left[\begin{array}{cccc}
\tilde{B}_{11} & \tilde{B}_{12} & \ldots & \tilde{B}_{1 p}  \tag{1}\\
\vdots & \ddots & \ddots & \vdots \\
\tilde{B}_{p 1} & \ldots & \ldots & \tilde{B}_{p p}
\end{array}\right]
$$

Assuming $I=\{1, \ldots, n\}$ 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 which equals $I$. $b=\sigma \times \tau \in I \times I$ is called a block cluster based on clusters $\sigma$ and $\tau$. A block clustering of $I \times I$ is then defined as a disjoint union of block clusters which equals $I \times I$. A block $B_{\sigma \tau}$ corresponds to an interaction between two subdomains $\sigma$ and $\tau$, where $\sigma$ contains the row indices of $B_{\sigma \tau}$ while $\tau$ 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 paper 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 strong 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 $\varepsilon$. $X_{\sigma \tau}$ is a $m_{\sigma} \times r_{\sigma \tau}$ matrix and $Y_{\sigma \tau}$ is a $m_{\tau} \times r_{\sigma \tau}$ matrix.
Manuscript submitted to ACM

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. In order 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., 1995) geqp3 routine), which is modified to stop the factorization when the diagonal coefficient of $\mathrm{R}, r_{i, i}$, falls below a prescribed threshold $\varepsilon$. We use a relative tolerance (i.e. we stop the factorization after $\left|r_{i, i}\right| /\left|r_{0,0}\right|<\varepsilon$ ).

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 above: FSUC, FSCU, FCSU, CFSU. These acronyms indicate the order in which the tasks are performed [31].

```
Algorithm 2 Right-looking BLR algorithm: FCSU
Input: a \(p \times p\) block matrix \(B\) of size \(n\),
\(B=\left[B_{i, j}\right]_{i=1: p, j=1: p}\)
    Construct tiles \(B_{i, j}, \forall i, j\)
    for \(i=1\) to \(p\) do
        Factor: \(B_{i, i}=L_{i, i} U_{i, i}\)
        for \(j=i+1\) to \(p\) do
            Compress: \(B_{i, j} \approx X_{i, j} Y_{i, j}^{T}\)
            Solve: \(B_{i, j} \leftarrow L_{i, i}^{-1} B_{i, j}\)
            Compress: \(B_{j, i} \approx X_{j, i} Y_{j, i}^{T}\)
            Solve: \(B_{j, i} \leftarrow B_{j, i} U_{i, i}^{-1}\)
        end for
        for \(j=i+1\) to \(p\) do
            for \(k=i+1\) to \(p\) do
                    Update: \(B_{k, j} \leftarrow B_{k, j}-X_{k, i}\left(Y_{k, i}^{T} X_{i, j}\right) Y_{i, j}^{T}\)
            end for
        end for
    end for
```

```
Algorithm 3 Left-looking BLR algorithm: UFCS
Input: a \(p \times p\) block matrix \(B\) of size \(n\),
\(B=\left[B_{i, j}\right]_{i=1: p, j=1: p}\)
    Construct tiles \(B_{i, j}, \forall i, j\)
    for \(i=1\) to \(p\) do
        for \(j=i\) to \(p\) do
            for \(k=1\) to \(i-1\) do
                    Update: \(B_{i, j} \leftarrow B_{i, j}-X_{i, k}\left(Y_{i, k}^{T} X_{k, j}\right) Y_{k, j}^{T}\)
            if \(j \neq i\) then
                    \(B_{j, i} \leftarrow B_{j, i}-X_{j, k}\left(Y_{j, k}^{T} X_{k, i}\right) Y_{k, i}^{T}\)
            end if
            end for
        end for
        Factor: \(B_{i, i}=L_{i, i} U_{i, i}\)
        for \(j=i+1\) to \(p\) do
            Compress: \(B_{i, j} \approx X_{i, j} Y_{i, j}^{T}\)
            Solve: \(B_{i, j} \leftarrow L_{i, i}^{-1} B_{i, j}\)
            Compress: \(B_{j, i} \approx X_{j, i} Y_{j, i}^{T}\)
            Solve: \(B_{j, i} \leftarrow B_{j, i} U_{i, i}^{-1}\)
        end for
    end for
```

These variants are so-called right-looking 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, see Fig. 2a. 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 $L U$ factorization of a dense BLR matrix. Based on the comparative study in [31] the FCSU variant seems the
most promising out of the right-looking versions, since it provides a good balance between efficiency and accuracy. Compressing earlier influences the accuracy negatively, an effect which is quantified in [18].

The right-looking algorithms can be rewritten in a left-looking ${ }^{1}$ form, where at each step $k$, blocks in row $k$ as well as column $k$ are updated using all the blocks already computed (those at its "left"): UFS, UFSC, UFCS, UCFS, CUFS.

In this paper, we focus on the UFCS version, since this version is most promising in terms of timing due to a different memory access pattern [8] as described in [4, 31]. See Algorithm 2 for the right-looking (RL) and Algorithm 3 for the left-looking (LL) implementation and Fig. 2a and Fig. 2b for a comparison between RL and LL.

(a)

(b)

(c)

Fig. 2. First steps of a BLR compression with FCSU/UFCS version. (2a) Right-looking. (2b) Left-looking. (2c) Hybrid with $\operatorname{col}_{\max }=3$.
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 a 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 sub-matrix are written and therefore they are loaded many times (at each step of the factorization), while the low rank blocks of the current panel are read once and

[^0]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, while the low rank blocks of all the previous panels are read, and therefore they are loaded many times during the entire factorization.

```
Algorithm 4 Hybrid BLR algorithm (Column-wise constructed)
Input: a \(p \times p\) block matrix \(B\) of size \(n, B=\left[B_{i, j}\right]_{i=1: p, j=1: p}\)
    for \(b=0\) to \(\left\lceil p / \operatorname{col}_{\text {max }}\right\rceil-1\) do \(\quad \triangleright\) Advance \(c o l_{\text {max }}\) block columns each time
        for \(i=c o l_{\text {max }} b+1\) to \(c o l_{\text {max }}(b+1)\) do \(\quad \triangleright\) Construct \(c o l_{\text {max }}\) 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 \(\quad \triangleright\) See Fig. 3, Subfigures 4-6
            for \(j=1\) to \(\mathrm{col}_{\text {max }} b\) do
                for \(i=c o l_{\text {max }} b+1\) to \(\operatorname{col}_{\text {max }}(b+1)\) 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: \(B_{k, i} \approx B_{k, i}-X_{k, j}\left(Y_{k, j}^{T} X_{j, i}\right) Y_{j, i}^{T}\)
                end for
                end for
            end for
        end if
        for \(i=\operatorname{col}_{\text {max }} b+1\) to \(\operatorname{col}_{\text {max }}(b+1)\) do \(\quad \triangleright\) See Fig. 3, Subfigures 1-3, 7-9
            Factor: \(P_{i} B_{i, i}=L_{i, i} U_{i, i} \quad \triangleright \mathrm{LU}\) with partial pivoting for the diagonal block
            for \(j=i+1\) to \(\operatorname{col}_{\text {max }}(b+1)\) 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 \(c o l_{\max }(b+1)\) do
                for \(k=i+1\) to \(p\) do
                    Update: \(B_{k, j} \approx B_{k, j}-X_{k, i}\left(Y_{k, i}^{T} X_{i, j}\right) Y_{i, j}^{T}\)
                end for
            end for
        end for
    end for
                \(\triangleright\) We assume that \(p\) is a multiple of \(\operatorname{col}_{\text {max }}\) for simplicity, but our code can easily handle any value of \(p\).
```

As visualized in Fig. 2, all BLR tiles of the entire frontal matrix for the LL as well as the 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 sub-matrix. For the LL version, the update task is executed on the immediate neighboring row and column block columns of the trailing sub-matrix.

Based on the observation that both the LL and the 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 the RL version which results in a column-wise construction of the frontal matrices, see Algorithm 4 and Fig. 2c. 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 $l_{\text {max }}$ row tiles, see Algorithm 4 for more details. Algorithm 4 is based on the assumption that $p$ is 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 $\# c o l_{\max }=$ 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 which eventually enables to solve larger problem sizes, see Section 5.
3.2.1 Parallel layout of the hybrid BLR matrix. In our implementation, the hybrid BLR matrix is parallelized with a two-dimensional 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 non-uniform 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 2 D block-cyclic data layout with $c o l_{\max }=P_{c}$. 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.

Fig. 3 presents the algorithmic steps for a parallel hybrid BLR method with twelve 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 $c o l_{\max }=3$ 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 one and all processes of column one. Afterwards the compression and solve steps are executed for all tiles in the first column as well as the two tiles in row one. The update operation as described in Algorithm 4 is executed for all tiles in columns two and three. These four tasks are repeated over and over such that all tiles that have been constructed already, are updated. Afterwards these steps are executed again for the next col $l_{\max }=3$ columns.


- Update
$\square$ used for Update or solve

(4)

(5)


Fig. 3. Hybrid BLR algorithm with 12 processes arranged as $4 \times 3$ process grid and $\operatorname{col}_{\max }=3$.
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 Fig. 2 and Fig. 3 the BLR(RL), BLR(LL) and BLR(Hybrid) algorithms consist of various different execution steps which 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 step, 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, while step 2 consists of two broadcast operations times the number of column/row processes each. These steps are repeated for each row of tiles, i.e.:

$$
\begin{array}{r}
(\lceil n / b\rceil-1) \times 3 \text { bcast_ops, } \\
(\lceil n / b\rceil-1) \times\left(2 \times\left(P_{r}+P_{c}\right)\right) \text { bcast_ops, }
\end{array}
$$

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 step, 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. While the broadcast operations are used for the communication of one tile along columns and one tile along rows, 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, while 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 $i \times\left(P_{r} \times 2+P_{c} \times 2\right)$ send operations, with $i$ current phase of the factorization. These steps are repeated for each row of tiles $i=1:\lceil n / b\rceil-1$, i.e.:

$$
\begin{array}{r}
(\lceil n / b\rceil-1) \times 3 \text { bcast_ops, } \\
\sum_{i=1}^{\lceil n / b\rceil-1}(i \times 4) \text { bcast_ops, } \\
\sum_{i=1}^{\lceil n / b\rceil-1}\left(i \times\left(2 \times\left(P_{r}+P_{c}\right)\right)\right) \text { send_ops. }
\end{array}
$$

The formula indicate that BLR(RL) uses less broadcast operations than BLR(RL). However, BLR(RL) consists of broadcast operations only, while BLR(LL) needs additional send operationswhich 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

Manuscript submitted to ACM
(a) After compression and solve step 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, i.e. 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 step 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.

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

$$
\begin{array}{r}
\left\lceil I / \text { col }_{\max }\right\rceil \times \text { col }_{\max } \times 3 \text { bcast_ops, } \\
\sum_{i=1}^{I}\left(\text { col }_{\max } \times i \times\left(2 \times P_{r}+2 \times P_{c}\right)\right) \text { bcast_ops, }
\end{array}
$$

with $I=\left\lceil\lceil n / b\rceil / c o l_{\max }\right\rceil$.

In summary, we notice additional communication cost for the two BLR variants (LL) and (Hybrid) compared to the $\operatorname{BLR}(\mathrm{RL})$ variant. $\operatorname{BLR}(\mathrm{RL})$ and BLR(Hybrid) consist of broadcast operations only, while BLR(LL) needs additional send operations. The formula indicate that $\operatorname{BLR}(\mathrm{RL})$ 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 Hierarchically Off-Diagonal Butterfly Compression

The HODBF matrix representation [28] is the butterfly extension of the HODLR matrix, i.e., $\mathcal{H}$-matrix with weak admissibility condition [16], that means 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 $\mathcal{T}_{H}=\mathcal{T}_{0}^{0}$ (see Fig. 4).


Fig. 4. Illustration of a 4-level hierarchically off-diagonal butterfly matrix. The root node is at level $l=0$, all the leaf nodes are at level $L=3$. The two largest off-diagonal blocks are approximated using 2-level butterfly matrices. The 4 off-diagonal blocks one level down in the hierarchy are approximated using a 1-level butterfly $\left(U^{1} B^{1} V^{0}\right)$. Finally, the smallest off-diagonal blocks are approximated as 0 -level butterfly matrices.

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, while the diagonal blocks at the leaf level $D_{\tau}=A\left(T_{\tau}^{L}, T_{\tau}^{L}\right)$ for cluster $\tau$ are stored as regular dense matrices (see Fig. 4). The butterfly representation of $L_{b}=L-l$ levels for the block $K=A\left(T_{O}, T_{S}\right)$ reads:

$$
\begin{equation*}
K \approx U^{L_{b}} R^{L_{b}-1} \cdots R^{h} B^{h}\left(W^{h}\right)^{*} \cdots\left(W^{1}\right)^{*}\left(V^{0}\right)^{*} \tag{2}
\end{equation*}
$$

where $h=L_{b} / 2, U^{L_{b}}$ and $V^{0}$ are block diagonal matrices, $R^{L_{b}-1}, \cdots R^{h}, B^{h}, W^{1}, \cdots, W^{h}$ 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 [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 $\tau$ of $\mathcal{T}_{H}$, the given communicator sharing $D_{\tau}$ is split into two sub communicators of similar number of processes. The two sub communicators store and compute $D_{\tau_{1}}, B_{\tau_{1}}$ and $D_{\tau_{2}}, B_{\tau_{2}}$, respectively. Layouts of $B_{\tau_{1}}$ and $B_{\tau_{2}}$ follow those described in [27]. Each of the two sub communicators is further recursively split unless the communicator has only one process that stores and computes $D_{\tau}$. Once constructed following this layout, the HODBF representation can also be inverted following this layout. See more detail on parallelization in [41].

## 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 $n_{\mathrm{Hmin}}$. In addition, BLR compression is only used for fronts corresponding to a separator size smaller than $n_{\mathrm{H} \min }$ and larger than a threshold $n_{\mathrm{Bmin}}$. We have the option to add zfp compression [24] for small fronts, below threshold $n_{\text {Bmin }}$ 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

Manuscript submitted to ACM
reduction in memory consumption, however the solve time increases due to the decompression of zfp fronts during the solve step, see for example Fig. 8c and Fig. 8e.

Typically, the larger fronts are found closer to the root of the multifrontal assembly tree. This is illustrated in Fig. 5 for a small regular $5 \times 5 \times 4$ mesh (Fig. 5a), and Fig. 5c 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.


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 $1\left(T_{0}^{1}\right.$ to $\left.T_{1}^{1}\right)$ and level $2\left(T_{0}^{2}\right.$ to $\left.T_{3}^{2}\right)$ 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 for the next level down; the following two levels in the assembly/frontal tree consists of BLR 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 $F_{\tau}$ 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: $F_{22 ; v_{1}}$ and $F_{22 ; v_{2}}$, where $v_{1}$ and $v_{2}$ are the two children of $\tau$. 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 the HODBF format. Hence, extracting frontal matrix elements requires getting them from fronts previously compressed. There are four different options that we describe in detail in the following subsections, Section 4.1 - Section 4.4: extracting from HODBF to construct HODBF, extracting from BLR to construct HODBF, extracting from LL/RL BLR to construct LL/RL BLR, extracting from hybrid BLR to construct hybrid BLR. In addition to the above mentioned extracting operations, also called extend-add operations, there are extracting operations that we don't 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 dissection), using METIS, of the graph corresponding to $A\left(I_{\tau}^{s}, I_{\tau}^{s}\right)$, where $I_{\tau}^{s}$ 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 $B_{i i}$ are always stored as full-rank matrices ( $\tilde{B}_{i i}=B_{i i}$ ). We also provide an alternative strategy, called strong admissibility, where we use the graph of $A\left(I_{\tau}^{s}, I_{\tau}^{s}\right)$ to determine whether a block is admissible. For BLR with strong admissibility, 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:

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 which 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 [27]. The $F_{11}$ block of $F \equiv F_{\tau}$ is compressed as an HODBF matrix, see [27] for details, which uses the extract_BF routine to extract elements from $F_{11}=A\left(I_{\tau}^{s}, I_{\tau}^{S}\right) \hat{\vartheta} F_{22 ; v_{1}} \hat{\vartheta} F_{22 ; v_{2}}$, 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 [27] for details.
(3) 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 $A\left(I_{\tau}^{\mathrm{s}}, I_{\tau}^{\mathrm{u}}\right) \hat{\downarrow} F_{22 ; v_{1}} \hat{\downarrow} F_{22 ; v_{2}}$ and $A\left(I_{\tau}^{\mathrm{u}}, I_{\tau}^{\mathrm{s}}\right) \hat{\downarrow} F_{22 ; v_{1}} \hat{\downarrow} F_{22 ; v_{2}}$ 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 $\tau$. $F_{22}$ as an HODBF matrix, again using element extraction, now from $F_{22 ; v_{1}} \stackrel{\imath}{\downarrow} F_{22 ; v_{2}}-S$, where $F_{22 ; v_{1}}$ and $F_{22 ; v_{2}}$ are in HODBF form and $S=F_{21} F_{11}^{-1} F_{12}$ is a single butterfly matrix compressed via the randomized algorithm in [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.

```
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: \(x \approx A^{-1} b\)
    \(A \leftarrow P\left(D_{r} A D_{c} Q_{c}\right) P^{\top} \quad \triangleright\) scaling, and permutation for stability and fill reduction
    \(A \leftarrow \hat{P} A \hat{P}^{\top} \quad \triangleright\) rank-reducing separator reordering
    Build assembly tree: define \(I_{\tau}^{\mathrm{s}}\) and \(I_{\tau}^{\mathrm{u}}\) for every frontal matrix \(F_{\tau}\)
    for nodes \(\tau\) in assembly tree in topological order do
        if dimension \(\left(I_{\tau}^{\mathrm{S}}\right)<8\) then
            construct \(F_{\tau}\) as a dense matrix \(\quad \triangleright\) Algorithm 1
        else if dimension \(\left(I_{\tau}^{\mathrm{S}}\right)<n_{\text {Bmin }}\) then
            construct \(F_{\tau}\) as a dense matrix \(\quad \triangleright\) Algorithm 1
            if zfp enabled then
                compress as zfp matrix
            end if
        else if dimension \(\left(I_{\tau}^{s}\right)<n_{\text {Hmin }}\) \&\& RL- or LL-BLR then
            \(F_{\tau} \leftarrow\left[\begin{array}{cc}A\left(I_{\tau}^{\mathrm{s}}, I_{\tau}^{\mathrm{s}}\right) \\ A\left(I_{\tau}^{\mathrm{u}}, I_{\tau}^{\tau}\right)\end{array} \quad A\left(I_{\tau}^{\mathrm{s}}, I_{\tau}^{\mathrm{u}}\right)\right]+F_{22 ; v_{1}}+F_{22 ; v_{2}}\)
            \(P_{\tau} L_{\tau} U_{\tau} \leftarrow F_{11} \quad \triangleright\) LU with partial pivoting
            \(F_{11}, F_{12}, F_{21} \leftarrow \operatorname{BLR}\) compress \(\left(F_{11}, F_{12}, F_{21}\right)\)
            \(F_{12} \leftarrow L_{\tau}^{-1} P_{\tau}^{\top} F_{12}\)
            \(F_{21} \leftarrow F_{21} U_{\tau}^{-1}\)
            \(F_{22} \leftarrow F_{22}-F_{21} F_{12} \quad \triangleright\) Schur update
        else if dimension \(\left(I_{\tau}^{\mathrm{S}}\right)<n_{\text {Hmin }} \& \&\) Hybrid-BLR then
            for columns \(C\) in frontal matrix \(F_{\tau}\) corresponding to \(I_{\tau}^{s}\) do
                \(F_{11 ; C} \leftarrow \operatorname{BLR}\) compress \(\left(A\left(I_{\tau}^{\mathrm{s}_{\tau} \text { partial }}, I_{\tau}^{\mathrm{s} \text { partial }}\right) \not \hat{\downarrow} F_{22 ; v_{1} ; \text { partial }} \hat{\downarrow} F_{22 ; \nu_{2} ; \text { partial }}\right)\)
                \(P_{\tau} L_{\tau} U_{\tau} \leftarrow F_{11 ; C} \quad \triangleright\) LU with partial pivoting
                \(F_{11 ; C}, F_{21 ; C} \leftarrow \operatorname{BLR} \operatorname{compress}\left(F_{11 ; C}, F_{21 ; C}\right)\)
                \(F_{21 ; C} \leftarrow F_{21 ; C} U_{\tau}^{-1}\)
            end for
            for columns \(C\) in frontal matrix \(F_{\tau}\) corresponding to \(I_{\tau}^{\mathrm{u}}\) do
                \(F_{12 ; C} \leftarrow \operatorname{BLR}\) compress \(\left(F_{12 ; C}\right)\)
                \(F_{12 ; C} \leftarrow L_{\tau}^{-1} P_{\tau}^{\top} F_{12 ; C}\)
                \(F_{22 ; C} \leftarrow F_{22 ; C}-F_{21} F_{12} \quad\) Schur update
            end for
        else
            \(F_{11} \leftarrow \operatorname{HODBF}\) compress \(\left(A\left(I_{\tau}^{\mathrm{s}}, I_{\tau}^{\mathrm{s}}\right) \stackrel{\jmath}{\downarrow} F_{22 ; v_{1}} \stackrel{\jmath}{\downarrow} F_{22 ; v_{2}}\right)\)
            \(F_{11}^{-1} \leftarrow\) HODBF invert \(\left(F_{11}\right)\)
            \(F_{12} \leftarrow\) butterfly compress \(\left(A\left(I_{\tau}^{\mathrm{s}}, I_{\tau}^{\mathrm{u}}\right) \hat{\jmath} F_{22 ; v_{1}} \hat{\jmath} F_{22 ; v_{2}}\right)\)
            \(F_{21} \leftarrow\) butterfly compress \(\left(A\left(I_{\tau}^{\mathrm{u}}, I_{\tau}^{\mathrm{S}}\right) \hat{\jmath} F_{22 ; v_{1}} \hat{\jmath} F_{22 ; v_{2}}\right)\)
            \(S \leftarrow\) butterfly compress \(\left(F_{21} F_{11}^{-1} F_{12}\right)\)
            \(F_{22} \leftarrow \operatorname{HODBF}\) compress \(\left(F_{22 ; v_{1}} \hat{\succ} F_{22 ; v_{2}}-S\right) \quad \triangleright\) Schur update
        end if
    end for
    \(x \leftarrow \operatorname{GMRES}\left(A, b, M: u \leftarrow D_{c} Q_{c} P^{\top} \hat{P}^{\top}\right.\) bwd-solve \(\left(\right.\) fwd-solve \(\left.\left.\left(\hat{P} P D_{r} v\right)\right)\right)\)
```


### 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:


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

- For line 32 in Algorithm 5 a different routine to extract elements from $F_{11}=A\left(I_{\tau}^{s}, I_{\tau}^{s}\right) \hat{\downarrow} F_{22 ; \nu_{1}} \hat{\downarrow} F_{22 ; \nu_{2}}$ needs to be executed. For RL- and LL-BLR the extract is straightforward since $F_{22 ; v_{1}}$ and $F_{22 ; v_{2}}$ 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 Hybrid-BLR children $F_{22 ; v_{1}}$ and $F_{22 ; v_{2}}$ are compressed. Hence, more computational effort is needed to extract from compressed BLR tiles.
- 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 $A\left(I_{\tau}^{\mathrm{s}}, I_{\tau}^{\mathrm{u}}\right) \hat{\lessgtr} F_{22 ; v_{1}} \hat{\downarrow} F_{22 ; v_{2}}$ and $A\left(I_{\tau}^{\mathrm{u}}, I_{\tau}^{\mathrm{s}}\right) \hat{\downarrow} F_{22 ; v_{1}} \hat{\downarrow} F_{22 ; v_{2}}$ respectively, where $F_{22 ; v_{1}}$ and $F_{22 ; v_{2}}$ are in compressed form as in Hybrid-BLR or dense as in LL/RL-BLR.
- The final step to construct the contribution block of $\tau, F_{22}$, as an HODBF matrix, uses element extraction, now from $F_{22 ; v_{1}} \hat{\lessgtr} F_{22 ; v_{2}}-S$, where $F_{22 ; v_{1}}$ and $F_{22 ; v_{2}}$ 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:

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 $F_{\tau} \leftarrow\left[\begin{array}{cc}A\left(I_{\tau}^{\mathrm{s}}, I_{\tau}^{\mathrm{s}}\right) & A\left(I_{\tau}^{\mathrm{s}}, I_{\tau}^{\mathrm{u}}\right) \\ A\left(I_{\tau}^{\mathrm{u}}, I_{\tau}^{\mathrm{s}}\right) & 0\end{array}\right] \hat{\succcurlyeq} F_{22 ; v_{1}} \hat{\lessgtr} F_{22 ; v_{2}}$, see line 13 in Algorithm 5. Since $F_{22 ; v_{1}}$ and $F_{22 ; v_{2}}$ are not compressed the extract is straightforward, we simply extract the elements from a dense submatrix using a scatter operation.
(2) Line 14 computes the $L U$ 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) Lines 16 and 17, the off-diagonal blocks $F_{12}$ and $F_{21}$ are updated.

Manuscript submitted to ACM
(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 $F_{22 ; v_{1}}$ and $F_{22 ; v_{2}}$. 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 (hybrid-BLR)

Extracting a BLR frontal matrix to construct a parent BLR matrix in fragments (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:


The tasks to be executed need to be updated as described below:
(1) The extend-add operations to update $F_{\tau} \leftarrow\left[\begin{array}{cc}A\left(I_{\tau}^{\mathrm{s}}, I_{\tau}^{\mathrm{s}}\right) & A\left(I_{\tau}^{\mathrm{s}}, I_{\tau}^{\mathrm{u}}\right) \\ A\left(I_{\tau}^{\mathrm{u}}, I_{\tau}^{\mathrm{s}}\right) & 0\end{array}\right] \hat{\downarrow} F_{22 ; v_{1}} \hat{\imath} \quad F_{22 ; v_{2}}$ requires 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. Eq. (6) visualizes the extend-add operation involving three newly constructed columns of the parent front $F_{\tau}$ and columns of the child fronts $F_{22 ; v_{1}}$ and $F_{22 ; v_{2}}$, highlighted in yellow. $F_{22 ; v_{1}}$ and $F_{22 ; v_{2}}$ are in BLR matrix form, fully constructed and compressed. Note, 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 $F_{22 ; v}$ needed for the extend-add operation are compressed, we first decompress the block columns and then apply a scatter operation to construct $F_{\tau}$.
- If the tiles of $F_{22 ; v}$ needed for the extend-add operation have been decompressed already, we simply apply a scatter operation to construct $F_{\tau}$.
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 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, i.e., 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 $F_{22 ; v_{1}}$ and $F_{22 ; v_{2}}$ 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 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 2133 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 $\left\|u_{i}\right\| \leq 10^{-10}$ or $\left\|u_{i}\right\| /\left\|u_{0}\right\| \leq 10^{-6}$, where $u_{i}=M^{-1}\left(A x_{i}-b\right)$ 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 [3].

### 5.1 Visco-Acoustic Wave Propagation

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

$$
\begin{equation*}
\left(\sum_{i} \rho(\mathbf{x}) \frac{\partial}{\partial x_{i}} \frac{1}{\rho(\mathbf{x})} \frac{\partial}{\partial x_{i}}\right) p(\mathbf{x})+\frac{\omega^{2}}{\kappa^{2}(\mathbf{x})} p(\mathbf{x})=-f(\mathbf{x}) \tag{7}
\end{equation*}
$$

Here $\mathbf{x}=\left(x_{1}, x_{2}, x_{3}\right), \rho(\mathbf{x})$ is the mass density, $f(\mathbf{x})$ is the acoustic excitation, $p(\mathbf{x})$ is the pressure wave field, $\omega$ is the angular frequency, $\kappa(\mathbf{x})=v(\mathbf{x})(1-i /(2 q(\mathbf{x})))$ is the complex bulk modulus with the velocity $v(\mathbf{x})$ and quality factor $q(\mathbf{x})$. We solve Eq. (7) by a finite-difference discretization on staggered grids using a 27-point stencil and 8 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 Eq. (7).

We consider a cubed domain with $v(\mathbf{x})=4000 \mathrm{~m} / \mathrm{s}, \rho(\mathbf{x})=1 \mathrm{~kg} / \mathrm{m}^{3}, q(\mathbf{x})=10^{4}$. The frequency is set to $\omega=8 \pi \mathrm{~Hz}$ 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, see Fig. 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 the BLR multifrontal solver outperform the exact solver in terms of factor flops, factor time, factor nonzeros and peak memory, see Fig. 6. The iteration counts are shown in Fig. 6e for BLR(Hybrid), see Table 1 for iteration counts Manuscript submitted to ACM


Fig. 6. Results for high frequency 3D Helmholtz using the BLR $\left(10^{-3}\right)$ 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 4 MPI ranks per node and 8 OpenMP threads per MPI process.
of other variants. The $\operatorname{BLR}(R L)$ 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, while BLR(LL) needs additional send operations. The formula in Section 3.2.2 indicate that BLR(RL) uses the least amount of broadcast operations and BLR(Hybrid) uses significantly more than the other two variants, see Fig. 7a, which is due to additional broadcast operations needed for each newly constructed set of columns. Fig. 7b 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 to solve larger problem sizes. BLR(RL) and BLR(LL) run out of memory when solving Eq. (7) with sizes larger than $320^{3}$. Among all three variants, $\operatorname{BLR}($ RL ) requires the least amount of communication and is the fastest, but it requires the largest amount of memory.

| 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 |  |  |  |  |

Table 1. Iteration counts corresponding to results in Fig. 6
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, see Fig. 8. For the BLR multifrontal solver,


Fig. 7. Amount of communication for the BLR(LL), BLR(RL) and BLR(Hybrid). (a) Number of broadcast operations. (b) Number of send operations. $\operatorname{BLR}(R L)$ and $\operatorname{BLR}($ Hybrid ) require zero send operations, see Section 3.2.2.
we use the $\operatorname{BLR}$ (Hybrid) variant which allows to solve problems of size up to $420^{3}$. We set the tolerance $\varepsilon=10^{-2}$ for HODBF and $\varepsilon=10^{-3}$ for BLR, respectively. The zfp compressed fronts use 16 bitplanes in HODBF_BLR_ZFP.

For HODBF compression, all fronts corresponding to separators with sizes $n_{\mathrm{Hmin}} \geq 7 \mathrm{~K}$ are compressed. For HODBF_BLR_ZFP and HODBF_BLR compression, all fronts corresponding to separators with sizes $n_{H m i n} \geq 15 \mathrm{~K}$ are compressed with HODBF, all fronts corresponding to separators $200 \leq n_{B m i n} \leq 15 \mathrm{~K}$ 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 relative error was $\|x-\tilde{x}\|_{2} /\|\tilde{x}\|_{2}<10^{-5}$. Compared to the $O\left(N^{2}\right)$ computation and $O\left(N^{4 / 3}\right)$ memory complexities using the exact multifrontal solver and the BLR multifrontal solver, we observe the predicted $O\left(N \log ^{2} N\right)$ computation and $O(N)$ memory complexities (see Fig. 8a-Fig. 8e) for the HODBF multifrontal solver variants. The iteration counts are shown in Fig. 8e 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 $\operatorname{BLR}$ (Hybrid) multifrontal solver allows to solve larger problem sizes up to $420^{3}$ with a low solve time.

| 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 |  |

Table 2. Iteration counts corresponding to results in Fig. 8

### 5.2 Singularly Perturbed PDE

Next, we consider the following three-dimensional singularly perturbed reaction-diffusion differential equation (SPDE) that arises in fluid dynamics, computational chemistry, and biological applications (see, for instance, [40]):

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

where the perturbation parameter $\delta$ is small and positive, $g$ and $f$ are some given functions. Solving large sparse systems arising from finite difference discretizations, even for the two-dimensional analogue of (8), is a challenging task. For example, [30] showed that standard Cholesky-based solvers exhibit poor performance when $\delta$ is small. The underlying Manuscript submitted to ACM


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 datapoint (additional data in Table 2). (f) Peak working memory. We use 64 compute nodes, with 4 MPI ranks per node and 8 OpenMP threads per MPI process.
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 [34].

Here, we set $\delta=10^{-4}$ and solve Eq. (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, see Fig. 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 $n_{H \min } \geq 5 \mathrm{~K}$ are compressed with HODBF, all fronts corresponding to separator sizes $200 \leq n_{\mathrm{Bmin}} \leq 5 \mathrm{~K}$ 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 $O\left(N \log ^{2} N\right)$ computation and $O(N)$ memory complexities. Adding zfp on top of BLR and HODBF can slightly improve the total factor memory, but can increase the solve time per iteration.


Fig. 9. Results for 3D singularly perturbed differential equations (SPDE) using the BLR, BLR-ZFP, BLR-HODBF and BLR-HODBF-ZFP $\left(10^{-2}\right)$ 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 2 MPI ranks per node and 16 OpenMP threads per MPI process.

### 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 $M x=b$, or

$$
\left[\begin{array}{cc}
A & G  \tag{9}\\
-D & 0
\end{array}\right]\left[\begin{array}{l}
x_{u} \\
x_{p}
\end{array}\right]=\left[\begin{array}{l}
b_{u} \\
b_{p}
\end{array}\right]
$$

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 ( 3 spatial components) and pressure respectively.

Since the pressure is only defined up to a constant, the matrix $M$ is singular. The nullspace is $Z=k^{-3 / 2}[0 \ldots 01 \ldots 1]$, with the last $k^{3}$ elements of $Z$ corresponding to the pressure degrees of freedom. We construct an exact solution
Manuscript submitted to ACM


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 4 MPI ranks per node and 8 OpenMP threads per MPI process.
$\tilde{x}=x_{\mathrm{r}}-Z Z^{T} x_{\mathrm{r}}$ with $x_{\mathrm{r}}$ 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 $x \leftarrow x-Z Z^{T} x$ 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 Fig. 10 with $k$ varying from 100 to 250 . For BLR, fronts corresponding to separator sizes $n_{\text {Bmin }}$ above 1000 are compressed with a relative compression tolerance $10^{-6}$ and all fronts corresponding to separators sizes below 1000 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 a (non-symmetric) 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 $\sqrt{\varepsilon_{\text {mach }}}\|M\|_{1}$. 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 $\tau \sqrt{\varepsilon_{\text {mach }}}\|M\|_{1}$ before starting the numerical factorization. Since we also apply a matrix equilibration similar to LAPACK's dgeequ/dlaqge, $\|M\|_{1}=1$. The factor $\tau$ needs to be chosen carefully. A larger $\tau$
reduces the condition number of the preconditioner and allows for better compression. However, a larger $\tau$ 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 then also used with BiCGStab (instead of iterative refinement). For all experiments, the relative error was $\|x-\tilde{x}\|_{2} /\|\tilde{x}\|_{2}<10^{-5}$. Fig. 10 shows a lower peak memory consumption when using BLR(Hybrid)_ZFP which allows to solve larger problem sizes. However, BLR(RL) and BLR(RL)_ZFP outperform BLR(Hybrid)_ZFP in terms of factor and solve time.

### 5.4 SuiteSparse Matrix Collection

|  |  |  | no compr. |  |  | BLR(RL, $10^{-2}$ ) compression |  |  |  |  |  |  |  | BLR(RL, $10^{-2}$ )_ZFP(32) compression |  |  |  |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  |  | fact | sol |  |  |  |  |  |  |  |  |  |  | onts |  | fact |  |  |  |  |  |
| matr | $\times 10^{3}$ |  |  | (s) | its | dens | R | (s) | (s) | its | (\%) | (\%) | err. | dense | ZFP | BLR | (s) | (s) | its | (\%) | (\%) | err. |
| boneS01 | 127 | 6,715 | 0.45 | 0.0 | 1 | 15 K | 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 |  | 3,866 | 0.49 | 0.02 | 1 | K | 8 | 0.47 | 0.23 | 12 | 87.4 | 102.0 | $9 \mathrm{e}-07$ | 3 K | 5K | 8 | 0.56 | 4.49 | 12 | 50.2 | 76.0 | $9 \mathrm{e}-07$ |
| scircuit |  | 959 | 0.12 | 0.01 | 1 | 1K | 0 | 0.12 | 0.01 |  | 100.0 | 2.0 | 1e-10 | 12K | 8K | 0 | 0.12 | 0.11 | 4 | 53.7 | 47.0 | $2 \mathrm{e}-08$ |
|  | 217 | , | 0.27 | 0.02 | 1 | 27K | 1 | 27 | 0.06 | 3 | . 7 | 9.0 | 6e-08 | 20K | 6K | 1 | 33 | 1.06 | 3 | 5.8 | 70.0 | 1e-06 |
| tor | 259 | 4,632 | 0.81 | 0.0 | 1 | 32K | 14 | 0.59 | 0.10 | 2 | 72.1 | 100.0 | $2 \mathrm{e}-07$ | 25 K | 7 K | 14 | 0.70 | 1.66 | 2 | 37. | 70.0 | $2 \mathrm{e}-07$ |
| cag |  | 7,479 | 75.11 | 0.2 | 1 | 55K | 56 | 28.74 | 0.53 | 2 | 24.1 | 54.0 | 6e-07 | 46K | 8K | 56 | 31.6 | 8.5 | 2 | 17.8 | 50.0 | 6e-07 |
| audikw | 943 | 77,651 | 12 | 0.42 | 1 | K | 124 | 60 | 4.90 | 48 | 47.7 | . 0 | 5e-04 | 94K | 22 K | 124 | 7.04 | 141.04 | 58 | 29.2 | 37.0 | 1e-03 |
| atmosmo | 1,270 | 8,814 | 82 | 0.12 | 1 | 158K | 63 | 60 | 1.03 | 8 | 44.5 | 4.0 | 5e-07 | 106K | 52K | 63 | 5.94 | 15.32 | 8 | 29.3 | 51.0 | 5e-07 |
| Ser | 1,391 | 64,531 | 42 | 0.36 | 1 | 173K | 134 | 18.66 | 1.87 | 10 | 34 | .0 | 2e-05 | 37 | 36K | 134 | 19.01 | 55.42 | 10 | 22.3 | 39.0 | $2 \mathrm{e}-02$ |
| Geo_1 | 1,437 | 63,156 | 26.76 | 0.23 | 1 | K | 177 | 12.76 | 2.01 | 13 | 45.4 | 47.0 | 8e-05 | 42 | 36K | 177 | 13.39 | 41.93 | 12 | 30.1 | 36.0 | 1e-03 |
| atmosmod | 1,48 | 10,319 | 18 | 0.13 | 1 | 186K | 64 | 40 |  | 4 | 45.8 | 4.0 | -07 | 126K | 59 K | 64 | 5.85 | 10.36 | 4 | 27.4 | 52.0 | 9 e |
| Hook_1498 | 1,4 | 60,917 | 15.00 | 0.22 | 1 | 187K | 135 | 81 | 4.81 | 34 | 46.9 | . 0 | $4 \mathrm{e}-05$ | 137K | K | 135 | 8.3 | 139.68 | 34 | 27 | 31.0 | 4e-05 |
| ML_Geer | 1,50 | 110,879 | 79 | 0.14 | 1 | 187K | 115 | 2.97 | 2.3 | 24 | 65.4 | 40.0 | $2 \mathrm{e}-05$ | 144 | 43K | 115 | 3.79 | 84.81 | 23 | 33.5 | 24.0 | $2 \mathrm{e}-0$ |
| Transport | 1,60 | 23,500 | 8.76 | 0.18 | 1 | 0K | 128 | 4.69 | 3.01 | 26 | 51.7 | 60.0 | 1e-05 | 152 | 47 K | 128 | 5.41 | 81.27 | 26 | 30 | 42. | 7e-06 |
| memchip | 2,70 | 15,950 | 0.3 | 0.13 | 1 | 338K | 0 | 0.35 | 0. | 1 | 100.0 | 100.0 | $7 \mathrm{e}-1$ | 204 K | 134 | 0 | 0.4 | 0.68 | 1 | 48.3 | 51.0 | 2 e |

Table 3. Results for the numerical factorization and solve for a number of matrices from the SuiteSparse matrix collection. Compression ratio (comp \%) refers to the size of the final LU factors relative to the exact solver without compression, while 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.

The problems shown so far where all defined on a regular three-dimensional 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., $n_{\mathrm{Bmin}}=512$, $n_{\text {Lmin }}=8$.

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 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 scircuit and memchip systems, using BLR only does not give any compression, whereas zfp compresses the factors by at least $2 \times$. 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 $n_{\mathrm{Bmin}}=512$. 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.
Manuscript submitted to ACM

## 6 CONCLUSION

This paper presents a fast and approximate multifrontal solver for large sparse linear systems. The solver leverages the Hierarchically Off-Diagonal Butterfly compression, HODBF, a reduced-memory version of the non-hierarchical Block Low-Rank format, BLR, and lossy compression. Depending on the application as well as problem sizes, we make 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 which allows to solve larger problem sizes. The resulting solver can attain the $O\left(N \log ^{2} N\right)$ computation and $O(N)$ memory complexities when adding the HODBF compression. Some of the presented results for smaller problem sizes do not make use of the HODBF compression since 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 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 ${ }^{3}$, lossy compression is provided with the software package zfp ${ }^{4}$.

## ACKNOWLEDGMENTS

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 No. 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 No. DE-AC02-05CH11231.

## REFERENCES

[1] Sivaram Ambikasaran and Eric Darve. 2013. An $O(N \log N)$ fast direct solver for partial hierarchically semi-separable matrices. SIAM 7. Sci. Comput 57, 3 (Dec. 2013), 477-501.
[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 7. 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. Software 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, Article 2 (Feb. 2019), 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 HeavyWeight Perfect Matching on Bipartite Graphs. SIAM fournal on Scientific Computing 42, 4 (2020), C143-C168. https://doi.org/10.1137/18M1189348
[7] Satish Balay, Shrirang Abhyankar, Mark Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, W Gropp, et al. 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, United States. https://doi.org/10.1137/1.9780898719611 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9780898719611
[9] Pieter Ghysels. 2014. STRUMPACK: STRUctured Matrix PACKage. http://portal.nersc.gov/project/sparse/strumpack/.

[^1][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 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, Orlando, FL, USA, 897-906.
[11] G.Karypis and V.Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM fournal on Scientific Computing 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. Journal of computational physics 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 f. 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 7 MATRIX ANAL A. 20, 4 (1999), 889-901.
[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 7. 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 Transactions on Visualization and Computer Graphics 20 (08 2014). https://doi.org/10.1109/TVCG.2014.2346458
[24] Peter Lindstrom. 2014. zfp: Compressed Floating-Point and Integer Arrays. 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. 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 fournal on Scientific Computing 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 fournal on Scientific Computing 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 7. 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. F. 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, 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 (2019), arXiv:1907.13442.
[36] Stéphane Operto, Jean Virieux, Patrick Amestoy, Jean-Yves L’Excellent, Luc Giraud, and Hafedh Ben Hadj Ali. 2007. 3D finite-difference frequencydomain 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, United States. https://doi.org/10.1137/1.9780898718072 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9780898718072
[38] Qiyuan Pang, Kenneth L. Ho, and Haizhao Yang. 2020. Interpolative decomposition butterfly factorization. SIAM 7. Sci. Comput. 42, 2 (2020), A1097-A1115. https://doi.org/10.1137/19M1294873
Manuscript submitted to ACM
[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 Heidelberg, Berlin, Heidelberg, 370-378. https://doi.org/10.1007/BFb0031609
[40] H.-G. Roos, M. Stynes, and L. Tobiska. 2008. Robust numerical methods for singularly perturbed differential equations (second ed.). Springer Series in Computational Mathematics, Vol. 24. Springer-Verlag, Berlin. xiv+604 pages.
[41] 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 Transactions on Antennas and Propagation 70, 5 (2022), 3549-3559. https://doi.org/10. 1109/TAP.2021.3137193
[42] 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.
[43] Raf Vandebril, Marc Van Barel, Gene Golub, and Nicola Mastronardi. 2005. A bibliography on semiseparable matrices. Calcolo 42, 3-4 (2005), 249-270.


[^0]:    ${ }^{1}$ This algorithmic variant is also called "left-up-looking" in the literature, for brevity we use "left-looking" throughout the paper.
    Manuscript submitted to ACM

[^1]:    ${ }^{2}$ https://github.com/pghysels/STRUMPACK
    ${ }^{3}$ https://github.com/liuyangzhuan/ButterflyPACK
    ${ }^{4}$ https://github.com/LLNL/zfp

