COMMUNICATION AVOIDING BLOCK LOW-RANK PARALLEL MULTIFRONTAL TRIANGULAR SOLVE WITH MANY RIGHT-HAND SIDES*

PATRICK AMESTOY†, OLIVIER BOITEAU‡, ALFREDO BUTTARI§, MATTHIEU GEREST‡, FABIENNE JÉZÉQUEL¶, JEAN-YVES L'EXCELLENT†, AND THEO MARY‖

Abstract. Block low-rank (BLR) compression can significantly reduce the memory and time costs of parallel sparse direct solvers. In this paper, we investigate the performance of the BLR triangular solve phase, which we observe to be underwhelming when dealing with many right-hand sides (RHS). We explain that this is because the bottleneck of the triangular solve is not in accessing the BLR LU factors, but rather in accessing the RHS, which are uncompressed. Motivated by this finding, we propose several new hybrid variants, which combine the right-looking and left-looking communication patterns to minimize the number of accesses to the RHS. We confirm via a theoretical analysis that these new variants can significantly reduce the total communication volume. We assess the impact of this reduction on the time performance on a range of real-life applications using the MUMPS solver, obtaining up to 20% time reduction.

Key words. numerical linear algebra, block low-rank matrices, data sparse matrices, LU factorization, triangular solve, linear systems, low-rank approximations, communication avoiding algorithms

AMS subject classifications. 65F05, 65F08, 65F10, 65F50, 65F55, 65Y05, 68W10, 68W25

1. Introduction.

We consider the solution of a large sparse system of linear equations

$$\mathcal{A}x = b$$

by means of direct solvers, such as the multifrontal method [8]. Direct methods first compute a sparse LU factorization of the matrix, $\mathcal{A} = \mathcal{L}\mathcal{U}$, and then directly solve the system by two triangular solves, $\mathcal{L}y = b$ and $\mathcal{U}x = y$. In many applications, the matrix $\mathcal{A}$ and its LU factors possess a block low-rank (BLR) structure [1]: we can permute them and partition them in blocks in such a way that most of the off-diagonal blocks can be accurately approximated by low-rank matrices. This BLR structure can be exploited to reduce the computational complexity of direct solvers [5] and therefore reduce their time and memory consumption significantly. The BLR format is notably used in the MUMPS [7, 6], PaStiX [10, 16, 15], and STRUMPACK [17, 9] solvers.

This paper is concerned with the performance of the triangular solve phase, which is critical in several contexts. Indeed, the BLR factorization is often used as a preconditioner for iterative solvers [3, 11], which require several iterations and thus solves. Moreover, even in a pure direct solver context, some of the real-life applications where BLR solvers have been the most successful are in the field of geosciences [14, 4, 18, 13], where we need to solve a system

$$\mathcal{A}\mathcal{X} = \mathcal{B}$$

with many right-hand sides (RHS): $\mathcal{B} \in \mathbb{R}^{n \times n_{\text{rhs}}}$ is a (possibly sparse) matrix with n_{rhs} columns,

*Version of April 25, 2023.

†Mumps Technologies, ENS Lyon, 46 Allée d'Italie, F-69007 Lyon, France (patrick.amestoy@mumps-tech.com; jean-yves.l.excellent@mumps-tech.com)

‡EDF R&D (olivier.boiteau@edf.fr; matthieu.gerest@edf.fr)

§CNRS, IRIT, 2 Rue Charles Camichel, F-31071 Toulouse, France (alfredo.buttari@irit.fr)

¶Sorbonne Université, CNRS, LIP6, and Université Paris-Panthéon-Assas, Paris, F-75005, France (fabienne.jezequel@lip6.fr)

‖Sorbonne Université, CNRS, LIP6, Paris, F-75005, France (theo.mary@lip6.fr)

where n_{rhs} is typically in the thousands or tens of thousands. In this case, the triangular solves $\mathcal{LY} = \mathcal{B}$ and $\mathcal{UX} = \mathcal{Y}$ are the bottleneck of the computation.

This work started from the observation that the performance of the BLR solve with multiple RHS is underwhelming, compared with the single RHS case. This is illustrated in Table 1.1, which provides the time for the forward solve $\mathcal{LY} = \mathcal{B}$ both for the BLR and dense (no compression) solvers, depending on the number of RHS, for a few problems of interest. For a single RHS ($n_{\text{rhs}} = 1$), the BLR compression reduces the solve time by significant factors for all problems. However, with multiple RHS ($n_{\text{rhs}} = 250$), the speedup achieved thanks to BLR compression becomes much smaller for all problems (for example, for the Poisson120 problem, the 3.9× speedup with $n_{\text{rhs}} = 1$ becomes only a 1.7× speedup with $n_{\text{rhs}} = 250$).

Table 1.1: Some motivating examples: forward solve time (s) for the dense (no compression) and BLR solvers on 2×18 cores.

	$n_{\text{rhs}} = 1$			$n_{\text{rhs}} = 250$		
Matrix	Dense	BLR	Ratio	Dense	BLR	Ratio
Poisson120	0.24	0.06	3.9×	1.37	0.78	1.7×
Geoazur100	0.23	0.10	2.4×	2.31	1.95	1.2×
atmosmodl	0.14	0.06	2.3×	0.67	0.52	1.3×
Geo_1438	0.19	0.12	1.6×	1.51	1.35	1.1×
Queen_4147	1.69	0.38	4.5×	11.13	5.90	1.9×
Serena	0.20	0.12	1.6×	1.81	1.23	1.5×
Transport	0.15	0.06	2.6×	0.77	0.73	1.0×

Therefore this motivated us to rethink the BLR solve algorithms. The key observation is that with BLR compression, the performance of the triangular solve is memory bound, even for multiple RHS. Therefore, the performance of the BLR solve is mainly determined by its communication costs. Crucially, while BLR compression reduces the size of the LU factors and therefore the cost of accessing them, it does not reduce the size of the RHS, which are uncompressed. The consequence is that when there are many RHS, the cost of accessing them is likely to dominate the cost of accessing the LU factors, thereby reducing (or even cancelling) the performance benefits of the BLR compression.

The main contribution of this paper is to overcome this limitation by proposing new hybrid algorithms that reduce the number of accesses to the RHS and therefore the total volume of communications.

The rest of this paper is organized as follows. After recalling some preliminaries on the existing BLR solve variants in section 2, we describe in section 3 several novel variants of the BLR solve that reduce its communication costs. We confirm that these new variants are indeed communication-avoiding by performing a theoretical communication volume analysis in section 4. To quickly analyze the performance of these new variants and assess their potential, we first develop a simplified prototype code and present our results on synthetic data in section 5. Based on these results, we implement a selected subset of the most promising variants in the MUMPS solver and test their performance on a range of real-life applications in section 6. Finally, we provide our concluding remarks in section 7.

Throughout the paper, we will discuss the case of the forward solve $\mathcal{LY} = \mathcal{B}$ without loss

of generality. The algorithms and ideas proposed in this paper also apply to the backward solve $\mathcal{U}\mathcal{X} = \mathcal{Y}$. For clarity of notation, we rename the forward solve $\mathcal{L}\mathcal{X} = \mathcal{B}$ hereinafter. In our experiments with the MUMPS solver, we will measure the time spent in the computations for the forward solve. The entire triangular solution phase of the solver also consists of the backward solve and some additional non-computational parts (data copies, etc.), whose cost represents a fixed overhead that is independent of the algorithm variants considered in this paper.

2. Preliminaries and notations.

2.1. Notations. With the multifrontal method, the forward solve $\mathcal{L}\mathcal{X} = \mathcal{B}$ amounts to a bottom-up traversal of a tree whose nodes are associated with the frontal matrices. The solution $\mathcal{X}$ of the global sparse problem is initialized to the right-hand side $\mathcal{B}$. Then, at each node, a partial forward elimination is performed with the corresponding frontal matrix. To be specific, let $L \in \mathbb{R}^{m \times n}$ be such a frontal matrix, with $m \geq n$, and let $X \in \mathbb{R}^{m \times n_{\text{rhs}}}$ be the rows of the solution $\mathcal{X}$ associated with the row variables of L. We denote as L_{fs} and X_{fs} the top $n \times n$ subparts of L and X, respectively, and as L_{cb} and X_{cb} their bottom $(m-n) \times n$ subparts. L_{fs} corresponds to the so called "fully-summed" (FS) variables of the frontal matrix; these variables are ready to be eliminated by computing $X_{\text{fs}} \leftarrow L_{\text{fs}}^{-1} X_{\text{fs}}$, which yields the final form of the solution X_{fs}. L_{cb} corresponds to the so-called "contribution block" (CB) variables of the frontal matrix, which are not ready to be eliminated; for these, the solution is merely updated as $X_{\text{cb}} \leftarrow X_{\text{cb}} - L_{\text{cb}} X_{\text{fs}}$. X_{cb} is not the final form of the solution, it will be further updated by variables from other fronts, until the fronts that have X_{cb} as fully-summed variables are reached.

When using BLR compression, the frontal matrix L is partitioned in $p \times p_{\text{fs}}$ blocks $L_{ij} \in \mathbb{R}^{b \times b}$, with $p = m/b$, $p_{\text{fs}} = n/b$, and where b denotes the block size (we assume for simplicity of notation that it is the same for all blocks). The solution X is also partitioned into p blocks $X_i \in \mathbb{R}^{b \times n_{\text{rhs}}}$. L_{fs} is partitioned into $p_{\text{fs}} \times p_{\text{fs}}$ blocks and L_{cb} is partitioned into $p_{\text{cb}} \times p_{\text{fs}}$ blocks, where $p_{\text{cb}} = p - p_{\text{fs}}$. The blocks L_{ij} that are low-rank are approximated as $L_{ij} \approx U_{ij} V_{ij}^T$, with $U_{ij}, V_{ij} \in \mathbb{R}^{b \times r}$, where r denotes the rank of the blocks (we again assume for simplicity of notation that it is the same for all blocks).

We summarize below the notations used for a given front L and its corresponding part of the solution X, which are illustrated in Figure 2.1.

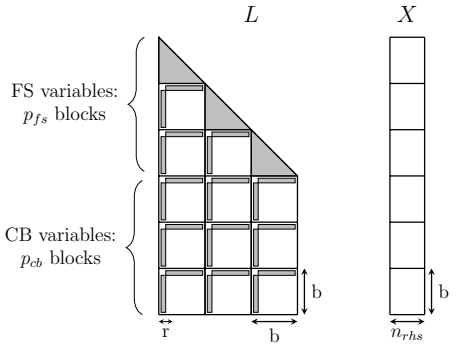


Fig. 2.1: A frontal BLR matrix L and its right-hand side X

- $\mathcal{A}$, $\mathcal{X}$, $\mathcal{B}$, $\mathcal{L}, \mathcal{U}$: the matrix, solution, RHS, and LU factors associated with the global sparse problem;
- A, X, B, L, U, the matrix, solution, RHS, and LU factors associated with a given dense frontal matrix;
- m, the number of rows of L and X;
- n, the number of columns of L;
- n_{rhs}, the number of columns of X (the number of right-hand sides);
- b, the block size;
- r, the rank of the low-rank blocks;
- $L_{ij} \in \mathbb{R}^{b\times b}$, the (i,j)th block of L, and $U_{ij}V_{ij}^T$, its low-rank representation;
- $X_i \in \mathbb{R}^{b\times n_{\text{rhs}}}$, the ith block of X;
- $p_{\text{fs}} = n/b$: the number of block rows in the FS part, also the number of block columns;
- $p_{\text{cb}} = (m-n)/b$: the number of block rows in the CB part;
- $p = p_{\text{fs}} + p_{\text{cb}} = m/b$, the total number of block rows.
- We also define $q = p_{\text{fs}}(p_{\text{fs}}-1)/2 + p_{\text{fs}}p_{\text{cb}}$, the total number of off-diagonal blocks in L;
- $FS = 1\colon p_{\text{fs}}$, the set of block indices for FS variables; and
- $CB = p_{\text{fs}}+1\colon p$, the set of block indices for CB variables.

With the block partitioning defined above, the FS elimination $X_{\text{fs}} \leftarrow L_{\text{fs}}^{-1}X_{\text{fs}}$ leads to the recurrence relation

$$X_i \leftarrow L_{ii}^{-1}(X_i - \sum_{j<i} L_{ij}X_j) \tag{2.1}$$

for $i \in FS$. The CB update $X_{\text{cb}} \leftarrow X_{\text{cb}} - L_{\text{cb}}X_{\text{fs}}$ takes the form

$$X_i \leftarrow X_i - \sum_{j\le p_{\text{fs}}} L_{ij}X_j \tag{2.2}$$

for $i \in CB$. The BLR representation of L is exploited by computing $L_{ij}X_j$ in the above expressions as $U_{ij}(V_{ij}^TX_j)$.

Computations (2.1) and (2.2) thus involve two types of tasks:

- update(i,j) for $i \in FS \cup CB$ and $j \in FS$ verifying $i > j$: $X_i \leftarrow X_i - U_{ij}(V_{ij}^TX_j)$, which can be performed only after trsm(j) has been completed; and
- trsm(i) for $i \in FS$: $X_i \leftarrow L_{ii}^{-1}X_i$, which can be performed only after update$(i,1)$, …, update$(i,i-1)$ have all been completed.

There are therefore some dependencies between tasks but also some independent tasks which can be performed in any order, possibly concurrently. We next describe two variants using different orders.

2.2. Right-looking and left-looking variants. Algorithm 2.1 describes the right-looking (RL) variant, which performs the update(i,j) tasks as soon as they are ready to be performed (eager approach): as soon as trsm(j) has been completed, all the update(i,j) for $i > j$ are immediately performed, as illustrated in Figure 2.2.

Conversely, Algorithm 2.2 describes the left-looking (LL) variant, which performs the update(i,j) tasks as late as possible (lazy approach): for a given i, the update(i,j) are delayed until they are all ready to be performed together, as illustrated in Figure 2.3.

The RL and LL variants perform the same computations in two different orders. In a sequential context, it is not clear whether one variant can be expected to yield better performance than the other. However, in a parallel context, a major difference appears. The RL variant can be

Algorithm 2.1 Right-looking variant.

1: **for** $j \in FS$ **do** ▷ Sequential loop
2: $X_j \leftarrow L_{jj}^{-1} X_j$ ▷ trsm(j)
3: **for** $i > j$ **do** ▷ Parallel loop
4: $X_i \leftarrow X_i - U_{ij}(V_{ij}^T X_j)$ ▷ update(i, j)
5: **end for**
6: **end for**

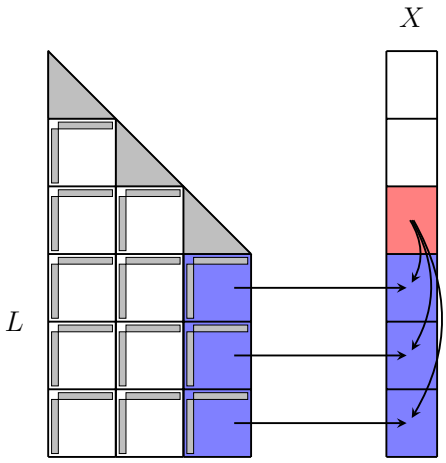
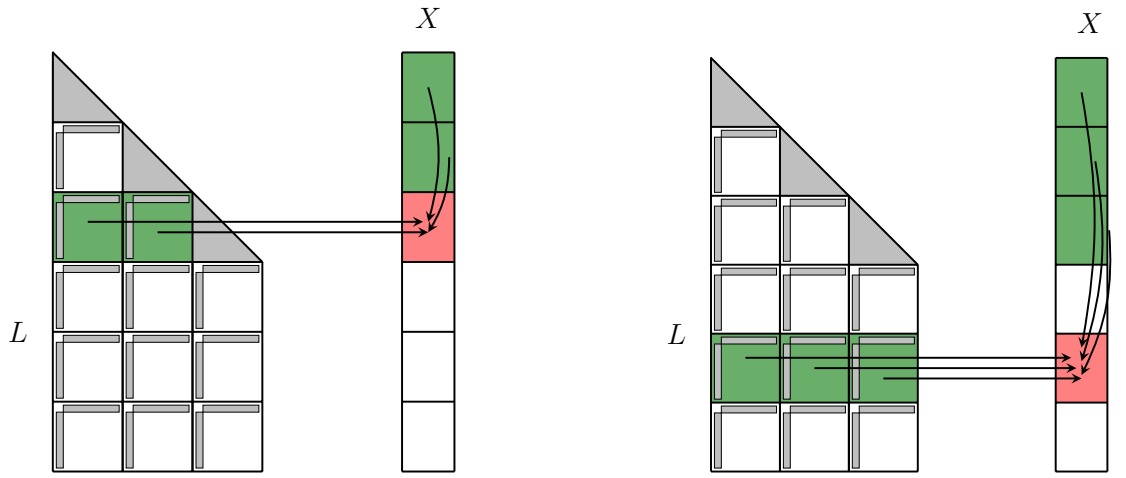


Fig. 2.2: Step $j = 3$ of Algorithm 2.1 (right-looking variant).

Algorithm 2.2 Left-looking variant.

1: **for** $i \in FS$ **do** ▷ Sequential loop
2: **for** $j < i$ **do** ▷ Parallel loop
3: $X_i \leftarrow X_i - U_{ij}(V_{ij}^T X_j)$ ▷ update(i, j)
4: **end for**
5: $X_i \leftarrow L_{ii}^{-1} X_i$ ▷ trsm(i)
6: **end for**
7: **for** $i \in CB$ **do** ▷ Parallel loop
8: **for** $j \in FS$ **do** ▷ Sequential loop
9: $X_i \leftarrow X_i - U_{ij}(V_{ij}^T X_j)$ ▷ update(i, j)
10: **end for**
11: **end for**

parallelized efficiently by executing the loop on block rows (line 3 of Algorithm 2.1) in parallel, since all update(i, j) tasks are independent for a fixed j. This approach is usually efficient because it does not lead to any conflict and the size of the loop, $p - j$, is large enough to expose a high amount of concurrency. In contrast, the parallelization of the LL variant is more difficult. The CB part of the update can be efficiently parallelized by executing the loop on the block-rows (line 7 of



(a) Step $i = 3$ of Algorithm 2.2: Update of an FS block.

(b) Step $i = 5$ of Algorithm 2.2: Update of a CB block.

Fig. 2.3: Left-looking variant.

Algorithm 2.2) in parallel, since the updates for different block-rows are independent and the size of the loop, p_{cb}, is large enough. However, for the FS part of the computation, the only loop that can be executed in parallel is the loop on the block columns (line 2 of Algorithm 2.2), which presents two difficulties. First, it requires a reduction operation to avoid conflicts since all update(i, j) tasks for a fixed i modify the same block X_i. Second, the loop is only of size $i - 1$, with $i \leq p_{\text{fs}}$, so that there is very little concurrency in the first steps of the loop, and even for the later steps because p_{fs} is typically much smaller than p_{cb}.

2.3. Parallelism in multifrontal solve. As mentioned above, in the multifrontal solution, we must carry out several partial solves with frontal matrices following a bottom-up traversal of a tree. As a result, two types of parallelism can be exploited.

- Node parallelism consists in processing a given front in parallel, by parallelizing the partial solve as described above for the RL and LL variants. The amount of work required by one partial solve is usually sufficient to be efficiently parallelized only for the largest fronts, which are at the top of the tree.
- Tree parallelism consists in processing multiple fronts on different branches concurrently, using only one process per front. This allows for efficiently parallelizing the bottom of the tree, which consists of many independent fronts of small size.

Since node and tree parallelism are more efficient for the top and bottom of the tree, respectively, the best approach is to combine both types of parallelism. One possibility to do so is to exploit tree parallelism for the bottom layers of the tree, and switch to node parallelism after a given layer (so-called the "$\mathcal{L}_0$" layer) is reached. This $\mathcal{L}_0$ approach is implemented in MUMPS [12].

With this approach, the frontal triangular solve algorithms described above can be called either in a sequential setting (corresponding to fronts under the $\mathcal{L}_0$ layer) or in a parallel one (corresponding to fronts above the $\mathcal{L}_0$ layer). Both settings are therefore of interest in the following.

An additional source of parallelism lies in the RHSs: these can be partitioned into blocks and all blocks can be handled concurrently. This type of parallelism must be used with care because partitioning the RHSs in excessively small blocks might degrade performance due to the small granularity of computations. In this work we focus on exploiting parallelism within a single block of RHSs.

3. New hybrid variants of the BLR triangular solve.

3.1. A novel hybrid variant. As mentioned in section 1, the performance of the BLR solve is underwhelming when dealing with many RHS. Based on the description of the RL and LL variants, we can see that one common weakness of both variants is that they require multiple accesses to the entire RHS. Indeed, at each step the RL variant only reads one block of the RHS, but needs to write all the bottom part of the RHS. Conversely, at each step the LL variant only writes one block, but needs to read all the top part of the RHS. We will precisely measure the volume of communications in the next section, but it is clear that when the number of RHS n_{rhs} is large, the accesses to the RHS can represent a much larger volume than the accesses to the BLR factors, and thus constitute the bottleneck of the computation.

Motivated by this observation, we propose in this section a novel BLR solve variant that is based on a hybrid scheme that only needs to read and write one block of the RHS per step. The main idea is to perform the read operations following a right-looking scheme, and the write operations following a left-looking scheme. To do so, we divide the update(i, j) task $X_i \leftarrow X_i - U_{ij}(V_{ij}^T X_j)$ into two separate subtasks:

- updateV(i, j): $W_{ij} = V_{ij}^T X_j$;
- updateU(i, j): $X_i \leftarrow X_i - U_{ij} W_{ij}$.

The updateV(i, j) tasks require to read the block X_j of the RHS; the updateU(i, j) tasks require to write the block X_i of the RHS. Thus, the idea of this hybrid variant is to perform the updateV tasks with a right-looking pattern (accessing X_j once and executing immediately all tasks for this j) and the updateU tasks with a left-looking pattern (waiting that all tasks for a given i are ready so as to access X_i only once). This is accomplished at the cost of having to store in a temporary workspace the W_{ij} matrices for $i > j$ (W_{ij} is created at step j and consumed at step i). However, these W_{ij} matrices are of small dimension $r \times n_{\text{rhs}}$, and so the cost of storing and accessing them should be small.

This new hybrid variant is outlined in Algorithm 3.1 and illustrated in Figure 3.1.

3.2. Parallelism-driven hybrid variant. One issue with the hybrid variant of Algorithm 3.1 is that it suffers from the same problems as the left-looking variant in a parallel setting: the updateU tasks corresponding to FS blocks of the RHS are performed following a left-looking pattern (loop on line 2 of Algorithm 3.1), which is of small size and requires a reduction.

To overcome this issue, we propose in Algorithm 3.2 a modified hybrid variant that is more amenable to a parallel execution. The idea is to use the hybrid scheme for the CB part of the computation only, which can be efficiently parallelized with a parallel loop on the block rows (line 11 of Algorithm 3.2), and to keep the efficiently parallelizable right-looking scheme for the FS part (with a parallel loop on line 3). Since the FS part is typically much smaller than the CB part, we can still expect to retain most of the benefits associated with the use of the hybrid scheme.

3.3. Low-rank updates accumulation. The hybrid variant described above minimizes the number of accesses to the RHS in order to reduce the volume of communications. It also increases arithmetic intensity of the BLR solve, defined as the ratio between the number of flops (which is

Algorithm 3.1 Hybrid variant.

1: **for** $k \in FS$ **do** ▷ Sequential loop
2: **for** $j < k$ **do** ▷ Parallel loop (left-looking)
3: $X_k \leftarrow X_k - U_{kj}W_{kj}$ ▷ updateU(k, j)
4: **end for**
5: $X_k \leftarrow L_{kk}^{-1}X_k$ ▷ trsm(k)
6: **for** $i > k$ **do** ▷ Parallel loop (right-looking)
7: $W_{ik} = V_{ik}^T X_k$ ▷ updateV(i, k)
8: **end for**
9: **end for**
10: **for** $k \in CB$ **do** ▷ Parallel loop
11: **for** $j \in FS$ **do** ▷ Sequential loop (left-looking)
12: $X_k \leftarrow X_k - U_{kj}W_{kj}$ ▷ updateU(k, j)
13: **end for**
14: **end for**

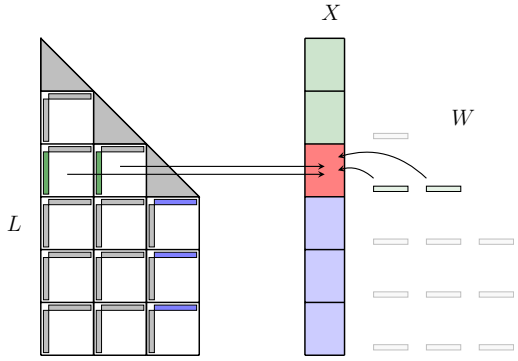
Algorithm 3.2 Parallelism-driven hybrid variant.

1: **for** $k \in FS$ **do** ▷ Sequential loop
2: $X_k \leftarrow L_{kk}^{-1}X_k$ ▷ trsm(k)
3: **for** $i > k$ **do** ▷ Parallel loop (right-looking)
4: **if** $i \in FS$ **then**
5: $X_i \leftarrow X_i - U_{ik}(V_{ik}^T X_k)$ ▷ update(i, k)
6: **else**
7: $W_{ik} = V_{ik}^T X_k$ ▷ updateV(i, k)
8: **end if**
9: **end for**
10: **end for**
11: **for** $k \in CB$ **do** ▷ Parallel loop
12: **for** $j \in FS$ **do** ▷ Sequential loop (left-looking)
13: $X_k \leftarrow X_k - U_{kj}W_{kj}$ ▷ updateU(k, j)
14: **end for**
15: **end for**

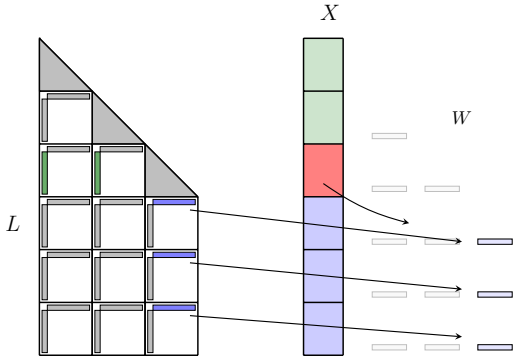
unchanged for all variants) and the volume of communications. In this section we now propose a further modification of the BLR solve algorithm to further increase its arithmetic intensity. The idea is based on using low-rank updates accumulation (LUA), that is, grouping together low-rank updates on the same block rows and/or block columns and applying them with a single matrix multiplication to increase the granularity of the computation.

The LUA technique has been originally proposed in [6] for the outer product operation in the BLR LU factorization, and is used by default in MUMPS. In the BLR factorization, we compute low-rank updates of the form

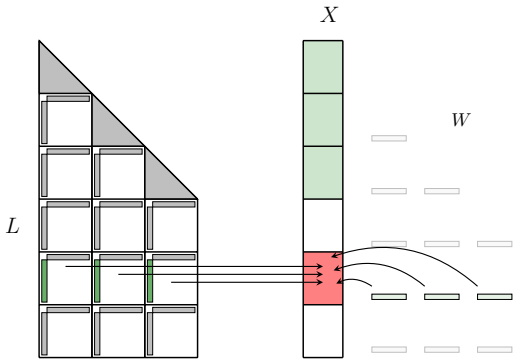
$$A_{ij} \leftarrow A_{ij} - (U_{ik}V_{ik}^T)(U_{kj}V_{kj}^T).$$



(a) updateU task for $k = 3$: access U_{kj} for $j < k$ in left-looking, and overwrite the FS block X_k.



(b) updateV task for $k = 3$: access V_{ik} for $i > k$ in right-looking. Each product $V_{ik}^T X_k$ is stored in W_{ik}.



(c) UpdateU for $k = 5$: access U_{kj} for $j \in FS$ in left-looking, and overwrite the CB block X_k.

Fig. 3.1: Hybrid variant.

In the left-looking variant of the BLR factorization, these low-rank updates can be grouped as

$$A_{ij} \leftarrow A_{ij} - \sum_{k=1}^{K} (U_{ik}V_{ik}^T)(U_{kj}V_{kj}^T) = A_{ij} - \sum_{k=1}^{K} \bar{U}_{ikj}\bar{V}_{ikj}^T,$$

where either $\bar{U}_{ikj} = U_{ik}(V_{ik}^T U_{kj})$ and $\bar{V}_{ikj} = V_{kj}^T$ or $\bar{U}_{ikj} = U_{ik}$ and $\bar{V}_{ikj} = (V_{ik}^T U_{kj})V_{kj}^T$, depending on the ranks. The sum in the above expression can then be efficiently evaluated as a single matrix multiplication $\bar{U}\bar{V}^T$. We emphasize that LUA in the BLR factorization is only possible with a left-looking scheme.

We now discuss how to adapt this LUA technique to the BLR solve. The situation is more complicated due to the presence of the RHS, which is not under low-rank form. In fact, we will see that LUA cannot be fully used in either the RL or LL variants of the BLR solve, but only partially. The hybrid variant allows us to take full advantage of LUA, as we now explain.

updateU accumulation: the updateU(k, j) tasks $X_k \leftarrow X_k - U_{kj}W_{kj}$ can be grouped together

for all j as

$$X_k \leftarrow X_k - \sum_{j=1}^{J} U_{kj} W_{kj} = X_k - \left[U_{k1} \cdots U_{kJ}\right] \left[W_{k1}^T \cdots W_{kJ}^T\right]^T, \tag{3.1}$$

with $J = k-1$ for the FS updates and $J = p_{\text{fs}}$ for the CB updates. The sum in (3.1) can be efficiently evaluated using only one matrix–matrix product, instead of J smaller ones. In order to do so, all U_{kj}, $j \leq J$, must be allocated contiguously in memory, as well as all W_{kj}, $j \leq J$.

updateV accumulation: the updateV(i,k) tasks $W_{ik} = V_{ik}^T X_k$ can be grouped together for all i as

$$\left[W_{i_0 k}^T \cdots W_{pk}^T\right]^T = \left[V_{i_0 k} \cdots V_{pk}\right]^T X_k \tag{3.2}$$

with $i_0 = k+1$. This operation can also be performed using only one matrix–matrix product instead of $p-k$. In order to do so, all V_{ik}, $i > k$, must be allocated contiguously, as well as all W_{ik}, $i > k$. Note that this allocation of the "W" workspaces is not compatible with the one needed for the updateU accumulation: updateU requires each block row to be a contiguous array, whereas updateV requires the same of each block column. As a result, if we wish to accumulate both types of tasks, the "W" arrays need to be transformed (that is, copied) from the updateU allocation scheme to the updateV one during the computation.

Note that the updateU tasks can only be accumulated with a left-looking scheme: thus, the RL variant can only benefit from the updateV accumulation. Conversely, the updateV tasks can only be accumulated with a right-looking scheme, so that the LL variant can only benefit from the updateU accumulation. The hybrid variant uses the right-looking scheme for the updateV tasks and the left-looking scheme for the updateU tasks: it can thus benefit from both LUA strategies. Naturally, in the parallelism-driven hybrid variant, LUA can only be used on the CB updates.

4. Communication volume analysis. In this section we develop a theoretical communication analysis that aims at measuring the total volume of communications required by the different BLR solve variants. In particular we seek to prove that the hybrid variant can significantly reduce the volume of accesses to the RHS, which should provide a benefit in the case where n_{rhs} is large.

To perform the analysis, we use a simple model of a two-level memory hierarchy: a fast but limited memory (such as a cache) and an unlimited but slow memory (such as RAM). To simplify, we assume that we have the control over the transfers of data between the two levels of memory, that is, that we can choose which data are discarded from the fast memory to make space for other data that we need to load from the slow memory. We also assume that the fast memory is large enough to accommodate all the data required to perform any given update(i,j) or trsm(j) tasks (essentially one BLR block and two RHS blocks, plus any temporary workspace associated with the computation). Conversely we assume that the fast memory is not large enough to accommodate all the data needed to perform more than one of these tasks, so that after one task is completed all the data required by the next task that were not already used by the previous task need to be loaded from the slow memory.

Under this model, we compute the volume of data that have to be transferred from the slow memory to the fast one for each BLR solve variant. This analysis provides an estimate of the cost of each variant, since the BLR solve tends to be a memory-bound computation in most practical cases.

Throughout the analysis, we distinguish three types of transfers: read-only (RO: the data are used but not modified), write-only (WO: data are created for the first time) and read/write (RW: existing data are used and modified). As an example, in the operation $C \leftarrow C - AB$, A and B

require RO transfers and C requires a RW transfer, whereas in the operation $C = AB$, C requires a WO transfer.

4.1. Analysis.

Right-looking variant. At each step $j \in FS$, the RL variant (Algorithm 2.1) requires the following transfers:

- trsm(j) reads the diagonal block L_{jj} and reads/writes the RHS block $X_j \to b^2$ RO transfers and bn_{rhs} RW transfers;
- each update(i, j) for $i > j$ reads the BLR block $U_{ij}V_{ij}^T$; X_j is also needed but is already in the fast memory $\to 2(p-j)br$ RO transfers; each update(i, j) also reads and writes the RHS block $X_i \to (p-j)bn_{\text{rhs}}$ RW transfers.

Summing over all steps $j \in FS$, the RL variant therefore requires a total volume of communications of

- $2qrb + p_{\text{fs}}b^2$ RO transfers,
- $qbn_{\text{rhs}} + p_{\text{fs}}bn_{\text{rhs}}$ RW transfers,

where we recall that $q = p_{\text{fs}}(p_{\text{fs}} - 1)/2 + p_{\text{fs}}p_{\text{cb}}$ denotes the total number of off-diagonal blocks in L.

Left-looking variant. At each step $i \in FS$, the LL variant (Algorithm 2.2) requires the following transfers:

- each update(i, j) for $j < i$ reads the BLR block $U_{ij}V_{ij}^T$ and the RHS block $X_j \to (i-1)(2br + bn_{\text{rhs}})$ RO transfers; each update(i, j) also reads and writes the RHS block X_i, but it only needs to be loaded once and can then be kept in the fast memory for all subsequent updates $\to bn_{\text{rhs}}$ RW transfers.
- trsm(i) reads L_{ii}; it also reads and writes X_i, which is already in the fast memory from the previous updates $\to b^2$ RO transfers.

Then, at each step $i \in CB$, it requires the following transfers:

- each update(i, j) for $j = 1$: p_{fs} reads the BLR block $U_{ij}V_{ij}^T$ and the RHS block $X_j \to p_{\text{fs}}(2br + bn_{\text{rhs}})$ RO transfers; each update(i, j) also reads and writes the RHS block X_i, but it only needs to be loaded once and can then be kept in the fast memory for all subsequent updates $\to bn_{\text{rhs}}$ RW transfers.

Summing over all steps $i \in FS \cup CB$, the LL variant therefore requires a total volume of communications of

- $2qrb + qbn_{\text{rhs}} + p_{\text{fs}}b^2$ RO transfers,
- pbn_{rhs} RW transfers.

Hybrid variant. At each step $k \in FS$, the hybrid variant (Algorithm 3.1) requires the following transfers:

- updateU(k, j) for $j < k$ reads U_{kj} and $W_{kj} \to (k-1)(b+n_{\text{rhs}})r$ RO transfers; updateU(k, j) also reads/writes X_k, which only needs to be loaded once $\to bn_{\text{rhs}}$ RW transfers;
- trsm(k) reads L_{kk}; X_k is already in the fast memory $\to b^2$ RO transfers;
- updateV(i, k) for $i > k$ reads V_{ik} and X_k, the latter still being in the fast memory; it also writes $W_{ik} \to (p-k)br$ RO transfers and $(p-k)n_{\text{rhs}}r$ WO transfers.

The following additional transfers are then required at each step $k \in CB$:

- updateU(k, j) for $j \in FS$ reads U_{kj} and $W_{kj} \to p_{\text{fs}}(b + n_{\text{rhs}})r$ RO transfers; updateU(k, j) also reads/writes X_k, which only needs to be loaded once $\to bn_{\text{rhs}}$ RW transfers.

Summing over all steps $k \in FS \cup CB$, the hybrid variant therefore requires a total volume of communications of

- $2qrb + qrn_{\text{rhs}} + p_{\text{fs}}b^2$ RO transfers,

- qrn_{rhs} WO transfers,
- pbn_{rhs} RW transfers.

Parallelism-driven hybrid variant. Finally, we compute the communication volume of the parallelism-driven hybrid variant (Algorithm 3.2), which is obtained by combining the RL volume for the FS part of the computation with the hybrid volume for the CB part. At each step $k \in FS$, the parallelism-driven hybrid variant requires the following transfers:

- trsm(k) reads L_{kk} and reads/writes $X_k \to b^2$ RO transfers and bn_{rhs} RW transfers;
- update(i,k) for $i > k$ and $i \in FS$ reads U_{ik} and V_{ik}; it also reads X_k which is already in the fast memory; and finally it also reads/writes $X_i \to 2(p_{\text{fs}} - k)br$ RO transfers and $(p_{\text{fs}} - k)bn_{\text{rhs}}$ RW transfers;
- updateV(i,k) for $i \in CB$ reads V_{ik} and X_k, the latter still being in the fast memory; it also writes $W_{ik} \to p_{\text{cb}}br$ RO transfers and $p_{\text{cb}}n_{\text{rhs}}r$ WO transfers.

The following additional transfers are then required at each step $k \in CB$:

- updateU(k,j) for $j \in FS$ reads U_{kj} and W_{kj}; it also reads/writes X_k, which only needs to be loaded once $\to p_{\text{fs}}br + p_{\text{fs}}n_{\text{rhs}}r$ RO transfers and bn_{rhs} RW transfers.

Summing over all steps $k \in FS \cup CB$, the parallelism-driven hybrid variant therefore requires a total volume of communications of

- $2qrb + p_{\text{fs}}p_{\text{cb}}rn_{\text{rhs}} + p_{\text{fs}}b^2$ RO transfers,
- $p_{\text{fs}}p_{\text{cb}}rn_{\text{rhs}}$ WO transfers,
- $(p + p_{\text{fs}}(p_{\text{fs}} - 1)/2)bn_{\text{rhs}}$ RW transfers.

4.2. Discussion. First, we note that the RL and LL variants are not completely equivalent in terms of communications: while they require the same overall volume regardless of transfer type, our analysis shows that qbn_{rhs} RW transfers in the RL variant have been replaced with the same volume of RO transfers in the LL variant. Therefore, in a context where RW transfers are more costly than RO ones the LL variant might outperform the RL one, at least in a sequential environment. This could for example occur if a RW transfer requires a first transfer from the slow to the fast memory and then a second transfer in the other direction.

We now seek to determine when the hybrid variant requires less communications than the LL one. To do so, we must make an assumption on the relative cost of RO, WO, and RW transfers. Under the assumption that RO and WO transfers are equally costly, and neglecting lower order terms in the expression of the communication volume, we obtain a ratio between the LL volume and the hybrid volume approximately equal to

$$\frac{2qrb + 2qrn_{\text{rhs}}}{2qrb + qbn_{\text{rhs}}} = \frac{1 + n_{\text{rhs}}/b}{1 + n_{\text{rhs}}/2r}. \tag{4.1}$$

Thus the condition for the hybrid variant to require less communications than the LL one is $2r \leq b$, which we can expect to be always satisfied, since it corresponds to the condition for a block to be low-rank (if $r > b/2$, the block requires less storage if represented as a dense block). We can thus conclude that the hybrid variant should always communicate less than the LL one.

A more important question is when can we expect the hybrid variant to communicate *much* less than the LL one: that is, when is ratio (4.1) much less than 1? In the regime where $n_{\text{rhs}}/2r$ is small (just one or few RHS), the ratio (4.1) is close to 1. The hybrid variant therefore does not significantly reduce the volume of communications for small numbers of RHS. However, in the regime where $n_{\text{rhs}}/2r \gg 1$, the ratio (4.1) is approximately equal to $2r/n_{\text{rhs}} + 2r/b$. This shows that (4.1) is much less than 1 if $\min(n_{\text{rhs}}, b) \gg 2r$: that is, the hybrid variant leads to significant gains when we have a large number of RHS and the blocks are very low-rank.

As for the parallelism-driven hybrid variant, without surprise it achieves a tradeoff between the hybrid and LL ones. It requires $p_{\text{fs}}(p_{\text{fs}}-1)/2(b-r)n_{\text{rhs}}$ extra transfers corresponding to the FS part of the computation which does not use the hybrid communication pattern. This extra volume can be expected to be small for typical cases where $p_{\text{fs}} \ll p_{\text{cb}}$.

We summarize in Table 4.1 the dominant terms in the total communication volume of the different variants.

Table 4.1: Summary of the communication volume analysis: dominant terms for each variant.

Variant	Communication volume		
	RO	WO	RW
Right-looking	$2qrb$		qbn_{rhs}
Left-looking	$2qrb + qbn_{\text{rhs}}$		
Hybrid	$2qrb + qrn_{\text{rhs}}$	qrn_{rhs}	
Parallelism-driven hybrid	$2qrb + p_{\text{fs}}p_{\text{cb}}rn_{\text{rhs}}$	$p_{\text{fs}}p_{\text{cb}}rn_{\text{rhs}}$	$(p + p_{\text{fs}}^2/2)bn_{\text{rhs}}$

The above analysis of these hybrid variants also applies to the case where LUA is used. We do not develop a specific analysis for the use of LUA: as mentioned, we can expect LUA to also reduce the volume of communications by reducing the number of cache misses; this is however a more complex phenomenon that our simple communication model used in this section does not capture.

5. Performance analysis based on a simplified prototype.

5.1. Experimental setting. In order to assess the potential of the new BLR solve variants, we have first developed a prototype which implements a partial BLR solve of a given frontal matrix. Since we aim to use this prototype for performance analysis only, we use a synthetic random matrix and we make some further simplifications by forcing both the block size b and the ranks of the blocks r to be constant.

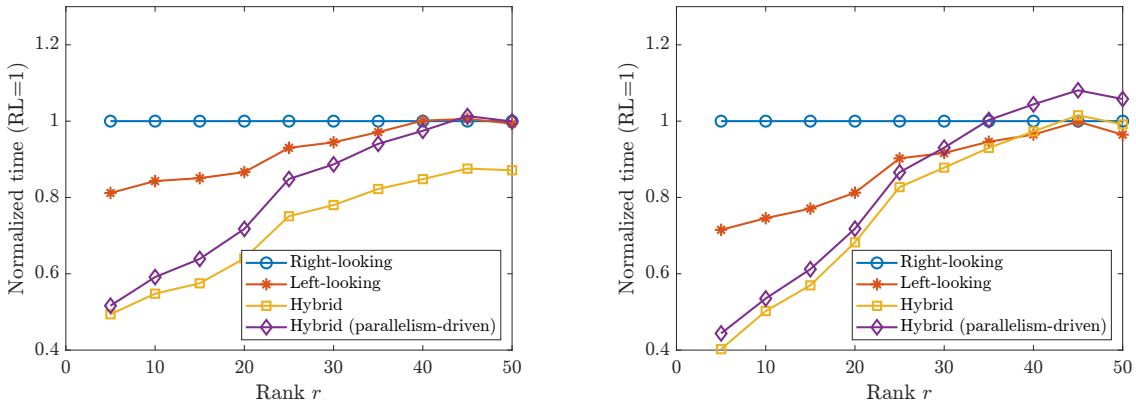
The prototype implements the four BLR solve variants: RL, LL, hybrid, and parallelism-driven hybrid. The prototype also allows for the optional use of LUA for the updateU and/or updateV tasks.

As explained in section 2.3, the frontal BLR solve algorithms are both needed for large fronts at the top of the tree, where node and tree parallelism are both exploited, but also for smaller fronts at the bottom of tree, where only tree parallelism is exploited. Therefore, our goal is to use the prototype to analyze the performance of the BLR solve in two types of configurations: the first configuration uses tree parallelism only by running 36 instances in parallel, each using a single thread; the second configuration uses both node and tree parallelism by running two instances in parallel, each parallelized with 18 threads.

All our experiments were run on the Olympe supercomputer; each node is equipped with two 18-core Intel Skylake 6140 processors running at 2.3 GHz (for a total of 36 cores per node). We use Intel MKL 2018 for the BLAS libraries.

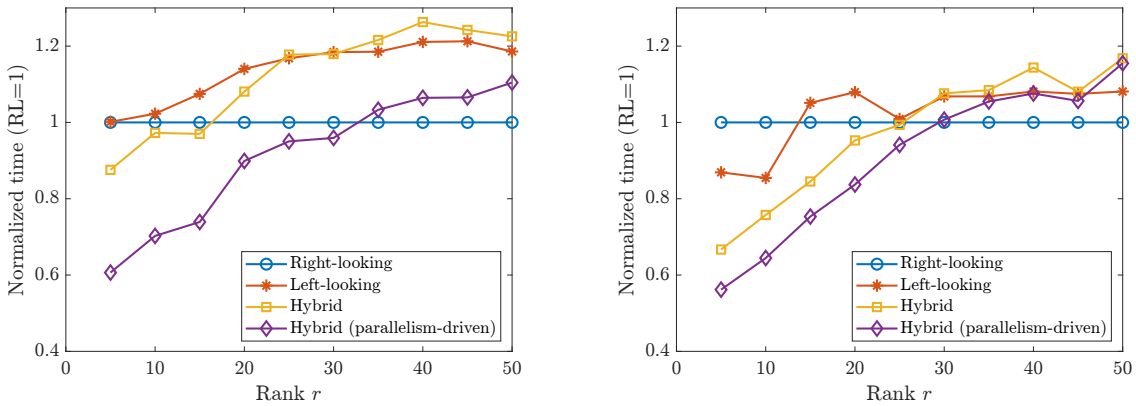
5.2. Performance analysis of hybrid variants. We report our performance results in Figures 5.1 and 5.2 for the tree and node parallelism configurations, respectively. For all experiments we set $n_{\text{rhs}} = 250$ and $p_{\text{fs}} = p/5$. For the node parallelism experiments, we use large fronts (BLR block size $b = 500$ and $p = 100$ or 200, leading to a front size of $50,000$ or $100,000$). For the tree

parallelism experiments, we use smaller fronts (BLR block size $b = 250$ and $p = 30$ or 50, leading to a front size of $7,500$ or $12,500$).



(a) $p = 30$ (b) $p = 50$

Fig. 5.1: Time spent in solve with tree parallelism (36 instances with 1 thread per instance), $b = 250$, $n_{\text{rhs}} = 250$, and $p_{\text{fs}} = p/5$.



(a) $p = 100$ (b) $p = 200$

Fig. 5.2: Time spent in solve with node+tree parallelism (2 instances with 18 threads per instance), $b = 500$, $n_{\text{rhs}} = 250$, and $p_{\text{fs}} = p/5$.

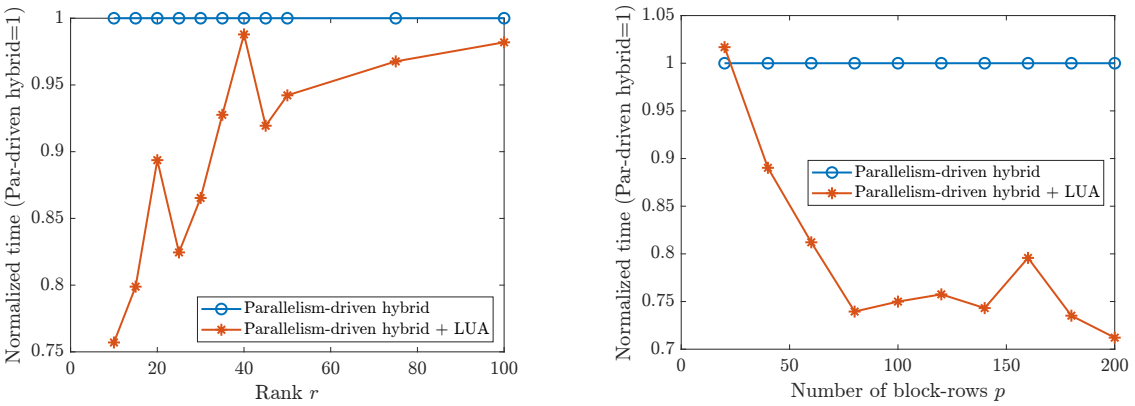
These results show that, as could be hoped, the hybrid variants can be faster than the RL and LL variants in many cases. The gains are the most significant when the rank r is small, and when the problem size is large, with speedups reaching a factor 2.5× in the best case. The parallelism-driven hybrid variant is not as fast as the standard hybrid one in the configuration with tree parallelism, but is significantly superior, as expected, in the configuration with node

parallelism. Indeed, as explained in section 3.2, the left-looking loop at line 2 of Algorithm 3.1 involves a reduction operation on a loop of small size, whose parallelization on 18 threads is not very efficient. This is also confirmed in Figure 5.2 by the observation that for larger values of p the performance of left-looking, hybrid and parallelism-driven hybrid variants gets closer.

5.3. Performance analysis of LUA. We finally analyze the time gains obtained when combining the hybrid variant with the LUA approach described in section 3.3. We report the performance with and without LUA in Figure 5.3. We use a node+tree parallelism configuration and therefore take the parallelism-driven hybrid variant as baseline. The figure consists of two plots, one where the number of block-rows p is fixed and the rank r varies, and the other where r is fixed and p varies. These two plots illustrate two opposite trends:

- As r increases, the benefits of using LUA diminish, since the granularity of the low-rank updates without accumulation is already large enough to achieve good performance.
- As p increases, the benefits of using LUA increase, since there are more blocks to accumulate together, which leads to a better granularity.

Overall, the use of LUA can lead to significant speedups on large fronts with small ranks.



(a) $p = 100$, r varies (b) $r = 10$, p varies

Fig. 5.3: Time gains obtained with LUA, with node+tree parallelism (2 instances with 18 threads per instance), $b = 500$, $n_{\text{rhs}} = 250$, and $p_{\text{fs}} = p/5$.

6. Results on real-life applications with the MUMPS solver. Based on the encouraging results obtained with our prototype in the previous section, we have implemented a subset of variants directly in the MUMPS solver. In addition to the existing RL variant (used so far by MUMPS), we have implemented the LL, hybrid, and parallelism-driven hybrid variants. For now we have not implemented the use of LUA, which we leave for future work.

We now present some experimental results obtained with these new variants of the MUMPS BLR solve on a range of real-life matrices, listed in Table 6.1. All the experiments have been performed on one node of the Olympe supercomputer previously described, using 2 MPI processes and 18 threads per MPI process, except for the tests with the Poisson200 and thmgaz matrices, which do not fit on a single node, and for which 4×18 threads were used. The BLR ϵ controls

the accuracy of the BLR factorization: the low-rank blocks $L_{ij} \approx U_{ij}V_{ij}^T$ are truncated such that $\|L_{ij} - U_{ij}V_{ij}^T\| \leq \epsilon$. We use double precision real arithmetic ("d") for all problems except the Geoazur ones, for which we use single precision complex arithmetic ("c").

Table 6.1: List of matrices used for the experiments.

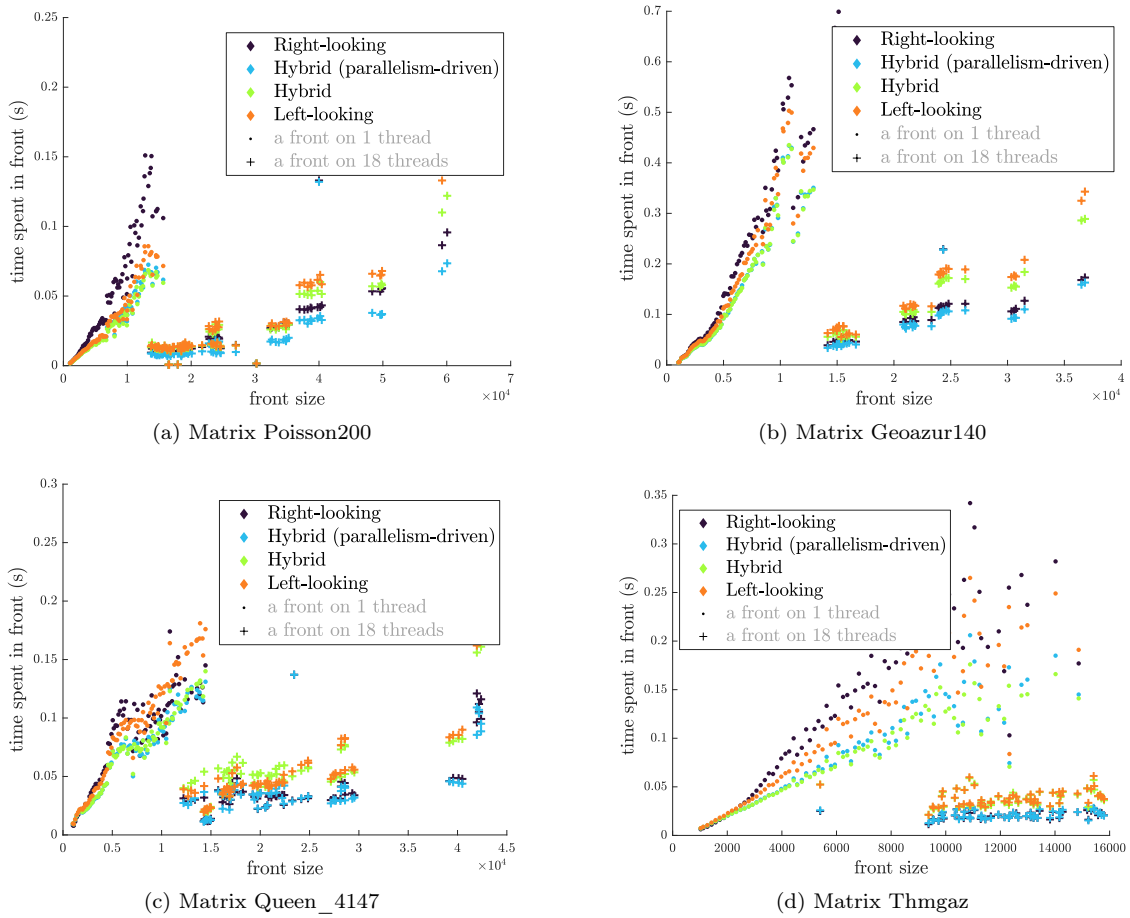
Matrix	Order	BLR ϵ	Arithmetic	BLR compression (% of dense)
Poisson120	1.7M	10^{-6}	d	27%
Poisson160	4.1M	10^{-6}	d	22%
Poisson200	8.0M	10^{-6}	d	19%
Geoazur100	1.6M	10^{-4}	c	52%
Geoazur140	3.8M	10^{-4}	c	47%
atmosmodl	1.5M	10^{-6}	d	30%
Geo_1438	1.4M	10^{-6}	d	50%
Queen_4147	4.1M	10^{-6}	d	28%
Serena	1.4M	10^{-6}	d	40%
Transport	1.6M	10^{-6}	d	42%
thmgaz	5.0M	10^{-6}	d	30%

First, in Figure 6.1, we plot the time spent in the forward solve of MUMPS for each BLR front. This leads to plots where we can distinguish two distinct groups of points. Those corresponding to the smaller fronts in the lower part of the tree where tree parallelism is used, are each processed using 1 thread. Those corresponding to the larger fronts higher in the tree, where node parallelism is used, are processed using 18 threads. The figure shows that for the larger fronts with node parallelism, the parallelism-driven hybrid variant is the fastest, whereas for the smaller fronts with tree parallelism, the hybrid variant seems to be the best choice for most fronts. This confirms the trends observed with the prototype experiments in the previous section.

The optimal approach therefore seems to be to combine the two types of hybrid variants by using the standard one when only tree parallelism is used and the parallism-driven when node parallelism is used.

To assess the impact of these per-front gains on the total time, we report in Table 6.2 the cumulative time spent in all fronts. This includes the time spent in the dense fronts (the very smallest fronts at the bottom of the tree, where BLR compression is not exploited), which are not shown in Figure 6.1. The table compares the standard RL variant to the optimized hybrid variant (which uses the parallelism-driven algorithm only on parallel fronts). The results confirm that the hybrid variant can achieve significant time reductions overall.

7. Conclusions. The performance of BLR sparse triangular solve, which is critical in several applications, is underwhelming when there are many RHS. This is explained by the fact that the computational bottleneck is the memory access to the RHS, which are dense and far heavier than the compressed BLR LU factors. To overcome this limitation, we have proposed novel hybrid algorithms that combine right-looking and left-looking communication patterns to minimize the number of accesses to the RHS. We have carried out a communication volume analysis that proves that these new variants are indeed communication-avoiding. Based on a performance analysis on



(a) Matrix Poisson200

(b) Matrix Geoazur140

(c) Matrix Queen_4147

(d) Matrix Thmgaz

Fig. 6.1: Time spent in each BLR front, in the forward solve of MUMPS.

synthetic data using a simple dense triangular solve prototype, we have selected a subset of the most promising hybrid variants and implemented them in the widely used MUMPS solver. Using this implementation, we have confirmed the potential of these new variants on several real-life applications, obtaining up to 20% time reduction.

In this paper we have focused on the double precision solver. However, we mention as an important perspective of this work that the new hybrid variants may be even more successful in a mixed precision context. Indeed, in recent work, we have shown that the BLR LU factors can be stored in mixed precision while preserving the same accuracy [2]. As a result, the mixed precision BLR LU factors are further compressed and can reduce the triangular solve time. However, the cost of accessing the RHS becomes even more dominant. This is illustrated in Table 7.1 for matrix Queen_4147, which shows that the mixed precision approach can reduce the BLR LU storage by a factor 1.4×. This storage reduction is translated into a comparable RL forward solve time reduction

Table 6.2: Time spent in the forward solve in MUMPS; $n_{\text{rhs}} = 250$ and 2×18 cores are used for all problems except Poisson200 (4×18).

Matrix	Time (s)		Gain
	Right-looking	Optimized hybrid	
Poisson120	0.78	0.69	**-12%**
Poisson160	2.15	1.76	**-18%**
Poisson200	2.30	1.83	**-20%**
Geoazur100	1.95	1.78	**-9%**
Geoazur140	5.26	4.70	**-11%**
atmosmodl	0.52	0.49	**-6%**
Geo_1438	1.35	1.30	**-3%**
Queen_4147	5.90	4.80	**-20%**
Serena	1.23	1.20	**-2%**
Transport	0.73	0.67	**-9%**
thmgaz	2.53	2.53	**-0%**

with a single RHS (1.3× speedup), but not with multiple ones (only 1.1× speedup). In future work it could therefore be promising to combine the use of mixed precision proposed in [2] and the new hybrid variants proposed in this article.

The inherent irregularity of low-rank operations and the limitations to parallelism in the triangular solve which we have highlighted in this work make our algorithms suitable to the use of task-based parallelism, as provided, for example, by the OpenMP standard. Nevertheless, it must be noted that a straightforward use of this parallel programming paradigm will not necessarily lead to better performance because favorable data locality properties might be lost due to the eager and asynchronous execution model. Although these properties might be preserved through a careful scheduling of tasks, this is a research challenge of its own, which we reserve for future work.

Table 7.1: A perspective of this work: BLR LU factors storage (GB) and RL forward solve time (s) for the BLR solver in double or mixed precision.

	BLR LU storage (GB)			Solve time ($n_{\text{rhs}} = 1$)			Solve time ($n_{\text{rhs}} = 250$)		
Matrix	Double	Mixed	Ratio	Double	Mixed	Ratio	Double	Mixed	Ratio
Queen_4147	33.6	23.4	1.4×	0.38	0.30	1.3×	5.90	5.31	1.1×

Acknowledgements. This work was done in the context of the CIFRE PhD thesis of Matthieu Gerest funded by EDF. All the experiments were performed on the Olympe supercomputer of the CALMIP center (project P0989).

REFERENCES

[1] P. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L'Excellent, and C. Weisbecker, *Improving multifrontal methods by means of block low-rank representations*, SIAM J. Sci. Comput., 37 (2015), pp. A1451–A1474, https://doi.org/10.1137/120903476.

[2] P. Amestoy, O. Boiteau, A. Buttari, M. Gerest, F. Jezequel, J.-Y. L'Excellent, and T. Mary, *Mixed precision low-rank approximations and their application to block low-rank LU factorization*, IMA J. Numer. Anal., (2022), pp. 1–30, https://doi.org/10.1093/imanum/drac037.

[3] P. Amestoy, A. Buttari, N. J. Higham, J.-Y. L'Excellent, T. Mary, and B. Vieublé, *Combining sparse approximate factorizations with mixed-precision iterative refinement*, ACM Trans. Math. Software, 49 (2023), pp. 4:1–4:29, https://doi.org/10.1145/3582493.

[4] P. R. Amestoy, R. Brossier, A. Buttari, J.-Y. L'Excellent, T. Mary, L. Métivier, A. Miniussi, and S. Operto, *Fast 3D frequency-domain full waveform inversion with a parallel Block Low-Rank multifrontal direct solver: application to OBC data from the North Sea*, Geophysics, 81 (2016), pp. R363–R383, https://doi.org/10.1190/geo2016-0052.1.

[5] P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary, *On the Complexity of the Block Low-Rank Multifrontal Factorization*, SIAM J. Sci. Comput., 39 (2017), pp. A1710–A1740, https://doi.org/10.1137/16M1077192.

[6] P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary, *Performance and Scalability of the Block Low-Rank Multifrontal Factorization on Multicore Architectures*, ACM Trans. Math. Software, 45 (2019), pp. 2:1–2:26, https://doi.org/10.1145/3242094.

[7] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent, *A fully asynchronous multifrontal solver using distributed dynamic scheduling*, SIAM J. Matrix Anal. Appl., 23 (2001), pp. 15–41.

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

[9] P. Ghysels, X. S. Li, F.-H. Rouet, S. Williams, and A. Napov, *An efficient multicore implementation of a novel hss-structured multifrontal solver using randomized sampling*, SIAM J. Sci. Comput., 38 (2016), pp. S358–S384, https://doi.org/10.1137/15M1010117.

[10] P. Hénon, P. Ramet, and J. Roman, *PaStiX: A high-performance parallel direct solver for sparse symmetric definite systems*, Parallel Computing, 28 (2002), pp. 301–321.

[11] N. J. Higham and T. Mary, *A new preconditioner that exploits low-rank approximations to factorization error*, SIAM J. Sci. Comput., 41 (2019), pp. A59–A82, https://doi.org/10.1137/18M1182802.

[12] J.-Y. L'Excellent and M. W. Sid-Lakhdar, *A study of shared-memory parallelism in a multifrontal solver*, Parallel Comput., 40 (2014), pp. 34–46.

[13] T. Mary, *Block Low-Rank multifrontal solvers: complexity, performance, and scalability*, PhD thesis, Université de Toulouse, Nov. 2017.

[14] S. Operto, P. Amestoy, H. Aghamiry, S. Beller, A. Buttari, L. Combe, V. Dolean, M. Gerest, G. Guo, P. Jolivet, J.-Y. L'Excellent, F. Mamfoumbi, T. Mary, C. Puglisi, A. Ribodetti, and P.-H. Tournier, *Is 3d frequency-domain fwi of full-azimuth/long-offset obn data feasible? the gorgon data fwi case study*, The Leading Edge, 42 (2023), pp. 173–183, https://doi.org/10.1190/tle42030173.1.

[15] G. Pichon, *On the use of low-rank arithmetic to reduce the complexity of parallel sparse linear solvers based on direct factorization techniques*, Ph.D. thesis, Université de Bordeaux, Nov. 2018, https://hal.inria.fr/tel-01953908/.

[16] G. Pichon, E. Darve, M. Faverge, P. Ramet, and J. Roman, *Sparse supernodal solver using block low-rank compression: Design, performance and analysis*, Journal of Computational Science, 27 (2018), pp. 255–270, https://doi.org/https://doi.org/10.1016/j.jocs.2018.06.007, http://www.sciencedirect.com/science/article/pii/S1877750317314497.

[17] F.-H. Rouet, X. S. Li, P. Ghysels, and A. Napov, *A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization*, ACM Trans. Math. Software, 42 (2016), pp. 27:1–27:35, https://doi.org/10.1145/2930660, http://doi.acm.org/10.1145/2930660.

[18] D. V. Shantsev, P. Jaysaval, S. de la Kethulle de Ryhove, P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary, *Large-scale 3D EM modeling with a Block Low-Rank multifrontal direct solver*, Geophys. J. Int., 209 (2017), pp. 1558–1571, https://doi.org/10.1093/gji/ggx106.