Performance and Scalability of the Block Low-Rank Multifrontal Factorization on Multicore Architectures

Patrick Amestoy, Alfredo Buttari, Jean-Yves L'Excellent, Théo Mary

▶ **To cite this version:**

Patrick Amestoy, Alfredo Buttari, Jean-Yves L'Excellent, Théo Mary. Performance and Scalability of the Block Low-Rank Multifrontal Factorization on Multicore Architectures. ACM Transactions on Mathematical Software, 2019, 45 (1), pp.1-26. 10.1145/3242094 . hal-01955766v1

HAL Id: hal-01955766

https://inria.hal.science/hal-01955766v1

Submitted on 21 Jan 2019 (v1), last revised 10 Oct 2019 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Performance and Scalability of the Block Low-Rank Multifrontal Factorization on Multicore Architectures

Patrick R. Amestoy* Alfredo Buttari† Jean-Yves L'Excellent‡
Theo Mary§

Abstract. Matrices coming from elliptic Partial Differential Equations have been shown to have a low-rank property which can be efficiently exploited in multifrontal solvers to provide a substantial reduction of their complexity. Among the possible low-rank formats, the Block Low-Rank format (BLR) is easy to use in a general purpose multifrontal solver and its potential compared to standard (full-rank) solvers has been demonstrated. Recently, new variants have been introduced and it was proved that they can further reduce the complexity but their performance has never been analyzed. In this paper, we present a multithreaded BLR factorization, and analyze its efficiency and scalability in shared-memory multicore environments. We identify the challenges posed by the use of BLR approximations in multifrontal solvers and put forward several algorithmic variants of the BLR factorization that overcome these challenges by improving its efficiency and scalability. We illustrate the performance analysis of the BLR multifrontal factorization with numerical experiments on a large set of problems coming from a variety of real-life applications.

1 Introduction

We are interested in efficiently computing the solution of large sparse systems of linear equations. Such a system is usually referred to as:

$$Ax = b\,, \tag{1}$$

where A is a sparse matrix of order n, x is the unknown vector of size n, and b is the right-hand side vector of size n.

This paper focuses on solving (1) with direct approaches based on Gaussian elimination and more particularly the multifrontal method, which was introduced by [22] and, since then, has been the object of numerous studies [30, 5, 20].

The multifrontal method achieves the factorization of a sparse matrix A as $A = LU$ or $A = LDL^T$ depending on whether the matrix is unsymmetric or symmetric, respectively. A is factored through a sequence of operations on relatively small dense matrices called *frontal matrices* or, simply, *fronts*, on which a partial factorization is performed, during which some variables (the fully-summed (FS) variables) are eliminated,

*Université de Toulouse, INPT-IRIT
†Université de Toulouse, CNRS-IRIT
‡Université de Lyon, INRIA-LIP
§Université de Toulouse, UPS-IRIT

i.e. the corresponding factors are computed, and some other variables (the non fully-summed (NFS) variables) are only updated. To know which variables come into which front, and in which order the fronts can be processed, an *elimination* or *assembly tree* [29, 34] is built, which represents the dependencies between fronts.

Consequently, the multifrontal factorization consists of a bottom-up traversal of the elimination tree where, each time a node is visited, two operations are performed.

- *assembly*: The frontal matrix is formed by summing the initial matrix coefficients in the rows and columns of the fully-summed variables of the tree node with coefficients produced by the factorization of its children.

- *factorization*: Once the frontal matrix is formed, a partial LU factorization is performed in order to eliminate the fully-summed variables associated with the tree node. The result of this operation is a set of columns/rows of the global LU factors and a Schur complement, also commonly referred to as *contribution block* (CB), containing coefficients that will be assembled into the parent node.

In many applications (e.g., those coming from the discretization of elliptic Partial Differential Equations), the matrix A has been shown to have a low-rank property: conveniently defined off-diagonal blocks of its Schur complements can be approximated by low-rank products [13]. Several formats have been proposed to exploit this property, mainly differing on whether they use a strong or weak admissibility condition and on whether they have a nested basis property. The most general of the hierarchical formats is the $\mathcal{H}$-matrix format [13, 25, 14], which is non-nested and strongly-admissible. The $\mathcal{H}^2$-matrix format [14] is its nested counterpart. HODLR matrices [9] are based on the weak admissibility condition and HSS [43, 15] and the closely related HBS [24] additionally possess nested basis.

These low-rank formats can be efficiently exploited within direct multifrontal solvers to provide a substantial reduction of their complexity. In comparison to the quadratic complexity of the Full-Rank (FR) solver, most sparse solvers based on hierarchical formats have been shown to possess near-linear complexity. To cite a few, [42], [41], and [23] are HSS-based, [24] is HBS-based, [10] is HODLR-based, and [33] is $\mathcal{H}^2$-based.

Previously, we have investigated the potential of a so-called Block Low-Rank (BLR) format [3] that, unlike hierarchical formats, is based on a flat, non-hierarchical blocking of the matrix which is defined by conveniently clustering the associated unknowns. We have recently shown [8] that the complexity of the BLR multifrontal factorization may be as low as $O(n^{4/3})$ (for 3D problems with constant ranks). While hierarchical formats may achieve a lower theoretical complexity, the simplicity and flexibility of the BLR format make it easy to use in the context of a general purpose, algebraic solver. In particular, BLR solvers have the following distinct advantages [3, 31]:

- No relative order is needed between blocks; this allows the clustering to be easily computed and delivers a great flexibility to distribute the data in a parallel environment.

- The size of the blocks is small enough for several of them to fit on a single shared-memory node; therefore, in a parallel environment, each processor can efficiently and concurrently work on different blocks.

- The simplicity of the BLR format makes it easy to handle dynamic numerical pivoting, a critical feature often lacking in other low-rank solvers.

- Non fully-structured BLR solvers, where the current active front is temporarily stored in full-rank, have the same asymptotic complexity as their fully-structured counterparts, contrarily to hierarchical formats. In particular, this allows for performing the assembly in full-rank and thus avoid the slow and complex low-rank extend-add operations.

Therefore, BLR solvers can be competitive with hierarchical formats, at least on some problem classes and for some range of sizes. We feel the differences between low-rank formats are a complicated issue and there are still many open questions about their practical behavior on low-rank solvers and their respective advantages and limits. A careful comparison between existing low-rank solvers would be an interesting issue but is out the scope of this article. Preliminary results of a comparison between the BLR-based MUMPS solver and the HSS-based STRUMPACK solver have been reported in [31].

The so-called standard BLR factorization presented in [3] has been shown to provide significant gains compared to the Full-Rank solver in a sequential environment. Since then, BLR approximations have been used in the context of a dense Cholesky solver for GPU [2], the PaStiX supernodal solver for multicores [32], and the `MUMPS` multifrontal solver for distributed-memory architectures [4, 36].

New variants that depend on the strategies used to perform, accumulate, and recompress the low-rank updates, and on the approaches to handle numerical pivoting have been presented in [11] and [7] and it was proved in [8] that they can further reduce the complexity of the BLR approach. The performance of these new variants has however never been studied. In this article, we present a multithreaded BLR factorization for multicore architectures and analyze its performance on a variety of problems coming from real-life applications. We explain why it is difficult to fully convert the reduction in the number of operations into a performance gain, especially in multicore environments, and describe how to improve the efficiency and the scalability of the BLR factorization.

To conclude, let us briefly describe the organization of this paper. Rather than first presenting all the algorithms and then their analysis, we will present the algorithms incrementally and interlaced with their analysis, to better motivate their use and what improvements they bring. In Section 2, we provide a brief presentation of the BLR format, the standard BLR factorization algorithm, so-called FSCU variant, and how it can be used within multifrontal solvers; for a more formal and detailed presentation, we refer the reader to [3] where this method was introduced. In Section 3, we describe our experimental setting. In Section 4, we motivate our work with an analysis of the performance of the FSCU algorithm in a sequential setting. We then present in Section 5 the parallelization of the BLR factorization in a shared-memory context, the challenges that arise, and the algorithmic choices made to overcome these challenges. In Section 6, we analyze the algorithmic variants of the BLR multifrontal factorization. We show how they can improve the performance of the standard algorithm. In Section 7, we provide a complete set of experimental results on a variety of real-life applications and in different multicore environments. We provide our concluding remarks in Section 8.

Please note that the BLR approximations can reduce the memory consumption of the factorization and improve the performance of the solution phase, and that this analysis does not depend on the BLR variant used, as explained in [8]. Both these aspects are out of the scope of this paper.

2 Preliminaries

2.1 Block Low-Rank approximations

Unlike hierarchical formats such as $\mathcal{H}$-matrices, the BLR format is based on a flat, non-hierarchical blocking of the matrix which is defined by conveniently clustering the associated unknowns. A BLR representation $\widetilde{F}$ of a dense matrix F is shown in Equation (2), where we assume that p sub-blocks have been defined. Sub-blocks B_{ij} of size $m_i \times n_j$ and numerical rank k_{ij}^{ε} are approximated by a low-rank product $\widetilde{B}_{ij} = X_{ij}Y_{ij}^T$ at accuracy ε, where X_{ij} is a $m_i \times k_{ij}^{\varepsilon}$ matrix and Y_{ij} is a $n_j \times k_{ij}^{\varepsilon}$ matrix.

$$\widetilde{F} = \begin{bmatrix} \widetilde{B}_{11} & \widetilde{B}_{12} & \cdots & \widetilde{B}_{1p} \\ \widetilde{B}_{21} & \cdots & \cdots & \vdots \\ \vdots & \cdots & \cdots & \vdots \\ \widetilde{B}_{p1} & \cdots & \cdots & \widetilde{B}_{pp} \end{bmatrix} \tag{2}$$

The $\tilde{B}_{ij}$ approximation of each block can be computed in different ways. We have chosen to use a truncated QR factorization with column pivoting; this corresponds to a QR factorization with pivoting which is truncated as soon as a diagonal coefficient of the R factor falls below the prescribed threshold ε. This choice allows for a convenient compromise between cost and accuracy of the compression operation.

2.2 Block Low-Rank LU or LDL^T factorization

We describe in Algorithm 1 the standard BLR factorization algorithm for dense matrices, introduced in [3].

In order to perform the LU or LDL^T factorization of a dense BLR matrix, the standard block LU or LDL^T factorization has to be modified so that the low-rank sub-blocks can be exploited to perform fast operations. Many such algorithms can be defined depending on where the compression step is performed. We present, in Algorithm 1, a version where the compression is performed after the so-called Solve step. We present Algorithm 1 in its LDL^T version, but it can be easily adapted to the unsymmetric case.

As described in detail in [3], this algorithm is fully compatible with threshold partial pivoting [20]. The pivots are selected inside the BLR blocks; to assess their quality, they are compared to the pivots of the entire column. Therefore, in practice, to perform numerical pivoting, the Solve step is merged with the Factor step and done in full-rank (i.e. before the Compress). The pivots that are too small with respect to a given threshold τ are delayed to the next BLR block, with a mechanism similar to the delayed pivoting between fronts [22]. These details are omitted in Algorithm 1 for the sake of clarity.

This algorithm is referred to as FSCU (standing for Factor, Solve, Compress, and Update), to indicate the order in which the steps are performed. The algorithm is presented in its Right-looking form. In Section 5, we will also study the performance of its Left-looking version, referred to as UFSC.

We recall that we denote the low-rank form of a block B by $\widetilde{B}$. Thus, the Outer Product on line 13 consists in decompressing the low-rank block $\widetilde{C}_{ik}^{(j)}$ into the corresponding full-rank block $C_{ik}^{(j)}$.

ALGORITHM 1: Dense BLR LDL^T (Right-looking) factorization: standard FSCU variant.

Input: A $p \times p$ block matrix F of order m; $F = [F_{ij}]_{i=1:p,j=1:p}$

for $k = 1$ **to** p **do**
  Factor: $F_{kk} = L_{kk} D_{kk} L_{kk}^T$
  for $i = k+1$ **to** p **do**
    Solve: $F_{ik} \leftarrow F_{ik} L_{kk}^{-T} D_{kk}^{-1}$
  end for
  for $i = k+1$ **to** p **do**
    Compress: $F_{ik} \approx \widetilde{F}_{ik} = X_{ik} Y_{ik}^T$
  end for
  for $i = k+1$ **to** p **do**
    for $j = k+1$ **to** i **do**
      Update F_{ij}:
      Inner Product: $\widetilde{C}_{ij}^{(k)} \leftarrow X_{ik}(Y_{ik}^T D_{kk} Y_{kj}) X_{kj}^T$
      Outer Product: $C_{ij}^{(k)} \leftarrow \widetilde{C}_{ij}^{(k)}$
      $F_{ij} \leftarrow F_{ij} - C_{ij}^{(k)}$
    end for
  end for
end for

2.3 Block Low-Rank multifrontal factorization

Because the multifrontal method relies on dense factorizations, the BLR approximations can be easily incorporated into the multifrontal factorization by representing the frontal matrices as BLR matrices as defined by Equation (2), and adapting Algorithm 1 to perform the partial factorization of the fronts.

In addition to the front factorization, one could also exploit the BLR approximations during the assembly. In this case, the factorization is called *fully-structured* [41] as the fronts are never stored in full-rank. In the BLR context, this corresponds to the CUFS variant [8], as explained in Section 6. The fully-structured factorization requires relatively complex low-rank extend-add operations and is out of the scope of this paper. In the experiments of this article, the assembly is therefore performed in full-rank.

To compute the BLR clustering while remaining in a purely algebraic context, we use the adjacency graph of the matrix A instead of the geometry. The clustering is computed with a k-way partitioning of the subgraph associated to the fully-summed variables of each front. A detailed description can be found in [3].

3 Experimental setting

The BLR factorization and all its variants have been developed and integrated into the general purpose symmetric and unsymmetric sparse multifrontal solver `MUMPS` [6], which was used to run all experiments and constitutes our reference Full-Rank solver.

The machines we used are listed in Table 1. All the experiments reported in this article, except those of Table 15, were performed on brunch, a machine equipped with 1.5 TB of memory and four Intel 24-cores Broadwell processors running at a frequency varying between 2.2 and 3.4 GHz, due to the turbo technology. We consider as peak per core the measured performance of the dgemm kernel with one core, 47.1 GF/s.

name	cpu model	np	nc	freq (GHz)	peak (GF/s)	bw (GB/s)	mem (GB)
brunch	E7-8890 v4	4	24	2.2−3.4*	47.1*	102	1500
grunch	E5-2695 v3	2	14	2.3	36.8	57	768

*frequency can vary due to turbo; peak is estimated as the dgemm peak

Table 1: List of machines used for Table 15 and their properties number of threads (nt), frequency (freq), peak performance, bandwidth (bw), and memory (mem).

Bandwidth is measured with the STREAM benchmark. For all experiments on brunch where several threads are used, the threads are scattered among the four processors to exploit the full bandwidth of the machine.

To validate our performance analysis, we report in Table 15 additional experiments performed on grunch, another machine with similar architecture but different properties (frequency and bandwidth), as described in Section 7.3. For the experiments on grunch, all 28 cores are used.

The above GF/s peak, as well as all the other GF/s values in this article, are computed counting flops in double-precision real (d) arithmetic, and assuming a complex flop corresponds to four real flops of the same precision.

3.1 Presentation of the test problems

In our experiments, we have used real life problems coming from three applications, as well as additional matrices coming from the SuiteSparse collection (previously named University of Florida Sparse Matrix Collection [17]). The complete set of matrices and their description is provided in Table 2. Each application is separated by a solid line while each problem subclass is separated by a dashed line.

Our first application is 3D seismic modeling. The main computational bulk of frequency-domain Full Waveform Inversion (FWI) [38] is the resolution of the forward problem, which takes the form of a large, single complex, sparse linear system. Each matrix corresponds to the finite-difference discretization of the Helmholtz equation at a given frequency (5, 7, and 10 Hz). In collaboration with the SEISCOPE consortium, we have shown in [4] how the use of BLR can reduce the computational cost of 3D FWI for seismic imaging on a real-life case-study from the North sea. We found that the biggest low-rank threshold ε for which the quality of the solution was still exploitable by the application was 10^{-3} [4] and this is therefore the value we chose for the experiments on these matrices.

Our second application is 3D electromagnetic modeling applied to marine Controlled-Source Electromagnetic (CSEM) surveying, a widely used method for detecting hydrocarbon reservoirs and other resistive structures embedded in conductive formations [16]. The matrices, arising from a finite-difference discretization of frequency-domain Maxwell equations, were used in [36] to carry out simulations over large-scale 3D resistivity models representing typical scenarios for the marine CSEM surveying. In particular, the S-matrices (S3, S21) correspond to the SEG SEAM model, a complex 3D earth model representative of the geology of the Gulf of Mexico. For this application, the biggest acceptable low-rank threshold is $\varepsilon = 10^{-7}$ [36].

Our third application is 3D structural mechanics, in the context of the industrial

application	matrix	ID	arith.	fact. type	n	nnz	flops	factor size
seismic modeling (SEISCOPE)	5Hz	1	c	LU	2.9M	70M	69.5 TF	61.4 GB
	7Hz	2	c	LU	7.2M	177M	471.1 TF	219.6 GB
	10Hz	3	c	LU	17.2M	446M	2.7 PF	728.1 GB
electromagnetic modeling (EMGS)	H3	4	z	LDL^T	2.9M	37M	57.9 TF	77.5 GB
	H17	5	z	LDL^T	17.4M	226M	2.2 PF	891.1 GB
	S3	6	z	LDL^T	3.3M	43M	78.0 TF	94.6 GB
	S21	7	z	LDL^T	20.6M	266M	3.2 PF	1.1 TB
structural mechanics (EDF Code_Aster)	perf008d	8	d	LDL^T	1.9M	81M	101.0 TF	52.6 GB
	perf008ar	9	d	LDL^T	3.9M	159M	377.5 TF	129.8 GB
	perf008cr	10	d	LDL^T	7.9M	321M	1.6 PF	341.1 GB
	perf009ar	11	d	LDL^T	5.4M	209M	23.6 TF	40.5 GB
computational fluid dynamics (SuiteSparse)	StocF-1465	12	d	LDL^T	1.5M	11M	4.7 TF	9.6 GB
	atmosmodd	13	d	LU	1.3M	9M	13.8 TF	16.7 GB
	HV15R	14	d	LU	2.0M	283M	1.9 PF	414.1 GB
structural problems (SuiteSparse)	Serena	15	d	LDL^T	1.4M	33M	31.6 TF	23.1 GB
	Geo_1438	16	d	LU	1.4M	32M	39.3 TF	41.6 GB
	Cube_Coup_dt0	17	d	LDL^T	2.2M	65M	98.9 TF	55.0 GB
	Queen_4147	18	d	LDL^T	4.1M	167M	261.1 TF	114.5 GB
DNA electrophoresis (SuiteSparse)	cage13	19	d	LU	0.4M	7M	80.1 TF	35.9 GB
	cage14	20	d	LU	1.5M	27M	4.1 PF	442.7 GB
optimization (SuiteSparse)	nlpkkt80	21	d	LDL^T	1.1M	15M	15.1 TF	14.4 GB
	nlpkkt120	22	d	LDL^T	3.5M	50M	248.4 TF	86.5 GB

Table 2: Complete set of matrices and their Full-Rank statistics

applications from Électricité De France (EDF). EDF has to guarantee the technical and economical control of its means of production and transportation of electricity. The safety and the availability of the industrial and engineering installations require mechanical studies, which are often based on numerical simulations. These simulations are carried out using Code_Aster[1] and require the solution of sparse linear systems such as the ones used in this paper. A previous study [39] showed that using BLR with $\varepsilon = 10^{-9}$ leads to an accurate enough solution for this class of problems.

To demonstrate the generality and robustness of our solver, we complete our set of problems with SuiteSparse matrices coming from different fields: computational fluid dynamics (CFD), structural mechanics and optimization. For these matrices, we have arbitrarily set the low-rank threshold to $\varepsilon = 10^{-6}$, except for the more difficult matrix nlpkkt120 where we used $\varepsilon = 10^{-9}$ (see Section 7).

For all experiments, we have used a right-hand side b such that the solution x is the vector containing only ones.

We provide in Section 7 experimental results on the complete set of matrices. For the sake of conciseness, the performance analysis in the main body of this paper (Sections 4 to 6) will focus on matrix S3.

Both the nested-dissection matrix reordering and the BLR clustering of the unknowns are computed with `METIS` in a purely algebraic way (i.e., without any knowledge of the geometry of the problem domain). For this set of problems, the time spent computing the BLR clustering is very small with respect to the time for analysis; its performance

[1] http://www.code-aster.org

	FR	BLR	ratio
flops ($\times 10^{12}$)	77.97	10.19	7.7
time (s)	7390.1	2241.9	3.3

Table 3: Sequential run (1 thread) on matrix S3

	FR				BLR			
step	flops ($\times 10^{12}$)	%	time (s)	%	flops ($\times 10^{12}$)	%	time (s)	%
Factor+Solve	1.51	1.9	671.0	9.1	1.51	14.9	671.0	29.9
Update	76.22	97.8	6467.0	87.5	7.85	77.0	1063.7	47.4
Compress	0.00	0.0	0.0	0.0	0.59	5.8	255.1	11.4
LAI parts	0.24	0.3	252.1	3.4	0.24	2.3	252.1	11.2
Total	77.97	100.0	7390.1	100.0	10.19	100.0	2241.9	100.0

Table 4: Performance analysis of sequential run of Table 3 on matrix S3

analysis is out of the scope of this article.

When threshold partial pivoting is performed during the FR and BLR factorizations, the Factor and Solve steps are merged together into a panel factorization operation. In order to improve the overall efficiency of the factorization, an internal blocking is used in the panel factorization to benefit from BLAS-3 kernels. The internal block size is set to 32 for all experiments.

In FR, the (external) panel size is constant and set to 128. In BLR, it is chosen to match the BLR cluster size (defined in Section 2.1). A good choice of block size should find a compromise between the number of operations and their speed (GF/s rate). That choice is automatically made according to the theoretical result reported in [8], which states that the block size should increase with the size of the fronts.

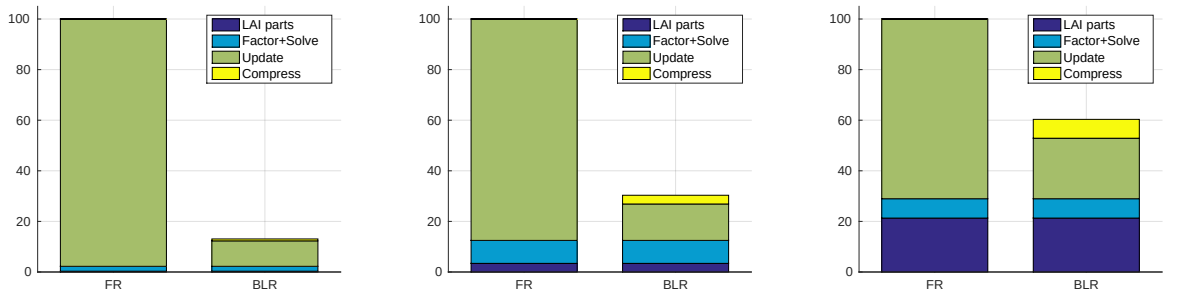
The threshold for partial pivoting is set to $\tau = 0.01$ for all experiments.

4 Performance analysis of sequential FSCU algorithm

In this section, we analyze the performance of the FSCU algorithm in a sequential setting. Our analysis underlines several issues, which will be addressed in subsequent sections.

In Table 3, we compare the number of flops and execution time of the sequential FR and BLR factorizations. While the use of BLR reduces the number of flops by a factor 7.7, the time is only reduced by a factor 3.3. Thus, the potential gain in terms of flops is not fully translated in terms of time.

To understand why, we report in Table 4 the time spent in each step of the factorization, in the FR and BLR cases. The relative weight of each step is also provided in percentage of the total. In addition to the four main steps Factor, Solve, Compress and Update, we also provide the time spent in parts with low arithmetic intensity (LAI parts). This includes the time spent in assembly, memory copies and factorization of the fronts at the bottom of the tree, which are too small to benefit from BLR and are thus treated in FR.



(a) Normalized flops (b) Normalized time (1 thread) (c) Normalized time (24 threads)

Figure 1: Normalized (FR = 100%) flops and time on matrix S3

The FR factorization is clearly dominated by the Update, which represents 87.5% of the total time. In BLR, the Update operations are done exploiting the low-rank property of the blocks and thus the number of operations performed in the Update is divided by a factor 9.7. The Factor+Solve and LAI steps remain in FR and thus do not change. From this result, we can identify three main issues with the performance of the BLR factorization:

Issue 1: lower granularity: the flop reduction by a factor 9.7 in the Update is not fully captured, as its execution time is only reduced by a factor 6.1. This is due to the lower granularity of the operations involved in low-rank products, which have thus a lower performance: the speed of the Update step is 47.1 GF/s in FR and 29.5 GF/s in BLR.

Issue 2: higher relative weight of the FR parts (Factor, Solve, and LAI parts): because the Update is reduced in BLR, the relative weight of the parts that remain FR increases from 12.5% to 41.1%. Thus, even if the Update step is further accelerated, one cannot expect the global reduction to follow as the FR part will become the bottleneck.

Issue 3: cost of the Compress step: even though the overhead cost of the Compress step is negligible in terms of flops (5.8% of the total), it is a very slow operation (9.2 GF/s) and thus represents a non-negligible part of the total time (11.4%).

A visual representation of this analysis is given on Figure 1 (compare Figures 1(a) and 1(b)).

In the next section, we first extend the BLR factorization to the multithreaded case, for which previous observations are even more critical. **Issues 1** and **2** will then be addressed by introducing algorithmic variants of the BLR factorization in Section 6. **Issue 3** is a topic of an article by itself; it is out of the scope of this article and we only comment on possible ways to reduce the cost of the Compress step in Section 8.2.

5 Multithreading the BLR factorization

In this section, we describe the shared-memory parallelization of the BLR factorization (Algorithm 1).

	FR	BLR	ratio
time (1 thread)	7390.1	2241.9	3.3
time (24 threads)	508.5	306.8	1.7
speedup	14.5	7.3	

Table 5: Multithreaded run on matrix S3

5.1 Performance analysis of multithreaded FSCU algorithm

Our reference Full-Rank implementation is based on a fork-join approach combining OpenMP directives with multithreaded BLAS libraries. While this approach can have limited performance on very small matrices, on the set of problems considered, it achieves quite satisfactory performance and speedups on 24 threads (around 20 for the largest problems) because the bottleneck consists of matrix-matrix product operations. It must be noted that better performance may be achieved using task-based parallelism as in state-of-the-art dense and sparse factorization libraries such as PLASMA [18] or qr_mumps [1]; however, for a fair and meaningful assessment of the gains that can be expected thanks to BLR approximations this would also require our BLR solver to be based on task parallelism and, as mentioned in the perspectives (Section 8.2), it raises open questions that are out of the scope of this article. Therefore, the fork-join approach will be taken as a reference for our performance analysis.

In the BLR factorization, the Update operations have a finer granularity and thus a lower speed and a lower potential for exploiting efficiently multithreaded BLAS. To overcome this obstacle, more OpenMP-based multithreading exploiting serial BLAS has been introduced. This allows for a larger granularity of computations per thread than multithreaded BLAS on low-rank kernels. In our implementation, we simply parallelize the loops of the Compress and Update operations on different blocks (lines 6, and 9-10) of Algorithm 1. The Factor+Solve step remains full-rank, as well as the FR factorization of the fronts at the bottom of the elimination tree. By skipping the compression step of the BLR code we obtain a "Block Full-Rank" variant relying on the same parallelization approach as the BLR one. We have observed that this "Block Full-Rank" variant is slower than the standard FR code based on multithreaded BLAS, and will therefore compare our BLR variants with the standard FR code.

Because each block has a different rank, the task load of the parallel loops is very irregular in the BLR case. To account for this irregularity, we use the dynamic OpenMP schedule (with a chunk size equal to 1), which achieves the best performance.

In Table 5, we compare the execution time of the FR and BLR factorization on 24 threads. The multithreaded FR factorization achieves a speedup of 14.5 on 24 threads. However, the BLR factorization achieves a much lower speedup of 7.3. The gain factor of BLR with respect to FR is therefore reduced from 3.3 to 1.7.

The BLR multithreading is thus less efficient than the FR one. To understand why, we provide in Table 6 the time spent in each step for the multithreaded FR and BLR factorizations. We additionally provide for each step the speedup achieved on 24 threads.

From this analysis, one can identify two additional issues related to the multithreading of the BLR factorization:

Issue 4: low arithmetic intensity parts become critical: the LAI parts expectedly achieve a very low speedup of 2.3. While their relative weight with respect to the total

	FR			BLR		
step	time	%	speedup	time	%	speedup
Factor+Solve	38.9	7.7	17.3	38.9	12.7	17.3
Update	361.2	71.0	17.9	121.6	39.6	8.8
Compress	0.0	0.0		37.9	12.4	6.7
LAI parts	108.4	21.3	2.3	108.4	35.3	2.3
Total	508.5	100.0	14.5	306.8	100.0	7.3

Table 6: Performance analysis of multithreaded run (24 threads) of Table 5 on matrix S3

remains reasonably limited in FR, it becomes quite significant in BLR, with over 35% of time spent in them. Thus, the impact of the poor multithreading of the LAI parts is higher on the BLR factorization than on the FR one.

Issue 5: scalability of the BLR Update: not only is the BLR Update less efficient than the FR one in sequential, it also achieves a lower speedup of 8.8 on 24 threads, compared to a FR speedup of 17.9. This comes from the fact that the BLR Update, due to its smaller granularities, is limited by the speed of memory transfers instead of the CPU peak as in FR. As a consequence, the Outer Product operation runs at the poor speed of 8.8 GF/s, to compare to 35.2 GF/s in FR.

A visual representation of this analysis is given on Figure 1 (compare Figures 1(b) and 1(c)).

In the rest of this section, we will revisit our algorithmic choices to address both of these issues.

5.2 Exploiting tree-based multithreading

In our standard shared-memory implementation, multithreading is exploited at the node parallelism level only, i.e. different fronts are not factored concurrently. However, in multifrontal methods, multithreading may exploit both node and tree parallelism. Such an approach has been proposed, in the FR context, by [27] and relies on the idea of separating the fronts by a so-called $\mathcal{L}_0$ layer, as illustrated in Figure 2. Each subtree rooted at the $\mathcal{L}_0$ layer is treated sequentially by a single thread; therefore, below the $\mathcal{L}_0$ layer pure tree parallelism is exploited by using all the available threads to process concurrently multiple sequential subtrees. When all the sequential subtrees have been processed, the approach reverts to pure node parallelism: all the fronts above the $\mathcal{L}_0$ layer are processed sequentially (i.e., one after the other) but all the available threads are used to assemble and factorize each one of them.

In Table 7, we quantify and analyze the impact of this strategy on the BLR factorization. The majority of the time spent in LAI parts is localized under the $\mathcal{L}_0$ layer. Indeed, all the fronts too small to benefit from BLR are under it; in addition, the time spent in assembly and memory copies for the fronts under the $\mathcal{L}_0$ layer represents 60% of the total time spent in the assembly and memory copies. Therefore, the LAI parts are significantly accelerated, by a factor over 2, by exploiting tree multithreading.

In addition, the other steps (the Update and especially the Compress) are also accelerated thanks to the improved multithreading behaviour of the relatively smaller BLR

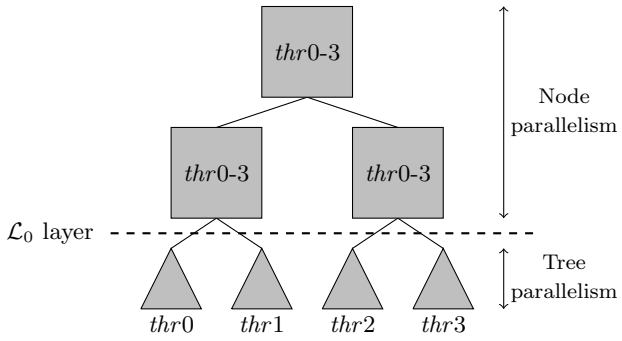


Figure 2: Illustration with four threads of how both node and tree multithreading can be exploited.

	FR			BLR		
step	time	%	speedup	time	%	speedup
Factor+Solve	33.2	7.9	20.2	33.2	15.1	20.2
Update	331.7	79.4	19.5	110.2	50.0	9.7
Compress	0.0	0.0		24.1	10.9	10.6
LAI parts	53.0	12.7	4.8	53.0	24.0	4.8
Total	417.9	100.0	17.4	220.5	100.0	10.2

Table 7: Execution time of FR and BLR factorizations on matrix S3 on 24 threads, exploiting both node and tree parallelism

fronts under the $\mathcal{L}_0$ layer which do not expose much node parallelism.

Please note that the relative gain due to introducing tree multithreading can be larger even in FR, for 2D or very small 3D problems, for which the relative weight of the LAI parts is important. However, for large 3D problems the relative weight of the LAI parts is limited, and the overall gain in FR remains marginal. In BLR, the weight of the LAI parts is much more important so that exploiting tree parallelism becomes critical: the overall gain is significant in BLR. We have thus addressed **Issue 4**, identified in Subsection 5.1.

The approach described in [27] additionally involves a so-called Idle Core Recycling (ICR) algorithm which consists in reusing the idle cores that have already finished factorizing their subtrees to help factorizing the subtrees assigned to other cores. This results in the use of both tree and node parallelism when the workload below the $\mathcal{L}_0$ layer is unbalanced.

The maximal potential gain of using ICR can be computed by measuring the difference between the maximal and average time spent under the $\mathcal{L}_0$ layer by the threads (this corresponds to the work unbalance). For the run of Table 7, the potential gain is equal to 3.3s in FR (i.e., 0.8% of the total) and 5.1s in BLR (i.e., 2.3% of the total). Thus, even though the potential gain in FR is marginal, it is higher in BLR, due to load inbalance generated by the irregularity of the compressions: indeed, the compression rate can greatly vary from front to front and thus from subtree to subtree.

Activating ICR brings a gain of 3.0s in FR and 4.7s in BLR; thus, roughly 90% of the potential gain is captured in both cases. While the absolute gain with respect to the

parallelism	step	FR RL	FR LL	BLR RL	BLR LL
1 thread	Update	6467.0	6549.8	1063.7	899.1
	Total	7390.1	7463.9	2241.9	2074.5
24 threads, node+tree//	Update	331.7	335.6	110.2	66.9
	Total	417.9	420.6	220.5	174.7

Table 8: Execution time of Right-looking and Left-looking factorizations on matrix S3

total is relatively small even in BLR, this analysis illustrates that Idle Core Recycling becomes an even more relevant feature for the multithreaded BLR factorization.

Exploiting tree multithreading is thus very critical in the BLR context. It will be used for the rest of the experiments for both FR and BLR.

5.3 Right-looking vs. Left-looking

Algorithm 1 has been presented in its Right-looking (RL) version. In Table 8, we compare it to its Left-looking (LL) equivalent, referred to as UFSC. The RL and LL variants perform the same operations but in a different order, which results in a different memory access pattern [19].

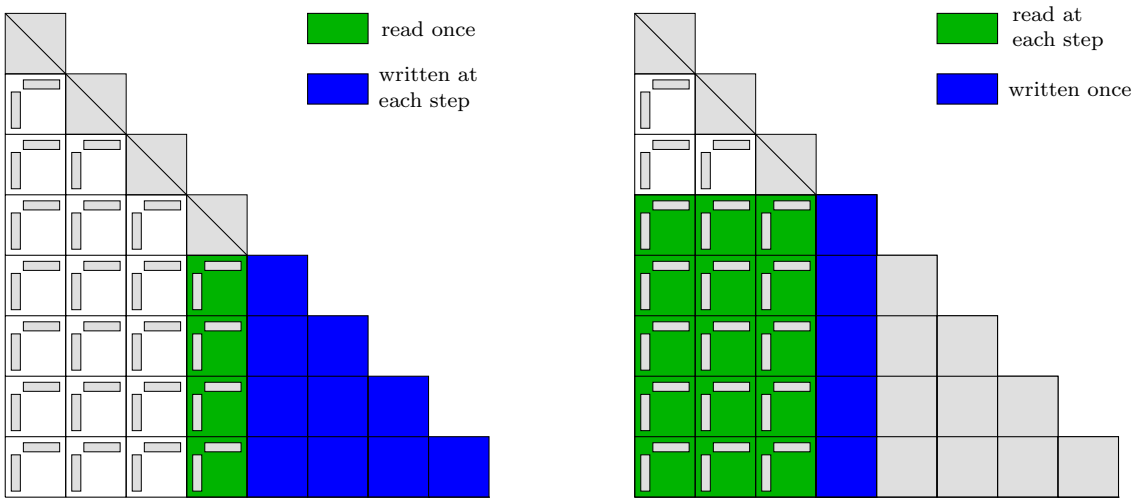
The impact of using a RL or LL factorization is mainly observed on the Update step. In FR, there is almost no difference between the two, RL being slightly (less than 1%) faster than LL. In BLR however, the Update is significantly faster in LL than in RL. This effect is especially clear on 24 threads (40% faster Update, which leads to a global gain of 20%).

We explain this result by a lower volume of memory transfers in LL BLR than RL BLR. As illustrated in Figure 3, during the BLR LDL^T factorization of a $p \times p$ block matrix, the Update will require loading the following blocks stored in main memory:

- in RL (Figure 3(a)), at each step k, the FR blocks of the trailing sub-matrix are written and therefore they are loaded many times (at each step of the factorization), while the LR blocks of the current panel are read once and never loaded again.
- in LL (Figure 3(b)), at each step k, the FR blocks of the current panel are written for the first and last time of the factorization, while the LR blocks of all the previous panels are read, and therefore they are loaded many times during the entire factorization.

Thus, while the number of loaded blocks is roughly the same in RL and LL (which explains the absence of difference between the RL FR and LL FR factorizations), the difference lies in the fact that the LL BLR factorization tends to load more often LR blocks and less FR blocks, while the RL one has the opposite behavior. To be precise:

- Under the assumption that one FR block and two LR blocks fit in cache, the LL BLR factorization loads $O(p^2)$ FR blocks and $O(p^3)$ LR blocks.
- Under the assumption that one FR block and an entire LR panel fit in cache (which is a stronger assumption so the number of loaded blocks may in fact be even worse), the RL BLR factorization loads $O(p^3)$ FR blocks and $O(p^2)$ LR blocks.



(a) RL factorization (b) LL factorization

Figure 3: Illustration of the memory access pattern in the RL and LL BLR Update during step k of the factorization of a matrix of $p \times p$ blocks (here, $p = 8$ and $k = 4$)

Thus, switching from RL to LL reduces the volume of memory transfers and therefore accelerates the BLR factorization, which addresses **Issue 5**, identified in Subsection 5.1.

Throughout the rest of this article, the best algorithm is considered: LL for BLR and RL for FR.

Thanks to both the tree multithreading and the Left-looking BLR factorization, the factor of gain due to BLR with respect to FR on 24 threads has increased from 1.7 (Table 5) to 2.4 (Table 8).

Next, we introduce algorithmic variants of the BLR factorization that further improve its performance.

6 BLR algorithmic variants

Thanks to the flexibility of the BLR format, it is possible to easily define variants of Algorithm 1. We present in Algorithm 2 the so-called UFCS+LUAR factorization. It consists of two main modifications of Algorithm 1, which are described in the following two subsections.

In [8], it was proved that they lead to a lower theoretical complexity; their performance has never been studied. In this section, we quantify the flop reduction achieved by these variants and how well this flop reduction can be translated into a time reduction. We analyze how they can improve the efficiency and scalability of the factorization.

6.1 LUAR: Low-rank Updates Accumulation and Recompression

The first modification is referred to as *Low-rank Updates Accumulation and Recompression* (LUAR). It consists in accumulating the update matrices $\widetilde{C}_{ik}^{(j)}$ together, as shown on line 6 of Algorithm 2:

$$\widetilde{C}_{ik}^{(acc)} := \widetilde{C}_{ik}^{(acc)} + \widetilde{C}_{ik}^{(j)}$$

Note that in the previous equation, the + sign denotes a low-rank sum operation. Specifically, if we note $A = C_{ik}^{(acc)}$ and $B = C_{ik}^{(j)}$, then

$$\widetilde{B} = \widetilde{C}_{ik}^{(j)} = X_{ij}(Y_{ij}^T D_{jj} Y_{jk}) X_{jk}^T = X_B C_B Y_B^T$$

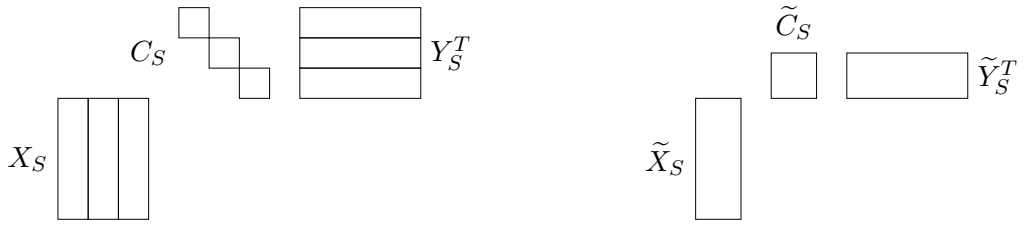
with $X_B = X_{ij}$, $C_B = Y_{ij}^T D_{jj} Y_{kj}$, and $Y_B = X_{kj}$. Similarly, $\widetilde{A} = \widetilde{C}_{ik}^{(acc)} = X_A C_A Y_A^T$. Then the low-rank sum operation is defined by:

$$\widetilde{A} + \widetilde{B} = X_A C_A Y_A^T + X_B C_B Y_B^T = \begin{pmatrix} X_A & X_B \end{pmatrix} \begin{pmatrix} C_A & \\ & C_B \end{pmatrix} \begin{pmatrix} Y_A & Y_B \end{pmatrix}^T = X_S C_S Y_S^T = \widetilde{S}$$

where $\widetilde{S}$ is a low-rank approximant of $S = A + B$.

This algorithm has two advantages: first, accumulating the update matrices together leads to higher granularities in the Outer Product step (line 9 of Algorithm 2), which is thus performed more efficiently. This should address **Issue 1**, identified in Section 4. Second, it allows for additional compression, as the accumulated updates $\widetilde{C}_{ik}^{(acc)}$ can be recompressed (as shown on line 8) before the Outer Product. A visual representation is given in Figure 4.

Note that there are several strategies to recompress the accumulated updates, which have been analyzed in [11]. On Figure 4(a), X_S, C_S, and Y_S can all be recompressed. In our experiments, we have observed that recompressing C_S only is the best strategy as, due to the small size of C_S, it leads to a lower Recompress cost while capturing most of the recompression potential.



(a) Accumulated updates before recompression

(b) Accumulated updates after recompression

Figure 4: Low-rank Updates Accumulation and Recompression

In Table 9, we analyze the performance of the UFSC+LUAR variant. We separate the gain due to accumulation (UFSC+LUA, without recompression) and the gain due to the recompression (UFSC+LUAR). We provide the flops, time and speed of both the Outer Product (which is the step impacted by this variant) and the total (to show the global gain). We also provide the average (inner) size of the Outer Product operation, which corresponds to the rank of $\widetilde{C}_{ik}^{(acc)}$ on line 9 in Algorithm 2. It also corresponds to the number of columns of X_S and Y_S in Figure 4.

Thanks to the accumulation, the average size of the Outer Product increases from 16.5 to 61.0. As illustrated by Figure 5, this higher granularity improves the speed of the Outer Product from 29.3 to 44.7 GF/s (compared to a peak of 47.1 GF/s) and thus accelerates it by 35%. The impact of accumulation on the total time depends on both the matrix and the computer properties and will be further discussed in Section 7.

ALGORITHM 2: Dense BLR LDL^T (Left-looking) factorization: UFCS+LUAR variant.

Input: A $p \times p$ block matrix F of order m; $F = [F_{ij}]_{i=1:p,j=1:p}$

for $k = 1$ **to** p **do**
  for $i = k$ **to** p **do**
    Update F_{ik}:
    for $j = 1$ **to** $k-1$ **do**
      Inner Product: $\widetilde{C}_{ik}^{(j)} \leftarrow X_{ij}(Y_{ij}^T D_{jj} Y_{kj}) X_{kj}^T$
      Accumulate update: $\widetilde{C}_{ik}^{(acc)} \leftarrow \widetilde{C}_{ik}^{(acc)} + \widetilde{C}_{ik}^{(j)}$
    end for
    $\widetilde{C}_{ik}^{(acc)} \leftarrow \mathsf{Recompress}(\widetilde{C}_{ik}^{(acc)})$
    $C_{ik}^{(acc)} \leftarrow \mathsf{Outer\ Product}(\widetilde{C}_{ik}^{(acc)})$
    $F_{ik} \leftarrow F_{ik} - C_{ik}^{(acc)}$
  end for
  Factor: $F_{kk} = L_{kk} D_{kk} L_{kk}^T$
  for $i = k+1$ **to** p **do**
    Compress: $F_{ik} \approx \widetilde{F}_{ik} = X_{ik} Y_{ik}^T$
  end for
  for $i = k+1$ **to** p **do**
    Solve: $\widetilde{F}_{ik} \leftarrow \widetilde{F}_{ik} L_{kk}^{-T} D_{kk}^{-1} = X_{ik}(Y_{ik}^T L_{kk}^{-T} D_{kk}^{-1})$
  end for
end for

Next, we analyze the gain obtained by recompressing the accumulated low-rank updates (Figure 4(b)). While the total flops are reduced by 20%, the execution time is only accelerated by 5%. This is partly due to the fact that the Outer Product only represents a small part of the total, but could also come from two other reasons:

- The recompression decreases the average size of the Outer Product back to 32.8. As illustrated by Figure 5, its speed remains at 44.4 GF/s and is thus not significantly decreased, but it can be the case for other matrices or machines.
- The speed of the Recompress operation itself is 0.7 GF/s, an extremely low value. Thus, even though the Recompress overhead is negligible in terms of flops, it can limit the global gain in terms of time. Here, the time overhead is 1.2s for an 8s gain, i.e. 15% overhead.

We also report in Table 9 the scaled residual $\frac{\|Ax-b\|_\infty}{\|A\|_\infty \|x\|_\infty}$, which on this test matrix is unchanged by the LUAR algorithm.

6.2 UFCS algorithm

In all the previous experiments, threshold partial pivoting was performed during the FR and BLR factorizations, which means the Factor and Solve steps were merged together as described in Section 3. For many problems, numerical pivoting can be restricted to a smaller area of the panel (for example, the diagonal BLR blocks). In this case, the Solve step can be separated from the Factor step and applied directly on the entire panel, thus solely relying on BLAS-3 operations.

Furthermore, in BLR, when numerical pivoting is restricted, it is natural and more efficient to perform the Compress before the Solve (thus leading to the so-called UFCS

			UFSC	+LUA	+LUAR
average size of Outer Product			16.5	61.0	32.8
flops	($\times 10^{12}$)	Outer Product	3.76	3.76	1.59
	($\times 10^{9}$)	Recompress	0.00	0.00	5.39
	($\times 10^{12}$)	Total	10.19	10.19	8.15
time (s)		Outer Product	21.4	14.0	6.0
		Recompress	0.0	0.0	1.2
		Total	174.7	167.1	160.0
speed (GF/s)		Outer Product	29.3	44.7	44.4
		Recompress			0.7
		Total	9.7	10.2	8.5
scaled residual			1.5e-09	1.5e-09	1.5e-09

Table 9: Performance analysis of the UFSC+LUAR factorization on matrix S3 on 24 threads

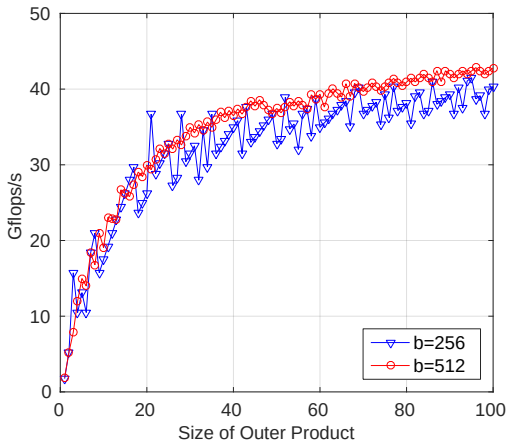


Figure 5: Performance benchmark of the Outer Product step on brunch. Please note that the average sizes (first line) and speed values (eighth line) of Table 9 cannot be directly linked using this figure because the average size would need to be weighted by its number of flops.

factorization). Indeed UFCS makes further use of the low-rank property of the blocks since the Solve step can then be performed in low-rank as shown on line 17 in Algorithm 2. Note that for the matrices where pivoting cannot be restricted, we briefly discuss possible extensions to pivoting strategies in Section 8.2.

In Table 10, we report the gain achieved by UFCS and its accuracy, measured by the scaled residual $\frac{\|Ax-b\|_\infty}{\|A\|_\infty\|x\|_\infty}$. We first compare the factorization with either standard or restricted pivoting. Restricting the pivoting allows the Solve to be performed with more BLAS-3 and thus the factorization is accelerated. This does not degrade the solution because on this test matrix restricted pivoting is enough to preserve accuracy.

	standard pivoting		restricted pivoting		
	FR	UFSC +LUAR	FR	UFSC +LUAR	UFCS +LUAR
flops ($\times 10^{12}$)	77.97	8.15	77.97	8.15	3.95
time (s)	417.9	160.0	401.3	140.4	110.7
scaled residual	4.5e-16	1.5e-09	5.0e-16	1.9e-09	2.7e-09

Table 10: Performance and accuracy of UFSC and UFCS variants on 24 threads on matrix S3

We then compare UFSC and UFCS (with LUAR used in both cases). The flops for the UFCS factorization are reduced by a factor 2.1 with respect to UFSC. This can at first be surprising as the Solve step represents less than 20% of the total flops of the UFSC factorization.

To explain the relatively high gain observed in Table 10, we analyze in detail the difference between UFSC and UFCS in Table 11. By performing the Solve in low-rank, we reduce its number of operations of the Factor+Solve step by a factor 4.2, which translates to a time reduction of this step by a factor of 1.9. Furthermore, the flops of the Compress and Update steps are also significantly reduced, leading to a time reduction of 15% and 35%, respectively. This is because the Compress is performed earlier, which decreases the ranks of the blocks. On our test problem, the average rank decreases from 21.6 in UFSC to 16.2 in UFCS, leading a very small relative increase of the scaled residual. The smaller ranks also lead to a smaller average size of the Outer Product, which decreases from 32.8 (last column of Table 9) to 24.4. This makes the LUAR variant even more critical when combined with UFCS: with no accumulation, the average size of the Outer Product in UFCS would be 10.9 (to compare to 16.5 in UFSC, first column of Table 9).

	flops ($\times 10^{12}$)		time (s)	
	UFSC	UFCS	UFSC	UFCS
Factor+Solve	1.52	0.36	12.4	6.6
Update	5.78	2.93	53.4	34.0
Compress	0.62	0.43	24.1	20.4
LAI parts	0.24	0.24	50.5	49.7
Total	8.15	3.95	140.4	110.7

Table 11: Detailed analysis of UFSC and UFCS results of Table 10 on matrix S3

Finally, note that it is possible to define a last variant, so-called CUFS [8], where the Compress is performed even earlier, before the Update. Since we perform the Solve in low-rank, we don't need to decompress the update matrices of the low-rank off-diagonal blocks. Thus, we can further reduce the cost of the factorization by keeping the recompressed accumulated updates $\widetilde{C}_{ik}^{(acc)}$ as the low-rank representation of the block F_{ik}, and thus suppress the Outer Product (line 9 of Algorithm 2). However, in a multifrontal context, this requires the assembly (or extend-add) operations to be performed on low-rank blocks, which is out of the scope of this paper.

Thanks to both the LUAR and UFCS variants, the factor of gain due to BLR with respect to FR on 24 threads has increased from 2.4 (Table 8) to 3.6 (Table 10).

7 Complete set of results

This section serves two purposes. First, we show that the results and the analysis reported on a representative matrix on a given computer hold for a large number of matrices coming from a variety of real-life applications and in different multicore environments. Second, we will further comment on specificities that depend on the matrix or machine properties.

The results on the matrices coming from the three real-life applications from SEISCOPE, EMGS and EDF (described in Section 3.1) are reported in Table 12. To demonstrate the generality and robustness of our solver, these results are completed with those of Table 13 on matrices from the SuiteSparse collection. We summarize the main results of Tables 12 and 13 with a visual representation in Figure 6. Then, for the biggest problems, we report in Table 14 results obtained using 48 threads instead of 24. We recall that the test matrices are described and assigned an ID in Table 2.

7.1 Results on the complete set of matrices

We report the flops and time on 24 threads for all variants of the FR and BLR factorizations and report the speedup and scaled residual $\frac{\|Ax-b\|_\infty}{\|A\|_\infty\|x\|_\infty}$ for the best FR and BLR variants. The scaled residual in FR is taken as a reference. In BLR, the scaled residual also depends on the low-rank threshold ε (whose choice of value is justified in Section 3). One can see in Tables 12 and 13 that in BLR the scaled residual correctly reflects the influence of the low-rank approximations with threshold ε on the FR precision. Matrix nlpkkt120 (matrix ID 22) is a numerically difficult problem for which the FR residual (1.9e-08) is several digits lower than the machine precision; on this matrix the low-rank threshold is set to a smaller value (10^{-9}) to preserve a scaled residual comparable to those obtained with the other matrices from the SuiteSparse collection.

On this set of problems, BLR always reduces the number of operations with respect to FR by a significant factor. This factor is never fully translated in terms of time, but the time gains remain important, even for the smaller problems.

Tree parallelism (tree//), the Left-looking factorization (UFSC) and the accumulation (LUA) always improve the performance of the BLR factorization. For some smaller problems where the factorization of the fronts at the bottom of the elimination tree represents a considerable part of the total computations, such as StocF-1465 and atmosmodd (matrix ID 12 and 13), exploiting tree parallelism is especially critical, even in FR.

Even though the recompression (LUAR) is always beneficial in terms of flops, it is not always the case in terms of time. Especially for the smaller problems, the low speed of the

low-rank threshold ε		10^{-3}			10^{-7}				10^{-9}			
matrix ID		1	2	3	4	5	6	7	8	9	10	11
flops ($\times 10^{12}$)	FR	69.5	471.1	2703.0	57.9	2188.0	78.0	3119.0	101.0	377.5	1616.0	23.6
	BLR	9.3	48.4	222.8	10.4	159.2	10.2	163.3	21.4	55.2	157.2	5.6
	+ LUAR	7.0	34.4	146.1	8.3	95.2	8.1	105.6	17.7	43.3	110.2	5.2
	+ UFCS	6.4	34.3	100.1	3.7	53.1	3.9	48.1	15.9	37.6	93.5	—
	flop ratio*	**10.8**	**13.7**	**27.0**	**15.8**	**41.2**	**19.7**	**64.8**	**6.4**	**10.0**	**17.3**	**4.2**
time (24 threads)	FR	235.2	1295.9	6312.5	376.4	10089.4	508.5	14362.3	211.7	662.2	2272.1	166.7
	+ tree//	196.2	1100.0	5844.8	321.4	9779.2	417.9	13979.7	174.3	577.3	2145.0	77.6
	+ rest. piv.	163.2	1013.0	5649.5	304.2	9655.6	401.3	13842.7	163.8	544.1	2066.9	—
	BLR	146.2	537.7	1998.8	229.0	2967.5	306.8	3702.9	161.5	416.7	1129.0	151.5
	+ tree//	92.8	373.6	1497.3	161.2	2586.7	220.5	3165.9	115.6	313.0	944.5	56.3
	+ UFSC	88.1	347.7	1334.3	150.1	1688.4	174.7	1971.6	99.3	245.2	632.9	50.0
	+ LUA	84.0	327.5	1245.4	145.6	1643.7	167.1	1856.6	97.3	232.2	570.2	49.1
	+ LUAR	91.0	362.5	1196.9	138.3	1509.4	160.0	—	92.7	216.5	515.8	79.6
	+ UFCS	49.7	194.8	773.7	91.0	652.7	110.7	736.1	78.2	176.7	377.9	—
	time ratio*	**3.3**	**5.2**	**7.3**	**3.3**	**14.8**	**3.6**	**18.8**	**2.1**	**3.1**	**5.5**	**1.6**
speedup (24 threads)	Best FR	17.8	18.8	—	17.7	—	18.2	—	15.9	17.5	—	14.2
	Best BLR	11.3	13.8	12.6	9.6	11.5	9.0	12.2	10.8	12.2	13.5	12.3
scaled residual	Best FR	1.7e-04	3.5e-04	2.9e-04	3.7e-16	7.0e-16	5.0e-16	8.1e-16	9.1e-15	5.2e-15	7.1e-15	1.4e-15
	Best BLR	3.1e-02	2.9e-02	4.2e-02	3.1e-10	2.8e-10	2.7e-09	2.0e-10	1.4e-08	3.7e-08	5.0e-08	4.5e-13

*between best FR and best BLR

Table 12: Experimental results on real-life matrices from SEISCOPE, EMGS, and EDF

low-rank threshold ε		10^{-6}			10^{-6}				10^{-6}		10^{-6}	10^{-9}
matrix ID		12	13	14	15	16	17	18	19	20	21	22
flops ($\times 10^{12}$)	FR	4.7	13.8	1872.0	31.6	39.3	98.9	261.1	80.1	4066.0	15.1	248.4
	BLR	0.4	1.2	216.2	5.0	2.0	14.7	21.5	9.9	161.8	1.9	22.8
	+ LUAR	0.4	1.0	173.1	3.9	1.9	12.8	18.8	7.9	111.5	1.7	17.6
	+ UFCS	0.2	0.8	133.9	4.1	1.8	12.7	14.6	6.1	75.5	—	—
	flop ratio*	**27.0**	**17.1**	**14.0**	**7.7**	**22.3**	**7.8**	**17.9**	**13.1**	**53.8**	**7.8**	**10.9**
time (24 threads)	FR	31.0	36.7	2349.1	81.0	85.5	210.2	513.1	123.6	4930.4	54.9	592.2
	+ tree//	15.2	27.7	2247.9	60.8	65.7	168.5	420.3	114.8	4864.1	37.0	523.2
	+ rest. piv.	12.8	23.4	2237.5	56.9	60.2	158.1	388.7	109.6	4626.2	—	—
	BLR	26.8	28.8	1019.6	66.3	52.2	153.1	321.0	76.6	1393.8	42.1	334.2
	+ tree//	10.1	18.5	840.8	43.3	27.9	105.2	202.3	67.9	1321.1	22.9	263.3
	+ UFSC	10.2	16.6	729.1	36.8	29.0	87.7	161.9	58.8	817.1	21.7	226.3
	+ LUA	10.9	16.6	697.3	35.2	29.6	82.1	152.8	55.0	734.2	20.9	215.6
	+ LUAR	10.5	18.1	681.4	35.1	28.7	80.9	151.2	53.6	662.2	24.7	228.8
	+ UFCS	7.5	11.4	565.0	28.7	20.1	68.5	106.1	43.4	517.3	—	—
	time ratio*	**1.7**	**2.1**	**4.0**	**2.0**	**3.0**	**2.3**	**3.7**	**2.5**	**8.9**	**1.8**	**2.4**
speedup (24 threads)	Best FR	10.9	14.1	—	14.5	15.2	15.8	17.0	16.3	—	17.7	20.4
	Best BLR	5.1	7.1	12.4	7.8	7.1	10.1	8.6	8.6	10.4	12.4	13.6
scaled residual	Best FR	1.6e-16	1.5e-15	1.8e-14	1.2e-15	1.8e-16	2.4e-15	2.7e-16	1.6e-15	3.4e-15	6.6e-12	1.9e-08
	Best BLR	1.9e-09	1.4e-04	6.5e-08	7.9e-07	6.9e-07	1.1e-08	2.1e-08	2.0e-05	4.1e-05	6.4e-04	1.7e-04

*between best FR and best BLR

Table 13: Experimental results on real-life matrices from the UFSMC

computations may lead to slowdowns. When LUAR is not beneficial (in terms of time), the "+UFCS" lines in Tables 12 and 13 correspond to a UFCS factorization without Recompression (LUA only). For all problems, the LUAR algorithm obtained a scaled residual of the same order of magnitude as the one obtained without recompression.

For most of the problems, the UFCS factorization obtained a scaled residual of the same order of magnitude as the one obtained by UFSC. This was the case even for some matrices where pivoting cannot be suppressed, but can be restricted to the diagonal BLR blocks, such as perf008{d,ar,cr} (matrix ID 8-10). Only for problems perf009ar and nlpkkt{80,120} (matrix ID 11 and 21-22), standard threshold pivoting was needed to preserve accuracy and thus the restricted pivoting and UFCS results are not available. To further improve the performance of this class of matrices and as mentioned in Sections 6.2 and 8.2, pivoting on the low-rank blocks could have been performed. This will be the object of future work.

We now analyze how these algorithmic variants evolve with the size of the matrix, by comparing the results on matrices of different sizes from the same problem class, such as perf008{d,ar,cr} (matrix ID 8-10) or {5,7,10}Hz (matrix ID 1-3). Tree parallelism becomes slightly less critical as the matrix gets bigger, due to the decreasing weight of the bottom of the elimination tree. On the contrary, improving the efficiency of the BLR factorization (UFSC+LUA variant, with reduced memory transfers and increased granularities) becomes more and more critical (e.g., 16% gain on perf008d compared to 40% gain on perf008cr). Both the gains due to the Recompression (LUAR) and the Compress before Solve (UFCS) increase with the problem size (e.g., 20% gain on perf008d compared to 34% gain on perf008cr), which is due to the improved complexity of these variants [8].

We also analyze the parallel efficiency of the FR and BLR factorization by reporting the speedup on 24 threads. The speedup achieved in FR for the small and medium problems is of 16.4 in average and up to 20.4. As for the biggest problems, they would take too long to run in sequential in FR; this is indicated by a "—" in the corresponding row of Tables 12 and 13. However, for these problems, we can estimate the speedup assuming they would run at the same speed as the fastest problem of the same class that can be run in sequential. Under this assumption (which is conservative because the smaller problems already run very close to the CPU peak speed), these big problems all achieve a speedup close to or over 20. Overall, it shows that our parallel FR solver is a good reference to be compared with.

The speedups achieved in BLR are lower than in FR, but they remain satisfactory, averaging at 10.5 and reaching up to 13.8, and leading to quite interesting overall time ratios between the best FR and the best BLR variants. It is worthy to note that bigger problems do not necessarily lead to better speedups than smaller ones, because they achieve higher compression and thus lower efficiency.

We summarize the main results of Tables 12 and 13 with a visual representation in Figure 6. We compare the time using 24 threads for four versions of the factorization: reference (ref.) FR and BLR, and improved (impr.) FR and BLR. Reference versions correspond to the initial versions of the factorization with only node parallelism, standard partial threshold pivoting and the standard FSCU variant for the BLR factorization. The improved FR version exploits tree parallelism and restricts numerical pivoting when possible. The improved BLR version additionally uses a UFCS factorization with accumulation (LUA), and possibly recompression (LUAR, only when beneficial). While the time ratio between the reference FR and BLR versions is only of 1.9 in average (and up

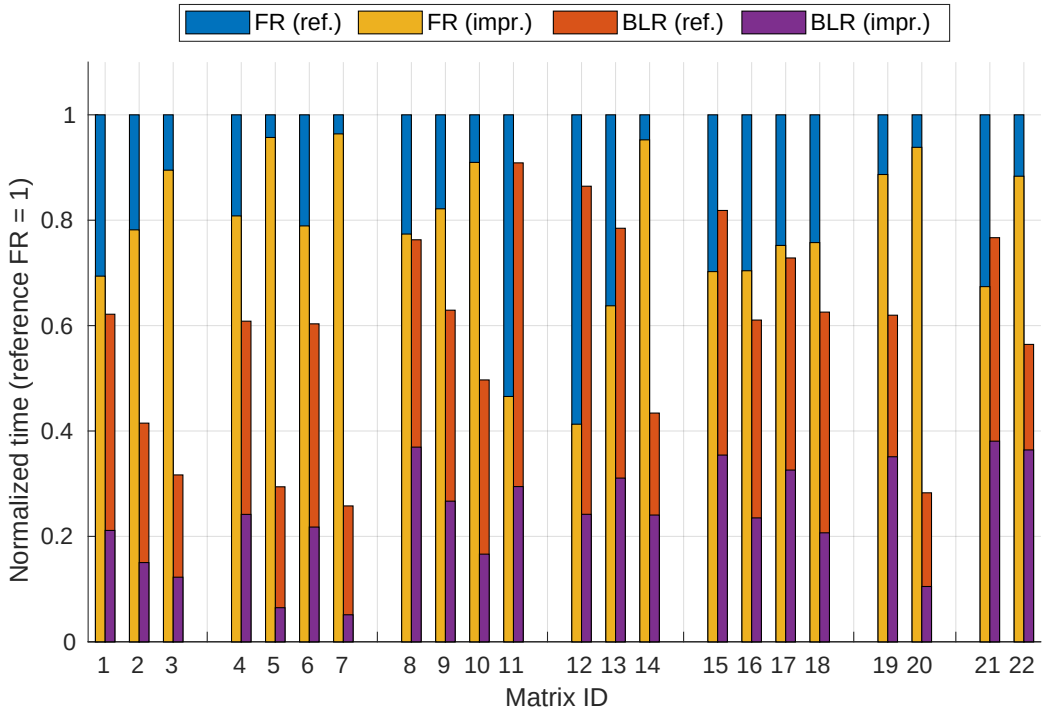


Figure 6: Visual representation of summarized results of Tables 12 and 13 (ref.: reference; impr.: improved).

to 6.9), that of the improved versions is of 4.6 in average (and up to 18.8).

7.2 Results on 48 threads

Next, we report in Table 14 the results obtained using 48 threads on brunch. For these experiments, we have selected the biggest of our test problems: 10Hz, H17, S21, and perf008cr (matrix ID 3, 5, 7, and 10). On these big problems, we compute the speedup obtained using 48 threads with respect to 24 threads (and thus, the optimal speedup value is 2). With the reference FR factorization, a speedup between 1.51 and 1.71 is achieved, which is quite satisfactory. The improved FR version, thanks to tree parallelism and restricted pivoting, increases the speedup to between 1.53 and 1.73, a relatively minor improvement.

The results are quite different for the BLR factorization. The speedup achieved by the reference version is much smaller: between 1.13 and 1.36, which illustrates that exploiting a high number of cores in BLR is a challenging problem. We then distinguish two types of improvements of the BLR factorization:

- The improvements that increase its scalability: tree parallelism but also the UFSC (i.e., Left-looking) factorization (due to a lower volume of memory accesses) and the LUA accumulation (due to increased granularities). All these changes combined lead to a major improvement of the achieved speedup, between 1.18 and 1.53, and illustrate the ability of the improved BLR factorization to scale reasonably well,

matrix ID	time (48 threads) 3	5	7	10	speedup w.r.t 24 threads 3	5	7	10
FR	4100.5	5949.8	8387.0	1508.7	1.54	1.70	1.71	1.51
+ tree// + rest. piv.	3402.1	5599.0	7987.4	1350.4	1.66	1.72	1.73	1.53
BLR	1764.9	2478.3	3142.7	828.8	1.13	1.20	1.18	1.36
+ tree// + UFSC + LUA	1056.7	1071.3	1234.3	389.9	1.18	1.53	1.50	1.46
+ LUAR + UFCS	590.1	519.0	604.9	328.2	1.31	1.26	1.22	1.15
ratio best FR/best BLR	**6.1**	**10.8**	**13.2**	**4.1**				

Table 14: Results on 48 threads

even on higher numbers of cores.

- The improvements that increase its compression: the recompression (LUAR) and the UFCS factorization. By decreasing the number of operations, these changes may degrade the scalability of the factorization. This explains why the achieved speedup may be lower than that of the UFSC+LUA variant, or sometimes even that of the reference BLR version. Despite this observation, these changes do reduce the time by an important factor and illustrate the ability of the improved BLR factorization to achieve significant gains, even on higher numbers of cores.

7.3 Impact of bandwidth and frequency on BLR performance

In this Section, we report additional experiments performed on two machines and analyze the impact of their properties on the performance.

The machines and their properties are listed in Table 1. brunch is the machine used for all previous experiments. grunch is a machine with very similar architecture but with lower frequency and bandwidth.

In Table 15, we compare the results obtained on brunch and grunch. We report the execution time of the BLR factorization in Right-looking (RL), Left-looking (LL), and with the LUA variant. On brunch, as observed and analyzed in Sections 5.3 and 6.1, the gain due to the LL factorization is significant while that of the LUA variant is limited. However, on grunch, we have the opposite effect, the difference between RL and LL is limited while the gain due to LUA is significant.

machine	RL	LL	LUA
brunch	220.5	174.7	167.1
grunch	247.7	228.3	196.8

Table 15: Time (s) for BLR factorization on matrix S3 (on 24 threads on brunch and 28 threads on grunch)

These results can be qualitatively analyzed using the Roofline Model [40]. This model provides an upper bound for the speed of an operation as a function of its arithmetic intensity, defined as the ratio between number of operations and number of memory transfers, the memory bandwidth and the CPU peak performance:

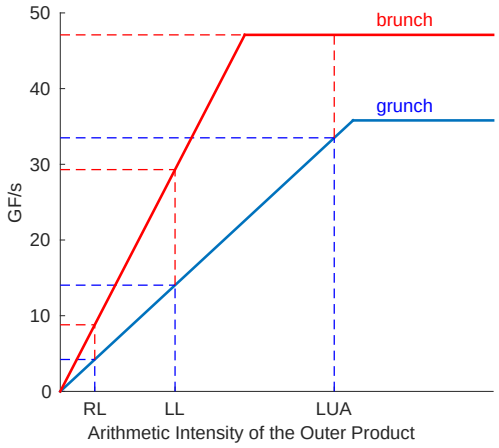


Figure 7: Roofline model analysis of the Outer Product operation.

$$\text{Attainable GF/s} = min \begin{cases} \text{Peak Floating-point Performance} \\ \text{Peak Memory Bandwidth} \times \text{Operational intensity} \end{cases}$$

The Roofline Model is plotted for the grunch and brunch machines in Figure 7 considering the bandwidth and CPU peak performance values reported in Table 1. Algorithms whose arithmetic intensity lies on the slope of the curve are commonly referred to as *memory-bound* because their performance is limited by the speed at which data can be transferred from memory; those whose arithmetic intensity lies on the plateau are referred to as *compute-bound* and can get close to the peak CPU speed.

Although it is very difficult to compute the exact arithmetic intensity for the algorithms presented above, the following relative order can be established:

- because of the unsuitable data access pattern (as explained in Section 5.3) and the low granularity of operations, the RL method is memory bound as proved by the fact that the Outer Product operation runs, on brunch, at the poor speed of 8.8 GF/s;
- as explained in Section 5.3, the LL method does the same operations as the RL one but in a different order which results is a lower volume of memory transfers. Consequently, the LL method enjoys a higher arithmetic intensity although it is still memory bound as shown by the fact that the Outer Product operation runs, on brunch, at 29.3 GF/s (see Table 9) which is still relatively far from the CPU peak;
- the LUA method is based on higher granularity operations; this likely allows for a better use of cache memories within BLAS operations which ultimately results in an increased arithmetic intensity; in conclusion the LUA method is compute-bound (or very close to) as shown by the fact that the Outer Product runs at 44.7 GF/s on brunch (see Table 9).

This leads to the following interpretation of the results of Table 15. Compared to grunch, brunch has a higher bandwidth; this translates by a steeper curve in the memory-

bound area of the roofline figure. As a consequence, the difference between the RL and LL algorithms (which are both memory-bound) is greater on brunch than on grunch. However, the higher bandwidth also makes the LL factorization closer to being compute-bound on brunch than on grunch. Therefore, the difference between LL and LUA (for which the Outer Product is compute-bound) is greater on grunch.

8 Conclusion

8.1 Summary

We have presented a multithreaded Block Low-Rank factorization for shared-memory multicore architectures.

We have first identified challenges of multithreaded performance in the use of BLR approximations within multifrontal solvers. This has motivated us to both revisit the algorithmic choices of our Full-Rank Right-looking solver based on node parallelism, and also to introduce algorithmic variants of the BLR factorization.

Regarding the algorithmic changes for the FR factorization, even though exploiting tree parallelism brings only a marginal gain in FR, we have shown that it is critical for the BLR factorization. This is because the factorization of the fronts at the bottom of the elimination tree is of much higher weight in BLR. We have then observed that, contrarily to the FR case, the Left-looking BLR factorization outperforms the Right-looking one by a significant factor. We have shown that it is due to a lower volume of memory transfers.

Regarding the BLR algorithmic variants, firstly we have shown that accumulating together the low-rank updates (so-called LUA algorithm) improves the granularity and the performance of the BLAS kernels. This approach also offers potential for recompression (so-called LUAR algorithm) which can often be translated into time reduction. Secondly, for problems on which the constraint of numerical pivoting can be relaxed, we have presented the UFCS variant which improves both the efficiency and compression rate of the factorization.

8.2 Perspectives

We briefly discuss remaining challenges and open questions that could be the object of further research.

A task-based multithreading could further improve the performance of the factorization; this approach (described, for example, in [11], [35], [2]) would allow for a pipelining of the successive stages of the factorization of each frontal matrix as opposed to the fork-join approach hereby presented. However, the taskification of the BLR factorization is not straightforward as it raises two questions: how to control the memory consumption; and how much of the gain due to the Left-looking factorization, which also makes possible the accumulation and recompression of low-rank updates, can be preserved?

We have shown that, compared to the FR factorization, the BLR factorization has a lower granularity of operations and is more memory-bound. These two issues will be even more critical in the context of accelerators such as GPUs or MICs which require larger granularities and higher amounts of parallelism.

Moreover, as mentioned in Section 4, the cost of the Compress step is not negligible in terms of time. With all the improvements proposed in this paper, this observation becomes even more critical: the Compress is close to being the bottleneck for several

problems. We leave the performance analysis of this step for future work. In particular, alternative compression kernels could be investigated, such as randomized QR with column pivoting [28, 26] or Adaptive Cross Approximation (ACA) [12].

Finally, as mentioned in Section 6.2, the threshold partial pivoting strategy needs to be extended for the UFCS variant. Assuming that QR with column pivoting is used for off-diagonal block compression, the quality of a candidate pivot could be estimated with respect to the column entries of the R matrices of the low-rank off-diagonal blocks. Strategies close to those suggested in [21] for distributed-memory settings, where off-diagonal blocks are not available locally, could also be applied.

8.3 Extension to distributed-memory

The extension of the BLR factorization to distributed-memory architectures is an ongoing effort which is out of the scope of this paper. We briefly indicate a few additional issues that should be addressed.

In a distributed-memory environment, the unpredictability of BLR compressions raises the difficulty of load balancing work between MPI processes. Mapping and scheduling strategies suitable to the BLR case should be designed. As the number of processes increases, synchronizations become more critical. Recent work from [37] aiming at avoiding such synchronizations in the FR case should be extended to the BLR case, for which it will certainly be even more critical. Finally, the LUAR variant presented in Section 6 requires the factorization to be performed in Left-looking. Its influence on the pattern of communications will have to be carefully analyzed.

Acknowlegments

We wish to thank our collaborators Romain Brossier, Ludovic Métivier, Alain Miniussi, Stéphane Operto, and Jean Virieux from SEISCOPE, Piyoosh Jaysaval, Sébastien de la Kethulle de Ryhove, and Daniil Shantsev from EMGS, and Olivier Boiteau from EDF for providing the test problems. We also thank Simon Delamare, Marie Durand, Guillaume Joslin, and Chiara Puglisi. This work was performed using HPC resources from the LIP laboratory of Lyon.

References

[1] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems. *ACM Trans. Math. Softw.*, 43(2):13:1–13:22, Aug. 2016.

[2] K. Akbudak, H. Ltaief, A. Mikhalev, and D. Keyes. Tile Low Rank Cholesky Factorization for Climate/Weather Modeling Applications on Manycore Architectures. In *Proceedings of ISC'17*, 2017.

[3] P. R. 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 Journal on Scientific Computing*, 37(3):A1451–A1474, 2015.

[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(6):R363 – R383, 2016.

[5] P. R. Amestoy, A. Buttari, I. S. Duff, A. Guermouche, J.-Y. L'Excellent, and B. Uçar. The multifrontal method. In D. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1209–1216. Springer, 2011.

[6] P. R. Amestoy, A. Buttari, I. S. Duff, A. Guermouche, J.-Y. L'Excellent, and B. Uçar. MUMPS. In D. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1232–1238. Springer, 2011.

[7] P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary. Complexity and performance of the Block Low-Rank multifrontal factorization. In *SIAM Conference on Parallel Processing (SIAM PP16)*, Paris, France, April 2016.

[8] P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary. On the complexity of the Block Low-Rank multifrontal factorization. *SIAM Journal on Scientific Computing*, 39(4):A1710–A1740, 2017.

[9] A. Aminfar, S. Ambikasaran, and E. Darve. A fast block low-rank dense solver with applications to finite-element matrices. *Journal of Computational Physics*, 304:170–188, 2016.

[10] A. Aminfar and E. Darve. A fast, memory efficient and robust sparse preconditioner based on a multifrontal approach with applications to finite-element matrices. *International Journal for Numerical Methods in Engineering*, 107(6):520–540, 2016.

[11] J. Anton, C. Ashcraft, and C. Weisbecker. A Block Low-Rank multithreaded factorization for dense BEM operators. In *SIAM Conference on Parallel Processing (SIAM PP16)*, Paris, France, April 2016.

[12] M. Bebendorf. Approximation of boundary element matrices. *Numerische Mathematik*, 86(4):565–589, 2000.

[13] M. Bebendorf. *Hierarchical Matrices: A Means to Efficiently Solve Elliptic Boundary Value Problems*, volume 63 of *Lecture Notes in Computational Science and Engineering (LNCSE)*. Springer-Verlag, 2008.

[14] S. Börm, L. Grasedyck, and W. Hackbusch. Introduction to hierarchical matrices with applications. *Engineering analysis with boundary elements*, 27(5):405–422, 2003.

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

[16] S. Constable. Ten years of marine CSEM for hydrocarbon exploration. *Geophysics*, 75(5):75A67–75A81, 2010.

[17] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, Dec. 2011.

[18] J. Dongarra, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, A. YarKhan, W. Alvaro, M. Faverge, A. Haidar, J. Hoffman, E. Agullo, A. Buttari, and B. Hadri. *PLASMA Users' Guide*. 2010.

[19] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. SIAM Press, Philadelphia, 1998.

[20] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, London, 1986.

[21] I. S. Duff and S. Pralet. Towards stable mixed pivoting strategies for the sequential and parallel solution of sparse symmetric indefinite systems. *SIAM Journal on Matrix Anaysis and Applications*, 29(3):1007–1024, 2007.

[22] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.

[23] P. Ghysels, X. S. Li, F.-H. Rouet, S. Williams, and A. Napov. An efficient multi-core implementation of a novel hss-structured multifrontal solver using randomized sampling. *SIAM Journal on Scientific Computing*, 38(5):S358–S384, 2016.

[24] A. Gillman, P. Young, and P.-G. Martinsson. A direct solver with $\mathcal{O}$(N) complexity for integral equations on one-dimensional domains. *Frontiers of Mathematics in China*, 7:217–247, 2012.

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

[26] N. Halko, P.-G. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53(2):217–288, 2011.

[27] J.-Y. L'Excellent and M. W. Sid-Lakhdar. A study of shared-memory parallelism in a multifrontal solver. *Parallel Computing*, 40(3-4):34–46, 2014.

[28] E. Liberty, F. Woolfe, P.-G. Martinsson, V. Rokhlin, and M. Tygert. Randomized algorithms for the low-rank approximation of matrices. *Proceedings of the National Academy of Sciences*, 104(51):20167–20172, 2007.

[29] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11:134–172, 1990.

[30] J. W. H. Liu. The multifrontal method for sparse matrix solution: Theory and Practice. *SIAM Review*, 34:82–109, 1992.

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

[32] G. Pichon, E. Darve, M. Faverge, P. Ramet, and J. Roman. Sparse Supernodal Solver Using Block Low-Rank Compression. In *18th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2017)*, Orlando, United States, jun 2017.

[33] H. Pouransari, P. Coulier, and E. Darve. Fast hierarchical solvers for sparse matrices using low-rank approximation. *arXiv preprint arXiv:1510.07363*, 2015.

[34] R. Schreiber. A new implementation of sparse Gaussian elimination. *ACM Transactions on Mathematical Software*, 8:256–276, 1982.

[35] M. Sergent, D. Goudin, S. Thibault, and O. Aumage. Controlling the memory subscription of distributed applications with a task-based runtime system. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 318–327, May 2016.

[36] D. 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. *Geophysical Journal International*, 209(3):1558–1571, 2017.

[37] W. M. Sid-Lakhdar. *Scaling multifrontal methods for the solution of large sparse linear systems on hybrid shared-distributed memory architectures*. Ph.D. dissertation, ENS Lyon, Dec. 2014.

[38] A. Tarantola. Inversion of seismic reflection data in the acoustic approximation. *Geophysics*, 49(8):1259–1266, 1984.

[39] C. Weisbecker. *Improving multifrontal solvers by means of algebraic block low-rank representations*. PhD thesis, Institut National Polytechnique de Toulouse, Oct. 2013.

[40] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, Apr. 2009.

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

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

[43] J. Xia, S. Chandrasekaran, M. Gu, and X. S. Li. Fast algorithms for hierarchically semiseparable matrices. *Numerical Linear Algebra with Applications*, 17(6):953–976, 2010.