Sparse supernodal solver using block low-rank compression: Design, performance and analysis

Grégoire Pichon, Eric Darve, Mathieu Faverge, Pierre Ramet, Jean Roman

Available online at www.sciencedirect.com

ScienceDirect

Journal of Computational Science 00 (2018) 1–24

Procedia Computer Science

Sparse Supernodal Solver Using Block Low-Rank Compression : design, performance and analysis

Grégoire Pichon[a], Eric Darve[b], Mathieu Faverge[a], Pierre Ramet[a], Jean Roman[a]

[a]*Inria, CNRS (LaBRI UMR 5800), Bordeaux INP, Université de Bordeaux, 33400 Talence, France*
[b]*Mechanical Engineering Department, Stanford University, United States*

Abstract

This paper presents two approaches using a Block Low-Rank (BLR) compression technique to reduce the memory footprint and/or the time-to-solution of the sparse supernodal solver PaStiX. This flat, non-hierarchical, compression method allows to take advantage of the low-rank property of the blocks appearing during the factorization of sparse linear systems, which come from the discretization of partial differential equations. The proposed solver can be used either as a direct solver at a lower precision or as a very robust preconditioner. The first approach, called *Minimal Memory*, illustrates the maximum memory gain that can be obtained with the BLR compression method, while the second approach, called *Just-In-Time*, mainly focuses on reducing the computational complexity and thus the time-to-solution. Singular Value Decomposition (SVD) and Rank-Revealing QR (RRQR), as compression kernels, are both compared in terms of factorization time, memory consumption, as well as numerical properties. Experiments on a shared memory node with 24 threads and 128 GB of memory are performed to evaluate the potential of both strategies. On a set of matrices from real-life problems, we demonstrate a memory footprint reduction of up to 4 times using the *Minimal Memory* strategy and a computational time speedup of up to 3.5 times with the *Just-In-Time* strategy. Then, we study the impact of configuration parameters of the BLR solver that allowed us to solve a 3D laplacian of 36 million unknowns a single node, while the full-rank solver stopped at 8 million due to memory limitation.

Keywords: Sparse linear solver, block low-rank compression, PaStiX sparse direct solver, multi-threaded architectures.

1. Introduction

Many scientific applications such as electromagnetism, geophysics or computational fluid dynamics use numerical models that require to solve linear systems of the form $Ax = b$, where the matrix A is sparse and large. In order to solve these problems, a classic approach is to use a sparse direct solver which factorizes the matrix into a product of triangular matrices before solving triangular systems.

Yet, there are still limitations to solve larger and larger systems in a black-box approach without any knowledge of the geometry of the underlying partial differential equation. Memory requirements and time-to-solution limit the use of direct methods for very large matrices. On the other hand, for iterative solvers, general black-box preconditioners that can ensure fast convergence for a wide range of problems are still missing.

In the context of sparse direct solvers, some recent works have investigated the low-rank representations of dense blocks appearing during the sparse matrix factorization, by compressing blocks through many possible compression formats such as Block Low-Rank (BLR), $\mathcal{H}$, $\mathcal{H}^2$, HSS, HODLR... These different approaches reduce the memory requirement and/or the time-to-solution of the solvers. Depending on the compression strategy, these solvers require knowledge of the underlying geometry to tackle the problem or can do it in a purely algebraic fashion.

Hackbusch [1] introduced the $\mathcal{H}$-LU factorization for dense matrices. It compresses the matrix into a hierarchical matrix format before applying low-rank operations instead of classic dense operations. The dense solver was extended to consider sparse matrices by using a nested dissection ordering to exhibit the hierarchical structure, as summarized in [2].

In [3], $\mathcal{H}$-LU factorization is used in an algebraic context. Performance, as well as a comparison of $\mathcal{H}$-LU with some sparse direct solvers is presented in [4]. Kriemann [5] and Lizé [6] implemented this algorithm using Directed Acyclic Graphs.

The Hierarchically Off-Diagonal Low-Rank (HODLR) compression technique was used in [7] to accelerate the elimination of large dense fronts appearing in the multifrontal method. It was fully integrated in a sparse solver in [8], and uses Boundary Distance Low-Rank (BDLR) to allow both time and memory savings, by avoiding the formation of dense fronts.

A supernodal solver using a compression technique similar to HODLR was presented in [9]. The proposed approach allows memory savings and can be faster than standard preconditioning techniques. However, it is slower than the direct approach in the benchmarks and requires an estimation of the rank to use randomized techniques and to accelerate the solver.

The use of Hierarchically Semi-Separable (HSS) matrices in sparse direct solvers has been investigated in several solvers. In [10], Xia et al. presented a solver for 2D geometric problems, where all operations are computed algebraically. In [11], a geometric solver was developed, but contribution blocks are not compressed, making memory savings limited. [12, 13] proposed an algebraic code that uses randomized sampling to manage low-rank blocks and to allow memory savings.

$\mathcal{H}^2$ arithmetic [14] has been used in several sparse solvers. In [15], a fast sparse $\mathcal{H}^2$ solver, called LoRaSp, based on extended sparsification was introduced. In [16], a variant of LoRaSp, aiming at improving the quality of the solver when used as a preconditioner, was presented, as well as a numerical analysis of the convergence with $\mathcal{H}^2$ preconditioning. In particular, this variant was shown to lead to a bounded number of iterations irrespective of problem size and condition number (under certain assumptions). In [17] a fast sparse solver was introduced based on interpolative decomposition and skeletonization. It was optimized for meshes that are perturbations of a structured grid. In [18], an $\mathcal{H}^2$ sparse algorithm was described. It is similar in many respects to [15], and extends the work of [17]. All these solvers have a guaranteed linear complexity, for a given error tolerance, and assuming a bounded rank for all well-separated pairs of clusters (the admissibility criterion in Hackbusch et al.'s terminology).

Block Low-Rank compression has been investigated for dense matrices [19, 20], and for sparse linear systems when using a multifrontal method [21, 22]. Considering that these approaches are similar to the current study, a detailed comparison will be described in Section 6. One of the differences of our approach with [22] is the supernodal context that leads to different low-rank operations, and possibly increases the memory savings. The main contribution of our work is the use of low-rank assemblies to avoid forming dense updates and to maximize memory savings.

The first objective of this work is to combine a generic sparse direct solver with recent work on matrix compression to solve larger problems, overcoming the memory limitations and accelerating the time-to-solution. The second objective is to keep the black-box algebraic approach of sparse direct solvers, by relying on methods that are independent of the underlying problem geometry. In this paper, we consider the multi-threaded sparse direct solver PaStiX [23] and we introduce a BLR compression strategy to reduce its memory and computational cost. We developed two strategies : *Minimal Memory*, which focuses on reducing the memory consumption, and *Just-In-Time* which focuses on reducing the time-to-solution (factorization and solve steps).

During the factorization, the first strategy compresses the sparse matrix before factorizing it, *i.e.* compresses A factors, and exploits dedicated low-rank numerical operations to keep the memory cost of the factorized matrix as low as possible. The second strategy compresses the information as late as possible, *i.e.* compresses L factors, to avoid the cost of low-rank update operations. The resulting solver can be used either as a direct solver for low accuracy solutions or as a high-accuracy preconditioner for iterative methods, requiring only a few iterations to reach the machine precision. The main contribution of this work is the introduction of low-rank compression in a supernodal solver with a purely algebraic method. Indeed, contrary to [9] which uses rank estimations (*i.e.* a non-algebraic criteria), our solver computes suitable ranks to maintain a prescribed accuracy.

A preliminary version of this work appeared in [24]. In this paper, we introduce new orthogonalization methods for the RRQR recompression kernels which leads to a better limitation of the rank growth during the factorization. We present a detailed analysis of the solver with a parallelism study and a comparison of efficiency of the low-rank kernels with respect to the original full-rank kernels. We also evaluate the impact of several parameters on the memory consumption and time to solution such as the blocking sizes, the maximum rank accepted for low-rank forms and the different orthogonalization methods. In Section 2, we go over basic aspects of sparse supernodal direct solvers. The two strategies, introduced in PaStiX, are then presented in Section 3, before detailing low-rank kernels in Section 4. Section 5 compares the two BLR strategies with the original approach, that uses only dense blocks, in terms of memory consumption, time-to-solution and numerical behaviour. We also investigate the efficiency of low-rank kernels, as well as the impact of the BLR solver parameters. Section 6 surveys in more detail related works on BLR for dense and/or sparse direct solvers, highlighting the differences with our approach, before discussing how to extend this work to a hierarchical format ($\mathcal{H}$, HSS, HODLR...).

2. Background on Sparse Linear Algebra

The common approach used by sparse direct solvers is composed of four main steps : 1) ordering of the unknowns, 2) computation of a symbolic block structure, 3) numerical block factorization, and 4) triangular system solves. In the rest of the paper, we consider that all problems have a symmetric pattern given by the pattern of $A + A^t$.

The purpose of the first step is to minimize the fill-in — zero becoming non-zero — that occurs during the numerical factorization to reduce the number of operations as well as the memory requirements to solve the problem. In order to both reduce fill-in and exhibit parallelism, the nested dissection [25] algorithm is widely used through libraries such as Metis [26] or Scotch [27]. Each set of vertices corresponding to a separator constructed during the nested dissection is called a supernode.

From the resulting supernodal partition, the second step predicts the symbolic block structure of the final factorized matrix (L) and the block elimination tree. This block structure is composed of one block of columns (column block) for each supernode of the partition, with a dense diagonal block and several dense off-diagonal blocks, as presented in Figure 1 for a 3D Laplacian.

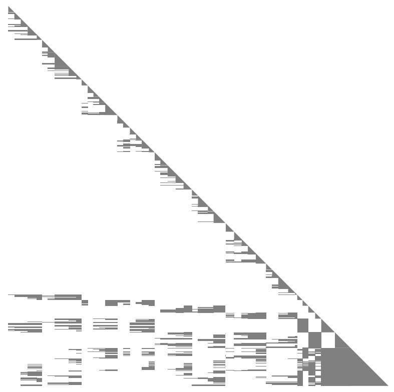


Figure 1. Symbolic factorization of a $10 \times 10 \times 10$ Laplacian matrix partitioned using Scotch.

The goal is to exhibit large block structures to leverage efficient Level 3 BLAS kernels during the numerical factorization. However, one may notice (cf. Figure 1) that the symbolic structure obtained with a general partitioning tool might be composed of many small off-diagonal blocks contributing to larger blocks. These off-diagonal blocks might be grouped together by adding zero to the structure if the BLAS efficiency gain is worthwhile and if the memory overhead induced by the fill-in is limited. Alternatively, it is also possible to reorder supernode unknowns to group off-diagonal blocks together without additional fill-in. A traveling salesman strategy is implemented in PaStiX [28] and divides by more than two the number of these off-diagonal blocks. Other approaches [12, 21] perform a k-way ordering of supernodes, starting from a reconnected graph of a separator, to order consecutively vertices belonging to a same local part of the separator's graph. Such reordering technique also allows to reduce ranks of the low-rank blocks as shown in [21]. To introduce more parallelism and data locality, the final structure can then be split into tiles as it is now commonly done in dense linear algebra libraries to fit the computational units granularity. These first two steps of direct solvers are preprocessing stages independent from the numerical values. Note that these steps can be computed once to solve multiple problems similar in structure but with different numerical values.

Finally, the last two steps, numerical factorization and triangular systems solves, perform the numerical operations. We consider here only the first one for the PaStiX solver. During the numerical factorization, the elimination of each supernode (column block) is similar to standard dense algorithms : 1) factorize the dense diagonal block, *xxTRF*, 2) solve the off-diagonal blocks belonging to this supernode, *TRSM*, and 3) apply the updates on the trailing submatrix, *GEMM*. Those steps are then adapted to the low-rank storage format as explained in Section 3.

3. Block Low-Rank solver

In this section, we describe the main contribution of this paper which is a BLR solver developed within the PaStiX library. First we introduce the notation used in this article, and the basics used to integrate low-rank blocks in the solver. Then, using the newly introduced structure, we describe two different strategies leading to a sparse direct solver that optimizes the memory consumption and/or the time-to-solution. In this work, we use the same static pivoting strategy as the full-rank version of the solver and which was first introduced in a sparse direct solver in [29].

3.1. Notation

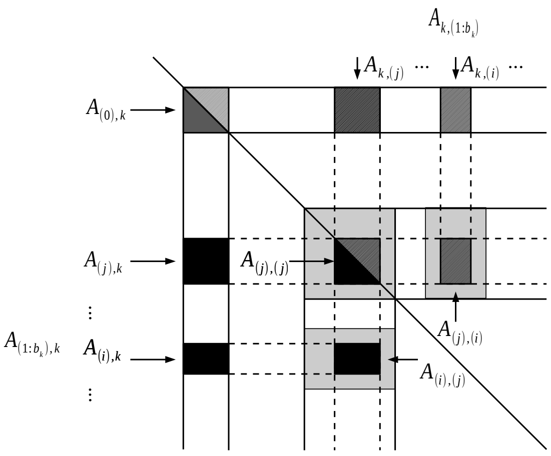


Figure 2. Symbolic block structure and notations used for the algorithms for the column block of index k, and its associated blocks. The larger gray boxes highlight the fact that the contributions modify only a part of the target blocks.

Let us consider the symbolic block structure of a factorized matrix L, obtained through symbolic block factorization. Initially, we allocate this structure initialized with the entries of A and perform an in-place factorization. We denote initial blocks by A and blocks in their final state by L (or U). The matrix is composed of N_{cblk} column blocks, where each column block is associated with a supernode, or to a subset of unknowns in a supernode when the latter is split to create parallelism. Each column block k is composed of $b_k + 1$ blocks, as presented in Figure 2 where :

— $A_{(0),k}(= A_{k,(0)})$ is the dense diagonal block ;

— $A_{(j),k}$ is the j^{th} off-diagonal block in the column block with $1 \leq j \leq b_k$, (j) being a multi-index describing the row interval of each block, and respectively, $A_{k,(j)}$ is the j^{th} off-diagonal block in the row block ;

— $A_{(1:b_k),k}$ represents all the off-diagonal blocks of the column block k, and $A_{k,(1:b_k)}$ all the off-diagonal blocks of the symmetric row block ;

— $A_{(i),(j)}$ is the rectangular dense block corresponding to the rows of the multi-index (i) and to the columns of the multi-index (j).

In addition, we denote $\hat{A}$ the compressed representation of a matrix A.

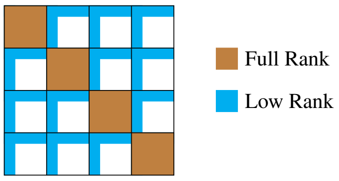


FIGURE 3. Block Low-Rank compression.

3.2. *Sparse direct solver using BLR compression*

The BLR compression scheme is a flat, non-hierarchical format, unlike others mentioned in the introduction. If we consider the example of a dense matrix, the BLR format clusters the matrix into a set of smaller blocks, as presented in Figure 3. Diagonal blocks are kept dense and off-diagonal blocks are admissible for compression. Thus, these off-diagonal blocks can be represented through a low-rank form uv^t, obtained with a compression technique such as Singular Value Decomposition (SVD) or Rank-Revealing QR (RRQR) factorization. Compression techniques are detailed in Section 4.

We propose in this paper to apply this scheme to the symbolic block structure of sparse direct solvers. Firstly, diagonal blocks of the largest supernodes in the block elimination tree can be considered as large dense matrices which are compressible with the BLR approach. In fact, as we have seen previously, it is common to split these supernodes into a set of smaller column blocks in order to increase the level of parallelism. Thus, the block structure resulting from this operation gives the cluster of the BLR compression format. Secondly, interaction blocks from two large supernodes are by definition long distance interactions, and thus can be represented by a low-rank form. It is then natural to store them as low-rank blocks as long as they are large enough. To summarize, if we take the final symbolic block structure (after splitting) used by the PaStiX solver, all diagonal blocks are considered dense, and all off-diagonal blocks might be stored using a low-rank structure. In practice, we limit this compression to blocks of a minimal size, and all blocks with relatively high ranks are kept dense.

From the original block structure, adapting the solver to block low-rank compression mainly relies on the replacement of the dense operations with the equivalent low-rank operations. Still, different variants of the final algorithm can be obtained by changing *when and how* the low-rank compression is applied. We introduce two scenarios : *Minimal Memory*, which compresses the blocks before any other operations, and *Just-In-Time* which compresses the blocks after they received all their contributions.

Algorithm 1 Right looking block sequential LU factorization with *Minimal Memory* scenario.

```
    ▷ /* Initialize A (L structure) compressed */
1:  For $k = 1$ to $N_{cblk}$ Do
2:      $\hat{A}_{(1:b_k),k} = \text{Compress}(A_{(1:b_k),k})$
3:      $\hat{A}_{k,(1:b_k)} = \text{Compress}(A_{k,(1:b_k)})$
4:  End For
5:  For $k = 1$ to $N_{cblk}$ Do
6:      Factorize $A_{(0),k} = L_{(0),k} U_{k,(0)}$
7:      Solve $\hat{L}_{(1:b_k),k} U_{k,(0)} = \hat{A}_{(1:b_k),k}$
8:      Solve $L_{(0),k} \hat{U}_{k,(1:b_k)} = \hat{A}_{k,(1:b_k)}$
9:      For $j = 1$ to $b_k$ Do
10:         For $i = 1$ to $b_k$ Do
    ▷ /* LR to LR updates (extend-add) */
11:             $\hat{A}_{(i),(j)} = \hat{A}_{(i),(j)} - \hat{L}_{(i),k} \hat{U}_{k,(j)}$        ▷ LR2LR
12:         End For
13:     End For
14: End For
```

3.2.1. Minimal Memory

This scenario, described by Algorithm 1, starts by compressing the original matrix A block by block without allocating the full matrix as it is done in the full-rank and *Just-In-Time* strategies. Thus, all low-rank blocks that are large enough are compressed directly from the original sparse form to the low-rank representation (lines $1-4$). Note that for a matter of conciseness, loops of compression and solve over all off-diagonal blocks are merged into a single operation. In this scenario, compression kernels and later operations could have been performed using a sparse format, such as CSC for instance, until we get some fill-in. However, for the sake of simplicity we use a low-rank form throughout the entire algorithm to rely on blocks and not just on sets of values. Then, each classic dense operation on a low-rank block is replaced by a similar kernel operating on low-rank forms, even for the usual matrix-matrix multiplication (*GEMM*) kernel that is replaced by the equivalent *LR2LR* kernel operating on three low-rank matrices (cf. Section 4).

Algorithm 2 Right looking block sequential LU factorization with *Just-In-Time* scenario.

```
 1: For $k = 1$ to $N_{cblk}$ Do
 2:     Factorize $A_{(0),k} = L_{(0),k} U_{k,(0)}$
    ▷ /* Compress L and U off-diagonal blocks */
 3:     $\hat{A}_{(1:b_k),k}$ = Compress( $A_{(1:b_k),k}$ )
 4:     $\hat{A}_{k,(1:b_k)}$ = Compress( $A_{k,(1:b_k)}$ )
 5:     Solve $\hat{L}_{(1:b_k),k} U_{k,(0)} = \hat{A}_{(1:b_k),k}$
 6:     Solve $L_{(0),k} \hat{U}_{k,(1:b_k)} = \hat{A}_{k,(1:b_k)}$
 7:     For $j = 1$ to $b_k$ Do
 8:         For $i = 1$ to $b_k$ Do
    ▷ /* LR to dense updates */
 9:             $A_{(i),(j)} = A_{(i),(j)} - \hat{L}_{(i),k} \hat{U}_{k,(j)}$        ▷ LR2GE
10:         End For
11:     End For
12: End For
```

3.2.2. Just-In-Time

This second scenario, described by Algorithm 2, delays the compression of each supernode after all contributions have been accumulated. The algorithm is thus really close to the previous one with the only difference being in the update kernel, *LR2GE*, at line 9, which accumulates contributions on a dense block, and not on a low-rank form.

This operation, as we describe in Section 4, is much simpler than the *LR2LR* kernel, and is faster than a classic *GEMM*. However, by compressing the initial matrix A, and maintaining the low-rank structure throughout the factorization with the *LR2LR* kernel, *Minimal Memory* can reduce more drastically the memory footprint of the solver. Indeed, the full-rank structure of the factorized matrix is never allocated, as opposed to *Just-In-Time* that requires it to accumulate the contributions. The final matrix is compressed with similar sizes in both scenarios.

3.2.3. Summary

For the sake of simplicity, we now compare the three strategies in a dense case to illustrate the potential of different approaches. Figure 4 presents the Directed Acyclic Graph (DAG) of *LU* operations for the original full-rank version of the solver on a 3-by-3 block matrix. On each node, the couple (i, j) represents respectively the row and column indexes of the block of the matrix being modified.

The *Minimal Memory* strategy generates the DAG described in Figure 5. In this context, the six off-diagonal blocks are compressed at the beginning. In the update process, dense diagonal blocks as well as low-rank off-diagonal blocks are updated. One can notice that off-diagonal blocks are never used in their dense form, leading to memory footprint reduction. However, as we will describe in Section 4, the *LR2LR* operation in sparse arithmetic is quite expensive and may lead to an increase of time-to-solution.

The *Just-In-Time* strategy generates the DAG described in Figure 6. In this case, the six off-diagonal blocks are compressed throughout the factorization. Since those off-diagonal blocks are used in their dense form before being

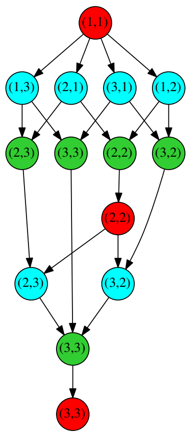


Figure 4. Full-rank.

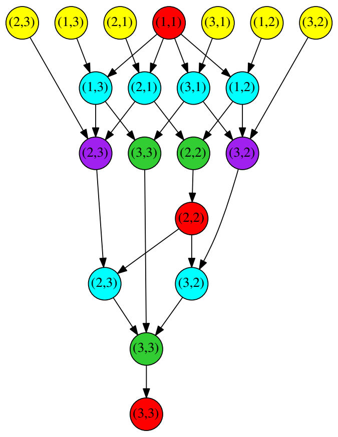


Figure 5. *Minimal Memory*.

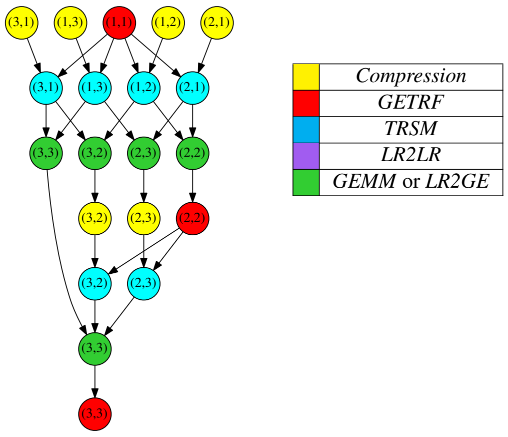


Figure 6. *Just-In-Time*.

compressed, there is less room for memory improvement. However, as we will see later, the *LR2GE* can still be performed efficiently in the sparse case.

4. Low-rank kernels

We introduce in this section the low-rank kernels used to replace the dense operations, and we present a complexity study of these kernels. Two families of operations are studied to reveal the rank of a matrix : Singular Value Decomposition (SVD) which leads to smaller ranks, and Rank-Revealing QR (RRQR) which has shorter time-to-solution.

4.1. Compression

The goal of low-rank compression is to represent a general dense matrix A of size m_A-by-n_A by its compressed version $\hat{A} = u_A v_A^t$, where u_A, and v_A, are respectively matrices of size m_A-by-r_A, and n_A-by-r_A, with r_A being the rank of the block supposed to be small with respect to m_A and n_A. In order to keep a given numerical accuracy we have to choose r_A such that $||A - \hat{A}|| \leq \tau ||A||$, where τ is the prescribed tolerance.

4.1.1. SVD

A is decomposed as $U\Sigma V^t$. The low-rank form of A consists of the first r_A singular values and their associated singular vectors such that : $\Sigma_{r_A+1} \leq \tau ||A||$, $u_A = U_{r_A}$, and $v_A^t = \Sigma_{1:r_A} V_{r_A}^t$ with U_{r_A} (respectively V_{r_A}) being the first r_A columns of U (respectively V). This solution presents multiple drawbacks. It has a large cost of $\Theta(m_A^2 n_A + n_A^2 m_A + n_A^3)$ operations to compress the matrix, and all singular values needs to be computed to get the first r_A ones. On the other hand, as the singular values of the matrix are explicitly computed, it leads to the lowest rank for a given tolerance.

4.1.2. RRQR

A is decomposed as QRP, where P is a permutation matrix, and QR the QR decomposition of AP^{-1}. The low-rank form of A of rank r_A is then formed by $u_A = Q_{(:,r_A)}$, the first r_A columns of Q, and by $v_A^t = R_{(r_A,:)}$, the first r_A rows of R. The main advantage of this process is that one can stop the factorization as soon as the norm of the trailing submatrix $\tilde{A}_{(r_A+1:m_A, r_A+1:n_A)} = A - Q_{(:,r_A)} R_{(r_A,:)} P$ is lower than $\tau ||A||$. Thus, the complexity is lowered to $\Theta(m_A n_A r_A)$ operations. However, it returns an evaluation of the rank which might be a little larger than those returned by SVD and generate more flops during the numerical factorization and solve.

In conclusion, SVD is much more expensive than RRQR. However, for a given tolerance, SVD returns lower ranks. Put another way, for a given rank, SVD will have better numerical accuracy. Thus, there is a trade-off between time-to-solution (RRQR) versus memory consumption (SVD).

Note that for the *Minimal Memory* scenario, the first compression (of sparse blocks) may be realized using Lanczos's methods, to take advantage of sparsity. However, both SVD and RRQR algorithms inherently take advantage of these zeros. In addition, most of the low-rank compression are applied to blocks stored as dense blocks and represent the main part of the computation.

4.2. Solve

The solve operation for a generic lower triangular matrix L is applied to blocks in low-rank forms in our two scenarios : $L\hat{x} = \hat{b} \Leftrightarrow Lu_x v_x^t = u_b v_b^t$. Then, with $v_x^t = v_b^t$, the operation is equivalent to applying a dense solve only to u_b, and the complexity is only $\Theta(m_L^2 r_x)$, instead of $\Theta(m_L^2 n_L)$ for the full-rank (dense) representation.

4.3. Update

Let us consider the generic update operation, $C = C - AB^t$. Note that the PaStiX solver stores L, and U^t if required. Then, the same update is performed for Cholesky and LU factorizations. We break the operation into two steps : the product of two low-rank blocks, and the addition of a low-rank block to either a dense block (*LR2GE*), or a low-rank block (*LR2LR*).

4.3.1. Low-rank matrix product

This operation can simply be expressed as two dense matrix products :

$$\hat{A}\hat{B}^t = (u_A(v_A^t v_B))u_B^t = u_A((v_A^t v_B)u_B^t),$$

where u_A is kept unchanged if $r_A \leq r_B$ (u_B^t is kept otherwise) to lower the complexity.

However, it has been shown in [19] that the rank r_{AB} of the product of two low-rank matrices of ranks r_A and r_B is usually smaller than $\min(r_A, r_B)$. As u_A and u_B are both orthogonal, the matrix $E = (v_A^t v_B)$ has the same rank as $\hat{A}\hat{B}^t$. Thus, the complexity can be further reduced by transforming the matrix product to the following series of operations :

$$E = v_A^t v_B \tag{1}$$
$$\hat{E} = \widehat{v_A^t v_B} = u_E v_E^t \tag{2}$$
$$u_{AB} = u_A u_E \tag{3}$$
$$v_{AB}^t = v_E^t v_B^t. \tag{4}$$

4.3.2. Low-rank matrix addition

Let us consider the next generic operation $C' = C - u_{AB} v_{AB}^t$, with $m_{AB} \leq m_C$ and $n_{AB} \leq n_C$ as it generally happens in the supernodal method. This is illustrated for example by the update block $A_{(i),(j)}$ in Figure 2.

If C is not compressed, as it happens in the *LR2GE* kernel, C' will be dense too, and the addition of the two matrices is nothing else than a *GEMM* kernel. The complexity of this operation grows as $\Theta(m_{AB}\ n_{AB}\ r_{AB})$.

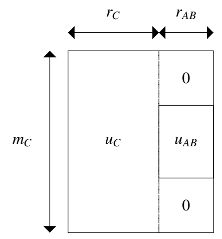


Figure 7. Accumulation of two low-rank matrices when sizes do not match.

If C is compressed as in the *LR2LR* kernel, C' will be compressed too, and

$$\hat{C'} = u_C v_C^t - u_{AB} v_{AB}^t \tag{5}$$
$$u_{C'} v_{C'}^t = [u_C, u_{AB}]([v_C, -v_{AB}])^t \tag{6}$$

where [,] is the concatenation operator. The update given at Equation (5) is the commonly named *extend-add* operation. Without further optimization, this operation costs only two copies. In the case of the supernodal method, adequate padding is also required to align the vectors coming from the AB and C matrices as shown in Figure 7 for the u vectors. The operation on v is similar.

One can notice that in this form, the rank of the updated C is now $r_C + r_{AB}$. When accumulating multiple updates, the rank grows quickly and the storage exceeds the full-rank version. In order to maintain a small rank for C, recompression techniques are used. As for the compression kernel, both SVD and RRQR algorithms can be used.

Recompression using SVD. We start by computing a QR decomposition for both composed matrices :

$$[u_C, u_{AB}] = Q_1R_1 \text{ and } [v_C, -v_{AB}] = Q_2R_2. \tag{7}$$

Then, the temporary matrix $T = R_1R_2^t$ is compressed using the SVD algorithm described previously. This gives the final $\hat{C}'$ with :

$$u_{C'} = (Q_1u_T) \text{ and } v_{C'} = (Q_2v_T). \tag{8}$$

The complexity of this operation is decomposed as follows : $\Theta((m_C{+}n_C)(r_C{+}r_{AB})^2)$ for the two QR decompositions of equation (7), $\Theta((r_C + r_{AB})^3)$ for the SVD decomposition, and finally $\Theta((m_C + n_C)(r_C + r_{AB})r_{C'})$ for the application of both Q_1 and Q_2.

Recompression using RRQR. This solution takes advantage of the fact that u_C is orthonormal to first orthogonalize u_{AB} with respect to u_C. For this operation, we refer to Section 4.3.3 to form an orthonormal basis $[u_C, \overline{u_{AB}}]$ such that

$$[u_C, u_{AB}] = [u_C, \overline{u_{AB}}]\ T. \tag{9}$$

We follow by applying the RRQR algorithm to :

$$T\left([v_C, -v_{AB}]\right)^t = QRP. \tag{10}$$

As for the compression, we keep the $k = r_{C'}$ first columns of Q and rows of R to form the final C' :

$$u_{C'} = ([u_C, \overline{u_{AB}}]\ Q_k) \text{ and } v_{C'}^t = R_kP. \tag{11}$$

Note that $u_{C'}$ is kept orthonormal for future updates.

When the RRQR algorithm is used, the complexity of the recompression is then composed of : $\Theta(r_C\ r_{AB}\ m_{AB})$ to form the intermediate product $u_C^t\ u_{AB}$, $\Theta(m_C\ r_C\ r_{AB})$ to form the orthonormal basis, $\Theta(n_{AB}\ r_{AB}\ r_C)$ to generate the temporary matrix used in (10), $\Theta((r_C + r_{AB})n_C\ r_{C'})$ to apply the RRQR algorithm, and finally again $\Theta((r_C + r_{AB})n_C\ r_{C'})$ to compute the final $u_{C'}$.

4.3.3. Orthogonalization

Let us consider the orthogonalization of $[u_C, u_{AB}] = QT$, taking advantage of u_C orthogonality. The following recompression of $T[v_C, -v_{AB}]^t$ will now be referred as *right recompression*.

A main issue is that $[u_C, u_{AB}]$ may not be full-rank in exact arithmetic if both matrices share a common spectrum. Thus, if some zero columns can be removed, it will reduce the cost of the *right recompression*, by ignoring zero rows. In practice, for each zero column, we permute both $[u_C, u_{AB}]$ and $[v_C, -v_{AB}]$ matrices and reduce the rank involved in next computations.

The objective is to reduce the number of operations depending on r_C, considering that in many cases $r_{AB} < r_C$ as many large off-diagonal blocks receive small contributions. In order to do so, we perform Gram-Schmidt (GS) projections to take advantage of u_C orthogonality. Two variants of GS are widely used : Classical Gram-Schmidt (CGS) and Modified Gram-Schmidt (MGS). Both may have stability issues, and several iterations may be performed to ensure a correct (at machine precision) orthogonality. In practice, we use CGS for its locality advantages and perform a second iteration if required. To verify that a second iteration is required, we use a widely used criterion [30] and did not experience any orthogonality issue for the set of problems we consider.

We now present several variants for orthogonalization.

Householder QR factorization. This method performs a Householder QR factorization on the full matrix $[u_C, u_{AB}] = Q_1 R_1$. Thus, it does not exploit the existing orthogonality in u_C, and cannot properly extract zero columns from the final solution to reduce the cost of later operations. However, it is probably the most stable approach and is the mostly tuned kernel in linear algebra libraries such as Intel MKL. The complexity of this operation grows as $\Theta(m_C(r_C+r_{AB})^2)$.

PartialQR. It performs a projection of u_{AB} into u_C :

$$\widetilde{u_{AB}} = u_{AB} - u_C(u_C^t\, u_{AB}). \tag{12}$$

From this projection, we obtain :

$$[u_C, u_{AB}] = [u_C, \widetilde{u_{AB}}]\begin{pmatrix} I & u_C^t u_{AB} \\ 0 & I \end{pmatrix}. \tag{13}$$

In practice, we perform two projections to ensure the stability. We now have $u_C \perp \widetilde{u_{AB}}$ and want to orthogonalize $\widetilde{u_{AB}}$ to obtain an orthogonal basis. In order to do so, we either perform a Householder QR factorization $\widetilde{u_{AB}} = \overline{u_{AB}} R$ or a rank-revealing QR factorization at the machine precision $\widetilde{u_{AB}} = Q_k R_k P$ to remove zero columns. The first approach can make a good use of Level 3 BLAS operations while the second is less efficient but may construct zero columns and then reduce the right recompression.

Classical Gram-Schmidt. In this variant, we orthogonalize one-by-one each vector of u_{AB} with one or two CGS iterations depending on the criterion presented before. The orthogonalization of the i-th column of u_{AB} to obtain the corresponding column of $\overline{u_{AB}}$ is performed with :

$$\overline{u_{AB_i}} = u_{AB_i} - u_{AB_{0:i-1}} u_{AB_{0:i-1}}^t u_{AB_i}. \tag{14}$$

Each column is then removed after its orthogonalization if it is a *zero* column. This reduces the final rank of the update and the cost of the following operations.

In the experimental study, we will compare the three approaches in terms of operation count and performance. For orthogonality, we did not observe any numerical issue in our test problems. Note that for both PartialQR and CGS, only last r_{AB} rows of $([v_C, -v_{AB}])^t$ are updated. For those two methods, the orthogonalization complexity grows as $\Theta(m_C\ r_C\ r_{AB})$.

4.4. Summary

Table 1 presents the computational complexity for the two low-rank strategies with respect to the original version of the solver. The main factor of complexity is computed under the assumption that $m_C \geq m_A \geq m_B$, $r_A \geq r_B$, $m_C \geq n_C$, and $r_C \leq r_{C'}$. It does not depend on n_A but on the ranks r_A and r_B : there are fewer operations to be performed. On the other hand, the *Minimal Memory* strategy requires using either the SVD or RRQR recompression, for which the complexity depends on m_C and n_C, the dimensions of the block C. This explains why this strategy appears as a higher complexity than the original solver.

When considering dense matrices, the low-rank matrix (blocks) are updated by contributions of the same size. In that case, the complexity of the low-rank update becomes asymptotically cheaper than the full-rank updates, and leads to performance gain. This is exploited in dense BLR solvers as [19], and in the CUFS (Compress, Update, Factor, Solve) strategy of the BLR-MUMPS solver, which compresses a dense front before applying operations between low-rank blocks of the same size.

In the supernodal approach, blocks belonging to last level separators receive many small contributions, as illustrated in Figure 1. Thus, $r_{C'}$ is often close to or equal to r_C and lower than $r_C + r_{AB}$: the rank is often invariant when applying a small contribution, which makes RRQR recompression more efficient than SVD recompression, and especially when an orthogonalization method that reduces the basis on the fly is used.

In terms of complexity, it is important to notice that the *LR2LR* update depends on the size of the target block C, and not on the contribution size as in full-rank. Thus, low-rank contributions between blocks of similar sizes are asymptotically cheaper than full-rank contributions on these blocks. Dense and multifrontal solvers exploit this property as they work only on regular block sizes. At the opposite, small updates to larger blocks, that regularly appear in supernodal solvers, have a higher cost in low-rank than in full-rank. In summary, we can decompose supernodal solvers in two groups of contributions :

Table 1. Summary of the operation complexities when computing $C = C - AB^t$.

	GEMM (*Dense*)	*LR2GE* (*Just-In-Time*)		*LR2LR* (*Minimal Memory*)	
		SVD	RRQR	SVD	RRQR
LR matrix product	–	(1) : $\Theta(n_A\ r_A\ r_B)$		(1) : $\Theta(n_A\ r_A\ r_B)$	
		(2) : $\Theta(r_A^2\ r_B)$	(2) : $\Theta(r_A\ r_B\ r_{AB})$	(2) : $\Theta(r_A^2\ r_B)$	(2) : $\Theta(r_A\ r_B\ r_{AB})$
		(3), (4) : $\Theta(m_A\ r_A\ r_{AB})$		(3), (4) : $\Theta(m_A\ r_A\ r_{AB})$	
LR matrix addition	–	–		(7) : $\Theta(m_C(r_C + r_{AB})^2)$	(12) : $\Theta(m_C\ r_C\ r_{AB})$
				(SVD) : $\Theta((r_C + r_{AB})^3)$	(10) : $\Theta(n_C(r_C + r_{AB})r_{C'})$
				(8) : $\Theta(m_C(r_C + r_{AB})r_{C'})$	(11) : $\Theta(m_C(r_C + r_{AB})r_{C'})$
Dense update	$\Theta(m_A\ m_B\ n_A)$	$\Theta(m_A\ m_B\ r_{AB})$		–	–
Main factor	$\Theta(m_A\ m_B\ n_A)$	$\Theta(m_A\ m_B\ r_{AB})$	$\Theta(m_A\ m_B\ r_{AB})$	$\Theta(m_C(r_C + r_{AB})^2)$	$\Theta(m_C(r_C + r_{AB})r_{C'})$

— the updates that will occur within each supernode at the top of the elimination tree and which are generated by regular split of the supernode to generate parallelism, as well as updates from direct descendants of the sons. These contributions work on regular sizes and will reduce the complexity of the solver as observed in dense and multifrontal solvers ;

— the updates from deeper descendants in the elimination tree that generate updates from small blocks to larger nodes of the elimination tree. In these contributions, the complexity is defined by the target block, and thus the cost of the low-rank updates may increase the complexity of the solver with respect to the full-rank solver.

The experiments showed that this overhead slows down the solver with the *Minimal Memory* strategy on small problems, but when the problem size increases this strategy outperforms the full-rank factorization.

The main advantage of the *Minimal Memory* scenario is that it can drastically reduce the memory footprint of the solver, since it compresses the matrix before the factorization. Thus, the structure of the full-rank factorized matrix is never allocated, and the low-rank structure needs to be maintained throughout the factorization process to lower the memory peak.

4.5. Kernel implementation

In practice, as we manage both dense and low-rank blocks in our solver, we adapt the extend-add operation for each basic case to be as efficient as possible. The optimizations are designed for the RRQR version, in practice SVD is only used as a reference for the optimal compression rates.

One of the important criteria for efficient low-rank kernels is the setup of the maximum rank above which the matrix will be considered as non compressible. The setting of this parameter is strategy and application dependent. In practice for an m-by-n matrix, if the main objective is the memory consumption, as in the *Minimal Memory* strategy, the ranks are limited to $\frac{mn}{m+n}$ to reduce as much as possible the final size of the factors without considering the number of flops that are generated. In contrast, if the objective is to reduce the time-to-solution, then the maximum ranks are defined by $\frac{min(m,n)}{4}$ for which the low-rank operations remains cheaper in number of operations than the full-rank ones. In practice, it also depends on the ranks of the blocks that will be involved in the update and cannot be computed before compression. For a real-life application, the criterion has to be set depending on the number of factorizations (second criterion is more important) and the number of solves (first criterion will reduce the size of the factors and thus the cost of the solve step). The impact of this parameter is studied in 5.6.3.

Eventually, when a low-rank block receives a contribution with a high rank, *i.e.*, $(r_C + r_{AB}) \leq max_{rank}$, the C matrix is decompressed to receive the update and then recompressed, instead of directly adding low-rank matrices. This is denoted as *Decompression / Recompression of C* in the following section.

There is also a trade-off between using a strict compression criterion and a smaller one due to the efficiency of low-rank operations with respect to dense operations. In practice, we add another parameter named rank ratio to control the maximum rank authorized during compression as a percentage of the strict theoretical rank.

5. Experiments

Experiments were conducted on the *Plafrim* [1] supercomputer, and more precisely on the `miriel` cluster. Each node is equipped with two Intel `Xeon E5-2680v3` 12-cores running at 2.50 GHz and 128 GB of memory. The Intel MKL `2017` is used for BLAS and SVD kernels. The RRQR kernel is issued from the BLR-MUMPS solver [21], and is an extension of the block rank-revealing QR factorization subroutines from LAPACK 3.6.0 (xGEQP3) to stop the factorization when the precision is reached.

The PaStiX version used for our experiments is available on the public git repository [2] as the tag `6.0.0`. The multi-threaded version used is the static scheduling version presented in [31]. Note that for low-rank strategies, we never perform LL^t factorization because compression can destroy the positive-definite property. In the case where the matrix is SPD, we use LDL^t factorization for low-rank strategies.

For the initial ordering step, we used Scotch [27] 6.0.4 with the configurable strategy string from PaStiX to set the minimal size of non-separated subgraphs, *cmin*, to 15. We also set the $frat$ parameter to 0.08, meaning that column aggregation is allowed by Scotch as long as the fill-in introduced does not exceed 8% of the original matrix.

In experiments, blocks that are larger than 256 are split into blocks of size within the range $128-256$ to create more parallelism while keeping sizes large enough. The same 128 criteria is used to define the minimal width of the column blocks that are compressible. An additional limit on the minimal height to compress an off-diagonal block is set to 20. We set the rank ratio to 1 to illustrate the results when the rank is as strict as possible to obtain Flops and/or memory gains. CGS orthogonalization is also used. Note that in Section 5.6 we try different blocking sizes to experiment the impact on the solver, relax the maximum ranks authorized, and compare the orthogonalization strategies.

Experiments were computed on a set of 3D matrices from The SuiteSparse Matrix Collection [32], described in Table 2.

Table 2. Real-life matrices used in experiments.

Matrix	Precision	Method	Size	Field	Ops (TFlops)	Memory (GB)
Atmosmodj	d	*LU*	1 270 432	atmospheric model	12.1	16.3
Audi	d	LL^t	943 695	structural problem	5.5	9.5
Hook	d	LDL^t	1 498 023	model of a steel hook	8.6	12.7
Serena	d	LDL^t	1 391 349	gas reservoir simulation	28.6	21.7
Geo1438	d	LL^t	1 437 960	geomechanical model of earth	18.0	20.1

We also used 3D Laplacian generators (7 point stencils), and defined $lap120$ as a Laplacian of size 120^3.

Note that when results showing numerical precision are presented, we used the backward error on b : $\frac{||Ax-b||_2}{||b||_2}$.

5.1. SVD versus RRQR

The first experiment studies the behaviour of the two compression methods coupled with both *Minimal Memory* and *Just-In-Time* scenarios on the matrix *Atmosmodj*. Table 3 presents the sequential timings of each operation of the numerical factorization with a tolerance of 10^{-8}, as well as the memory used to store the final coefficients of the factorized matrix.

1. `https://www.plafrim.fr`
2. `https://gitlab.inria.fr/solverstack/pastix`

Table 3. Cost distribution on the Atmosmodj matrix with $\tau = 10^{-8}$.

	Full-rank	*Just-In-Time*		*Minimal Memory*	
		SVD	RRQR	SVD	RRQR
Factorization time (s)					
Compression	-	4.1e+02	3.4e+01	1.8e+02	5.6+00
Block factorization (GETRF)	7.2e-01	7.4e-01	7.3e-01	7.8e-01	7.6e-01
Panel solve (TRSM)	1.7e+01	6.9e+00	7.4e+00	7.6e+00	7.9e+00
Update					
Formation of contribution	-	-	-	9.9e+01	4.2e+01
Addition of contribution	-	-	-	3.0e+03	7.3e+02
Dense udpate (GEMM)	4.6e+02	1.3e+02	9.7e+01	2.8e+01	2.4e+01
Total	*4.7e+02*	*5.5e+02*	***1.4e+02***	*3.6e+03*	*8.1e+02*
Solve time (s)	6.3e+00	1.9e+00	3.0e+00	**1.5e+00**	3.2e+00
Factor final size (GB)	16.3	6.95	7.49	**6.85**	7.31
Memory peak for the factors (GB)	16.3	16.3	16.3	**6.85**	7.31

We can first notice that SVD compression kernels are much more time consuming than the RRQR kernels in both scenarios following the complexity study from Section 4. Indeed, RRQR compression kernels stop the computations as soon as the rank is found which reduces by a large factor the complexity, and this reduction is reflected in the time-to-solution. However, the SVD allows, for a given tolerance, to get better memory reduction in both scenarios.

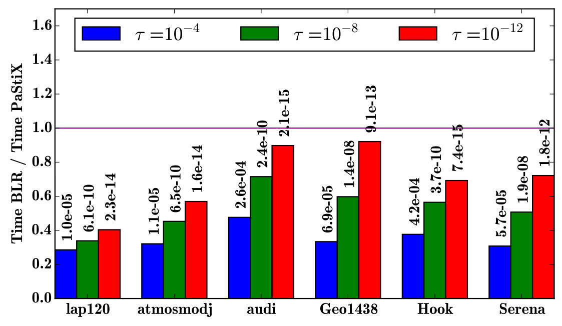
Comparing the *Minimal Memory* and the *Just-In-Time* scenario, the compression time is minimized in the *Minimal Memory* scenario because the compression occurs on the initial blocks which hold more zero and are lower rank than when they have been updated. The time for the update addition, *extend-add* operation, becomes dominant in the *Minimal Memory* scenario, and even explodes when SVD is used. This is expected as the complexity depends on the largest blocks in the addition even for small contributions (see Section 4). Note that this ratio will evolve in favor of the *extend-add* operation on larger matrices where the ratio of updates of same size becomes dominant with respect to the number of updates from small blocks. For both compression methods, both scenarios compress the final coefficients with similar rates.

The diagonal block factorization time is invariant in the five strategies : the block sizes and kernels are identical. Panel solve, update product, and solve times are reduced in all low-rank configurations compared to the dense factorization and the timings follow the final size of the factors, since this size reflects the final ranks of the blocks.

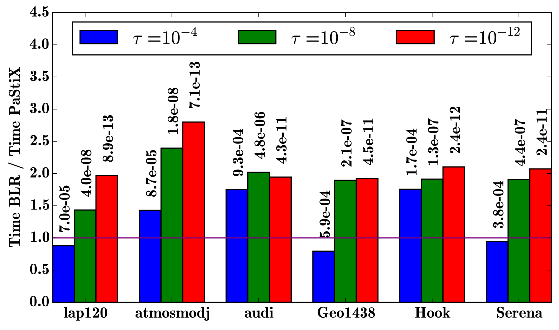
To conclude, the *Minimal Memory* scenario is not always able to compete with the original direct factorization on these test cases due to the costly update addition. However, it reduces the memory peak of the solver to the final size of the factors. The *Minimal Memory*/RRQR offers a 50% memory reduction while doubling the sequential time-to-solution. The *Just-In-Time* scenario competes with the original direct factorization, and divides by three the time-to-solution with RRQR kernels.

5.2. Performance

Figure 8 presents the overall performance achieved by the two low-rank scenarios with respect to the original version of the solver (where lower is better) on the previously introduced set of six matrices. All versions are multi-threaded implementations and use the 24 cores of one node. The scheduling used is the PaStiX static scheduler developed for the original version, this might have a negative impact for low-rank strategies by creating a load imbalance. We study only the RRQR kernels as the SVD kernels have shown to be much slower. Three tolerance thresholds are studied for their impact on the time-to-solution and the accuracy of the backward error. The backward errors printed on top of each bar correspond to the use of one refinement step.



(a) *Just-In-Time* scenario using RRQR.



(b) *Minimal Memory* scenario using RRQR.

Figure 8. Performance of both low-rank strategies with three tolerance thresholds. The backward error of the solution is printed on top of each bar.

Figure 8(a) shows that the *Just-In-Time*/RRQR scenario is able to reduce the time-to-solution in almost all cases of tolerance, and for all matrices which have a large spectrum of numerical properties. These results show that applications which require low accuracy, as the seismic application for instance, can benefit by up to a factor of 3.5. Figure 8(b) shows that it is more difficult for the *Minimal Memory*/RRQR scenario to be competitive. The performance is often degraded with respect to the original PaStiX performance, with an average loss of around a factor 2, and the tolerance has a much lower impact than for the previous case.

For both scenarios, the backward error of the first solution is close to the entry tolerance. It is a little less accurate in the *Minimal Memory* scenario, because approximations are made earlier in the computation, and information is lost from the beginning. However, these results show that we are able to catch algebraically the information and forward it throughout the update process.

5.3. Memory consumption

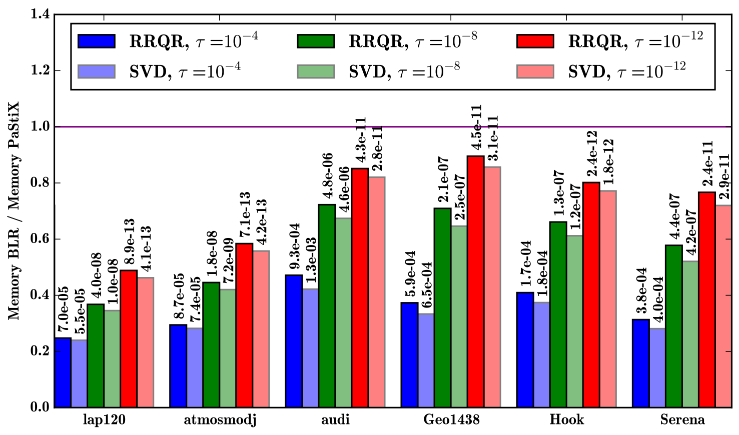
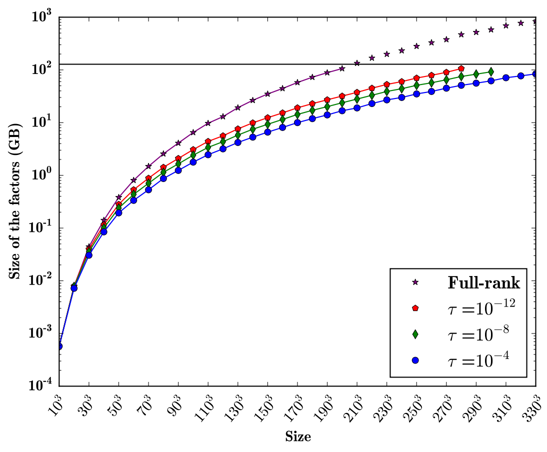


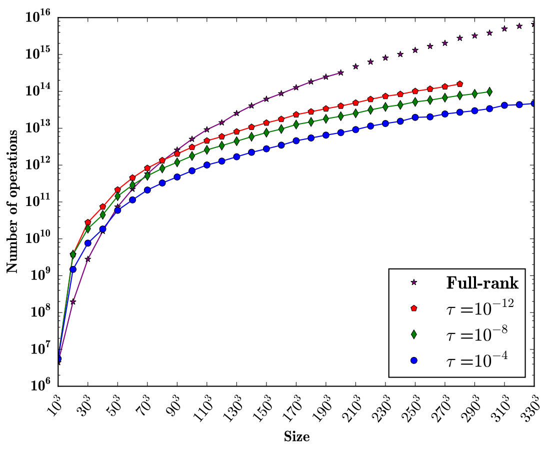
Figure 9. Memory peak for the *Minimal Memory* scenario with three tolerance thresholds and both SVD and RRQR kernels. The backward error of the solution is printed on top of each bar.

The *Minimal Memory* scenario is slower than the original solver in most cases, but it is a strategy that efficiently reduces the memory peak of the solver. Figure 9 presents the gain in the memory used to store the factors at the end of the factorization of the set of six matrices with respect to the *block dense* storage of PaStiX. In this figure, we also compare the memory gain of the SVD and RRQR kernels. We observe that in all cases, SVD provides better compression rate by finding smaller ranks for a given matrix and a given tolerance. The quality of the backward error is in general slightly better with the SVD kernels despite the smaller ranks. The second observation is that the smaller the tolerance (10^{-12}), the larger the ranks and the memory consumption. However, the solver always presents a memory gain larger than 50% with larger tolerance (10^{-4}).

Figure 10(a) presents the evolution of the size of the factors as well as the full consumption of the solver (factors and management structures) on 3D Laplacians with an increasing size. The memory limit of the system is 128GB. The original version is limited on this system to a 3D Laplacian of 8 million unknowns, and the size of the factors quickly increases for larger number of unknowns. With the *Minimal Memory*/RRQR scenario, we have now been able to run a 3D problem on up to 36 million unknowns when relaxing the tolerance to 10^{-4}. From the same experiment, Figure 10(b) presents the number of operations evolution depending on the Laplacians size. One can note that for a small number of unknowns, *Minimal Memory*/RRQR scenario performs more operations than the original, full-rank, factorization. However, for a large number of unknowns, the number of operations is reduced by a large factor. For instance, for a 280^3 Laplacian with a 10^{-8} tolerance, the number of operations is reduced by a factor larger than 36. It demonstrates the potential of our approach : even if operations are less efficient and may lead to a time-to-solution overhead for relatively small problems, the *Minimal Memory* strategy enables the computation of larger problem and reduces its time-to-solution. In this experiment, the *Minimal Memory* strategy becomes faster for Laplacian larger than 150^3 with a 10^{-8} tolerance.



(a) Memory.



(b) Number of operations.

FIGURE 10. Scalability of the memory on top and the number of operations on bottom with three tolerance thresholds of the *Minimal Memory*/RRQR scenario for 3D Laplacians of size n^3 with $n \in [10, 330]$. The full-rank scenario, in purple, is given as a reference with the numbers computed from the symbolic factorization.

The memory of the *Just-In-Time* scenario has not been studied, because in our supernodal approach, each supernode is fully allocated in a full-rank fashion in order to accumulate the updates before being compressed. Thus, the memory peak corresponds to the totality of the factorized matrix structure without compression and is identical to the original version. To reduce this memory peak, a solution would be to modify the scheduler to a *Left-Looking* approach that would delay the allocation and the compression of the original blocks. However, it would need to be carefully implemented to keep a certain amount of parallelism in order to save both time and memory. A possible solution is the scheduling strategy presented in [33] to keep the memory consumption of the solver under a given limit.

5.4. Convergence and numerical stability

Figure 11 presents the convergence of the GMRES iterative solver preconditioned with the low-rank factorization at tolerances of 10^{-4} and 10^{-8}. The iterative solver is stopped after reaching 20 iterations or a backward error lower than 10^{-12}.

With a tolerance of 10^{-8}, only a few iterations are required to converge to the solution. Note that on the *Audi* and *Geo1438* matrices, which are difficult to compress, a few more iterations are required to converge. With a larger tolerance 10^{-4}, it is difficult to recover all the information lost during the compression, but this is enough to quickly get solutions at 10^{-6} or 10^{-8}. Note that the GMRES process benefits from the compression, as the solve step.

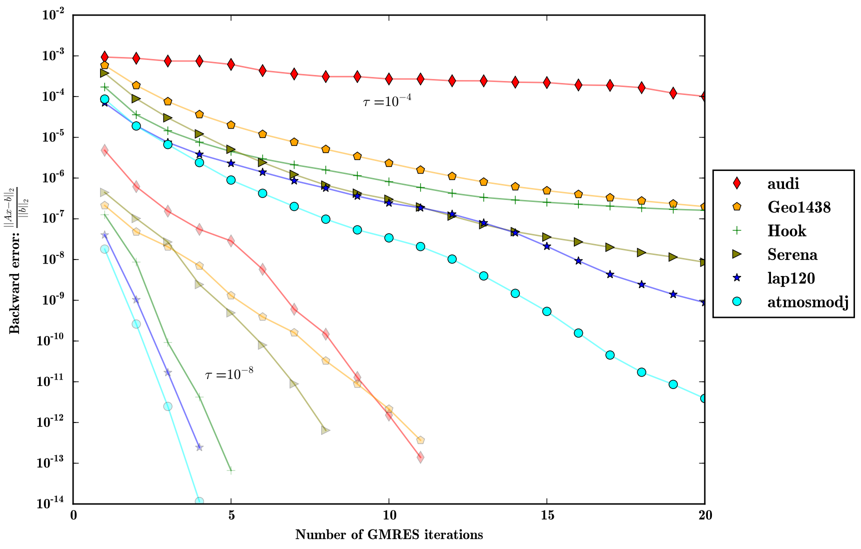


FIGURE 11. Convergence speed for the *Minimal Memory*/RRQR scenario with two tolerance thresholds.

5.5. *Parallelism*

For the full-rank factorization, supernodes are split between processors depending on the corresponding number of operations. When a target block C receives a contribution, a lock is used to ensure that the block is not modified simultaneously by several threads. For the *Just-In-Time* strategy, a similar lock is used since the update operations directly apply dense modifications. However, for the *Minimal Memory* strategy, the update operation is decomposed into two parts : *Formation of contribution* and *Addition of contribution*. Because the formation of contribution does not depend on the target block C, the lock is only positioned for the addition of contribution, which may increase the level of parallelism.

Figure 12 presents the speedup of the full-rank factorization and low-rank strategies using tolerances of 10^{-4} and 10^{-8} for the *atmosmodj* matrix. The speedup for the full-rank version is above 12, for a relatively small matrix. The speedup of the *Minimal Memory* strategy is above 11, while the speedup of the *Just-In-Time* strategy is around 8. As the supernodes distribution cannot predict the ranks and the corresponding number of operations, load balancing may be degraded. *Minimal Memory* strategy scalability exceeds *Just-In-Time* strategy because there are less constraints on locks. Note that on recent architectures, the maximal speedup can not be obtained, as the CPU frequency is reduced when all cores are used on the node, while it is increased for single core operation. We thus computed an approximate maximum speedup of 20.7 on this architecture for the Intel MKL BLAS GEMM operation.

5.6. *Kernels analysis*

In this section, we focus on the *atmosmodj* matrix to illustrate the performance of basic kernels as well as the impact of several parameters. As in practical cases we use RRQR instead of SVD, we will focus on this compression kernel for each strategy.

5.6.1. *Performance of basic kernels*

We evaluate the performance rate (in GFlops/s) for the full-rank factorization and for the two low-rank strategies, and use a 10^{-8} tolerance. Figure 13(a) (respectively Figure 13(b)) presents the runtime distribution among kernels for the full-rank (respectively *Just-In-Time*) factorization. We can note that in both cases, the *Update* process is the most time-consuming. For the *Just-In-Time* strategy, *Compression* and *TRSM* are not negligible, because the *Update* runtime is much reduced with respect to full-rank factorization.

Figure 13(c) presents the runtime distribution among kernels for the *Minimal Memory* strategy with three different levels from the left to the right : the main steps of the solver, the details for the *Update* kernels, and the details for

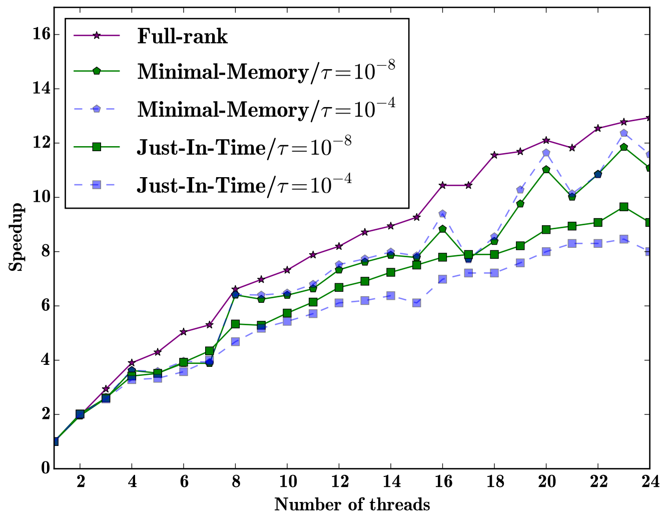


FIGURE 12. Speedup of the factorization for the *atmosmodj* matrix with 1 to 24 threads.

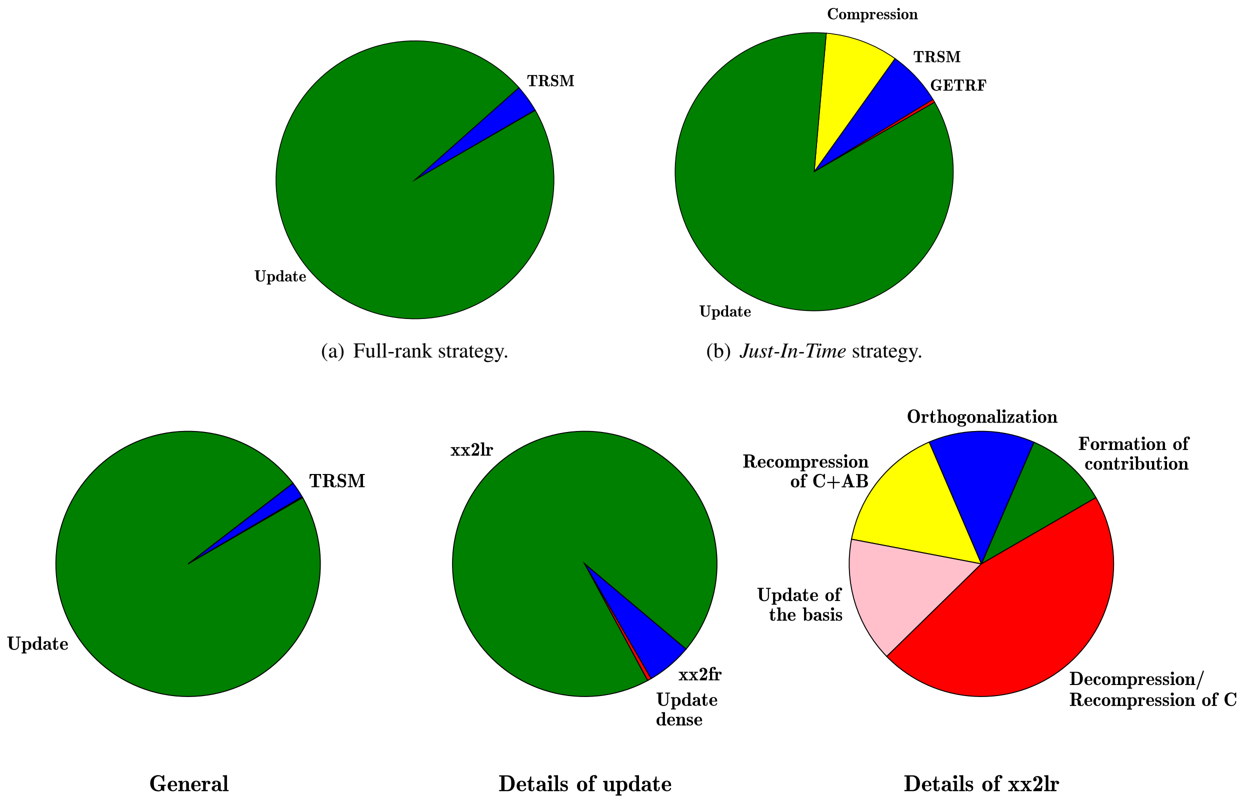


FIGURE 13. Breakdown of the most time-consuming kernels for the full-rank strategy (*top-left*), the *Just-In-Time* strategy (*top-right*), and the *Minimal Memory* strategy (*bottom*) on the *atmosmodj* case using the RRQR kernels with $\tau = 10^{-8}$. The analysis of the *Minimal Memory* strategy is shown at three different levels from the left to the right : the global operations, the partition of the updates between low-rank and full-rank, and the details of the low-rank updates.

the low-rank updates. Note that *xx2fr* refers to a full-rank update within a compressible supernode (*i.e.*, an update to a dense block which was originally considered compressible) while *xx2lr* is a low-rank update, *xx* being one of the four possible matrix products : low-rank/low-rank, low-rank/full-rank, full-rank/low-rank or full-rank/full-rank. *Update dense* corresponds to blocks that were not considered compressible and managed in dense arithmetic throughout all operations ; as expected the underlying operations represent a small time of computation.

We observe that the low-rank update is the most time consuming part. In practice, the *Formation of contribution* (cf. Section 4.3.1) is quite cheap, while applying the update is expensive ; as we have seen in Table 1, it depends on the size and the rank of the target C. This update addition is decomposed into three main operations : *Orthogonalization* of the $[u_C, u_{AB}]$ matrix (cf. Section 4.3.3), *Recompression of* $C + AB$ (cf. Eq (2)) and *Update of the basis* (cf. Eq (11)). If the contribution rank is too high to take advantage of recompression, we perform a *Decompression / Recompression of* C (cf. Section 4.5).

Table 4 presents the performance of most time-consuming kernels for each type of factorization on a machine where around 32 GFlops/s can be raised for each CPU core when all cores are used. One can note that the performance of kernels for the full-rank factorization is close to the machine peak : the original solver makes good use of Level 3 BLAS even if there are many small blocks. On the other hand, low-rank kernels performing Level 3 BLAS kernels, *i.e.*, *Formation of contribution*, *Update of the basis*, *TRSM*, are running at $\frac{1}{3}$ of the peak. It is due to lower granularity generated by the smaller blocks, which reduce the arithmetic intensity of the kernels. For *Compression* and *Recompression of C+AB* kernels, the efficiency is even worse due to the behaviour of RRQR.

TABLE 4. Kernels efficiency per core for full-rank, *Just-In-Time* and *Minimal Memory* strategies, on *atmosmodj* with $\tau = 10^{-8}$.

Strategy	Name	Nb calls	Time (s)	GFlops	Perf (GFlops/s)
-	GETRF	74665	981.6	6.4	6.5
Full-rank	TRSM	1572464	17862.0	421.2	23.6
	Update	10993468	446003.1	12869.6	28.9
Just-In-Time	Compression	44544	32831.8	146.6	4.5
	TRSM	1572464	9312.8	111.1	11.9
	Update	10993468	126681.3	1464.1	11.6
Minimal Memory	TRSM	1572464	7355.9	103.8	14.1
	Update				
	dense	5782698	4838.5	24.6	5.1
	xx2fr	2207642	31280.6	285.0	9.1
	xx2lr				
	Formation of contribution	3036024	39319.2	493.7	12.6
	Orthogonalization	2688031	115957.9	630.2	5.4
	Recompression of $C + AB$	2499058	301204.3	755.4	2.5
	Update of the basis	2499058	54132.7	742.5	13.7
	Decompression / Recompression of C	556650	256821.7	2236.3	8.7

In our implementation, RRQR is a modification of LAPACK `xgeqp3` and `xlaqps` routines. Some stability issues presented in [34] prevent to make efficient use of Level 3 BLAS kernels.

To summarize, low-rank strategies are useful to reduce the overall number of operations. However, due to the poor efficiency of low-rank kernels, the gain in flops does not directly translate into timing reduction. One can expect that for larger problems, the reduction in flops will more easily translate into time-to-solution reduction.

5.6.2. *Impact of blocking parameters*

In previous experiments, we always used a splitting criterion of 256 with a minimal size of 128 : blocks larger than 256 are split into a subset of blocks larger than 128. For low-rank strategies, we consider only blocks larger than 128 as compressible. In Table 5, we evaluate the impact of using different blocking sizes (between 128 and 256 or between 256 and 512) for the full-rank factorization and for three different tolerances using *Minimal Memory* and *Just-In-Time* strategies. The minimum width of compressible supernodes is set to the minimum blocksize : either 128 or 256. All results are performed using 24 threads on the *atmosmodj* matrix.

Table 5. Impact of the blocking size parameter on the number of operations, the factorization time, and the memory for the *atmosmodj* on the Full-rank strategy (top) and on both *Minimal Memory* and *Just-In-Time* strategies (bottom). Both low-rank strategies are using RRQR kernels.

(a) Full-rank strategy.

	Blocksize	128–256	256–512
Full-rank	Operations (TFlops)	12.1	12.1
	Fact. time (s)	**39.5**	51.5
	Memory (GB)	**16.3**	16.5

(b) Low-rank strategies.

	Precision	$\tau = 10^{-4}$		$\tau = 10^{-8}$		$\tau = 10^{-12}$	
	Blocksize	128–256	256–512	128–256	256–512	128–256	256–512
Minimal Memory	Operations (TFlops)	**1.8**	3.9	**4.8**	10.8	**8.3**	18.0
	Fact. time (s)	**49.7**	76.4	**100.1**	166.6	**147.7**	253.0
	Memory (GB)	**4.7**	5.8	**6.53**	7.29	**8.31**	8.81
Just-In-Time	Operations (TFlops)	**0.5**	0.7	**0.8**	1.1	**1.3**	1.5
	Fact. time (s)	12.0	**10.0**	15.6	**14.4**	20.5	**20.2**
	Memory (GB)	**4.9**	6.0	**6.71**	7.47	**8.43**	8.93

The first observation concerns the full-rank factorization. The blocking sizes impact both the granularity and the level of parallelism. For a sequential run, it is suitable to use large blocking sizes to increase granularity and thus the efficiency of Level 3 BLAS operations. On the other hand, it may degrade parallel performance if there are many workers and not enough supernodes.

For the *Minimal Memory* strategy, we have seen in Table 1 that the number of operations depends on the target size and rank and thus increases a lot with the blocking size. This is true even in the case where ranks are not that much impacted by the blocking size. In practice, we observe that for each tolerance, increasing blocking size degrades factorization time. In addition, as less data is considered as compressible, the size of the factors is growing and this will increase the solve cost.

For the *Just-In-Time* strategy, the impact of blocking size will mostly depend on ranks. If ranks are small, using larger blocks will increase the performance of RRQR and thus reduce the time-to-solution. However if ranks are higher, it will reduce the level of parallelism as in the full-rank factorization. We always observe a gain using a larger blocking size, but the ratio between the use of 256/512 versus 128/256 is decreasing when the tolerance is lower : the granularity gain causes some parallelism issues.

5.6.3. *Impact of rank ratio parameter*

In previous experiments, we use strict maximum ranks, *i.e.,* the limit ranks to reduce the number of flops or the memory consumption : $\frac{mn}{m+n}$ for *Minimal Memory* strategy and $\frac{min(m,n)}{4}$ for *Just-In-Time* strategy. In practice, we have seen that for both strategies, kernel efficiency is poor with respect to classical Level 3 BLAS operations. In Table 6 we evaluate the impact of relaxing the constraint on a strict rank for the *Minimal Memory* and the *Just-In-Time* strategies. The ratio parameter corresponds to a percentage of the strict maximum rank (used to obtain smaller ranks) and avoid the overhead of managing low-rank blocks with a high rank by turning back these blocks into full-rank form. All results are performed using 24 threads on the *atmosmodj* matrix.

For the *Minimal Memory* strategy, there are two main observations. Firstly, considering that the update complexity depends on the size but also on the rank of the target C, the burden on recompression of high-rank blocks is reduced,

Table 6. Impact of the maximum rank ratio on the number of operations, the factorization time, and the memory for the *atmosmodj* case with RRQR kernels for both *Minimal Memory* and *Just-In-Time* strategies.

	Precision	$\tau = 10^{-4}$			$\tau = 10^{-8}$			$\tau = 10^{-12}$		
	Ratio	1	0.5	0.25	1	0.5	0.25	1	0.5	0.25
Minimal Memory	Operations (TFlops)	**1.8**	1.9	2.3	4.8	4.8	**4.5**	8.3	**7.2**	7.4
	Fact. time (s)	48.7	49.7	**43.7**	100.0	85.4	**56.0**	146.0	105.1	**65.6**
	Memory (GB)	**4.7**	**4.7**	5.4	**6.53**	7.0	8.94	**8.31**	9.26	11.71
Just-In-Time	Operations (TFlops)	**0.5**	0.6	0.9	**0.8**	1.4	2.6	**1.3**	2.8	5.1
	Fact. time (s)	11.9	11.6	**10.9**	**14.2**	15.4	17.8	**19.0**	20.8	25.3
	Memory (GB)	**4.9**	5.0	5.9	**6.71**	7.23	9.12	**8.43**	9.36	11.73

and sometimes even the number of operations. Secondly, managing more blocks in full-rank fashion can reduce time-to-solution due to the poor efficiency of low-rank kernels. In practice, we observe that using a relaxed criterion always reduces time-to-solution while having only a little impact on the memory.

For the *Just-In-Time* strategy, the maximum rank criterion cannot be set theoretically, because it depends on all ranks within a same supernode. However, contrary to the *Minimal Memory* approach, the number of operations being really reduced with respect to full-rank factorization, there is a time-to-solution gain and it seems suitable to compress as many blocks as possible. In practice, relaxing the max-rank criteria is only interesting for 10^{-4} tolerance, for which the granularity is really small.

Note that in both cases, relaxing the burden on large ranks increases the size of the factors, and in the same way the cost of the solve. Thus, selecting a suitable criteria will depend on the application and the ratio between the number of factorizations and the number of solves.

5.6.4. Orthogonalization cost

In previous *Minimal Memory* experiments, we used CGS as orthogonalization process. As presented in Section 4.3.3, some other approaches can be investigated. In Table 7, we present the impact of using CGS, Householder QR or PartialQR on the number of operations as well as on the efficiency of the solver. It only impacts intermediate ranks and not the final size of the factors.

Table 7. Impact of the orthogonalization method on the number of operations and the factorization time for *Minimal Memory* strategy on *atmosmodj*.

Precision	$\tau = 10^{-4}$			$\tau = 10^{-8}$			$\tau = 10^{-12}$		
Orth. Method	CGS	QR	PartialQR	CGS	QR	PartialQR	CGS	QR	PartialQR
Operations (TFlops)	**1.8**	2.8	2.0	**4.8**	8.0	5.1	**8.3**	13.2	8.7
Fact. time (s)	50.5	60.3	**48.5**	99.1	128.9	**96.3**	145.2	191.8	**138.6**

As predicted by its complexity, Householder QR factorization is more expensive than both CGS and PartialQR. The difference between CGS and PartialQR is related to the number of columns of zero. While CGS can remove those columns during computations, PartialQR can only deal with those zero after all operations : the number of operations increases for each tolerance.

However, in terms of time, PartialQR outperforms CGS especially for small tolerances (10^{-12}) due to the efficiency of Level 3 BLAS operations. One can expect that for a larger tolerance (10^{-4}), CGS may be faster than PartialQR, because the smaller granularity will degrade the assets of Level 3 BLAS operations.

6. Discussion

In this section, we discuss the positioning of our low-rank strategies with the closest related works and we give some perspectives in extending this work to a hierarchical format.

Sparse $\mathcal{H}$ solver. In [2], different approaches using $\mathcal{H}$-matrices are summarized, including the extension to the sparse case. The authors use a nested dissection ordering as in our solver, and thus there is no fill-in between distinct branches of the elimination tree. However, contributions of a supernode to its ancestors are considered as full, in the sense that all structural zero are included to generate the low-rank representation. Thus, they do not have extend-add (*LR2LR*) operations between low-rank blocks of different sizes : assembly is performed as in our *Minimal Memory* scenario, but without zero padding. The memory consumption is higher with their approach because some structural zeros are not managed : unlike them, we perform a block symbolic factorization to consider sparsity in all contributions between supernodes.

Dense BLR solver. A dense BLR solver was designed by Livermore Software Technology Corporation (LSTC) [19]. In this work, the full matrix is compressed at the beginning and operations between low-rank blocks are performed. This approach performs low-rank assembly as in our *Minimal Memory* scenario. As it handles dense matrices, the extend-add process is performed between low-rank matrices of the same size and zero padding is not required. Thus, the *LR2LR* operation is less costly than the full-rank update in this context.

Sparse Multifrontal BLR solver. A BLR multifrontal sparse direct solver was designed for the Mumps solver. The different strategies are summarized in [22] and a theoretical study of the complexity of the solver for regular meshes is presented in [35]. In the current implementation, the fronts are always assembled in a full-rank form before being compressed. The authors suggest as an alternative solution to assemble the fronts panel by panel to avoid allocating the full front in a dense fashion, but this was not implemented and it will impact the potential parallelism. Mumps is a multifrontal solver, and BLR is considered when eliminating a dense front and not between fronts while our supernodal approach has a more global view of all supernodes.

Our scenario *Just-In-Time* is similar to the FCSU (Factor, Compress, Solve, Update) strategy from [22] where the fronts are compressed panel by panel during their factorization and where only full-rank blocks are updated (*LR2GE* kernel). The LUAR (Low-Rank Update Accumulation with Recompression) groups together multiple low-rank products to exploit the memory locality during the product recompression process. This could be similarly used in the *Just-In-Time* strategy, but would imply larger ranks in the extend-add operations of the *Minimal Memory* strategy.

The CUFS (Compress, Update, Factor, Solve) [22] strategy is the closest to the *Minimal Memory* scenario, as it performs low-rank assembly (*LR2LR* kernel) within a front. However, the fronts are still first allocated in full-rank to assemble the contributions from the children. Within a front, the CUFS strategy is similar to what was proposed by LSTC for dense matrices. In that case, since blocks are dense and low-rank operations are performed between blocks of equal sizes, zeros padding is not necessary as in the *Minimal Memory* strategy.

We believe that there is more room for memory savings using the *Minimal Memory* strategy for two reasons : 1) we avoid the extra cost of fronts inherent to the multifrontal method and 2) the low-rank assembly avoids forming dense blocks before compressing them. We demonstrated that the *Minimal Memory* strategy reduces the number of operations performed on 3D Laplacians and other matrices coming from various applications, which is the main contribution of this paper. A theoretical complexity study has to be performed to investigate if such an approach performs asymptotically the same number of operations as *Just-In-Time* or Mumps strategies with a constant factor overhead. Low-rank assembly without randomization techniques has only been studied in [10] for 2D problems with the knowledge of the underlying problem, and computing the complexity bounds for a more general case is still an open research problem, we are currently investigating.

Extension to hierarchical formats. With the aim of extending our solver to hierarchical compression schemes, such as $\mathcal{H}$, HSS, or HODLR, we consider graphs coming from real-life simulations of 3D physical problems. From a theoretical point of view, the majority of these graphs have a bounded degree or are bounded density graphs [36], and thus good separators [37] can be built. For a n-vertices mesh, time complexity of a direct solver is in $\Theta(n^2)$, and we expect to build a low-rank solver requiring $\Theta(n^{\frac{4}{3}})$ operations. For memory requirements, the direct approach leads to an overall storage of $\Theta(n^{\frac{4}{3}})$, while we target a $\Theta(n \log(n))$ complexity.

Let us consider the last separator of size $\Theta(n^{\frac{2}{3}})$ for a 3D mesh, and one of the largest low-rank blocks of this separator in a hierarchical clustering. They have asymptotically the same size. Previous studies have shown that such a block may have a rank of order $\Theta(n^{\frac{1}{3}})$. For the *Just-In-Time* scenario, maintaining such a block in a dense form before compressing it requires $\Theta(n^{\frac{4}{3}})$ memory. Thus, we will still have the same memory peak, but we might encounter a large overhead when compressing off-diagonal blocks with current RRQR and SVD kernels. For the *Minimal Memory* scenario, we have seen that the cost of the solver can be split in two stages. The low-level one from the elimination tree that generates small updates to large contribution blocks and that might become more expensive with the hierarchical compression, and the high-level where blocks fit the hierarchical structure and generate flops savings. To overcome the issue of the low-level contributions, new ordering techniques need to be investigated to minimize the number of updates on larger off-diagonal blocks. We will also investigate the use of randomization techniques to lower the complexity of the updates from the bottom of the elimination tree.

Conclusion

We presented a new Block Low-Rank sparse solver that combines an existing sparse supernodal direct solver PaStiX and low-rank compression kernels. This solver reduces the memory consumption and/or the time-to-solution depending on the scenario. Two scenarios were developed. For the set of real-life problems studied, *Minimal Memory* saves memory up to a factor of 4 using RRQR kernels, with a time overhead that is limited to 2.8. Large problems that could not fit into memory when the original solver was used can now be solved thanks to the lower memory requirements, especially when low accuracy solutions. For larger problems, one can expect that the reduction of the number of operations will translate into a time-to-solution reduction. We experienced this behaviour with large Laplacians : we are now able to solve a 330^3 unknowns Laplacian while the original solver was limited to a 200^3 unknowns Laplacian, the time-to-solution being reduced over 150^3 unknowns.

Just-In-Time reduces both the time-to-solution by a factor up to 3.5 and the memory requirements of the final factorized matrix with similar factors to *Minimal Memory*. However, with the current scheduling strategy, this gain is not reflected in the memory peak.

Two compression kernels, SVD and RRQR, were studied and compared. We have shown that, for a given tolerance, both approaches provide correct solutions with the expected accuracy, and that RRQR, despite larger ranks, provides faster kernels. In addition, we demonstrated that the solver can be used either as a low-tolerance direct solver or as a good preconditioner for iterative methods, that normally require only a few iterations before reaching the machine precision. A comparison with other preconditioners (AMG, ILU(k)) will be performed in future work to measure the impact of using a low-rank factorization as a preconditioner.

In the future, new kernel families, such as RRQR with randomization techniques, will be studied in terms of accuracy and stability in the context of a supernodal solver. To further improve the performance of *Minimal Memory* and close up the gap with the original solver, aggregation techniques on small contributions will also be studied. This will lead to the extension of this work to hierarchical compression in large supernodes that could further reduce the memory footprint and the solver complexity.

Regarding *Just-In-Time*, future work is focused on studying smart scheduling strategies that combine *Right-Looking* and *Left-Looking* approaches in order to find a good compromise between memory and parallelism for the target architecture. Sergent et al. [38] studied scheduling with memory constraints for dense factorization using BLR, and we expect it can be adapted for sparse direct solvers. This will follow up recent work on applying parallel runtime systems [31] to the PaStiX solver.

Acknowledgments

This material is based upon work supported by the DGA under a DGA/Inria grant. Experiments presented in this paper were carried out using the PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine (see https ://www.plafrim.fr/).

Références

[1] W. Hackbusch, A Sparse Matrix Arithmetic Based on $\mathcal{H}$-Matrices. Part I : Introduction to $\mathcal{H}$-Matrices, Computing 62 (2) (1999) 89–108.

[2] W. Hackbusch, Hierarchical Matrices : Algorithms and Analysis, Vol. 49, Springer Series in Computational Mathematics, 2015.

[3] L. Grasedyck, R. Kriemann, S. Le Borne, Parallel black box $\mathcal{H}$-LU preconditioning for elliptic boundary value problems, Computing and Visualization in Science 11 (4-6) (2008) 273–291.

[4] L. Grasedyck, W. Hackbusch, R. Kriemann, Performance of H-LU preconditioning for sparse matrices, Computational methods in applied mathematics 8 (4) (2008) 336–349.

[5] R. Kriemann, H-LU factorization on many-core systems, Computing and Visualization in Science 16 (3) (2013) 105–117.

[6] B. Lizé, Résolution directe rapide pour les éléments finis de frontière en électromagnétisme et acoustique : H-matrices. parallélisme et applications industrielles., Ph.D. thesis, École Doctorale Galilée (Jun. 2014).

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

[8] A. Aminfar, 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) (2016) 520–540.

[9] J. N. Chadwick, D. S. Bindel, An Efficient Solver for Sparse Linear Systems Based on Rank-Structured Cholesky Factorization, CoRR abs/1507.05593.

[10] J. Xia, S. Chandrasekaran, M. Gu, X. Li, Superfast Multifrontal Method For Large Structured Linear Systems of Equations, SIAM Journal on Matrix Analysis and Applications 31 (2009) 1382–1411.

[11] S. Wang, X. S. Li, F.-H. Rouet, J. Xia, M. V. De Hoop, A Parallel Geometric Multifrontal Solver Using Hierarchically Semis-Separable Structure, ACM Trans. Math. Softw. 42 (3) (2016) 21.

[12] P. Ghysels, X. S. Li, F.-H. Rouet, S. Williams, A. Napov, An Efficient Multicore Implementation of a Novel HSS-Structured Multifrontal Solver Using Randomized Sampling, SIAM Journal on Scientific Computing 38 (5) (2016) S358–S384.

[13] P. Ghysels, X. S. Li, C. Gorman, F.-H. Rouet, A robust parallel preconditioner for indefinite systems using hierarchical matrices and randomized sampling, in : 2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017, 2017, pp. 897–906.

[14] W. Hackbusch, S. Börm, Data-sparse Approximation by Adaptive $\mathcal{H}^2$-Matrices, Computing 69 (1) (2002) 1–35.

[15] H. Pouransari, P. Coulier, E. Darve, Fast hierarchical solvers for sparse matrices using extended sparsification and low-rank approximation, SIAM Journal on Scientific Computing 39 (3) (2017) A797–A830.

[16] K. Yang, H. Pouransari, E. Darve, Sparse Hierarchical Solvers with Guaranteed Convergence, arXiv preprint arXiv :1611.03189.

[17] K. L. Ho, L. Ying, Hierarchical Interpolative Factorization for Elliptic Operators : Differential Equations, Communications on Pure and Applied Mathematics 8 (69) (2016) 1415–1451.

[18] D. A. Sushnikova, I. V. Oseledets, “Compress and eliminate” solver for symmetric positive definite sparse matrices, arXiv preprint arXiv :1603.09133v3.

[19] J. Anton, C. Ashcraft, C. Weisbecker, A Block Low-Rank Multithreaded Factorization for Dense BEM Operators, in : SIAM Conference on Parallel Processing for Scientific Computing (SIAM PP 2016), Paris, France, 2016.

[20] K. Akbudak, H. Ltaief, A. Mikhalev, D. Keyes, Tile Low Rank Cholesky Factorization for Climate/Weather Modeling Applications on Manycore Architectures, Springer International Publishing, Cham, 2017, pp. 22–40.

[21] P. R. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L'Excellent, C. Weisbecker, Improving Multifrontal Methods by Means of Block Low-Rank Representations, SIAM Journal on Scientific Computing 37 (3) (2015) A1451–A1474.

[22] T. Mary, Block Low-Rank multifrontal solvers : complexity, performance, and scalability, Ph.D. thesis, Toulouse University, Toulouse, France (Nov. 2017).

[23] P. Hénon, P. Ramet, J. Roman, PaStiX : A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems, Parallel Computing 28 (2) (2002) 301–321.

[24] G. Pichon, E. Darve, M. Faverge, P. Ramet, 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, 2017.

[25] A. George, Nested dissection of a regular finite element mesh, SIAM Journal on Numerical Analysis 10 (2) (1973) 345–363.

[26] G. Karypis, V. Kumar, METIS : A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices (1995).

[27] F. Pellegrini, Scotch and libScotch 5.1 User's Guide, user's manual, 127 pages (Aug. 2008).

[28] G. Pichon, M. Faverge, P. Ramet, J. Roman, Reordering Strategy for Blocking Optimization in Sparse Linear Solvers, SIAM Journal on Matrix Analysis and Applications 38 (1) (2017) 226 – 248.

[29] X. S. Li, J. W. Demmel, SuperLU_DIST : A Scalable Distributed-memory Sparse Direct Solver for Unsymmetric Linear Systems, ACM Trans. Math. Softw. 29 (2) (2003) 110–140.

[30] L. Giraud, J. Langou, A Robust Criterion for the Modified Gram-Schmidt Algorithm with Selective Reorthogonalization, SIAM Journal on Scientific Computing 25 (2) (2003) 417–441.

[31] X. Lacoste, Scheduling and memory optimizations for sparse direct solver on multi-core/multi-gpu cluster systems, Ph.D. thesis, Bordeaux University, Talence, France (Feb. 2015).

[32] T. A. Davis, Y. Hu, The University of Florida sparse matrix collection, ACM Trans. Math. Softw. 38 (1) (2011) 1 :1–1 :25. doi :10.1145/2049662.2049663.

[33] E. Agullo, P. R. Amestoy, A. Buttari, A. Guermouche, J.-Y. L'Excellent, F.-H. Rouet, Robust memory-aware mappings for parallel multifrontal factorizations, SIAM Journal on Scientific Computing 38 (3) (2016) C256–C279.

[34] G. W. Howell, J. W. Demmel, C. T. Fulton, S. Hammarling, K. Marmol, Cache efficient bidiagonalization using blas 2.5 operators, ACM Trans. Math. Softw. 34 (3) (2008) 14 :1–14 :33.

[35] P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, T. Mary, On the complexity of the block low-rank multifrontal factorization, SIAM Journal on Scientific Computing 39 (4) (2017) A1710–A1740.

[36] G. L. Miller, S. A. Vavasis, Density graphs and separators, in : Second Annual ACM-SIAM Symposium on Discrete Algorithms, 1991, pp. 331–336.

[37] R. J. Lipton, R. E. Tarjan, A separator theorem for planar graphs, SIAM Journal on Applied Mathematics 36 (1979) 177–189.

[38] M. Sergent, D. Goudin, S. Thibault, O. Aumage, Controlling the Memory Subscription of Distributed Applications with a Task-Based Runtime System, in : 21st International Workshop on High-Level Parallel Programming Models and Supportive Environments, Chicago, United States, 2016.