Sparse Supernodal Solver Using Block Low-Rank Compression

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

Sparse Supernodal Solver Using Block Low-Rank Compression

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

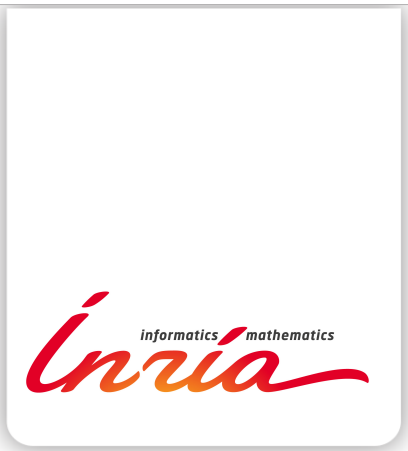
RESEARCH REPORT

N° 9022

January 2017

Project-Team HiePACS

ISSN 0249-6399 ISRN INRIA/RR--9022--FR+ENG



Sparse Supernodal Solver Using Block Low-Rank Compression

Grégoire Pichon*†‡§, Eric Darve¶, Mathieu Faverge*†‡§ ‖,

Pierre Ramet ‡†*§, Jean Roman †*‡§

Project-Team HiePACS

Research Report n° 9022 — January 2017 — 20 pages

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 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 single node with 24 threads and 128 GB of memory are presented on a set of matrices from real-life problems. We demonstrate a memory footprint reduction of up to 4.4 times using the *Minimal Memory* strategy and a computational time speedup of up to 3.3 times with the *Just-In-Time* strategy.

Key-words: Sparse linear solver, block low-rank compression, PaStiX direct solver, multi-threaded architectures

* Bordeaux INP, Talence, France
† Inria Bordeaux - Sud-Ouest, Talence, France
‡ University of Bordeaux, Talence, France
§ CNRS (Labri UMR 5800), Talence, France
¶ Mechanical Engineering Department, Stanford University, United States
‖ University of Tennesse, ICL, Knoxville, USA

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Un solveur supernodal creux utilisant une compression de rang faible par bloc

Résumé : Ce papier présente deux approches utilisant les techniques de compression de rang faible par bloc (BLR) afin de réduire l'empreinte mémoire et/ou le temps de résolution du solveur superndal PaStiX. Cette technique de compression à plat, non hiérarchique, permet de tirer parti des propriétés de rang faible dans les blocs obtenus lors de la factorisation du système linéaire provenant par exemple de la discrétisation des équations différentielles partielles. La première approche, appelée *Minimal Memory*, montre le gain mémoire maximum qu'il est possible d'obtenir avec une compression BLR, alors que la seconde approche, appelée *Just-In-Time*, se concentre principalement sur la réduction du temps de calcul pour la résolution du système. Dans cette étude, nous comparons, en termes de temps de calcul et de consommation mémoire, les noyaux de compression qui utilisent soit la technique de décomposition en valeurs propres singulières (SVD) soit la factorisation QR avec détermination du rang (RRQR). Nous avons réalisé les expériences sur un noeud composé de 24 coeurs avec 128 GB de mémoire sur une collection de matrices issues d'applications réelles. Nous montrons que la consommation mémoire peut-être réduite jusqu'à un facteur 4.4 avec la stratégie *Minimal Memory* et que le temps de calcul peut être divisé par 3.3 en utilisant la stratégie *Just-In-Time*.

Mots-clés : Solveur linéaire creux, compression de rang faible par bloc, PaStiX solveur direct, architectures multi-thread

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 matrices 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 allow a reduction of the memory requirement and/or the time to solution. Depending on the compression strategy, solvers require knowledge of the underlying geometry to tackle the problem or can do it in a purely algebraic fashion.

Hackbusch [**?**] introduced the $\mathcal{H}$-LU factorization for dense matrices, which compresses the matrix into a hierarchical matrix before applying low-rank operations instead of classic dense operations. In the same paper, an extension of the dense version was designed for sparse matrices using nested dissection ordering. In [**?**], $\mathcal{H}$-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 [**?**]. Kriemann [**?**] implemented this algorithm using Direct Acyclic Graphs.

The Hierarchically Off-Diagonal Low-Rank (HODLR) compression technique was used in a multifrontal sparse direct solver in [**?**] to accelerate the elimination of large fronts. It was fully extended for a sparse purpose in [**?**] and uses Boundary Distance Low-Rank (BDLR) to allow both time and memory savings. A supernodal solver using a compression technique close to HODLR was presented in [**?**]. The proposed approach allows memory savings and can be faster than standard preconditioned techniques. However, it is slower than the direct approach in the benchmarks and requires an estimation of the rank to use randomized techniques and accelerate the solver.

There have been different works around the use of Hierarchically Semi-Separable (HSS) matrices in sparse direct solvers. In [**?**], Xia et al. presented a solver for 2D geometric problems, where all operations are realized algebraically. In [**?**], a geometric solver was developed, but contribution blocks are not compressed, making memory savings impossible. [**?**] proposed an algebraic code that uses randomized sampling to manage low-rank blocks and to allow memory savings.

$\mathcal{H}^2$ arithmetic has also been applied to sparse solvers. In [**?**], a fast sparse $\mathcal{H}^2$ solver, called LoRaSp, based on extended sparsification was introduced. In [**?**], a variant of LoRaSp, aimed 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 [**?**] 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 [**?**], an $\mathcal{H}^2$ sparse algorithm was described. It is similar in many respects to [**?**], and extends the work of [**?**]. 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 have also been investigated for dense matrices [**?**], and for sparse linear systems considering a multifrontal method [**?**]. Considering that these approaches are close to the current study, a detailed comparison will be described in Section 5.

The first objective of this work is to combine a generic sparse direct solver with recent work on matrix compression to come up with a way 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 [**?**] 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 from the beginning and exploits complicated low-rank numerical operations to keep the memory cost of the factorized matrix as low as possible. The second one compresses the information as late as possible 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 machine precision.

In Section 1, we go over basic aspects of sparse supernodal direct solvers. The two strategies, introduced in PaStiX, are then presented in Section 2, before detailing low-rank kernels in Section 3. In Section 4, we perform experiments comparing the two BLR strategies with the original approach — that uses only dense blocks — in terms of memory consumption, time-to-solution and numerical behavior. Section 5 surveys in more details 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...).

1 Background

The common approach used by 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 systems solves. In the rest of the paper, we focus on problems leading to sparse systems with a symmetric pattern.

The purpose of the first step is to minimize the fill-in — zeros becoming non-zeros during factorization — 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 [**?**] algorithm is widely used through libraries such as Metis [**?**] or Scotch [**?**]. 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.

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 zeros 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

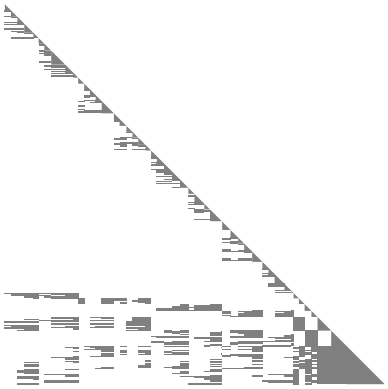


Figure 1: Symbolic factorization of a $10 \times 10 \times 10$ Laplacian partitioned using SCOTCH.

to group off-diagonal blocks together without additional fill-in. A traveler salesman strategy is implemented in PASTIX [**?**] and divides by more than two the number of off-diagonal blocks. Other approaches like [**?**, **?**] 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 re-ordering technique also allows to reduce ranks of the low-rank blocks as shown in [**?**]. To introduce more parallelism and data locality, the final structure can then be split in tiles as it is now commonly done in dense linear algebra libraries. These first two steps of direct solvers are preprocessing stages independent from 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, 2) solve the off-diagonal blocks belonging to this supernode, and 3) apply the updates on the trailing submatrix (cf. Section 2).

2 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 notations 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 or the time-to-solution.

2.1 Notations

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

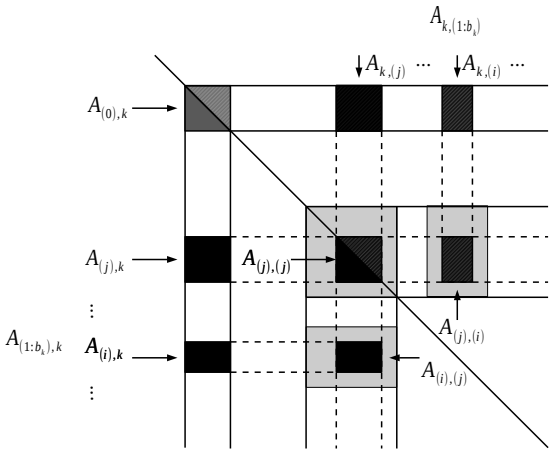


Figure 2: Symbolic block structure and notations used for the algorithms for one column block k, and its associated blocks.

- $A_{(0),k} (= A_{k,(0)})$ 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.

2.2 Sparse direct solver using BLR compression

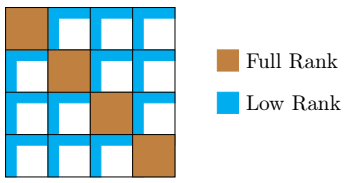


Figure 3: Block Low-Rank compression.

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, which represent long distance interactions in the graph, are low-rank. 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 3.

We propose in this paper to similarly apply this scheme to the symbolic block structure of sparse direct solvers. First, 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. Second, 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 high ranks are kept dense.

Relying on 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: $\hat{A}_{(1:b_k),k}$ = Compress($A_{(1:b_k),k}$)
3: $\hat{A}_{k,(1:b_k)}$ = Compress($A_{k,(1:b_k)}$)
4: **End For**
5: **For** $k = 1$ to N **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**

2.2.1 *Minimal Memory*

This scenario, described by Algorithm 1, starts by compressing the original matrix A. 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 on 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 3).

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: $\hat{A}_{(1:b_k),k} = \text{Compress}(\ A_{(1:b_k),k}\ )$
4: $\hat{A}_{k,(1:b_k)} = \text{Compress}(\ A_{k,(1:b_k)}\ )$
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**

2.2.2 *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 described in Section 3, 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 final *dense* 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 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 allows a faster implementation.

3.1 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 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.

3.1.1 SVD

A is decomposed as $U\sigma V^t$. The low-rank form of A is thus made out of the first r_A singular values and their associated singular vectors such that: $\sigma_{r_A+1} \leq \tau$, $u_A = U_{r_A}$, and $v_A^t = \sigma_{1:r_A} V_{r_A}^t$ with U_{r_A} being the first r_A columns of U, and respectively for V. This process requires $\Theta(m_A^2 n_A + n_A^2 m_A + n_A^3)$ operations.

3.1.2 RRQR

A is decomposed as PQR, where P is a permutation matrix, and QR the QR decomposition of $P^{-1}A$. The rank-r_A form of A is then formed by $u_A = Q_{r_A}$, the first r_A columns of Q, and $v_A^t = R_{r_A}$, the first r_A rows of R. The main advantage of this process is that it can stop the factorization as soon as the norm of the trailing submatrix $\tilde{A}_{(r_A+1:m_A, r_A+1:n_A)} = A - PQ_{r_A}R_{r_A}$ is lower than τ. Thus, the complexity is lowered to $\Theta(n_A r_A^2)$ operations.

SVD compression 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 a better numerical accuracy. Thus, there is a trade-off between time-to-solution (RRQR) versus memory consumption and numerical accuracy (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 take inherently advantage of these zeros. In addition, most of the low-rank compressions are applied to blocks stored as dense blocks and it represents the main part of the computations.

3.2 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 apply a dense solve only to u_b^t, and the complexity is only $\Theta(m_L^2 r_x)$, instead of $\Theta(m_L^2 n_L)$ for the dense representation.

3.3 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 in two steps: the product of two low-rank blocks, and the addition of a low-rank block and either a dense block (*LR2GE*), or a low-rank block (*LR2LR*).

3.3.1 Low-rank matrices 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 [**?**] that the rank r_{AB} of the product of two low-rank matrices of ranks r_A and r_B is usually smaller than $\min(r_A, r_B)$. Moreover, u_A and u_B are both orthogonal, so the matrix $T = (v_A^t v_B)$ 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:

$$T = v_A^t v_B \tag{1}$$
$$\hat{T} = \widehat{v_A^t v_B} = u_T v_T^t \tag{2}$$
$$u_{AB} = u_A u_T \tag{3}$$
$$v_{AB}^t = v_T^t v_B^t. \tag{4}$$

3.3.2 Low-rank matrices 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 supernodal methods. This is illustrated for example by the update block $A_{(i),(j)}$ in Figure 2.

If C is not compressed as 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 $\Theta(m_{AB}\ n_{AB}\ r_{AB})$.

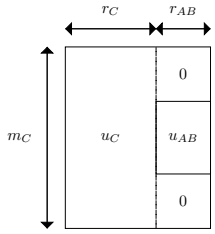


Figure 4: Accumulation of two low-rank matrices when sizes do not match.

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. This is the commonly named *extend-add* operation. Without further optimization, this operation costs only two copies. In the case of supernodal method, adequate padding is also required to align the vectors coming from AB, and C matrices as it is presented in Figure 4 for the u vectors. The operation on v is similar.

One can notice that, kept as this, 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 original dense 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 it first requires to compute a QR decomposition for both composed matrices:

$$[u_C, u_{AB}] = Q_1 R_1 \text{ and } [v_C, -v_{AB}] = Q_2 R_2. \tag{7}$$

Then, the temporary matrix $T = R_1 R_2^t$ is compressed using the SVD algorithm described previously. This gives the final $\hat{C'}$ with:

$$u_{C'} = (Q_1 u_T) \text{ and } v_{C'} = (Q_2 v_T). \tag{8}$$

The complexity of this operation is decomposed as follows: $\Theta((m_C + n_C)(r_C + r_{AB})^2)$ for the QR decomposition 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 orthogonality of both u_C and u_{AB} to first orthogonalize u_{AB} with respect to u_C:

$$u^*_{AB} = u_{AB} - u_C(u_C^t\ u_{AB}). \tag{9}$$

We obtain an orthonormal basis $[u_C, u^*_{AB}]$ such that:

$$[u_C, u_{AB}] = [u_C, u^*_{AB}] \times \begin{pmatrix} I & u_C^t u_{AB} \\ 0 & I \end{pmatrix}. \tag{10}$$

We follow by applying the RRQR algorithm to:

$$\begin{pmatrix} I & u_C^t u_{AB} \\ 0 & I \end{pmatrix} \times \left([v_C, v_{AB}]\right)^t = PQR. \tag{11}$$

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

$$u_{C'} = ([u_C, u^*_{AB}]PQ_k) \text{ and } v^t_{C'} = R_k. \tag{12}$$

Note that $u_{C'}$ is kept orthogonal 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 (11), $\Theta((r_C + r_{AB})n_C\ r_{C'})$ to apply the RRQR algorithm, and finally again $\Theta((r_C + r_{AB})n_C\ r_{C'})$ to compute the final $u_{C'}$.

3.4 Summary

Table 1 presents the computational complexity for the two low-rank strategies with respect to the original version of the solver. To get the main factor of the complexity, we make the assumption that $m_C \geq m_A \geq m_B$, $r_A \geq r_B$, $m_C \geq n_C$, and $r_C \leq r_{C'}$. One can note that the *Just-In-Time* strategy performs the calculation of the low-rank contribution before assembling the matrix explicitly to apply a dense modification. The main factor of the complexity 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 to use either SVD or RRQR recompression, for which the complexity depends on m_C and n_C, the dimensions of the block C. It explains why this strategy is slower than the original solver.

When considering dense matrices, a low-rank matrix is usually modified by a contribution of the same size: the low-rank extend-add process may be efficient and lead to performance gain [**?**]. It is also the case for the CUFS strategy in BLR-MUMPS, which compresses a dense front before applying operations between low-rank blocks of the same size.

In our case, a block C receives many small contributions, see Figure 1, as stated by the separator theorem [**?**] describing how the size of supernodes is evolving during the nested dissection process. According to our experiments, it is still interesting to have low-rank blocks at the end of the factorization, meaning that ranks remain lower than $\min(m_C, n_C)/4$ (otherwise compression will not help), even if blocks received a large number of contributions. Thus, $r_{C'}$ is often close or equal to r_C and lower than $r_C + r_{AB}$: the rank is often invariant applying a small contribution. So it is less expensive to use RRQR recompression (and operations are more suitable for performances). In terms of complexity, the recompression depends on the size of the target block C and not on the size of the contribution blocks A and B. As huge low-rank blocks are recompressed many times, it makes the *Minimal Memory* scenario slower than the dense version, but allows consequent memory savings.

Finally, 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 final structure of the dense factorized matrix is never allocated, and the low-rank structure needs to be maintained throughout the factorization process to lower the memory peak.

In order to overcome the issue of expensive low-rank additions, an idea would be to consider randomized techniques to allow an extend-add process depending on the size of contributing blocks and not on the size of the target block.

4 Experiments

Experiments were conducted on the *Plafrim*[1] supercomputer, and more precisely on the `miriel` cluster. Each node is equipped with two Intel `Xeon E5-2680 v3` 12-cores running at 2.50 GHz and 128 GB of memory. The Intel MKL `2016` is used for BLAS and SVD kernels. The RRQR kernel is coming from the BLR-MUMPS solver [**?**], and is an extension of the block rank-revealing QR factorization subroutines from LAPACK 3.6.0 (xGEQP3).

The PaStiX version used for our experiments is available on the public git repository[2] as the tag `papers/pdsec17`. The multi-threaded version used is the static scheduling version presented in [**?**].

For the initial ordering step, we used Scotch [**?**] 5.1.11 with the configurable strategy string from PaStiX to set the minimal size of non separated sub-graphs, *cmin*, to 15. We also set the $frat$ parameter to 0.08, meaning that columns 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 in blocks of size at least 128 to create more parallelism while keeping sizes large enough. The same 128 criteria is used to defined 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.

Experiments were computed on a set of 3D matrices extracted from The SuiteSparse Matrix Collection [**?**]:

- *Atmosmodj*: atmospheric model (1 270 432 dofs)
- *Audi*: structural problem (943 695 dofs)
- *Hook*: model of a steel hook (1 498 023 dofs)
- *Serena*: gas reservoir simulation (1 391 349 dofs)
- *Geo1438*: geomechanical model of earth (1 437 960 dofs)

We also used 3D laplacian generators (7 points stencils), and defined *lap*120 as a laplacian of size 120^3.

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

4.1 SVD versus RRQR

The first experiment studies the behavior of the two compression methods coupled both with *Minimal Memory*, and *Just-In-Time* scenario, on the matrix *Atmosmodj*. Table 2 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 coefficient of the factorized matrix.

[1] `https://plafrim.bordeaux.inria.fr`
[2] `https://gitlab.inria.fr/solverstack/pastix`

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 3. 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 a 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 zeros and are lower ranks than once they have been updated. The time of 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 3). In both scenarios, SVD kernels are able to keep the useful information and compress the final coefficients with similar rates, while the RRQR kernels are not as efficient to capture the information and to compress the blocks efficiently with the *Minimal Memory* scenario.

The diagonal blocks 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 factors final size, since this size reflects the final ranks of the blocks.

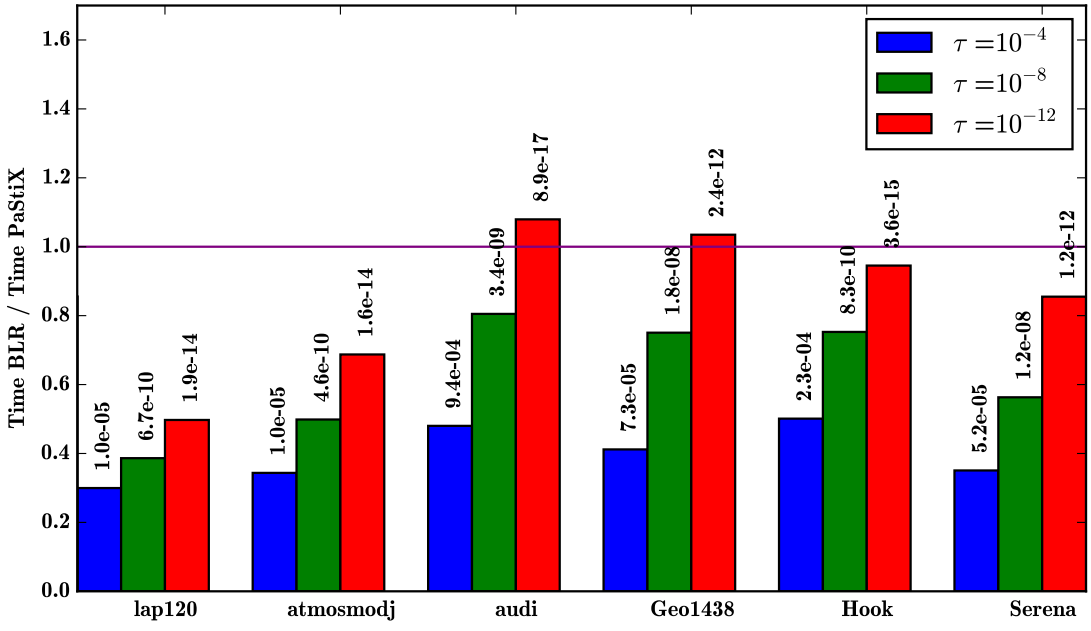
To conclude, the *Minimal Memory* scenario is not able to compete with the original direct factorization due to the costly update addition. However, it reduced the memory peak of the solver to the factors final size. The *Minimal Memory*/RRQR offers a 25% memory reduction with a time to solution doubled in sequential. The *Just-In-Time* scenario competes with the original direct factorization, and divide by two the time-to-solution with RRQR kernels.

4.2 Performance

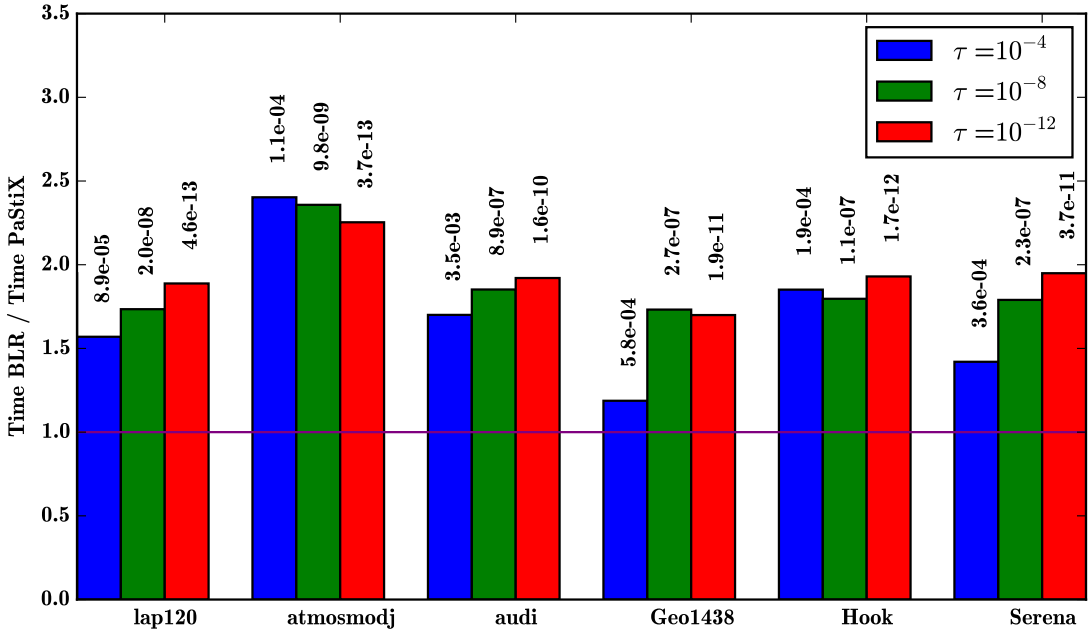
Figure 5 presents the overall performance achieved by the two low-rank scenarios with respect to the original version of the solver (where lower is better) on the previously introduced set of 6 matrices. All versions are multi-threaded implementations and use all the 24 cores of one node. The scheduling used is the PaStiX static scheduler developed for the original version, that is the only one available in the new development branch for now. This might have a negative impact on the low-rank implementations 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 first residual of the solver, which went through one refinement step is shown with the backward errors printed on top of each bar.

Figure 5(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 requires low accuracy, as seismic for instance, can benefit up to a 3.3 speedup. Figure 5(b) shows that it is more difficult for the *Minimal Memory*/RRQR scenario to be competitive. The performance is always degraded with respect to the original PaStiX performance, with an average loss around a factor of 1.8, 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 computations, 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.



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



(b) *Minimal Memory* scenario using RRQR.

Figure 5: Performance of both strategies with 3 tolerance thresholds, backward error of the solution is printed on top of each bar.

4.3 Memory consumption

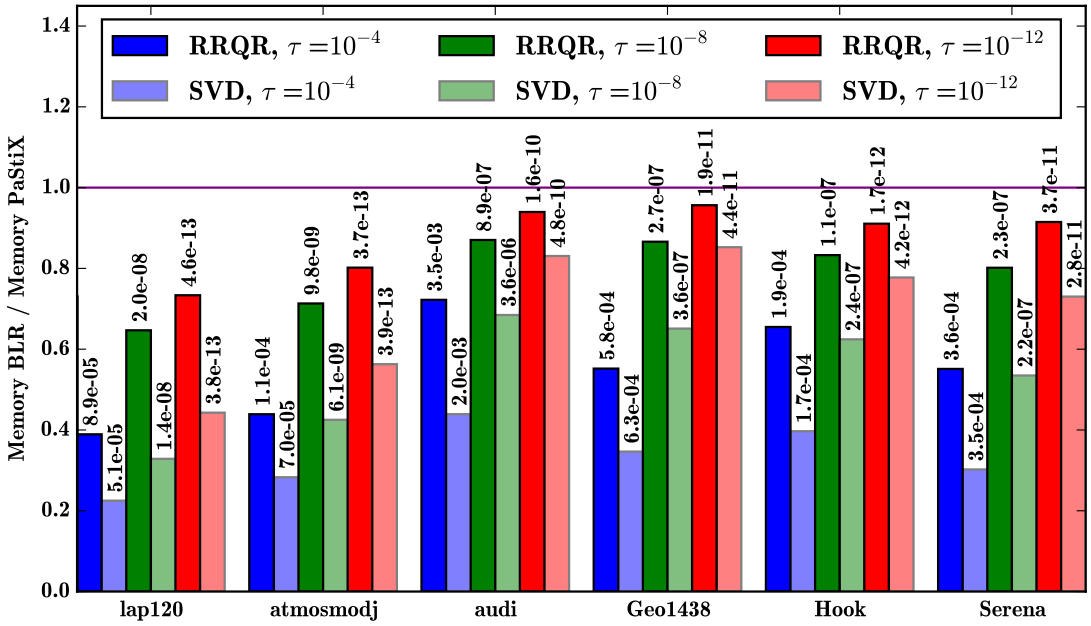


Figure 6: Memory peak for the *Minimal Memory* scenario with 3 tolerance thresholds and both SVD and RRQR kernels.

The *Minimal Memory* scenario is slower than the original solver, but it is a strategy that efficiently reduces the memory peak of the solver. Figure 6 presents the gain on the memory used to store the factors at the end of the factorization of the set of 6 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 a better compression rate by finding smaller ranks for a given matrix and a given tolerance. The quality of the first residual is also 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 which can be more than 50% with larger tolerance (10^{-4}).

Figure 7 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 4 million unknowns, and the size of the factors quickly increase for larger sizes. With the *Minimal Memory*/RRQR scenario, we have now been able to run a 3D problem up to 12 million unknowns when relaxing the tolerance to 10^{-4}.

The memory of the *Just-In-Time* scenario has not been studied, as long as in our supernodal approach, each supernode is fully allocated in a dense fashion in order to accumulate the update 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

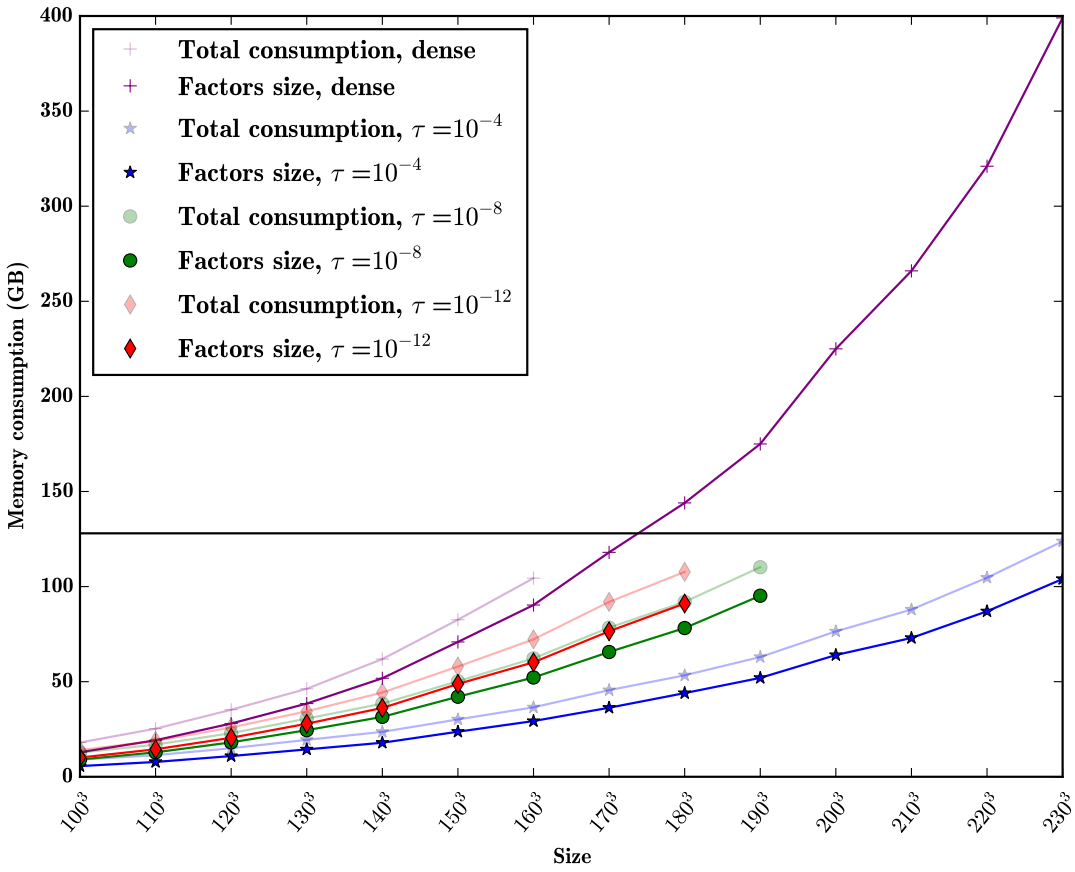


Figure 7: Memory scalability with 3 tolerance thresholds for the *Minimal Memory*/RRQR scenario when increasing the size of 3D Laplacians.

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 are the scheduling strategies presented in [**?**] to keep the memory consumption of the solver under a given limit.

4.4 Convergence and numerical stability

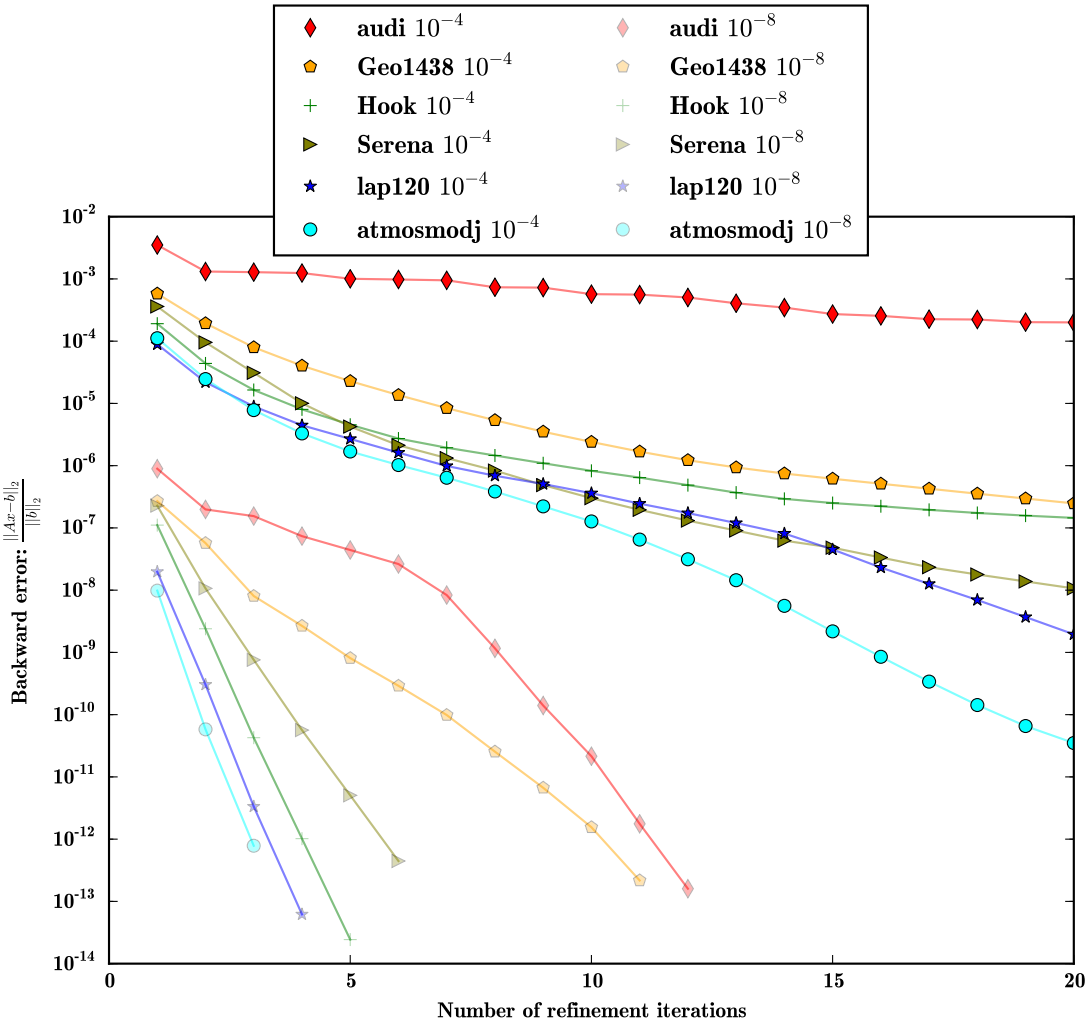


Figure 8: Convergence speed for the *Minimal Memory*/RRQR scenario with 2 tolerance thresholds.

Figure 8 presents the convergence of the iterative solver — GMRES for general matrices and Conjugate Gradient (CG) for SPD matrices — 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 iterative refinement process benefits from the compression, as the solve step, and is accelerated.

5 Discussion

In this section, we discuss the positioning of our solver with the closest related works and we give some limitations to extend this work to a hierarchical format.

Contrary to the approach studied in [**?**], we perform a symbolic block factorization. In their approach, as in our proposition, 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 zeros are included to generate the low-rank representation. Thus, they do not have extend-add (*LR2LR*) operation between low-rank blocks of different sizes, but the memory consumption is higher because some structural zeros are not managed.

A dense BLR solver was designed by Livermore Software Technology Corporation [**?**]. In this work, the full matrix is compressed at the beginning and operations between low-rank blocks are performed. This approach is similar to our *Minimal Memory* scenario in the context of dense matrices. Due to this restriction, the extend-add process concerns low-rank matrices of the same size, without zeros padding. Thus, the *LR2LR* operation is less costly than the dense update in their context.

A BLR multifrontal sparse direct solver was designed for the MUMPS solver. The strategy is described in [**?**] and a theoretical study of the complexity of the solver is presented in [**?**]. In this work, contribution blocks are not compressed. When a front is eliminated, different strategies are proposed to enhance the time-to-solution. Our scenario *Just-In-Time* is close to their FCSU (Factor, Compress, Solve, Update) strategy. 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*, but would implies larger ranks in the extend-add operations of the *Minimal Memory*. The CUFS (Compress, Update, Factor, Solve) is the strategy closest to our *Minimal Memory* scenario. However, only a dense front is fully compressed before being eliminated: contributing blocks are not compressed and low-rank operations occur within a dense matrix, similarly to the previous work from LSTC. If the time-to-solution is better with BLR-MUMPS, there is more room for memory savings in our approach.

With the aim of extending our solver to hierarchical compression schemes, such as $\mathcal{H}$, HSS, or HODLR, we consider graphs issued from finite element meshes coming from real-life simulations of 3D physical problems. From a theoretical point of view, the majority of these graphs have a bounded degree and thus good separators respecting the separator theorems [**?**] can be built. For a n-vertices mesh, the 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 the memory requirements, the direct approach leads to an overall storage in $\Theta(n^{\frac{4}{3}})$, 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 block of this separator. They are asymptotically the same size. Previous studies have shown that such a block may have a rank in order of $\Theta(n^{\frac{1}{3}})$.

For the *Minimal Memory* scenario, we have seen that the time-to-solution is longer than the dense version. As low-rank blocks become larger in the hierarchy, it will be even worse than the

solution we developed. For the *Just-In-Time* scenario, maintaining such a block in a dense form before compressing block requires $\Theta(n^{\frac{4}{3}})$ memory and does not satisfy the memory complexity we target. It also means that a compromise between *Minimal Memory* and *Just-In-Time* strategies using a *Left-Looking* approach might not be a relevant solution. Currently, no sparse solver is able to perform efficiently the extend-add operations using compression techniques such as SVD or RRQR, and it is still an open problem.

Conclusion

We presented a new Block Low-Rank sparse solver that combines an existing sparse 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. *Minimal Memory* saves memory up to a factor of 2.6 using RRQR kernels, with a time overhead that is limited to 2.4 despite the higher complexity. 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 and/or large number of right hand sides are involved.

Just-In-Time reduces both the time-to-solution by a factor up to 3.3, and the memory requirements of the final factorized matrix with similar factors to *Minimal Memory*. However, with the actual scheduling strategy, this gain is not reflected on 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 will require only a few iterations before reaching the machine precision.

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 combines *Right-Looking* and a *Left-Looking* approach in order to find a good compromise between memory and parallelism for the targeted architecture. This will follow up recent work on applying parallel runtime systems [**?**] to the PaStiX solver.

Acknowledgments

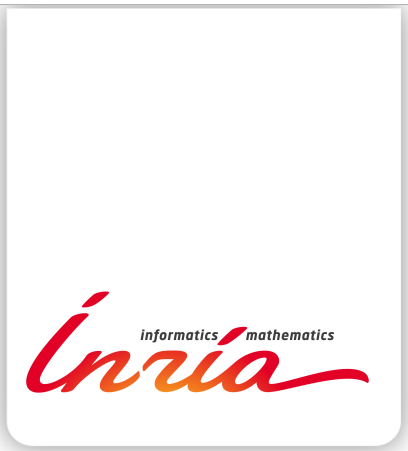
This material is based upon work supported by the DGA under a DGA/Inria grant. Experiments presented in this paper were carried out using the PLAFRIM experimental platform.

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 matrices 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 matrices addition	–	–		(7): $\Theta(m_C(r_C + r_{AB})^2)$	(9): $\Theta(m_C\ r_C\ r_{AB})$
				(SVD): $\Theta((r_C + r_{AB})^3)$	(11): $\Theta(n_C(r_C + r_{AB})r_{C'})$
				(8): $\Theta(m_C(r_C + r_{AB})r_{C'})$	(12): $\Theta(m_C(r_C + r_{AB})r_{C'})$
Dense update	$\Theta(m_A\ m_B\ n_A)$	$\Theta(m_A\ m_B\ r_{AB})$		–	–
Main factor	$\Theta(m_A\ m_B\ n_A)$	$\Theta(m_A\ m_B\ r_{AB})$	$\Theta(m_A\ m_B\ r_{AB})$	$\Theta(m_C(r_C + r_{AB})^2)$	$\Theta(m_C(r_C + r_{AB})r_{C'})$

Table 2: Costs distribution on the Atmosmodj matrix with $\tau = 10^{-8}$

	Dense	*Just-In-Time*		*Minimal Memory*	
		RRQR	SVD	RRQR	SVD
Factorization time (s)					
Compression	-	49.53	418.5	15.20	180.9
Block factorization	0.9635	1.000	1.003	1.074	1.104
Panel solve	15.80	6.970	6.526	11.16	6.946
LR product	-	64.10	91.15	193.1	94.36
LR addition	-	-	-	774.6	6523
Dense udpate	418.7	47.94	47.03	-	-
Total	*436*	*169*	*564*	*995*	*6806*
Solve time (s)	2.43	1.54	1.8	2.22	1.29
Factors final size (GB)	15.9	7.4	6.86	11.4	6.76



RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399