Using Block Low-Rank compression in mixed precision for sparse direct linear solvers

Matthieu Gerest

THÈSE DE DOCTORAT
DE SORBONNE UNIVERSITÉ

Spécialité : Informatique

École doctorale nº130: Informatique, Télécommunications et Électronique

réalisé au

Laboratoire d'Informatique de Paris 6 (UMR 7606)

présentée par

Matthieu GEREST

Sujet de la thèse :

Using Block Low-Rank compression in mixed precision for sparse direct linear solvers

soutenue le 8 novembre 2023

devant le jury composé de :

M^{me}	Barucq Hélène	Directrice de recherche, INRIA	Examinatrice
M.	Boiteau Olivier	Industriel, EDF R&D	Co-encadrant
M.	Duff Iain	Directeur de recherche, Rutherford Appleton Laboratory	Rapporteur
M.	Giraud Luc	Directeur de recherche, INRIA	Rapporteur
M^{me}	Jézéquel Fabienne	Maîtresse de conférence, LIP6	Directrice de thèse
M.	Mary Théo	Chargé de recherche, LIP6	Co-encadrant
M.	Nataf Frédéric	Directeur de recherche, Laboratoire Jacques-Louis Lions	Président du jury

Acknowledgements

Je souhaite tout d'abord remercier mes encadrants pour leur aide durant ces trois années. Merci à tous pour votre soutien, votre bienveillance, et vos retours constructifs tout au long de ma thèse. Je tiens à remercier tout particulièrement Théo Mary, qui m'a accompagné et guidé sur le plan scientifique et technique au cours de la thèse, ainsi qu'au cours du stage qui l'a précédée. Nos réunions de travail ont régulièrement soulevé des pistes intéressantes et prometteuses, dont certaines ont été approfondies dans cette thèse. Je souhaite remercier Olivier Boiteau, mon tuteur à EDF, qui m'a sensibilisé au contexte d'utilisation de MUMPS dans les codes d'EDF. Merci pour ton suivi, tes bons conseils et ton humour. Merci à ma directrice de thèse, Fabienne Jézéquel, pour ses relectures ainsi que pour son aide lors des différentes procédures administratives. Je tiens à remercier également mes autres encadrants de l'équipe MUMPS : Patrick Amestoy, Alfredo Buttari et Jean-Yves L'Excellent.

Merci à Iain Duff et à Luc Giraud qui ont accepté d'être rapporteurs de cette thèse, qui ont relu attentivement le manuscrit et qui ont fait des retours pertinents. Je tiens à remercier également les autres membres du jury, Frédéric Nataf et Hélène Barucq.

Je souhaite remercier Alexei Mikchevitch, qui, étant chef de projet à l'époque, a été d'un grand support pour le lancement de cette thèse CIFRE à EDF R&D. Merci aux membres de l'ancien groupe I23 d'EDF, avec qui j'ai partagé les (longues) pauses café quotidiennes, aux discussions animées. Je remercie également mes différents collègues thésards, notamment Bastien, Roméo, Matthieu et Dimitri, à qui je souhaite une bonne continuation.

Enfin, je tiens à remercier ma famille, qui m'a soutenu pendant cette thèse : mes parents, ma sœur Adeline et mon frère Corentin.

Contents

Introduction

Many industrial and scientific simulations require to solve linear systems of the form $Ax = b$. Being able to perform such an algorithmic step with enough accuracy is crucial for of feasibility and performance of the simulation and may take a huge part of the computation time. It is also the case for the codes developed at EDF R&D, such as code_aster, code_carmel and code_saturne.

In order to solve a linear system in a robust and accurate way, one might need to use a direct method such as performing an LU factorization $A = LU$, before solving triangular systems $Ly = b$ and $Ux = y$. The multifrontal method and the supernodal method are both part of this category of algorithms, while handling sparse matrices, whose coefficients are mostly zero.

However, direct methods can be very costly in terms of memory consumption and computation time. As an addition to the use of massive parallelism, one may try to take advantage of the properties of data sparsity of the matrices manipulated in order to further reduce the complexity. Indeed, carefully chosen sub-matrices of A, L and U can often be approximated by matrices of low ranks. By doing this we obtain a Block Low-Rank compression (BLR) that can be exploited to reduce the computational complexity of direct solvers (Amestoy et al., 2017) and therefore reduce their time and memory consumption significantly. The BLR format is notably used in the MUMPS (Amestoy et al., 2001, 2019a), PaStiX (Hénon et al., 2002; Pichon et al., 2018; Pichon, 2018), and STRUMPACK (Rouet et al., 2016; Ghysels et al., 2016) solvers.

Another perspective to improve the performance of numerical methods is the use of mixed-precision algorithms. Low precision is used to improve the performance of the computations: speed, memory, and energy consumption. However, those low-precision formats are carefully combined with higher ones so that the result will have a high accuracy. This category of algorithms has recently known a renewed interest due to the growing availability of low-precision formats on modern hardware, such as fp16 and bfloat16.

The goal of this work is to develop new techniques that aim at further improving the performance gains of using BLR compression within a direct method, in particular based on mixed precision.

In chapter 2 we investigate the possibility of combining low-rank approximations with mixed-precision arithmetic. The main idea is that not all coefficients of a low-rank approximation are equally important. We try to store different components in different precision formats, the components with the lowest norms being kept and used in low

precision. We justify this approach with an error analysis in the case where an arbitrary number of floating-point formats are used. As a result, we obtain a criterion determining what precision format should be used for each component. Our numerical experiments suggest that the proposed mixed-precision low-rank representation presents a high potential: a large fraction of both the entries needed to represent BLR matrices and the floating-point operations needed to compute their LU factorization could be switched to lower precisions.

In chapter 3 we adapt the approach described in chapter 2 for dense matrices to the factorization of sparse matrices based on a multifrontal method. Indeed, in the multifrontal method we process a sequence of dense frontal matrices on which the algorithms presented on chapter 2 for the dense LU factorization and its BLR variant can be applied. We propose two main contributions, while being more focused on the practical aspects of the algorithms. First, we describe how mixed-precision BLR (MPBLR) can be used as a storage format only, and could benefit from using custom floating-point formats. We implemented it in the multifrontal solver MUMPS. Our experiments show that the memory peak is significantly reduced at the price of a small time overhead, without compromising the accuracy. Second, we describe algorithms that perform most computations using low precision in the phases of factorization and triangular solution. We wrote a high-performance implementation of the solution phase within the MUMPS solver. The computation time is reduced, while preserving the previous memory gains, in the case there is only one right-hand side (RHS) to the linear system.

Our experiments revealed a weakness of the BLR triangular solution phase in case we have several RHS: in MUMPS, the time gains of both the classical and the mixed-precision BLR variants diminish when increasing the number of RHS. After analyzing the algorithm in case of multiple RHS, we conclude that the data locality is rather poor, which motivates us to rethink the communication and memory access patterns. In chapter 4 we propose a new algorithm in which the order of the block operations has been modified. We carry out a communication volume analysis that shows that the data movement is reduced and therefore the data locality is improved. We implement several variants of the algorithms, including one that is better adapted to the parallelism scheme of MUMPS. The latter obtains time gains compared to the standard algorithm, in case of multiple RHS.

In order to make this thesis self-contained, we also provide a chapter of general background (chapter 1). In particular we explore the techniques used in the multifrontal method, how to handle BLR compressions, the basics of rounding error analysis, as well as the state of the art of the mixed-precision algorithms that are somewhat related to this thesis.

Chapter 1

Background

1.1 The multifrontal method

1.1.1 Gaussian elimination

Gaussian elimination (Algorithm 1.1) is one of the most basic direct methods for solving a linear system $Ax = b$. Its main step consists in performing a series of updates on each coefficient: overall, we will subtract $\sum_{k<\min(i,j)} l_{ik}u_{kj}$ from each coefficient a_{ij}. The factors L and U are progressively filled with their final values, one column at a time for L, and one row at a time for U. After the last step of the algorithm (should there be no division by 0), we obtain a decomposition $A = LU$ of the original matrix, where L is a lower triangular matrix and U an upper triangular matrix. Then, we solve the two triangular systems $Ly = b$ and $Ux = y$ using Algorithms 1.2 and 1.3 respectively.

Algorithm 1.1 Gaussian elimination (right-looking)

1: $L \leftarrow 0$
2: $U \leftarrow 0$
3: **for** k=1 **to** n **do**
4: **for** i=k **to** n **do**
5: $l_{i,k} \leftarrow a_{i,k}/a_{k,k}$
6: $u_{k,i} \leftarrow a_{k,i}$
7: **end for**
8: **for** i=k+1 **to** n **do**
9: **for** j=k+1 **to** n **do**
10: $a_{i,j} \leftarrow a_{i,j} - l_{i,k}u_{k,j}$
11: **end for**
12: **end for**
13: **end for**

Algorithm 1.2 Forward elimination: Solve Ly=b (left-looking)

1: $y \leftarrow b$
2: **for** i=1 **to** n **do**
3: $y_i \leftarrow (y_i - \sum_{j=1}^{i-1} l_{ij}y_j)/l_{ii}$
4: **end for**

Algorithm 1.3 Backward substitution: Solve Ux=y (left-looking)

1: $x \leftarrow y$
2: **for** i=1 **to** n **do**
3: $\quad x_i \leftarrow (x_i - \sum_{j=i+1}^{n} u_{ij} y_j)/u_{ii}$
4: **end for**

Gaussian elimination is often implemented in-place in practice, and it will also be the case for the variants presented in this thesis. The computed columns of L are successively stored in the lower part of A, overwriting the corresponding coefficients of A that are no longer needed. Similarly, the computed rows of U are stored in the upper part of A. We note that, as Algorithm 1.1 is written, the diagonal coefficients of L have a value 1. Thus, it is possible not to store them explicitly, which leaves just enough space for storing the coefficients of U, including the diagonal. Also, the original matrix A is overwritten when performing the updates. In short, the matrix that is manipulated by actual implementations of the algorithm is, after step k, $(L - I_k) + U + A_{sub}$, where A_{sub} is obtained from the current matrix A by replacing all unused coefficients by 0, leaving only the $k \times k$ coefficients of the bottom-right block. I_k is a matrix whose k first diagonal coefficients are 1, the others being 0.

In a less naive approach, one may want to add a pivoting step to Algorithm 1.1. After line 3, we may exchange row k with another row $i_p \geq k$, and column k with a column $j_p \geq k$. As a consequence, the result obtained at the end of the algorithm is a factorization $PAQ = LU$, and the right-hand side of the linear system will have to be adapted as well by performing pivoting operations on b and/or on x.

One of the reasons behind the need to add such a pivoting step is that we cannot perform the division by $a_{k,k}$ if its value is 0. Moreover, we try to avoid dividing by a small value of $a_{k,k}$ for numerical reasons when using an inexact floating-point arithmetic. As a result, a first strategy known as complete pivoting consists of choosing as pivot the largest eligible coefficient, which requires pivoting on both the rows and columns. However, complete pivoting is rather costly, and this is why we will rather consider a cheaper alternative: we only perform interchanges on the rows, taking the largest coefficient of the column as the pivot for instance. This pivoting strategy is known as partial pivoting, and in practice it is often sufficient to obtain numerical stability. As a result, we obtain a decomposition $PA = LU$ of the original matrix.

This algorithm can be adapted to the symmetric case: given a symmetric matrix A, we compute a factorization $A = LDL^T$. The number of coefficients to store and the number of operations to perform on them should both be divided by a factor 2. The pivoting strategies must be adapted in order to keep the matrix symmetric at each step: the pivots are typically chosen on the diagonal instead of the current column.

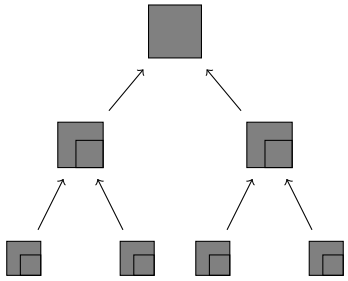


Figure 1.1: Elimination tree of a sparse matrix $\mathcal{A}$

1.1.2 Adapting Gaussian elimination to sparse matrices

Other versions of the Gaussian elimination handle the case where we want to factorize a matrix $\mathcal{A}$ that is structurally sparse, with most of its coefficients being 0. However, with a naive choice of pivots, most of the coefficients of the factors are non-zero: a huge fill-in occurs, and the sparsity is lost. On the other hand, if the pivots are chosen so as to minimize this fill-in, then we may obtain factors $\mathcal{L}$ and $\mathcal{U}$ that are relatively sparse (although not quite as much as the original matrix). The number of operations performed is greatly reduced as well, because all operations performed on structural zeros are skipped.

The multifrontal method is one of these algorithms. It was first formally introduced in Duff and Reid (1983). It is composed of three main phases:

- During the analysis phase, we choose a main pivoting strategy, choosing pivots on the diagonal in a way that reduces the fill-in of the factorization. This is done by running a certain graph algorithm on the adjacency graph of the matrix. At the end of this phase, we obtain an elimination tree (see Figure 1.1), which is the structure that will be manipulated. It is a representation of the matrix $\mathcal{A}$.

- Factorization phase: The elimination tree of $\mathcal{A}$ is modified, by performing the updates of its coefficients. After each of its coefficient has been fully summed, we obtain a tree representing the factors $\mathcal{L}$ and $\mathcal{U}$.

- Solution phase: We solve the triangular systems $\mathcal{L}x = b$ and $\mathcal{U}y = x$.

At the end of the analysis, we obtain an elimination tree of the sparse matrix $\mathcal{A}$. Each of its nodes corresponds to a dense submatrix of $\mathcal{A}$, and is called a front.

We perform a bottom-up traversal of this elimination tree. At each node we perform a partial LU factorization of the current front, as illustrated in Figure 1.2. As a result, we obtain new fully summed parts, that is to say coefficients on which all updates have been performed. They are replaced by L and U, which are submatrices of $\mathcal{L}$ and $\mathcal{U}$: they respectively contain all the non-zero coefficients of a set of columns of $\mathcal{L}$, and all the non-zero coefficients of a set of rows of $\mathcal{U}$. The contribution block (CB) corresponds to a certain partial sum of updates, to be added with the right coefficients of the parent front during the assembly step.

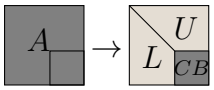


Figure 1.2: Partial LU factorization of a front. The fully summed parts of A are the first rows and the first columns. They become L and U after the partial factorization. The contribution block (CB) corresponds to a certain sum of updates, to be added with the right variables of the parent front during the assembly step.

Factorization of a frontal matrix

The partial LU factorization of a frontal matrix $\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ can be seen as performing the following steps:

- Compute an LU factorization $L_{11}U_{11}$ of A_{11}
- Compute $L_{21} = A_{21}U_{11}^{-1}$ (triangular solve)
- Compute $U_{12} = L_{11}^{-1}A_{12}$ (triangular solve)
- Update the contribution block A_{22}, which becomes the Schur complement of the original frontal matrix : $A_{22} \leftarrow A_{22} - L_{21}U_{12}$

In order to save some memory space, these four operations are once again performed in-place in practice. As a result, U_{11} and L_{11} are stored in the same memory space as A_{11} (without storing the 1s of the diagonal of L_{11}), and A_{21} and A_{12} are overwritten by L_{21} and U_{12}.

We note that all these operations can be performed using well-parallelized linear algebra routines, such as BLAS-3 routines. Therefore, we can expect them to be performed very efficiently in practice.

Assembly

Before performing the factorization of a node, the contributions of all its descendants have to be added to its coefficients. Therefore, the contribution blocks S_1 and S_2 of the children nodes are added through an extend-add operation of the form $A \leftarrow A \oplus S_1 \oplus S_2$: this is the assembly step.

However, the cost for storing the contribution blocks before the assembly can be quite high, and those temporary variables represent a huge part of the memory peak.

Triangular solution

The factorization of $\mathcal{A}$ produced a modified elimination tree illustrated in Figure 1.3, containing the factors $\mathcal{L}$ and $\mathcal{U}$. We now solve the two triangular systems $\mathcal{L}x = b$ and $\mathcal{U}y = x$. The nodes are visited from the bottom to the top of the tree for the former, and from the top to the bottom for the latter.

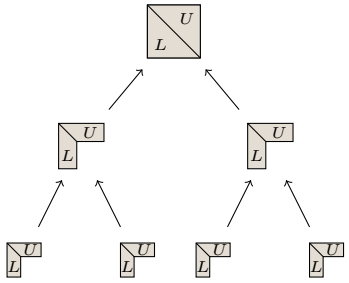


Figure 1.3: Elimination tree after the factorization of $\mathcal{A}$: it now contains a representation of the factors $\mathcal{L}$ and $\mathcal{U}$.

On each front, we perform a series of operations on the right-hand side $\begin{bmatrix} X_1 \\ X_2 \end{bmatrix}$ using the L factors $\begin{bmatrix} L_1 \\ L_2 \end{bmatrix}$:

- Triangular solve: $X_1 \leftarrow L_1^{-1} X_1$
- Update: $X_2 \leftarrow X_2 - L_2 X_2$

Similarly, the operations performed during the backward substitution on the right-hand side $\begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix}$ using the factors $\begin{bmatrix} U_1 & U_2 \end{bmatrix}$ are the following :

- Update: $Y_1 \leftarrow Y_1 - U_2 Y_2$
- Triangular solve: $Y_1 \leftarrow U_1^{-1} Y_1$

Before treating a front, the right-hand side X has to be initialized from the children node, similarly to the assembly step of the factorization.

We may want to solve a linear system with several right-hand sides (RHS) instead of having only one. In this case, x, y, X and Y will not be a 1-dimensional vectors, but matrices with several columns instead. This modification is mostly transparent: for example, matrix-vector products are replaced with matrix-matrix products.

1.1.3 Parallelism

We distinguish two main sources of parallelism in the multifrontal method:

- Different branches of the elimination tree are independent and can be traversed concurrently. This is referred to as tree parallelism.
- The partial LU factorization of large enough fronts may also be performed on several processes/threads: this is node parallelism.

We note that tree parallelism induces an extra memory cost: if the factorizations of two nodes are performed in parallel, then the working space is duplicated. In particular, the contribution blocks are large-size temporary variables. Using tree parallelism will

duplicate them, and they will end up taking a huge part of the memory consumption. On the other hand, node parallelism requires a huge amount of data transfers. Thus, node parallelism is mostly suited to the top of the tree, where there is not enough potential for tree parallelism (e.g., the root node is only eligible for node parallelism), while tree parallelism is best suited to the bottom of the tree.

1.2 Exploiting data sparsity

In many applications, the matrices manipulated are structurally sparse, with most of their coefficients being zero. As a result, the complexity of the problem can be greatly reduced. However, this aspect fails to fully take into account how sparse the problem really is, which part is significant, and which operations can be performed approximately for numerical reasons. In fact, one may often benefit from data sparsity, given the fact that the matrices may still be compressed much further for a given accuracy. This is also often the case for the frontal matrices taken from an elimination tree of the multifrontal method, especially when the matrix has been obtained from a finite element method. Their off-diagonal blocks correspond to interactions between two groups of variables that are weakly connected. They contain little information, and we could try to compress them.

1.2.1 Low-rank approximations

Definition

Let A be a matrix of size (m, n). A rank-r approximation of A is a matrix T of rank r situated within a certain distance τ of A. In other words, it means that there exist two matrices $X \in \mathbb{R}^{m\times r}$ and $Y \in \mathbb{R}^{n\times r}$ such that $\|A - XY^T\| \leq \tau$. We say that r is the numerical rank of A. If r is sufficiently small, we say that $T = XY^T$ is a low-rank approximation of A. Storing X and Y requires fewer coefficients than storing A: such a representation may be used as a matrix compression of A.

We might want to use a threshold τ that is relative to the norm of A instead of a constant. There are several possible choices for the norm $\|.\|$. In this thesis, we will normally choose the Frobenius norm: $\forall M, \|M\| = \|M\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} m_{ij}^2}$.

Truncated SVD

One possible choice for computing a low-rank approximation of A is through a truncated SVD. Let $U\Sigma V = \sum_{i=1}^{min(m,n)} \sigma_i U_i V_i^T$ be the singular value decomposition of A (SVD). Given a target accuracy τ, we choose r as the smallest integer such that $T = \sum_{i=1}^{r} \sigma_i U_i V_i^T$ verifies $\|A - T\| \leq \tau$.

We know from Eckard and Young (1936) that using a truncated SVD leads to the optimal low-rank approximation for the Frobenius norm: for a given rank, it leads to the most accurate low-rank approximation. Said otherwise, for a given required accuracy, this technique leads to an approximation of minimal rank. This property of optimality also stands when using the 2-norm instead of the Frobenius norm.

QR decomposition

However, computing a truncated SVD is rather costly in terms of time. In practice one may prefer to perform a non-optimal low-rank approximation, easier to compute. One of those possibilities is to compute a truncated QR factorization with column pivoting. Therefore, we obtain an approximation $AP \approx QR$, where P is a permutation matrix, Q is a matrix of dimensions $m \times r$ with orthonormal columns, and R an upper triangular matrix of size $r \times n$.

1.2.2 BLR matrices

Let $A \in \mathbb{R}^{n \times n}$ be a dense square matrix partitioned into $q \times q$ blocks. A block low-rank (BLR) representation T of A is a block matrix of the form

$$T = \begin{bmatrix} T_{11} & T_{12} & \cdots & T_{1q} \\ T_{21} & \cdots & \cdots & \vdots \\ \vdots & & & \vdots \\ T_{q1} & \cdots & \cdots & T_{qq} \end{bmatrix}, \tag{1.1}$$

where the blocks A_{ij} of size $b \times b$ are approximated by matrices T_{ij} satisfying

$$\|A_{ij} - T_{ij}\| \leq \varepsilon \beta_{ij}, \tag{1.2}$$

where $\beta_{ij} > 0$. Some of the blocks are left uncompressed, i.e., $T_{ij} = A_{ij}$. We will refer to them as being "full-rank" (FR). In particular, this is the case for all the diagonal blocks. The other blocks are approximated as low-rank (LR): each T_{ij} matrix, of rank r_{ij}, is expressed as

$$T_{ij} = \begin{cases} X_{ij} Y_{ij}^T, & i > j, \\ Y_{ij} X_{ij}^T, & i < j, \end{cases} \tag{1.3}$$

where X_{ij} and Y_{ij} are $b \times r_{ij}$ matrices, and where X_{ij} has orthonormal columns.

Even though, in general, each block can be of different dimensions, we assume for simplicity that they are all of the same dimensions $b \times b$, and so $n = qb$.

Representation (1.3) guarantees the so-called outer orthonormality property (Higham and Mary, 2021; Mary, 2017), which is used in MUMPS. This property will mostly be reused in chapter 2. For simplifications, we will tend to ignore it in chapters 3 and 4, and consider that all low-rank blocks are under the form $T_{ij} = X_{ij} Y_{ij}^T$.

Importantly, the β_{ij} parameters in Equation 1.2 are used to distinguish two types of BLR compression, local and global, depending on whether block T_{ij} approximates A_{ij} with error ε relative to the local norm $\beta_{ij} = \|A_{ij}\|$ or relative to the global norm $\beta_{ij} = \|A\|$. In their error analysis of the BLR factorization, Higham and Mary (2021) show that global compression achieves a better tradeoff between compression and accuracy, and is therefore to be preferred. Throughout this thesis, we will thus use a global compression, with $\beta_{ij} = \beta = \|A\|$. On a BLR matrix, it leads to a global error bound that is proportional to ε:

$$\|A - T\| \leq q\varepsilon\|A\|. \tag{1.4}$$

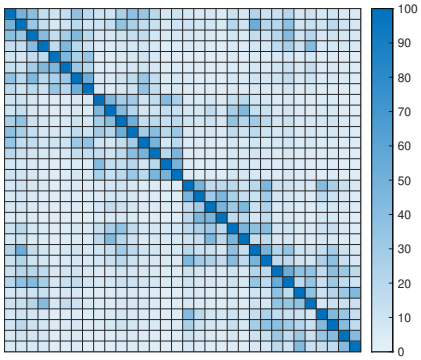


Figure 1.4: Ranks of the blocks T_{ij} in the BLR approximation of matrix P64 (see Table 2.1), for $\varepsilon = 10^{-10}$, expressed as a percentage of the block size. Dark blue blocks are full-rank, while light blue blocks have very low ranks.

1.2.3 Adapting the multifrontal method to BLR compression

We observe that, in the multifrontal method, large frontal matrices tend to have a huge potential for BLR compression. For example, Figure 1.4 was obtained from the root node of the elimination tree of a sparse matrix, and most of the blocks are highly compressible. We will now present how to take advantage of the BLR compression described previously in order to reduce the complexity of the multifrontal method. Such modifications were first described during the PhD thesis of Clément Weisbecker (see Weisbecker, 2013), and further improvements were made in the PhD thesis of Théo Mary (see Mary, 2017). In order to avoid confusion with the BLR variant of the multifrontal method that we present in this section, we will sometimes refer to the classical multifrontal method presented in section 1.1.2 as the full-rank algorithm (FR).

Factorization of a frontal matrix

Contrary to the full-rank case, where we could apply basic linear algebra operations on large parts of the matrix at once, we now have to follow the block structure of the BLR matrix. Therefore, we must adapt Algorithm 1.1 into a tile algorithm, which manipulates blocks of size $b \times b$ as basic elements. The main difference is that the update formula, formerly $a_{ij} \leftarrow a_{ij} - a_{ik}a_{kk}^{-1}a_{kj}$, is now generalized to a block: $A_{ij} \leftarrow A_{ij} - (A_{ik}U_{kk}^{-1})(L_{kk}^{-1}A_{kj})$.

We note that computing a decomposition $A_{kk} = L_{kk}U_{kk}$ and solving the triangular systems is equivalent to a multiplication by A_{kk}^{-1}.

We now introduce the use of low-rank approximations in order to obtain Algorithm 1.4. The main idea is that we store the computed LU factors as a BLR matrix. Moreover, whenever a low-rank block appears in a formula that involves a matrix product, choosing the right associativity leads to a reduced number of operations.

Algorithm 1.4 LU factorization of a BLR frontal matrix (Left-looking)

1: /* **Input:** a $q \times q$ block frontal matrix A; $A = [A_{i,j}]_{i=1:q,j=1:q}$; $q = q_{\text{fs}} + p_{nfs}$ */
2: **for** $k = 1$ **to** q_{fs} **do**
3: **for** $i = k$ **to** q **do**
4: Update (L): $A_{i,k} \leftarrow A_{i,k} - \sum_{l=1}^{k-1} L_{i,l}U_{l,k}$
5: **end for**
6: **for** $j = k + 1$ **to** q **do**
7: Update (U): $A_{k,j} \leftarrow A_{k,j} - \sum_{l=1}^{k-1} L_{l,j}U_{k,j}$
8: **end for**
9: Factor: Compute LU factorization $L_{k,k}U_{k,k} = A_{k,k}$
10: **for** $i = k + 1$ **to** q **do**
11: Compress (L): compute $L_{i,k} \approx A_{i,k}$
12: Compress (U): compute $U_{k,j} \approx A_{k,j}$
13: Solve (L): $L_{i,k} \leftarrow L_{i,k}U_{k,k}^{-1}$
14: Solve (U): $U_{k,i} \leftarrow L_{k,k}^{-1}U_{k,j}$
15: **end for**
16: **end for**
17: **for** $i = q_{\text{fs}} + 1$ **to** q **do**
18: **for** $j = q_{\text{fs}} + 1$ **to** q **do**
19: Update (CB): $A_{i,k} \leftarrow A_{i,k} - \sum_{l=1}^{k-1} L_{l,j}U_{k,j}$
20: **if** CB compression is activated **then**
21: Compress (CB): compute $T_{i,k} \approx A_{i,k}$
22: **end if**
23: **end for**
24: **end for**

When the block (i, j) is in low-rank form, we can reduce the complexity of the Solve step (lines 13 and 14) by performing the following operations:

- If $i > j$: Solve (L) can be rewritten as $Y_{i,j} \leftarrow (U_{j,j}^T)^{-1}Y_{i,j}$
- If $i < j$: Solve (U) can be rewritten as $Y_{i,j} \leftarrow L_{i,i}^{-1}Y_{i,j}$

As a result, whether the block is considered as low-rank or not, the operations performed for the Solve step require solving a triangular system with several right-hand sides (r_{ij} right-hand sides instead of b if low-rank is used, which systematically reduces the complexity).

Similarly, a product of the form $C = L_{ik} \times U_{kj}$ from line 4, 7, or 19 may have its complexity reduced if at least one of the two blocks is in low-rank form. Several cases appear:

- LR×FR product: $C = X_{ik} \times (Y_{ik}^T \times U_{kj})$
- FR×LR product: $C = (L_{ik} \times Y_{kj}) \times X_{kj}^T$
- LR×LR product: $C = (X_{ik} \times (Y_{ik}^T \times Y_{kj})) \times X_{kj}^T$ or $C = X_{ik} \times ((Y_{ik}^T \times Y_{kj}) \times X_{kj}^T)$ depending on which side leads to the lower complexity.

Triangular solution on a frontal matrix

We consider the triangular solution phase presented in section 1.1.2, and we adapt it to the case where a front of the factors is stored as a BLR matrix (see Figure 1.5). Similarly to what have been done for the factorization, it is possible to drastically reduce the complexity because of the BLR compression.

We first rewrite the forward and backward substitutions as tile algorithms, and we obtain Algorithms 1.5 and 1.6. Their main cost is a product between a block of the factor and a block of the RHS (lines 4 and 3 respectively), by calling the function *prod*.

In the context of the use of BLR compression, the expression $prod(\widetilde{M}, N)$ is considered as a LR×FR product if $\widetilde{M}$ is in low-rank form XY^T, and evaluated with the following associativity: $X \times (Y^T \times N)$. On the other hand, if $\widetilde{M}$ is a full-rank matrix, then we simply have to perform the matrix product $\widetilde{M} \times N$ instead.

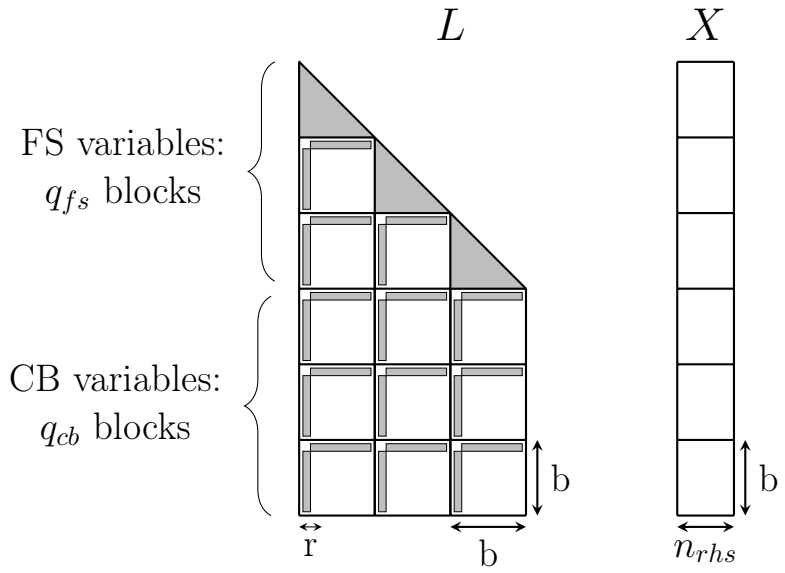


Figure 1.5: A frontal BLR matrix L and its right-hand side X

Algorithm 1.5 Forward elimination (Right-looking algorithm)

1: **for** $j = 1$ **to** q_{fs} **do**
2: $\quad B_j \leftarrow L_{jj}^{-1} B_j$
3: $\quad$ **for** $i = j + 1$ **to** q **do**
4: $\quad\quad B_i \leftarrow B_i - prod(\widetilde{L}_{i,j}, B_j)$
5: $\quad$ **end for**
6: **end for**

Algorithm 1.6 Backward elimination (Left-looking algorithm)

1: **for** $i = q_{\text{fs}}$ **to** 1 **do**
2: $\quad$ **for** $j = i + 1$ **to** q **do**
3: $\quad\quad X_i \leftarrow B_i - prod(\widetilde{U}_{i,j}, B_j)$
4: $\quad$ **end for**
5: $\quad B_i \leftarrow U_{ii}^{-1} B_i$
6: **end for**

Low-Rank updates accumulation

Low-rank updates accumulation (LUA) consists in grouping together low-rank updates on the same block rows and/or block columns and applying them with a single matrix multiplication to increase the granularity of the computation.

The LUA technique was originally proposed by Amestoy et al. (2019a) for the outer product operation in the BLR LU factorization, and is used by default in MUMPS. In the BLR factorization, we compute low-rank updates of the form

$$A_{ij} \leftarrow A_{ij} - (X_{ik}Y_{ik}^T)(Y_{kj}X_{kj}^T).$$

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

$$A_{ij} \leftarrow A_{ij} - \begin{bmatrix} \bar{X}_{i,1,j} & \cdots & \bar{X}_{i,K,j} \end{bmatrix} \times \begin{bmatrix} \bar{Y}_{i,1,j}^T \\ \cdots \\ \bar{Y}_{i,K,j}^T \end{bmatrix}$$

where either $\bar{X}_{ikj} = X_{ik}(Y_{ik}^T Y_{kj})$ and $\bar{Y}_{ikj} = X_{kj}^T$ or $\bar{X}_{ikj} = X_{ik}$ and $\bar{Y}_{ikj} = (Y_{ik}^T Y_{kj})X_{kj}^T$, depending on the ranks. The product of block matrices in the above expression can be efficiently evaluated as a single matrix multiplication $\bar{X}_{ij}\bar{Y}_{ij}^T$.

1.3 Floating-point arithmetic and rounding error analysis

1.3.1 Floating-point arithmetic

Definition

In computer science, it is not possible to represent any real number $f \in \mathbb{R}$. Instead, we have to use an inexact representation, for example floating-point arithmetic. The idea is to approximate f in such a way that the relative error of approximation is bounded by a certain constant u. In order to do so, its approximation $\widehat{f}$ has a mantissa with a fixed number of digits, scaled by an exponent allowing cover of a large range of values. Thus, we define a floating-point number as follows:

$$\begin{aligned} \widehat{f} &= (-1)^s \times b^{e-t+1} \times m \\ &= (-1)^s \times b^e \times \sum_{i=0}^{t-1} m_i b^{-i} \end{aligned} \tag{1.5}$$

- b is the base
- t is the number of significant digits

- s is the bit of sign (0 or 1)
- $e \in [\![e_{min}; e_{max}]\!]$ is the exponent
- $m \in [\![0; b^t[\![$ is the mantissa, whose representation in base b is $m_0...m_{t-1}$

The constants b, t, e_{max} and $e_{min} = 1 - e_{max}$ define the floating-point format. On the other hand, s, e and m are the parameters corresponding to the encoding of $\widehat{f}$ in this format. We will only focus on binary formats in this thesis, so we will take $b = 2$.

With this definition, $\widehat{f}$ can have several encodings in a floating-point format. However, such a property is not always acceptable. Therefore, we define the normalized representation of a floating-point number, in which the constraint $m \geq b^{t-1}$ has to be respected. As a result, we obtain the property of uniqueness of a normalized representation.

The IEEE-754 standard was first published in 1985 and revised in 2008 (see IEEE Computer Society, 2008). It defines floating-point arithmetics that are portable and reproducible, but also the behavior of their basic operations, their rounding rules, as well as their handling of exceptions. In particular, the standard defines a set of binary interchange formats, having normalized representations whose bit encodings are entirely specified.

In binary interchange formats, the integer e is encoded as a positive unsigned integer, offset by a bias in order to reach negative values as well. We notice that, for a binary format, the first bit of the mantissa is $m_0 = 1$ for a normalized representation. Therefore, m_0 does not have to be stored explicitly, it can be an implicit bit instead: the mantissa is encoded by the bit string $m_1 \cdots m_{t-1}$. Also, it is specified that the bit string representing a floating-point number in an interchange format follows the order s, e, m. In particular, the least significant bits of m are stored last. We will make use of this property in chapter 3.

format	bits for encoding the exponent e	significant bits: t	float min (normalized): $2^{e_{min}}$	float max: $(1-2^{-t})2^{e_{max}+1}$	unit roundoff: $u = 2^{-t}$
fp64 (double precision)	11 bits	53 bits	2.2×10^{-308}	1.8×10^{308}	1.1×10^{-16}
fp32 (single precision)	8 bits	24 bits	1.2×10^{-38}	3.4×10^{38}	6.0×10^{-8}
fp16 (half precision)	5 bits	11 bits	6.1×10^{-5}	6.6×10^{4}	4.9×10^{-4}
bfloat16	8 bits	8 bits	1.2×10^{-38}	3.4×10^{38}	3.9×10^{-3}

Table 1.1: Some common floating-point formats. fp64, fp32 and fp16 are defined by the IEEE-754 standard as binary interchange formats.

Range of representation

A floating-point format represents a finite set of values, and therefore has a limited range. The result of an operation may be outside this range. If the exponent is beyond e_{max}, we obtain an overflow: the result of the operation is considered as $+\infty$ or $-\infty$.

This may very well lead to an irrelevant result because it causes a propagation of NaN (Not a Number) values. Thus, overflows should generally be avoided at all costs. This is especially the case when handling precision formats such as fp16, whose range is small. As shown in Table 1.1, converting to fp16 a number higher than 10^5 will result in an overflow.

On the other hand, if the value to be stored in a floating-point format is smaller than the minimum value of the format, that is to say if the exponent should be smaller than e_{min}, we say that an underflow occurs. A common choice for handling this case is to round the result to zero. Alternatively, by not imposing the use of a normalized representation, the result can still be approximated as a so-called subnormal number: the first bits of the mantissa are zero. However, the relative accuracy is inferior to the normalized numbers, given the fact that the number of significant bits is reduced. Besides, the use of subnormal numbers is not often implemented in hardware but in software instead, leading to a significant slowdown.

However, in practice the overflows and underflows tend to be rare enough that they usually do not have too much impact on computations. This is especially true for fp64 which has a very large range: $[10^{-308}; 10^{308}]$. The smaller ranges of fp32 and bfloat16 ($[10^{-38}; 10^{38}]$) are still generally sufficient for avoiding most issues of overflow and underflow, contrary to fp16. In this thesis, we will consider that the presence of overflows and underflows can be neglected, and that the exponent e can take any integer value.

Unit Roundoff

In addition to the range of its representable values, an important parameter describing the properties of a floating-point format is its unit roundoff u. Let us consider that there is no restriction on the exponent: $e \in \mathbb{Z}$. For any real x we note $fl(x)$ its floating-point approximation in the format.

The unit roundoff u of the format is a constant approaching the maximal relative error of approximation of a real number in this format. Such a maximum is obtained when approximating $x = 1 + 2^{-t}$ by $fl(x) = 1$ for instance, obtaining a relative error of $\frac{|x-fl(x)|}{|x|} \approx 2^{-t}$. Therefore, we define $u = 2^{-t}$. For every real x within the range, we find that there is a δ such that

$$fl(x) = x(1 + \delta), \qquad |\delta| < u \tag{1.6}$$

Equation 1.6 is a rather fundamental property of the rounding operator. It is often used instead of the original definition of floating-point rounding (Equation 1.5). However, it does not contain the same level of information.

In order to carry out a rounding error analysis of an algorithm, we need to make some assumptions about the accuracy of its basic arithmetic operations, namely $+$, $-$, $\times$ and $/$. Indeed, the result of such an operation is not always representable in the same format, and the computation cannot be exact in this case. The most common assumption is the

standard model of floating-point arithmetic. Given two floating-point numbers x and y, if we note $x\ op\ y$ the result of one of the basic operations and $fl(x\ op\ y)$ its computed value, this model states that the relative error introduced is at most u:

$$fl(x\ op\ y) = (x\ op\ y)(1+\delta), \qquad |\delta| \leq u, \quad op \in \{+,-,\times,/\} \tag{1.7}$$

This property is a consequence of the IEEE-754 standard in which those operations are entirely specified: it is stated that $fl(x\ op\ y)$ should give the same result as computing $x\ op\ y$ in infinite precision before rounding it to the floating-point format in use. However, this specification gives more information than the standard model. In particular, Equation 1.7 loses track of the fact that some of the operations are performed exactly.

1.3.2 The basics of rounding error analysis

Backward and forward errors

Let us consider a function $f : \mathbb{R}^n \mapsto \mathbb{R}^m$. We want to assess the quality of a computed approximation $\widehat{y}$ of $y = f(x)$. For example, if f consists in a series of exact basic operations, such an approximation may be the result of using floating-point arithmetic. There are several possibilities of error indicators available, depending on the context. A first approach is to compute the absolute error committed on the output variable y, i.e., $||\widehat{y} - y||$. A bound on this term is referred to as the (normwise) forward error.

However, one may prefer to focus on finding for which modified set of data we have actually solved the exact problem, i.e., which perturbation Δx of the input x is needed in order to obtain $\widehat{y} = f(x + \Delta x)$. Among the many Δx that satisfy this relation we will consider the one having the smallest norm. As a result, a formal definition of the (normwise) backward error of $\widehat{y} \approx f(x)$ would be:

$$\eta(y) = min\left\{ \frac{||\Delta x||}{||x||} : \Delta x \in \mathbb{R}^n \text{ such that } \widehat{y} = f(x+\Delta x) \right\} \tag{1.8}$$

The concept of backward error is illustrated in Figure 1.6.

Accumulation of rounding errors

We illustrate the definition of the backward error by trying to quantify the impact of rounding errors on the computation of a certain sum $s_n = \sum_{k=1}^{n} x_k$, in the order of its indices. We note $\widehat{s}_n$ its value computed using floating-point additions (more generally, we will tend to use the notation $\widehat{var}$ to denote the computed value approximating a variable var in this thesis).

We note $\delta_1, \cdots, \delta_{n-1}$ the coefficients obtained for the relative errors of each addition (Equation 1.7). We have the recursive relation $\widehat{s}_n = fl(\widehat{s}_{n-1} + x_n) = (1+\delta_{n-1})(\widehat{s}_{n-1} + x_n)$, and $\widehat{s}_1 = x_1$. As a result, we give the expression of the computed sum: $\widehat{s}_n = (\pi_1 x_n + \cdots \pi_{n-1} x_2) + \pi_{n-1} x_1$, after having defined each quantity $\pi_k = (1+\delta_1)\cdots(1+\delta_k)$.

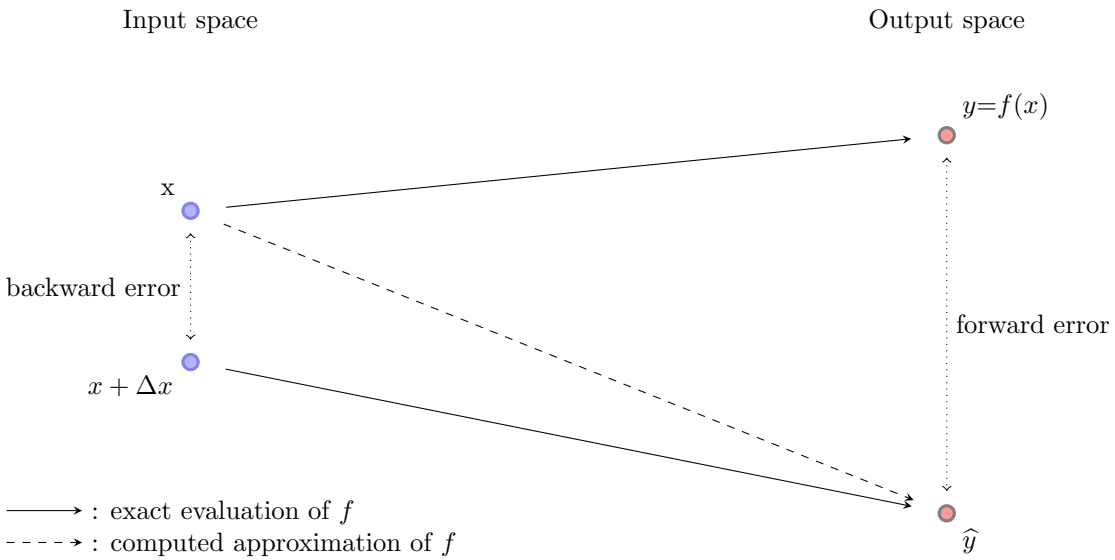


Figure 1.6: Illustration of the forward and backward errors for the computation $\widehat{y} \approx f(x)$.

We choose to express each π_k under the form $\pi_k = 1 + \theta_k$. Then we obtain $|\theta_k| \leq \gamma_k$, with $\gamma_k = \frac{ku}{1-ku}$. The presence of such a weight γ_k keeps track of the fact that x_{n-k+1} appears in k additions.

As a result, we obtain $\widehat{s}_n = \sum_{k=1}^{n}(1+\theta_k)x_k$, with a bound on the relative error committed on the input variable $x = (x_1, \cdots, x_n)$: $\forall k, |\theta_k| \leq \gamma_k \leq \gamma_{n-1}$. Because $||\theta|| \leq \gamma_{n-1}$, the relative backward error of the computation of the sum is γ_{n-1} (at most). This constant is quite common in rounding error analysis.

On the other hand, the error committed on the output variable s_n is the forward error. For the sum, a rough componentwise bound would be $|\widehat{s}_n - s_n| = |\sum_{k=1}^{n}\theta_k x_k| \leq \gamma_{n-1}\sum_{k=1}^{n}|x_k|$. Therefore, the normwise forward error would be $||x||$

Matrix-vector product

The rounding error analysis of a matrix-vector product $y = Ax$ is quite similar to that of the sum that we did previously. As explained in detail in Higham (2002), the approximate result $\widehat{y}$ can be expressed as $\widehat{y} = (A + \Delta A)x$, with $|\Delta A| \leq \gamma_n|A|$ (backward error). As a consequence, the componentwise forward error bound is $|\widehat{y} - y| \leq \gamma_n|A|\ |x|$. We note that, here and throughout the rest of the thesis, we use the notation $A \leq B$ to denote the elementwise inequality $a_{ij} \leq b_{ij}$ on the coefficients of matrices A and B. Moreover, we use the notation $|A|$ to denote the matrix obtained by taking the absolute values of the coefficients of A.

LU factorization

We consider the solution of $Ax = b$ via a Gaussian elimination of A (by combining Algorithms 1.1, 1.2 and 1.3). Although we could use a multi-word arithmetic in order to obtain an exact solution, it is much more common to use floating-point arithmetic instead,

which is much faster. However, one may wonder whether the impact on the accuracy of the solution would be acceptable or not. Therefore, we would like to obtain its backward error bound. We look for a ΔA such that $(A + \Delta A)\widehat{x} = b$. We know from Rigal and Gaches (1967) that such a ΔA of minimal norm verifies:

$$\eta(x) = ||\Delta A|| = \frac{||A\widehat{x} - b||}{||A||\ ||\widehat{x}||} \tag{1.9}$$

This result stands for any subordinate matrix norm. $\eta(x)$ is the relative normwise backward error, defined in Equation 1.8.

We would like to obtain a reasonable bound on ΔA, ideally in $O(u)||A||$. However, the best obtainable componentwise bound obtained is not as optimistic:

$$|\Delta A| \leq \gamma_{3n}|\widehat{L}|\ |\widehat{U}| \tag{1.10}$$

Indeed, the coefficients of $|\widehat{L}||\widehat{U}|$ are generally much larger than the coefficients of A. In order to quantify this increase, we define the growth factor of the factorization:

$$\rho_n = \frac{\max\limits_{i,j,k} |a_{ij}^{(k)}|}{\max\limits_{i,j} |a_{ij}|} \tag{1.11}$$

where $a_{ij}^{(k)}$ is an intermediate result obtained after performing k updates on the original coefficient a_{ij}. James Wilkinson showed in the early 1960s that the numerical stability of Gaussian elimination intrinsically depends on the growth factor. When using partial pivoting, we can find ΔA such that

$$||\Delta A||_\infty \leq n^2 \gamma_{3n} \rho_n ||A||_\infty \tag{1.12}$$

Therefore, one may wonder how large the growth factor ρ_n is. In the worst-case scenario, we may have $\rho_n = 2^{n-1}$ despite the use of partial pivoting: the backward error obtained is rather catastrophic and the algorithm cannot be considered numerically stable in this case. However, it is highlighted in (Higham, 2002, Section 9.4) that fortunately, "despite the existence of matrices for which ρ_n is large with partial pivoting, the growth factor is almost invariably small in practice". Thus, solving linear systems using Gaussian elimination with partial pivoting is stable enough in practice.

More recently, this result of backward stability has been extended to the case of a BLR LU factorization (Algorithm 1.4) in (Higham and Mary, 2021, Theorem 4.3). It is shown that:

$$A = \widehat{L}\widehat{U} + \Delta A, \qquad ||\Delta A|| \leq (q\varepsilon + \gamma_q)||A|| + \gamma_c||\widehat{L}||\ ||\widehat{U}|| + O(u\varepsilon) \tag{1.13}$$

if the matrix A has q^2 blocks of size b and of rank r, and if $c = b + 2r^{3/2} + q$. If $u \ll \varepsilon$ then the main term of error will be $q\varepsilon||A||$.

1.4 Mixed-precision algorithms

When an accurate result is needed in scientific computing, the default choice is often to use double precision everywhere, with 64 bits floating-point numbers (fp64). However, lower precision formats like fp32 also exist. They require less storage space and usually run faster: we can typically hope to gain a 2× speedup using fp32 instead of fp64. However, the computations are not as accurate: the relative error of elementary operations (unit roundoff) is higher, and the range is slightly narrower (see Table 1.1).

Mixed-precision algorithms result from the will to do part of the computations in low precision, obtaining time and memory gains, while controlling the accuracy by performing key operations in high precision. In this section, we give some examples of mixed-precision techniques used in the literature in contexts similar to our own.

The recent availability of half precision formats in hardware raises the question of how to use them in HPC. In fact, they would allow us to reduce all storage by a factor of 4 compared to double precision (and therefore reduce communication cost), and to possibly have an even better gain regarding computation time.

In some deep learning models where great accuracy is not needed, algorithms tend to use Google's bfloat16 format. The range is the same as in fp32, so the algorithms do not necessarily require many modifications. In linear solvers where we need more precision, fp16's greater relative accuracy would be an advantage. However, fp16 has only a limited range available: positive numbers lie between 6×10^{-8} and 7×10^{4}. Standard algorithms can therefore lead to overflow, underflow, or subnormal numbers, all of which are undesirable. This is a serious issue that is almost non-existent for the other main precision formats, and it restricts the use of fp16.

1.4.1 Iterative refinement

One of the best known mixed-precision algorithms is iterative refinement, which was first programmed by Wilkinson in 1948. It aims at improving the accuracy of the solution of a linear system $Ax = b$.

Algorithm 1.7 Iterative refinement

Solve $Ax_0 = b$ in precision u_f
for i=0:∞ **do**
  Compute $r_i = b - Ax_i$ at precision u_r
  Solve $Ad_i = r_i$ at precision u_s
  $x_{i+1} \leftarrow x_i + d_i$ at working precision u
end for

A version of this algorithm consists in computing an LU factorization of the matrix at the beginning and then using it for all the *Solve* steps. Each task can be performed using a different precision format. For example, with $u = u_r =$ fp64 and $u_f = u_s =$ fp32, the most expensive part of the computation is done entirely in single precision. The computation

can therefore be expected to run twice as fast as the double-precision algorithm. This usage was proposed and analyzed by Langou et al. (2006). However, the convergence of this algorithm is not guaranteed if the matrix A is ill conditioned, with a condition number of u_f^{-1} or more.

This is why Carson and Higham introduced another version of mixed-precision iterative refinement that does not have this limitation (Carson and Higham, 2017). In their algorithm, GMRES-IR (GMRES-based Iterative Refinement), each *Solve* step is computed using GMRES (Generalized Minimal Residual), an iterative method. A low-precision LU factorization is used as a preconditioner. The reason is that $U^{-1}L^{-1}A$ is far better conditioned than A.

In Carson and Higham (2018), the same authors then implemented a version of GMRES-IR with 3 precision formats instead of 2. The *Solve* step is now handled by GMRES at precision u, except matrix-vector products with A that are computed at precision u_r. Their choice of (u_f, u, u_r) = (fp16, fp32, fp64) or (fp32, fp64, fp128) allowed them to obtain a much better convergence condition and a smaller forward error than classic iterative refinement using the same working precisions.

1.4.2 Mixed precision on GPUs

GPUs are a type of hardware that is well adapted to matrix computations and to the use of low precisions, which makes them a target of particular interest for mixed-precision algorithms in linear algebra. They handle computations in fp32 at least twice as fast as in fp64. Recent models tend to have support for fp16 and/or bfloat16 arithmetics, with great speedups. This the case for NVIDIA's model A100, available since September 2020, in which fp16 computation offers a 4x speedup compared to fp32 (peak performance).

Moreover, one of the preferred operations on GPUs is block FMA (Block Fuse Multiple-Add), computed in special units referred to as "Tensor Cores" by some vendors. This kind of unit can typically compute operation $D = C + AB$ in one clock cycle, where all matrices have a size of 4×4 for instance.

The tensor cores in the NVIDIA Volta, Turing, Ampere and Hopper architectures can perform these FMAs using different combinations of precision formats, including an fp16 input and an fp32 output. The accumulation inside the matrix multiplication is done in fp32. Therefore, mixed precision is implemented at a hardware level, and algorithms may be able to take this fact into account.

$$\underbrace{D}_{fp32} = \underbrace{C}_{fp32} \underbrace{+}_{fp32} \underbrace{A}_{fp16} \underbrace{\times}_{fp32} \underbrace{B}_{fp16}$$

Haidar et al. (2018) introduced a new class of mixed-precision dense matrix factorization algorithms based on this basic operation. An error analysis of the mixed block FMA as well as the LU factorization based on it was done in Blanchard et al. (2020).

1.4.3 A scaling algorithm for handling low-precision formats

Using the fp16 format can lead to a huge number of overflows and underflows due to its narrow range. In order to deal with this issue, Higham and Pranesh published in 2019 an algorithm for rounding correctly a matrix to fp16 format (see Higham et al., 2019). First a two-sided diagonal scaling is applied in order to balance the matrix's coefficients. We obtain a factorization $D_l A D_r$, with D_l and D_r diagonal matrices, and every row and column of A has an ∞-norm of 1. Then we multiply A by a scalar to bring the elements of largest magnitude within a factor $\theta \leq 1$ of the overflow level. We finally round to half precision. The multiplication by a scalar ensures that a large part of the limited range of half precision arithmetic is used, and therefore the number of underflows is reduced.

The authors presented an application to GMRES-IR iterative solver. After rounding in fp16 with the previous algorithm (for negligible cost of $O(n^2)$ operations), they compute an LU factorization of the scaled matrix. The parameter θ is chosen carefully to ensure the absence of overflow. With this factorization as a preconditioner, GMRES-IR has a much better convergence rate than using a standard method for rounding to fp16.

1.4.4 A mixed-precision Cholesky factorization

A first method for using mixed precision inside a dense tile Cholesky factorization was proposed in Abdulah et al. (2019). The authors chose to address specifically the factorization of a dense covariance matrix in the field of geostatistics, rather than a more general approach. No compression technique is used in this algorithm.

Their mixed-precision approach consists in storing some blocks of the matrix in double precision, and the others in single precision. A simple criterion was used to choose between the two possibilities: every block within a certain distance from the diagonal is kept in double precision, and the others are stored instead in single precision (Figure 1.7).

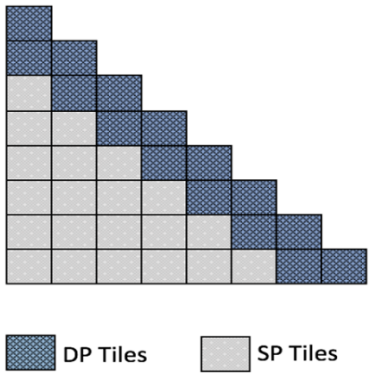


Figure 1.7: Block structure used in Abdulah et al. (2019) for mixed-precision Cholesky

The precision of the computation is based on the precision of the target block being updated. One consequence of this choice is that each computed block is needed at some point both in single precision *and* in double precision for the updates. As a result, without

a careful implementation, this mixed precision algorithm could introduce an extra memory cost (up to 50%).

On the other hand, they succeeded in obtaining a 1.6× performance speedup on massively parallel architectures while maintaining the accuracy necessary for their modeling and prediction.

1.4.5 A mixed-precision representation of $\mathcal{H}$-matrices

The article Ooi et al. (2020a) introduces a mixed-precision algorithm that uses $\mathcal{H}$-matrices, a low-rank matrix format that is based on hierarchical partitioning of the matrix. No error analysis was considered.

Each low-rank block is separated between a part in double precision and another in single (Figure 1.8).

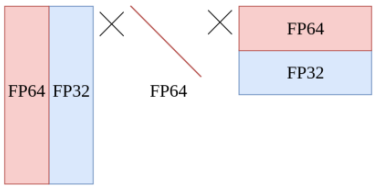


Figure 1.8: A mixed-precision representation of a low-rank matrix, used by Ooi et al.

A diagonal scaling is used. If we have a first low-rank approximation:

$$A = (v_1, ..., v_p) \times (w_1, ..., w_p)^T$$

Then we obtain our scaling from diagonal scalings of the 2 factors:

$$D_v = \text{diag}(\|v_1\|_\infty, ..., \|v_p\|_\infty)$$
$$D_w = \text{diag}(\|w_1\|_\infty, ..., \|w_p\|_\infty)$$

We then scale using diagonal matrix $D = D_v \times D_w$.

This type of scaling can guarantee the absence of any overflow and at the same time avoid most underflows. While its usefulness may be limited if we only work with single or double precision, if we consider the use of fp16 format as well, such a scaling method might be helpful due to the huge risk of overflow and underflow associated with the very narrow range of this format.

1.4.6 Using low precision for storing or accessing data

Another strategy for using mixed precision consists in using low precision formats for storage and/or communications, while keeping the computations in high precision formats.

This strategy not only reduces the memory footprint and communication volume of the algorithm, but also reduce its runtime, thanks to the reduced data movements.

In fact, there are several examples in the literature where data movement reductions have successfully been turned into time gains. In Mukunoki and Imamura (2016), a set of 7 floating-point formats was used for data storage (the same set as our own, see Table 3.1). On several GPU architectures, they called memory-bound functions from cuBLAS[1], such as GEMV and AXPY, using a custom format for memory accesses and a certain working precision (double or single) for computations. They succeeded in obtaining very decent speedups, up to a factor 2.

Grützmacher et al. (2023) developed a similar idea of decoupling the formats for computation and storage, based on Ginkgo's memory accessors[2] this time. By doing this, double precision should only be handled at the register level. They applied their method to memory-bound functions in double precision, while accessing single-precision variables. On GPUs as well as on CPUs, they attained a performance comparable to the single-precision functions.

[1]The cuBLAS library is an implementation of BLAS on top of the NVIDIA CUDA runtime. It allows the user to access the computational resources of NVIDIA GPU.

[2]Ginkgo is a high-performance linear algebra library for manycore systems (including GPUs), with a focus on the solution of sparse linear systems. It is implemented using modern C++.

Chapter 2

Dense LU factorization in mixed precision

2.1 Introduction

In this chapter, we investigate the potential of combining mixed-precision arithmetic with low-rank approximations. Let $A \in \mathbb{R}^{m\times n}$ and let $X\Sigma Y^T = \sum_{i=1}^{\min(m,n)} x_i\sigma_i y_i^T$ be its singular value decomposition (SVD). As mentioned in section 1.2.1, given a target accuracy ε, a low-rank approximation T of A satisfying $\|T - A\| \leq \varepsilon\|A\|$ can be built from the truncated SVD $T = \sum_{i=1}^{r} x_i\sigma_i y_i^T$, where r is the rank of T.

Our starting idea for this work is to ask what precision should be used to store T and to operate on it. In the literature, the truncated SVD (or any other form of low-rank decomposition such as truncated QR with column pivoting), tends to be stored in the lowest possible precision with unit roundoff safely smaller than ε. For example, if $\varepsilon = 10^{-12}$ and we have access to the floating-point arithmetics defined by the IEEE standard, existing algorithms would use double precision (for which the unit roundoff is $u_d \approx 1 \times 10^{-16}$), because the next lower precision, single precision, has a unit roundoff $u_s \approx 6 \times 10^{-8}$ that is much larger than the prescribed ε.

However, we explain why and how we can actually exploit much lower precisions, with almost no loss of accuracy. We show that singular vectors associated with sufficiently small singular values can be stored at precisions with unit roundoff larger than $\varepsilon\|A\|$ while maintaining an overall approximation accuracy of order $\varepsilon\|A\|$. For example, with $\varepsilon = 10^{-12}$, any singular vector x_i whose associated singular value σ_i is smaller than $\varepsilon\|A\|/u_s \approx 2 \times 10^{-5}\|A\|$ can be stored in single precision. Indeed, the single precision vector $\widehat{x}_i$ satisfies $\|\widehat{x}_i - x_i\| \leq u_s$, but the overall error introduced by replacing x_i by $\widehat{x}_i$ is bounded by $\|(\widehat{x}_i - x_i)\sigma_i y_i^T\| \leq (\varepsilon\|A\|/u_s)u_s = \varepsilon\|A\|$. As can be seen from this example, the reason we can afford to convert some singular vectors to lower precision is because the error introduced by this conversion is demagnified by the singular value; hence the error may be safely bounded if σ_i is small enough.

In the following, we formalize this intuition with an error analysis considering an arbi-

trary number of precisions. Moreover, our analysis applies to any low-rank decomposition of the form $T = XY^T$ where X has orthonormal columns. Indeed, the mixed-precision approach presented here is general and can be used for several other low-rank approximations, in particular rank-revealing QR decompositions.

Clearly, the potential of the proposed approach depends on whether the singular values of the matrices to be approximated decay rapidly. In the second part of this chapter, we apply this approach to an important class of matrices: data sparse, rank-structured matrices, whose off-diagonal blocks have low numerical rank (Bebendorf, 2008). We focus in particular on the block low-rank (BLR) format (Amestoy et al., 2015, 2017), although the approach is also applicable to hierarchical formats. Our numerical experiments demonstrate that the proposed mixed precision low-rank representation presents a very high potential in this context: a large fraction of both the entries needed to represent BLR matrices and the floating-point operations (flops) needed to compute their LU factorization can be switched to lower precisions.

The rest of this chapter is organized as follows. In section 2.2, we describe the proposed mixed precision low-rank representation and show that the loss of accuracy introduced by the use of lower precisions can be rigorously bounded. We then apply this representation to BLR matrices in section 2.3. In section 2.4, we analyze how to compute the LU factorization of a BLR matrix using mixed precision arithmetic. We present numerical experiments on a range of real-life matrices in section 2.5, before concluding in section 2.6.

Throughout the chapter, we define $\gamma_k^{(\ell)} = ku_\ell/(1-ku_\ell)$ for any unit roundoff u_ℓ and for any k such that $ku_\ell < 1$. Given two matrices A and B, we use the notation $|A| \leq |B|$ to denote the elementwise inequality $|a_{ij}| \leq |b_{ij}|$.The unsubscripted norm $\|\cdot\|$ denotes the Frobenius norm:

$$\|A\| = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}^2 \right)^{1/2} = \left(\sum_{i=1}^{\min(m,n)} \sigma_i^2 \right)^{1/2}. \tag{2.1}$$

In our experiments, we will work with IEEE double and single precision arithmetics (denoted as fp64 and fp32, respectively) and bfloat16 arithmetic. The unit roundoffs for these three arithmetics are $u_d = 2^{-53} \approx 1 \times 10^{-16}$, $u_s = 2^{-24} \approx 6 \times 10^{-8}$, and $u_h = 2^{-8} \approx 4 \times 10^{-3}$, respectively.

2.2 Low-rank approximations in mixed precision

Let $A \in \mathbb{R}^{m\times n}$ and let T be a low-rank approximation of A satisfying

$$\|A - T\| \leq \varepsilon\beta, \tag{2.2}$$

where $\varepsilon > 0$ is the target accuracy and where β is a scaling parameter chosen by the user: a natural choice is $\beta = \|A\|$, which leads to an accuracy of ε relative to $\|A\|$, but other choices are possible, as mentioned in section 1.2.2.

Hereinafter, we refer to the precision that T is stored in as the working precision, and

we assume that its unit roundoff u_1 is safely smaller than ε, that is, $u_1 \ll \varepsilon$.

Given the SVD $A = \sum_{i=1}^{n} x_i \sigma_i y_i^T$, it is well known that the approximation of A of lowest rank is given by the truncated SVD

$$T = X\Sigma Y^T = \sum_{i=1}^{r} x_i \sigma_i y_i^T, \quad X \in \mathbb{R}^{m\times r},\ Y \in \mathbb{R}^{n\times r}, \tag{2.3}$$

where the rank r is the smallest integer such that (2.2) is satisfied. Neglecting any noise associated with the working precision, r is given by

$$r = \min\left\{k\colon \|A - \sum_{i=1}^{k} x_i\sigma_i y_i\| \le \varepsilon\beta\right\} = \min\left\{k\colon \left(\sum_{i=k+1}^{\min(m,n)} \sigma_i^2\right)^{1/2} \le \varepsilon\beta\right\}. \tag{2.4}$$

The goal of this section is to prove that depending on the singular values of T, we can use lower precisions than the working precision (with unit roundoff larger than ε), and still preserve an overall approximation error of order ε. We first carry out our analysis for the SVD, but also provide at the end of this section its extension to other types of low-rank approximation methods, such as rank-revealing QR.

Our analysis assumes that the use of lower precision arithmetic with fewer exponent bits than the working precision does not lead to any overflow or underflow. To ensure that this assumption is satisfied in our experiments, we focus on the use of bfloat16 arithmetic (which has the same range as fp32), rather than fp16 (which has a much narrower range).

Let us assume that p different floating-point arithmetics are available (including the working precision u_1), and that their unit roundoffs satisfy

$$u_1 \ll \varepsilon < u_2 < \ldots < u_p. \tag{2.5}$$

Let us consider a partition of matrix T into p groups

$$T = X\Sigma Y^T = \begin{bmatrix} X_1 \ldots X_p \end{bmatrix} \begin{bmatrix} \Sigma_1 & & \\ & \ddots & \\ & & \Sigma_p \end{bmatrix} \begin{bmatrix} Y_1 \ldots Y_p \end{bmatrix}^T, \tag{2.6}$$

where

$$T_k = X_k \Sigma_k Y_k^T, \quad X_k \in \mathbb{R}^{m\times r_k},\ \Sigma_k \in \mathbb{R}^{r_k\times r_k},\ Y_k \in \mathbb{R}^{n\times r_k} \tag{2.7}$$

is a matrix of rank r_k formed by the subset of the singular values and vectors of T assigned to group k.

We now analyze the effect of converting T_k to precision u_k. We assume that only the singular vectors X_k and Y_k are converted, whereas the singular values Σ_k are kept in precision u_1. This is because the storage for Σ_k is negligible compared with that of X_k and Y_k. We however note that adapting the analysis to the case where Σ_k is also converted to precision u_k is straightforward and only slightly increases the constants in

the error bounds. We write $\widehat{X}_k$ and $\widehat{Y}_k$ for the converted vectors, and $\widehat{T}_k = \widehat{X}_k \Sigma_k \widehat{Y}_k^T$ (note that since T_1 is already in precision u_1, $\widehat{T}_1 = T_1$). The following lemma bounds $\|\widehat{T}_k - T_k\|$ for $k \geq 2$. Since X_k and Y_k are stored in precision u_1, they only have approximately orthonormal columns. In the following lemma and throughout this chapter, we neglect this loss of orthonormality, which would only introduce a lower order term $O(\varepsilon u_1)$ in the error bounds.

Lemma 2.1. *Let $T_k = X_k \Sigma_k Y_k^T$ where X_k and Y_k have approximately orthonormal columns (stored in precision u_1), and let $\widehat{T}_k = \widehat{X}_k \Sigma_k \widehat{Y}_k^T$ be obtained by converting X_k and Y_k to precision u_k. Then*

$$\|\widehat{T}_k - T_k\| \leq (2 + \sqrt{r_k} u_k) u_k \|\Sigma_k\|. \tag{2.8}$$

Proof. The converted $\widehat{X}_k$ and $\widehat{Y}_k$ satisfy, for $k \geq 2$,

$$\widehat{X}_k = X_k + E_k, \quad |E_k| \leq u_k |X_k|, \tag{2.9}$$
$$\widehat{Y}_k = Y_k + F_k, \quad |F_k| \leq u_k |Y_k|. \tag{2.10}$$

Therefore, we have

$$\|T_k - \widehat{T}_k\| \leq \|E_k \Sigma_k Y_k^T\| + \|X_k \Sigma_k F_k^T\| + \|E_k \Sigma_k F_k^T\|. \tag{2.11}$$

For the first term, we observe that

$$\|E_k \Sigma_k Y_k^T\|^2 = \|E_k \Sigma_k\|^2 \tag{2.12}$$
$$= \sum_j \sigma_j^2 \sum_i e_{ij}^2 \tag{2.13}$$
$$\leq \sum_j \sigma_j^2 \sum_i u_k^2 x_{ij}^2 \tag{2.14}$$
$$= u_k^2 \sum_j \sigma_j^2 = u_k^2 \|\Sigma_k\|^2, \tag{2.15}$$

where we have used the fact that the columns of X_k have a norm of 1. Therefore

$$\|E_k \Sigma_k Y_k^T\| \leq u_k \|\Sigma_k\|. \tag{2.16}$$

Similarly, we also have

$$\|X_k \Sigma_k F_k^T\| \leq u_k \|\Sigma_k\|. \tag{2.17}$$

Finally, for the third term, we have $\|E_k \Sigma_k F_k^T\| \leq \|E_k \Sigma_k\| \|F_k\|$ and so

$$\|E_k \Sigma_k F_k^T\| \leq \sqrt{r_k} u_k^2 \|\Sigma_k\|. \tag{2.18}$$

Reinjecting (2.16), (2.17), and (2.18) into (2.11) yields the result. □

Lemma 2.1 shows that converting T_k to precision u_k introduces an error of order $u_k\|\Sigma_k\|$, which is thus proportional to the size of the singular values in Σ_k. This fundamental observation is at the foundation of the mixed precision representation that we propose. Indeed, Lemma 2.1 suggests that we can preserve an overall accuracy of order $\varepsilon\beta$ by partitioning the singular values in such a way that $\|\Sigma_k\| \approx \varepsilon\beta/u_k$.

In order to build such a partitioning where the size of the groups stored in lower precision is as large as possible, it is easy to see that we should start by including the smallest singular values in the last group first, until its norm exceeds $\varepsilon\beta/u_p$; at this point, we can start building group $p-1$ with the remaining singular values, and so on. Therefore, the Σ_k are formed from consecutive singular values:

$$\Sigma_k = \operatorname{diag}(\sigma_i), \quad i = i_k \colon i_{k+1} - 1, \tag{2.19}$$

where the indices i_k and i_{k+1} define which singular values are part of Σ_k, and can be computed by the recursive formula:

$$i_k = \min\left\{ i \colon \left(\sum_{j=i}^{i_{k+1}-1} \sigma_j^2 \right)^{1/2} \leq \varepsilon\beta/u_k \right\}, \quad k \in [2:p], \tag{2.20}$$

starting with $i_{p+1} = r+1$ and ending with $i_1 = 1$. We thus end up with a partitioning of the SVD as defined by (2.19)–(2.20). Note that this definition may lead to some empty Σ_k, in which case T_k is a rank-0 matrix. We note that this partitioning is similar to the Method 3 proposed by Ooi et al. (2020b). Our analysis justifies the use of this partitioning and gives a precise rule to define the p groups depending on the singular values and on the precisions.

This partitioning guarantees that $\|\Sigma_k\| \leq \varepsilon\beta/u_k$ for all $k \geq 2$ and so, by Lemma 2.1, converting T_k to precision u_k introduces an error bounded by

$$\|T_k - \widehat{T}_k\| \leq (2 + \sqrt{r_k}u_k)\varepsilon\beta. \tag{2.21}$$

By combining (2.21) over $k = 2\colon p$, we readily obtain a bound on the overall error introduced by converting each T_k to precision u_k.

Theorem 2.1. *Let T be a low-rank approximation of A satisfying $\|A - T\| \leq \varepsilon\beta$. If T is partitioned into p groups $T_k = X_k\Sigma_k Y_k^T$ as defined by (2.19)–(2.20), and the X_k and Y_k are converted to precision u_k, the resulting matrix $\widehat{T} = \sum_{k=1}^p \widehat{T}_k = \sum_{k=1}^p \widehat{X}_k\Sigma_k\widehat{Y}_k$ satisfies*

$$\|A - \widehat{T}\| \leq \left(2p - 1 + \sum_{k=2}^{p} \sqrt{r_k}u_k\right)\varepsilon\beta. \tag{2.22}$$

Proof. The triangle inequality $\|A - \widehat{T}\| \leq \|A - T\| + \sum_{k=2}^p \|T_k - \widehat{T}_k\|$ together with (2.21) readily yields the result. □

Theorem 2.1 shows that the mixed precision low-rank matrix $\widehat{T}$ approximates A with

an accuracy of order ε. To first order, the constant in this error bound is $2p-1$, instead of 1 for the uniform precision matrix T: the introduction of lower precisions therefore only increases the overall error by a very modest quantity. Moreover, we note that by means of a more sophisticated proof that avoids the use of the triangle inequality, this constant can be reduced to $1+2\sqrt{p-1}$:

$$\|A-\widehat{T}\| \leq \left(1+2\sqrt{p-1}+O(u_2)\right)\varepsilon\beta. \tag{2.23}$$

However, we will not use such proofs for the sake of readability, and because the precise value of the constants in the error bounds is unimportant, as long as they are not too large.

Note that while Theorem 2.1 guarantees an approximation error $\|A-\widehat{T}\|$ in $O(\varepsilon)$, the errors on the low-rank factors $\|X-\widehat{X}\|$ and $\|Y-\widehat{Y}\|$ are in $O(u_p)$, the lowest precision used. Moreover, the loss of orthogonality of X and Y is also in $O(u_p)$. Therefore, one should be careful before using the proposed method in applications that make use of the singular vectors or their orthogonality.

Importantly, the proposed method can be applied to other types of low-rank decompositions, not necessarily based on SVD. For example, Theorem 2.1 can be easily extended to decompositions of the form XBY^T, where X and Y have orthonormal columns (the difference with the SVD being that B is not diagonal), bound (2.22) holds with a slightly larger constant; one example of this form is the UTV decomposition (Fierro and Hansen, 1997). Our analysis can also be adapted to decompositions of the form XY^T, where X has orthonormal columns (but Y does not). This second form is particularly of interest because it applies to rank-revealing QR decompositions. We state the analogue to Lemma 2.1 for XY^T decompositions below.

Lemma 2.2. *Let $T_k = X_k Y_k^T$ where X_k has approximately orthonormal columns (stored in precision u_1), and let $\widehat{T}_k = \widehat{X}_k \widehat{Y}_k^T$ be obtained by converting X_k and Y_k to precision u_k. Then*

$$\|T_k-\widehat{T}_k\| \leq (2+\sqrt{r_k}u_k)u_k\|Y_k\|. \tag{2.24}$$

Thus, the error introduced by converting group k now depends on $\|Y_k\|$, and this means that the $X\Sigma Y^T$ partitioning (2.19)–(2.20) should be adapted by replacing $\|\Sigma_k\|$ by $\|Y_k\|$. Then, it is easy to show that Theorem 2.1 still holds. Lastly, the proposed method can even be applied to decompositions XY^T where neither X nor Y have orthonormal columns. In this case the error introduced by the conversion of T_k to precision u_k is

$$\|T_k-\widehat{T}_k\| \leq (2+u_k)u_k\|X_k\|\|Y_k\|, \tag{2.25}$$

which shows that there is potential for mixed precision as long as the low-rank components $x_i y_i^T$ are of decreasing norm.

In the rest of this chapter, we will focus on low-rank decompositions of the form XY^T, computed by means of a truncated QR factorization with column pivoting.

An important question is under what condition the low-rank compression is beneficial, that is, when does the low-rank approximation T require less storage than the original matrix $A \in \mathbb{R}^{m\times n}$. In the standard uniform precision case, $T = XY^T$ can be represented with $r(m+n)$ entries, and so the condition is

$$r(m+n) \leq mn. \tag{2.26}$$

With a mixed precision representation, this condition changes due to the fact that entries belonging to groups $k \geq 2$ are stored in lower precision. The condition becomes

$$(m+n)\sum_{k=1}^{p} c_k r_k \leq mn, \tag{2.27}$$

where c_k quantifies the cost of storing a floating-point number in precision u_k instead of u_1. For example, if we use three precisions, fp64, fp32, and bfloat16, (2.27) takes the form $(m+n)(r_1+0.5r_2+0.25r_3) \leq mn$. Interestingly, the difference between conditions (2.26) and (2.27) means that a matrix that is not "low-rank enough" in uniform precision can become so when using mixed precision arithmetic.

Crucially, the size r_k of each group depends on the singular values. Indeed, group k must satisfy $\|\Sigma_k\| \leq \varepsilon\beta/u_k$, so if the singular values of A decay slowly, most of them must be kept in the first group and little gain can be expected from the use of mixed precision. Conversely, if A possesses many small singular values, the low precision groups will be very large and the use of mixed precision will be very beneficial. Therefore, the potential gains achieved by the proposed approach completely depend on the singular values of the matrix. From now on, we will focus on an important class of matrices that exhibit off-diagonal blocks with rapidly decaying singular values, and therefore present a high potential for the use of mixed precision.

2.3 Mixed precision BLR compression

Data sparse matrices are rank-structured matrices most of whose off-diagonal blocks have low numerical rank. In this section, we show how this property can be exploited to represent these matrices in mixed precision. We focus on a specific class of data sparse matrices, called the block low-rank (BLR) format (Amestoy et al., 2015, 2017, 2019a). The approach described here could also be extended to other formats, such as hierarchical (Hackbusch, 2015) or multilevel (Amestoy et al., 2019b) ones.

2.3.1 Background on BLR matrices

As explained in section 1.2.2, a BLR representation T of a dense square matrix $A \in \mathbb{R}^{n\times n}$ has the block $q \times q$ form

$$T = \begin{bmatrix} T_{11} & T_{12} & \cdots & T_{1q} \\ T_{21} & \cdots & \cdots & \vdots \\ \vdots & \cdots & \cdots & \vdots \\ T_{q1} & \cdots & \cdots & T_{qq} \end{bmatrix}, \tag{2.28}$$

where some of the off-diagonal blocks A_{ij} of size $b\times b$ have been approximated by matrices T_{ij} of ranks r_{ij}:

$$T_{ij} = \begin{cases} X_{ij}Y_{ij}^T & \text{if } i > j \\ Y_{ij}X_{ij}^T & \text{if } i < j \end{cases} \tag{2.29}$$

where X_{ij} and Y_{ij} are $b \times r_{ij}$ matrices, and where X_{ij} has orthonormal columns. The other blocks are left uncompressed, with $T_{ij} = A_{ij}$. In both cases, we ensure that

$$\|A_{ij} - T_{ij}\| \leq \varepsilon\beta_{ij}, \tag{2.30}$$

with $\beta_{ij} = \|A\|$ if we use a global threshold.

On a BLR matrix, it leads to the global error bound

$$\|A - T\| \leq q\varepsilon\|A\|. \tag{2.31}$$

2.3.2 Error analysis of mixed precision BLR compression

We now seek to combine BLR compression with the mixed precision representation proposed in section 2.2. The natural approach is to simply use this mixed precision representation on every low-rank off-diagonal block of the BLR matrix, leaving the full-rank blocks in the working precision u_1. Then, it is easy to show that (2.22) becomes

$$\|A_{ij} - \widehat{T}_{ij}\| \leq \big(2p - 1 + \sum_{k=2}^{p} \sqrt{r_{ij}^{(k)}}u_k\big)\varepsilon\beta_{ij}, \tag{2.32}$$

where $r_{ij}^{(k)}$ is the rank of the matrix $\widehat{T}_{ij}^{(k)} = \widehat{X}_{ij}^{(k)}(\widehat{Y}_{ij}^{(k)})^T$, that is, the number of columns of X_{ij} and Y_{ij} stored in precision u_k. The next result bounds the error introduced by mixed precision BLR compression.

Theorem 2.2 (mixed precision BLR compression)**.** *Let T be a BLR approximation of A defined by (2.28)–(2.30) with $\beta_{ij} = \|A\|$ (global compression). If the off-diagonal blocks T_{ij} are represented with the mixed precision representation $\widehat{T}_{ij}$ described in section 2.2,*

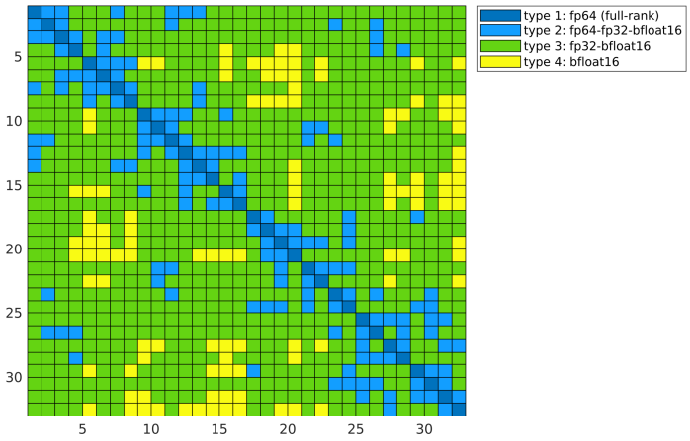


Figure 2.1: Precision formats used for each block of a mixed-precision BLR matrix (matrix P64, $\varepsilon = 10^{-10}$).

the resulting BLR matrix $\widehat{T}$ *satisfies*

$$\|A - \widehat{T}\| \leq q\left(2p - 1 + \sum_{k=2}^{p} c_k u_k\right)\varepsilon\|A\|, \tag{2.33}$$

with $c_k = \max_{i,j} \sqrt{r_{ij}^{(k)}}$.

Proof. Using

$$\|A - \widehat{T}\|^2 = \sum_{i=1}^{q}\sum_{j=1}^{q} \|A_{ij} - \widehat{T}_{ij}\|^2 \tag{2.34}$$

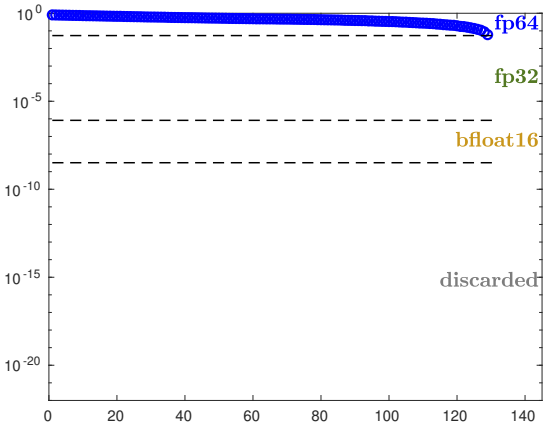
and (2.32), we readily obtain the result. □

Compared with the uniform precision bound (2.31), the mixed precision bound (2.33) is thus larger by a modest factor of about $2p-1$. Theorem 2.2 therefore shows that we can exploit mixed precision arithmetic in the BLR compression while preserving an accuracy of order ε.

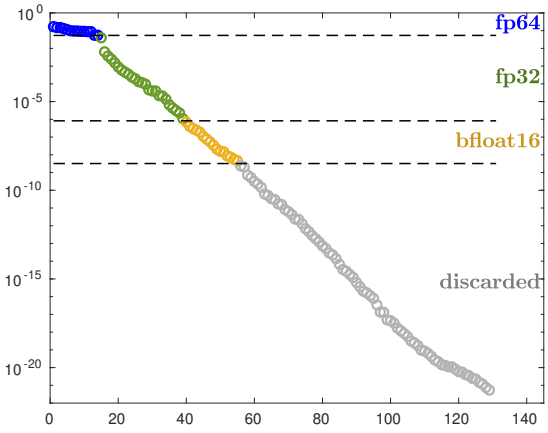
2.3.3 Types of mixed precision blocks

Figure 2.1 shows an example of a mixed precision BLR matrix, plotting for each of its blocks the precisions that are effectively used to represent it. With $\varepsilon = 10^{-10}$ and with three available precisions (fp64, fp32, and bfloat16), we can distinguish four types of blocks. The singular values of a representative example of each type are plotted in Figure 2.2.

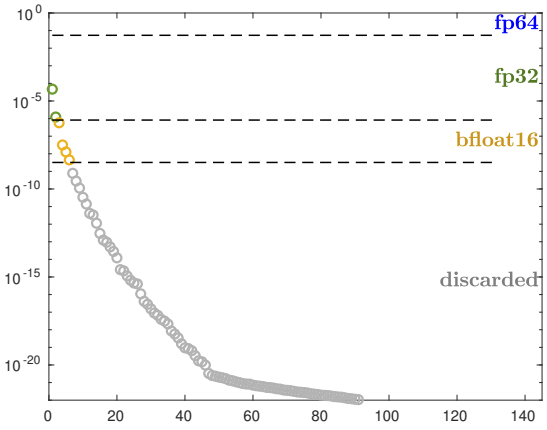
First, the full-rank blocks (type 1, dark blue blocks in Figure 2.1, consisting of only the diagonal blocks here) are stored in the working precision (fp64). An example of diagonal block is given in Figure 2.2a, showing that its singular values decay too slowly to benefit from the use of a mixed precision representation. However, there is only a small number of such blocks: the majority of the blocks therefore benefits from the use of lower precisions.



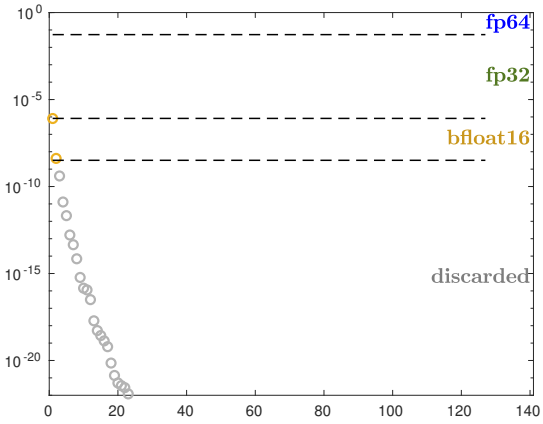
(a) Diagonal block in position (15,15).



(b) Near field block in position (15,16).



(c) Mid field block in position (15,22).



(d) Far field block in position (15,27).

Figure 2.2: Distribution of the singular values of four blocks of different type in Figure 2.1. The dashed lines indicate the thresholds $\varepsilon\beta/u_s$, $\varepsilon\beta/u_h$, and $\varepsilon\beta$, with $\beta = \|A\|$ and where u_s and u_h denote the unit roundoffs of the fp32 and bfloat16 arithmetics, respectively.

Interestingly, the number of low-rank blocks that effectively use all three precisions is quite small (type 2, light blue blocks, example given by Figure 2.2b). Most blocks actually do not need to store any of their entries in fp64. This is a consequence of using global compression: if $\beta_{ij} = \|A\| \gg \|A_{ij}\|$, fp64 is not needed to represent A_{ij}. In other words, blocks of sufficiently small norm can be stored entirely in lower precision. Among these blocks, we can further distinguish two types: those that are represented in mixed precision using both fp32 and bfloat16 (type 3, green blocks, example given by Figure 2.2c) and those that are stored entirely in bfloat16 (type 4, yellow blocks, example given by Figure 2.2d). The type-4 blocks are those whose norm is smaller than $\varepsilon\|A\|/u_h$, where $u_h = 2^{-8}$ is the unit roundoff of bfloat16. Note that a fifth type of block could arise, those whose norm is smaller than $\varepsilon\|A\|$: these blocks can simply be dropped, that is, replaced by zero (but no such blocks appear in the example of Figure 2.1).

The observation that blocks of small norm can be stored entirely in lower precision is important. It justifies why the simpler approach proposed by Abdulah et al. (2019) and Doucet et al. (2019) can already achieve significant gains. Their approach consists in storing each block in uniform precision, but possibly differing from one block to another. For example, for the matrix in Figure 2.1, all type-2 blocks (light blue) would need to be stored entirely in double precision, but type-3 blocks (green) could be stored entirely in single precision. Moreover, our error analysis provides the criterion that should be used to choose each block's precision: blocks A_{ij} such that $\|A_{ij}\| \leq \varepsilon\|A\|/u_i$ can be stored in precision u_i.

2.4 Mixed precision BLR LU factorization

We now present how to exploit the mixed precision BLR representation described previously in order to accelerate the LU factorization of BLR matrices. There exists several BLR LU factorization algorithms, and here we focus on the so-called UCF (a.k.a. UCFS or FCSU) variant described in Algorithm 2.1. This variant has been successfully used in the literature, for example in the MUMPS (Amestoy et al., 2019a) and PaStiX (Pichon et al., 2018) sparse direct solvers, and its rounding error analysis in the uniform precision case has been carried out by Higham and Mary (2021, sect. 4.2). We note that BLR LU factorization can and usually does incorporate numerical pivoting for stability, but we describe Algorithm 2.1 without pivoting for simplicity. Note that in the UCF variant, the matrix is not compressed from the beginning: instead, we perform the compression on the fly during the LU factorization (step 11 of Algorithm 2.1). To compress the R_{ik} matrices, we use the same truncation criterion as in (2.30)

$$\|R_{ik} - T_{ik}\| \leq \varepsilon\beta_{ik}, \tag{2.35}$$

where $\beta_{ik} > 0$ is a parameter whose role has been discussed in section 1.2.2 (in practice we will use global compression by setting $\beta_{ik} = \|A\|$).

Algorithm 2.1 BLR LU factorization.

1: /* **Input**: a $q \times q$ block matrix A with blocks A_{ij} of size $b \times b$. */
2: /* **Output**: its BLR LDU factors LDU. */
3: **for** $k = 1$ **to** q **do**
4: UPDATE:
5: $R_{kk} = A_{kk} - \sum_{j=1}^{k-1} L_{kj} D_{jj} U_{jk}$.
6: **for** $i = k+1$ **to** q **do**
7: $R_{ik} = A_{ik} - \sum_{j=1}^{k-1} L_{ij} D_{jj} U_{jk}$ and $R_{ki} = A_{ki} - \sum_{j=1}^{k-1} L_{kj} D_{jj} U_{ji}$.
8: **end for**
9: COMPRESS:
10: **for** $i = k+1$ **to** q **do**
11: Compute low-rank approximations $T_{ik} \approx R_{ik}$ and $T_{ki} \approx R_{ki}$.
12: **end for**
13: FACTOR:
14: Compute the LU factorization $L_{kk} D_{kk} U_{kk} = R_{kk}$.
15: **for** $i = k+1$ **to** q **do**
16: Solve $L_{ik} D_{kk} U_{kk} = T_{ik}$ for L_{ik} and $L_{kk} D_{kk} U_{ki} = T_{ki}$ for U_{ki}.
17: **end for**
18: **end for**

The error analysis presented in this section has a double purpose. First, it proves that the numerical stability of the uniform precision BLR LU factorization (proven by Higham and Mary (2021)) is preserved in mixed precision arithmetic. Second, for each operation required by Algorithm 2.1, it determines which level of accuracy is needed to maintain the overall error of order ε. Our analysis therefore guides us towards an implementation of mixed precision BLR LU that is both robust and efficient.

One technical difficulty of this analysis is the handling of the scaling factors hidden inside the U factor. To make these details more apparent, we analyze instead the LDU factorization, where L and U are unitriangular matrices (with ones on the diagonal). For the sake of readability, we will not always keep track of lower order error terms; we use the notations $\approx$ and $\lesssim$ to indicate when these terms (of order at most $u_p \varepsilon$) have been dropped.

We first analyze each kernel separately. Algorithm 2.1 requires computing products of the form $L_{ij} D_{jj} U_{jk}$ (on line 7), where L_{ij} and U_{jk} may be either uniform precision full-rank blocks or mixed precision low-rank blocks (analysis of sections 2.4.1 and 2.4.2). We also analyze in section 2.4.3 the solution of a triangular system $L_{kk} D_{kk} U_{ki} = T_{ki}$ (needed on line 16), where the right-hand side T_{ki} is a mixed precision low-rank matrix. Finally, we combine the analysis of these kernels to obtain a backward error bound on the mixed precision BLR LU factorization in section 2.4.4.

2.4.1 Low-rank matrix times full-rank matrix

Let us begin with the computation of a product $P = BC$, where C is a full-rank matrix and $B = XY^T$ is a mixed precision low-rank matrix partitioned into p groups $B_\ell = X_\ell Y_\ell^T$ satisfying $\|B_\ell\| \leq \varepsilon\beta/u_\ell$ for $\ell > 1$, and where the output P is needed under

full-rank form.

In which precision should we compute $P = BC$? The natural approach is to compute each product $P_\ell = B_\ell C$ in precision u_ℓ. Then, using Higham and Mary (2021, Lemma 3.2), the computed $\widehat{P}_\ell$ satisfies

$$\widehat{P}_\ell = B_\ell C + \Delta P_\ell, \quad \|\Delta P_\ell\| \leq \gamma_{c_\ell}^{(\ell)} \|B_\ell\| \|C\| \tag{2.36}$$

with $c_\ell = b + r_\ell^{3/2}$. For $\ell > 1$, we thus obtain

$$\widehat{P}_\ell = B_\ell C + \Delta P_\ell, \quad \|\Delta P_\ell\| \lesssim c_\ell \varepsilon \beta \|C\|, \tag{2.37}$$

since $\gamma_{c_\ell}^{(\ell)}/u_\ell = c_\ell(1 + \gamma_{c_\ell}^{(\ell)}) \approx c_\ell$. This shows that the partial product P_ℓ associated with the part of B stored in precision u_ℓ can itself be computed in precision u_ℓ, since the introduced error remains of order ε.

The next question is what precision should be used to combine the partial results into $P = \sum_{\ell=1}^{p} P_\ell$. Since for $\ell > 1$ $\|P_\ell\| \leq \varepsilon/u_\ell \beta \|C\|$, it is easy to see that $P_i + P_j$ must be computed in precision $\min(u_i, u_j) = u_{\min(i,j)}$. Knowing this, in order to maximize the performance gains associated with the use of lower precisions, the best approach is to compute $\sum_{\ell=1}^{p} \widehat{P}_\ell$ in reverse order. The approach to compute P suggested by our analysis is summarized in Algorithm 2.2.

Algorithm 2.2 Mixed precision low-rank matrix times full-rank matrix.

1: /* **Input**: a mixed precision low-rank matrix B and a full-rank matrix C. */
2: /* **Output**: $P = BC$. */
3: Initialize P to zero.
4: **for** $\ell = p$ **to** 1 **do**
5: Compute $P_\ell = B_\ell C$ in precision u_ℓ.
6: Update $P \leftarrow P + P_\ell$ in precision u_ℓ.
7: **end for**

With this algorithm, each component of $\widehat{P}_\ell$ is involved in exactly $\min(\ell, p-1)$ additions, one in each precision $u_1, \ldots, u_{\min(\ell,p-1)}$. Therefore, the computed $\widehat{P}$ satisfies:

$$\widehat{P} = \sum_{\ell=1}^{p} \widehat{P}_\ell \circ (J + \Theta_\ell), \quad |\Theta_\ell| \lesssim u_\ell, \tag{2.38}$$

where J is the matrix of ones, $\circ$ denotes the Hadamard product, and the inequality

$|\Theta_\ell| \lesssim u_\ell$ holds componentwise. Overall, we obtain

$$\widehat{P} = \sum_{\ell=1}^{p} (B_\ell C + \Delta P_\ell) \circ (J + \Theta_\ell) \tag{2.39}$$

$$= BC + \sum_{\ell=1}^{p} B_\ell C \circ \Theta_\ell + \Delta P_\ell + \Delta P_\ell \circ \Theta_\ell \tag{2.40}$$

$$= BC + \Delta P, \quad \|\Delta P\| \lesssim \big((c_1+1)u_1\|B^{(1)}\| + \sum_{\ell=2}^{p}(c_\ell+1)\varepsilon\beta\big)\|C\| \tag{2.41}$$

$$= BC + \Delta P, \quad \|\Delta P\| \lesssim (pb + r^{3/2} + p)\max(u_1\|B^{(1)}\|, \varepsilon\beta)\|C\|. \tag{2.42}$$

We summarize this analysis in the next theorem.

Theorem 2.3. *Let $B = \sum_{\ell=1}^{p} B_\ell \in \mathbb{R}^{b\times b}$ be a mixed precision low-rank matrix of rank r such that $\|B_\ell\| \le \varepsilon\beta/u_\ell$ for $\ell > 1$, and let $C \in \mathbb{R}^{b\times b}$. If $P = BC$ is computed as described by Algorithm 2.2, then the computed $\widehat{P}$ satisfies*

$$\|\widehat{P} - BC\| \lesssim c\max(u_1\|B\|, \varepsilon\beta)\|C\|, \tag{2.43}$$

with $c = pb + r^{3/2} + p$.

Theorem 2.3 shows that we can perform many of the flops in Algorithm 2.2 in lower precisions and still maintain an error of order ε. We now prove similar results for the other kernels.

2.4.2 Low-rank matrix times low-rank matrix

Next we analyze the product $P = BDC$ of two mixed precision low-rank matrices $B = X_B Y_B^T$ and $C = Y_C X_C^T$, where we also incorporate a diagonal scaling matrix D, which will be useful for the LU factorization analysis of section 2.4.4.

The product P, which is needed in full-rank form, can be computed in the following three steps:

1. Compute the inner product $M = (Y_B)^T D Y_C$.
2. Compute the middle product $W = X_B M$ (or $W = M X_C^T$).
3. Compute the outer product $P = W X_C^T$ (or $P = X_B W$).

A trivial extension of Higham and Mary (2021, Lem. 3.2) to incorporate D shows that if P is computed in uniform precision u, the computed $\widehat{P}$ satisfies

$$\widehat{P} = BDC + \Delta P, \quad \|\Delta P\| \lesssim cu\|B\|\|D\|\|C\|, \tag{2.44}$$

where $c = b + 2r^{3/2}$.

We now consider the case where B and C are partitioned into p groups $B_\ell = X_{B\ell}Y_{B\ell}^T$ and $C_m = Y_{Cm}X_{Cm}^T$, stored in precision u_ℓ and u_m, respectively. We assume that matrices B and C satisfy $\|B_\ell D\| \leq \varepsilon\beta_B/u_\ell$ and $\|DC_m\| \leq \varepsilon\beta_C/u_m$ for $\ell, m > 1$. We analyze each of the three steps separately.

Inner product $M = Y_B^T D Y_C$

Let us first analyze the computation of the inner product M. Assume $M_{\ell m} = Y_{B\ell}^T D Y_{Cm}$ is computed in a given precision denoted as $u_{\ell m}^M$. The computed $\widehat{M}_{\ell m}$ satisfies $\widehat{M}_{\ell m} = M_{\ell m} + \Delta M_{\ell m}$, with

$$|\Delta M_{\ell m}| \lesssim b u_{\ell m}^M |Y_{B\ell}|^T |D| |Y_{Cm}|. \tag{2.45}$$

By taking norms, we obtain

$$\|\Delta M_{\ell m}\| \lesssim b u_{\ell m}^M \min(\alpha_1, \alpha_2, \alpha_3), \tag{2.46}$$

where

$$\alpha_1 = \|B_\ell D\|\|C_m\|, \qquad \alpha_1 \leq \varepsilon\beta_B\|C_m\|/u_\ell \ \text{ if } \ell > 1, \tag{2.47}$$

$$\alpha_2 = \|B_\ell\|\|DC_m\|, \qquad \alpha_2 \leq \varepsilon\|B_\ell\|\beta_C/u_m \ \text{ if } m > 1, \tag{2.48}$$

$$\alpha_3 = \|B_\ell D\|\|D^{-1}\|\|DC_m\|, \quad \alpha_3 \leq \varepsilon^2\beta_B\beta_C\|D^{-1}\|/(u_\ell u_m) \ \text{ if } \ell, m > 1. \tag{2.49}$$

From this we can deduce the optimal choices of precisions $u_{\ell m}^M$ that still guarantee an error of order ε.

- If $\ell = m = 1$, in general we must take $u_{11}^M = u_1$ since $\|B_1\|$ and $\|C_1\|$ are not bounded in terms of ε. We obtain

$$\|\Delta M_{11}\| \lesssim b u_1 \|B_1\|\|D\|\|C_1\|. \tag{2.50}$$

- If $\ell = 1$ and $m > 1$, $\alpha_2 \leq \varepsilon/u_m\|B\|\beta_C$, and so taking $u_{1m}^M = u_m$ yields an error of order ε:

$$\|\Delta M_{1m}\| \lesssim b\varepsilon\|B_1\|\beta_C. \tag{2.51}$$

  Similarly, we can take $u_{\ell 1}^M = u_\ell$ and obtain

$$\|\Delta M_{\ell 1}\| \lesssim b\varepsilon\beta_B\|C_1\|. \tag{2.52}$$

- If $\ell, m > 1$, we can safely take $u_{\ell m}^M = \max(u_\ell, u_m) = u_{\max(\ell,m)}$. Indeed, if $\ell \geq m$ we can use (2.47) and if $\ell < m$ we can use (2.48), and so, in any case, we have

$$\|\Delta M_{\ell m}\| \lesssim b\varepsilon \max(\beta_B\|C_1\|, \|B_1\|\beta_C). \tag{2.53}$$

Combining (2.50), (2.51), (2.52), and (2.53), we obtain for $\ell, m \geq 1$

$$\|\Delta M_{\ell m}\| \lesssim b \max(\varepsilon\beta_B\|C_1\|, \varepsilon\|B_1\|\beta_C, u_1\|B_1\|\|D\|\|C_1\|). \tag{2.54}$$

In summary, for any value of ℓ and m, we can compute the product between the part of B stored in precision u_ℓ and the part of C stored in precision u_m in the *lower* of the two precisions. This is a crucial observation that allows us to maximize the use of lower precision.

Moreover, in some cases we may actually take $u^M_{\ell m} > \max(u_\ell, u_m)$ because of (2.49). To see why, let us take an example where $\|D^{-1}\|$, β_B, β_C, and $\|A\|$ are all approximately equal to 1. In this case, for $\ell, m > 1$, (2.46) reduces to

$$\|\Delta M_{\ell m}\| \lesssim b u^M_{\ell m}\varepsilon^2/(u_\ell u_m) \tag{2.55}$$

and so the requirement to obtain an error of order ε is

$$b u^M_{\ell m}\varepsilon \leq u_\ell u_m, \tag{2.56}$$

which thus depends not only on u_ℓ and u_m, but also on ε. If ε is small enough, (2.56) may be satisfied even for $u^M_{\ell m} > \max(u_\ell, u_m)$. For example, assume we have three precisions $u_1 = u_d = 2^{-53}$, $u_2 = u_s = 2^{-24}$, and $u_3 = u_h = 2^{-8}$. We may consider that we also have access to a 0-bit precision format, $u_4 = 1$, whose only representable value is 0. Then, ignoring the constant b in (2.56):

- The condition $u^M_{22}\varepsilon \leq u_2^2$ is satisfied for $u^M_{22} = u_3$ if $\varepsilon \leq u_s^2/u_h \approx 9 \times 10^{-13}$. Thus, if ε is small enough, M_{22} need only be computed in half precision.

- The condition $u^M_{23}\varepsilon \leq u_2 u_3$ is satisfied for $u^M_{23} = 1$ if $\varepsilon \leq u_s u_h \approx 2 \times 10^{-10}$. The same holds for u^M_{32}. Thus, if ε is small enough, the computation of M_{23} and M_{32} may be skipped altogether. Indeed, by replacing the result of the operation by 0, we would obtain a relative error of 1, which is affordable in this particular case.

- Finally, the condition $u^M_{33}\varepsilon \leq u_3^2$ is satisfied for $u^M_{33} = 1$ if $\varepsilon \leq u_h^2 \approx 2 \times 10^{-5}$. Again, the computation of M_{33} may be skipped in this case.

Going back to the general case, the precise requirement on $u^M_{\ell m}$ depends on u_ℓ, u_m, ε, β_B, β_C, and $\|D^{-1}\|$. For global compression ($\beta_B = \beta_C = \|A\|$), we obtain

$$\|\Delta M_{\ell m}\| \lesssim b u^M_{\ell m}\varepsilon^2\|A\|^2\|D^{-1}\|/(u_\ell u_m). \tag{2.57}$$

Middle product $W = X_B M$ (or $W = M X_C^T$)

We analyze the product $W = X_B M$, the case of $W = M X_C^T$ being analogous. Let $W_m = \sum_{\ell=1}^{p} X_{B\ell} M_{\ell m}$, for $m = 1\colon p$. Assume the product $W_m^{(\ell)} = X_{B\ell} M_{\ell m}$ is computed in

precision $u_{\ell m}^W$, then the computed $\widehat{W}_m^{(\ell)}$ satisfies

$$\widehat{W}_m^{(\ell)} = X_{B\ell}\widehat{M}_{\ell m} + \Delta W_m^{(\ell)}, \tag{2.58}$$

$$\|\Delta W_m^{(\ell)}\| \lesssim r_\ell u_{\ell m}^W \|X_{B\ell}\|\|\widehat{M}_{\ell m}\| \lesssim r_\ell^{3/2} u_{\ell m}^W \|B_\ell D C_m\|. \tag{2.59}$$

This bound on $\|\Delta W_m^{(\ell)}\|$ is similar to the bound (2.46) on $\|\Delta M_{\ell m}\|$, and we should therefore set $u_{\ell m}^W = u_{\ell m}^M$. Then, similarly to Algorithm 2.2, the partial results $W_m^{(\ell)}$ should be summed in reverse order and in increasing precision, since $W_m^{(\ell)} + W_m^{(\ell+1)}$ must be computed in precision $u_{\ell m}^W$. Overall, with $u_{\ell m}^W = u_{\ell m}^M = \max(u_\ell, u_m)$, the computed $\widehat{W}_m$ satisfies

$$\widehat{W}_m = \sum_{\ell=1}^{p} \widehat{W}_m^{(\ell)} \circ (J + \Theta_\ell), \quad |\Theta_\ell| \lesssim u_\ell, \tag{2.60}$$

$$= \sum_{\ell=1}^{p} (X_{B\ell} M_{\ell m} + X_{B\ell}\Delta M_{\ell m} + \Delta W_m^{(\ell)}) \circ (J + \Theta_\ell), \tag{2.61}$$

$$= \sum_{\ell=1}^{p} W_m^{(\ell)} + \Delta\widehat{W}_m^{(\ell)} = W_m + \Delta W_m, \tag{2.62}$$

with

$$\|\Delta\widehat{W}_m^{(\ell)}\| \lesssim (b + r_\ell^{3/2} + 1)\max(\varepsilon\beta_B\|C\|, \varepsilon\|B\|\beta_C, u_1\|B\|\|D\|\|C\|) \tag{2.63}$$

and so

$$\|\Delta W_m\| \lesssim (pb + r^{3/2} + p)\max(\varepsilon\beta_B\|C\|, \varepsilon\|B\|\beta_C, u_1\|B\|\|D\|\|C\|). \tag{2.64}$$

Outer product $P = WX_C^T$ (or $P = X_B W$)

It remains to analyze the final product $P = WX_C^T$ (or $P = X_B W$, which is analogous). Let $P_m = W_m X_{Cm}^T$ be computed in precision u_m^P. Then the computed $\widehat{P}_m$ satisfies

$$\widehat{P}_m = \widehat{W}_m X_{Cm}^T + \Delta P_m, \tag{2.65}$$

$$\|\Delta P_m\| \lesssim r_m u_m^P \|\widehat{W}_m\|\|X_{Cm}\| \le r_m^{3/2} u_m^P \|\widehat{W}_m\| \lesssim r_m^{3/2} u_m^P \sum_{\ell=1}^{p} \|B_\ell D C_m\|. \tag{2.66}$$

Since $\sum_{\ell=1}^{p} \|B_\ell D C_m\|$ is at least as large as $\|B_1 D C_m\|$, by (2.48) we must take $u_m^P = u_m$. Then, (2.66) becomes

$$\|\Delta P_m\| \lesssim r_m^{3/2}\max(\varepsilon\|B\|\beta_C, u_1\|B\|\|D\|\|C\|). \tag{2.67}$$

Finally, as previously for W_m, we sum P_m over m in reverse order and in increasing precision, to obtain a computed $\widehat{P}$ satisfying

$$\widehat{P} = \sum_{m=1}^{p} \widehat{P}_m \circ (J + \Theta_m), \quad |\Theta_m| \lesssim u_m, \tag{2.68}$$

$$= \sum_{m=1}^{p} (\widehat{W}_m (X_{Cm})^T + \Delta P_m) \circ (J + \Theta_m), \tag{2.69}$$

$$= \sum_{m=1}^{p} P_m + (\Delta W_m X_{Cm}^T + \Delta P_m) \circ (J + \Theta_m), \tag{2.70}$$

$$= P + \Delta P, \tag{2.71}$$

with

$$\|\Delta P\| \lesssim (p^2 b + (p+1) r^{3/2} + p^2 + p) \max(\varepsilon \beta_B \|C\|, \varepsilon \|B\| \beta_C, u_1 \|B\| \|D\| \|C\|). \tag{2.72}$$

This concludes the analysis of the product P. We summarize the approach suggested by this analysis in Algorithm 2.3, for which the following theorem holds.

Algorithm 2.3 Mixed precision low-rank matrix times mixed precision low-rank matrix.

1: /* **Input**: mixed precision low-rank matrices $B = X_B Y_B^T$ and $C = Y_C X_C^T$ and a diagonal matrix D. */
2: /* **Output**: $P = BDC$. */
3: Initialize P to zero.
4: **for** $m = p$ **to** 1 **do**
5: Initialize W_m to zero.
6: **for** $\ell = p$ **to** 1 **do**
7: Compute $M_{\ell m} = Y_{B\ell} D Y_{Cm}^T$ in precision $\max(u_\ell, u_m)$.
8: Compute $W_m^{(\ell)} = X_{B\ell} M_{\ell m}$ in precision $\max(u_\ell, u_m)$.
9: Update $W_m \leftarrow W_m + W_m^{(\ell)}$ in precision $\max(u_\ell, u_m)$.
10: **end for**
11: Compute $P_m = W_m X_{Cm}^T$ in precision u_m.
12: Update $P \leftarrow P + P_m$ in precision u_m.
13: **end for**

Theorem 2.4 (Low-rank times low-rank)**.** *Let $B = \sum_{\ell=1}^{p} B_\ell$ and $C = \sum_{m=1}^{p} C_m$ be two mixed precision low-rank matrices satisfying*

$$\|B_\ell D\| \le \varepsilon \beta_B / u_\ell \quad \text{for } \ell > 1, \tag{2.73}$$

$$\|D C_m\| \le \varepsilon \beta_C / u_m \text{ for } m > 1, \tag{2.74}$$

and D a diagonal matrix, and let $P = BDC$ be computed as described in Algorithm 2.3. Then, the computed $\widehat{P}$ satisfies

$$\widehat{P} = BDC + \Delta P, \quad \|\Delta P\| \lesssim c \max\left(\varepsilon \beta_B \|C\|,\ \varepsilon \|B\| \beta_C,\ u_1 \|B\| \|D\| \|C\|\right), \tag{2.75}$$

with $c = p^2 b + (p+1)r^{3/2} + p^2 + p$.

2.4.3 Triangular system with low-rank right-hand side

The last kernel that we need to analyze is the solution of a triangular system $LDZ = B$, where $L \in \mathbb{R}^{b\times b}$ is lower triangular, D is diagonal, and the right-hand side $B = YX^T$ is a mixed precision low-rank matrix (used in Algorithm 2.1, line 16). We analyze the kernel for a lower triangular matrix L, the upper triangular case ($ZDU = B$) being analogous. For this kernel, the output (the solution Z) is needed under low-rank form.

In the uniform precision case, if the system $LDZ = B$ is solved in uniform precision u, the computed solution $\widehat{Z}$ satisfies (Higham and Mary, 2021, Lemma 3.5)

$$L D \widehat{Z} = B + \Delta B, \quad \|\Delta B\| \lesssim bu\|L\|\|D\|\|\widehat{Z}\|. \tag{2.76}$$

Let $B_\ell = Y_\ell X_\ell^T$ be the part of B that is stored in precision u_ℓ, satisfying $\|B_\ell\| = \|Y_\ell\| \leq \varepsilon\beta/u_\ell$ for $\ell > 1$. Then, the natural approach to solve $LDZ = B$ in mixed precision is to solve each system $LDV_\ell = Y_\ell$ in precision u_ℓ and to define $Z_\ell = V_\ell X_\ell^T$, which yields the mixed precision low-rank solution $Z = \sum_{\ell=1}^{p} Z_\ell$. However, a traditional normwise analysis based on (2.76) does not provide a satisfactory bound here: if we apply (2.76) to $LDV_\ell = Y_\ell$ and use $\widehat{V}_\ell \approx D^{-1}L^{-1}Y_\ell$, we obtain the bound

$$L D \widehat{V}_\ell = Y_\ell + \Delta Y_\ell, \quad \|\Delta Y_\ell\| \lesssim b\varepsilon\beta\kappa(L)\kappa(D). \tag{2.77}$$

This bound is very weak due to the presence of the normwise condition numbers $\kappa(L)\kappa(D) = \|L\|\|L^{-1}\|\|D\|\|D^{-1}\|$.

A stronger bound can be obtained by using a componentwise analysis:

$$L D \widehat{V}_\ell = Y_\ell + \Delta Y_\ell, \quad |\Delta Y_\ell| \lesssim bu_\ell |L||D||\widehat{V}_\ell|. \tag{2.78}$$

Replacing $\widehat{V}_\ell$ by $D^{-1}L^{-1}(Y_\ell + \Delta Y_\ell)$ in the bound on ΔY_ℓ yields

$$|\Delta Y_\ell| \lesssim bu_\ell |L||D||D^{-1}L^{-1}Y_\ell| \leq bu_\ell |L||L^{-1}||Y_\ell|. \tag{2.79}$$

We can now take norms, obtaining for $\ell > 1$

$$\|\Delta Y_\ell\| \lesssim bu_\ell \operatorname{cond}(L, Y_\ell)\|Y_\ell\| \leq b\varepsilon\beta \operatorname{cond}(L, Y_\ell) \tag{2.80}$$

where $\operatorname{cond}(L, Y_\ell)$ is the condition number introduced by Skeel (1979) (Higham, 2002, Eq. (7.13)):

$$\operatorname{cond}(L, Y_\ell) = \frac{\||L||L^{-1}||Y_\ell|\|}{\|Y_\ell\|}. \tag{2.81}$$

Multiplying both sides of (2.78) by X_ℓ^T on the right yields

$$LD\widehat{Z}_\ell = B_\ell + \Delta Y_\ell X_\ell^T. \tag{2.82}$$

Summing (2.82) over ℓ, we obtain

$$LD\widehat{Z} = B + \sum_{\ell=1}^{p} \Delta Y_\ell X_\ell^T = B + \Delta B, \tag{2.83}$$

$$\|\Delta B\| \lesssim bu_1 \|L\|\|D\|\|\widehat{Z}_1\| + pb\varepsilon\beta \,\mathrm{cond}(L), \tag{2.84}$$

where $\mathrm{cond}(L) = \||L^{-1}||L|\|$.

The use of intermediate componentwise bounds presents two advantages. First, we obtain a bound with $\mathrm{cond}(L)$, which is in general smaller than $\kappa(L)$ (see Higham (2002, p. 123) for a discussion on the difference between these two quantities). Second and more importantly, we have dropped the matrix D from the term proportional to $\varepsilon\beta$, which shows that this term is invariant under scaling, and which represents a significant improvement since $\kappa(D) \approx \kappa(A)$ can be arbitrarily large. Importantly, in the case of an LDU factorization with partial pivoting, both L and U are well conditioned, and $\mathrm{cond}(L)$ is in practice a small constant. Therefore, in the context of Algorithm 2.1, the mixed precision triangular solution analyzed here is backward stable. However, for a general system $LDZ = B$, bound (2.84) does not guarantee backward stability, and indeed some examples can be built where the use of mixed precision arithmetic in the solution of the system leads to a large increase of the backward error (we note however that such examples are very hard to find and we were only able to construct one using direct search optimization (Higham, 1993)).

We summarize the proposed approach to solve $LDZ = B$ in Algorithm 2.4 and its error analysis in the following theorem.

Algorithm 2.4 Solution to $LDZ = B$ (triangular system with low-rank right-hand side).

1: /* **Input**: a mixed precision low-rank matrix $B = YX^T$, a lower triangular matrix L, and a diagonal matrix D. */
2: /* **Output**: a mixed precision low-rank matrix Z, solution to $LDZ = B$. */
3: **for** $\ell = p$ **to** 1 **do**
4: Solve the triangular system $LDV_\ell = Y_\ell$ in precision u_ℓ.
5: Define $Z_\ell = V_\ell X_\ell^T$ (no computation performed: output is low-rank).
6: **end for**

Theorem 2.5. *Let $L \in \mathbb{R}^{b\times b}$ be a lower triangular full-rank matrix and let $B = \sum_{\ell=1}^{p} B_\ell$ be a mixed precision low-rank matrix satisfying $\|B_\ell\| \le \beta\varepsilon/u_\ell$. If the system $LDZ = B$ is solved by Algorithm 2.4, the computed solution $\widehat{Z}$ satisfies*

$$LD\widehat{Z} = B + \Delta B, \quad \|\Delta B\| \lesssim pb\varepsilon\beta \,\mathrm{cond}(L) + bu_1\|L\|\|D\|\|\widehat{Z}\|. \tag{2.85}$$

2.4.4 Putting everything together: error analysis of mixed precision BLR LU factorization

Now that we have analyzed all the kernels of Algorithm 2.1, we are ready to prove the backward stability of the BLR LU factorization in mixed precision arithmetic. We define

$$\lambda_1 = \max_{k=1:\,q} \max\left(\|L_{kk}^{-1}\|, \|U_{kk}^{-1}\|, \mathrm{cond}(L_{kk}), \mathrm{cond}(U_{kk})\right). \tag{2.86}$$

If partial pivoting is performed, λ_1 is almost always small in practice and of order a constant (Higham, 2002, Chapter 8), even though in theory it can only be bounded by $2^b - 1$ (Higham, 2002, Lemma 8.6). We also define

$$\lambda_2 = \max_{i \geq j} \max\left(\|L_{ij}\|, \|U_{ji}\|\right). \tag{2.87}$$

If partial pivoting is performed, $\lambda_2 \leq b$.

Let us bound the error incurred in the computation of some (i,k) block of the L factor, the U factor analysis being similar. For $i < k$, L_{ik} is obtained by solving

$$L_{ik} D_{kk} U_{kk} = T_{ik}, \tag{2.88}$$

where T_{ik} is the compressed form of

$$R_{ik} = A_{ik} - \sum_{j=1}^{k-1} \widehat{L}_{ij} D_{jj} \widehat{U}_{jk}, \tag{2.89}$$

where $\widehat{L}_{ij}$ and $\widehat{U}_{jk}$ are the LU factors computed at the previous steps, and are represented as mixed precision low-rank matrices, and the product $\widehat{L}_{ij} D_{jj} \widehat{U}_{jk}$ is computed with Algorithm 2.3. Note that if one of $\widehat{L}_{ij}$ or $\widehat{U}_{jk}$ is a full-rank matrix, the analysis is similar and relies on Theorem 2.3; if both are full-rank, the computation is done in uniform precision $u_1 \ll \varepsilon$ and introduces an error term $b u_1 \|\widehat{L}_{ij}\| \|D_{jj}\| \|\widehat{U}_{jk}\|$. The difficulty is that $\widehat{L}_{ij}$ and $\widehat{U}_{jk}$ are not directly the result of a compression, and so we cannot directly control the norms of $\widehat{L}_{ij}^{(\ell)}$ and $\widehat{U}_{jk}^{(m)}$, the parts of $\widehat{L}_{ij}$ and $\widehat{U}_{jk}$ stored in precision u_ℓ and u_m, respectively. Instead, they are given by

$$\widehat{L}_{ij}^{(\ell)} \approx T_{ij}^{(\ell)} \widehat{U}_{jj}^{-1} D_{jj}^{-1}, \tag{2.90}$$

$$\widehat{U}_{jk}^{(m)} \approx D_{jj}^{-1} \widehat{L}_{jj}^{-1} T_{jk}^{(m)}, \tag{2.91}$$

where T_{ij} and T_{jk} have been compressed such that

$$\|T_{ij}^{(\ell)}\| \leq \varepsilon \beta_{ij} / u_\ell \quad \text{for } \ell > 1, \tag{2.92}$$

$$\|T_{jk}^{(m)}\| \leq \varepsilon \beta_{jk} / u_m \text{ for } m > 1. \tag{2.93}$$

Therefore, the norms of $\widehat{L}_{ij}^{(\ell)}$ and $\widehat{U}_{jk}^{(m)}$ depend on β_{ij} and β_{jk}, respectively, but also on the

scaling factors in D_{jj}. However, one of the two D_{jj}^{-1} in (2.90)–(2.91) is canceled by the D_{jj} in (2.89) and $\|\widehat{L}_{jj}^{-1}\|$ and $\|\widehat{U}_{jj}^{-1}\|$ are both bounded by λ_1. As a result, we can rewrite the product $R_{ik,j} = \widehat{L}_{ij}D_{jj}\widehat{U}_{jk}$ as BDC, where

$$\|B_\ell D\| \lesssim \lambda_1 \varepsilon \beta_{ij}/u_\ell \quad \text{for } \ell > 1, \tag{2.94}$$

$$\|DC_m\| \lesssim \lambda_1 \varepsilon \beta_{jk}/u_m \text{ for } m > 1. \tag{2.95}$$

By Theorem 2.4, the computed $\widehat{R}_{ik,j}$ satisfies

$$\widehat{R}_{ik,j} = \widehat{L}_{ij}D_{jj}\widehat{U}_{jk} + \Delta R_{ik,j}, \tag{2.96}$$

$$\|\Delta R_{ik,j}\| \lesssim c \max(\lambda_1 \varepsilon \beta_{ij}\|\widehat{U}_{jk}\|, \lambda_1 \varepsilon \beta_{jk}\|\widehat{L}_{ij}\|, u_1\|\widehat{L}_{ij}\|\|D_{jj}\|\|\widehat{U}_{jk}\|). \tag{2.97}$$

By (2.89), we obtain a computed $\widehat{R}_{ik}$

$$\widehat{R}_{ik} = A_{ik} \circ (J + \Theta_k) - \sum_{j=1}^{k-1} \left(\widehat{L}_{ij}D_{jj}\widehat{U}_{jk} + \Delta R_{ik,j}\right) \circ (J + \Theta_j), \tag{2.98}$$

where $|\Theta_j| \le \gamma_j^{(1)} J$ accounts for the errors in the additions of the products $\widehat{R}_{ik,j}$ to A_{ik}. We thus obtain

$$\widehat{R}_{ik} = A_{ik} - \sum_{j=1}^{k-1} \widehat{L}_{ij}D_{jj}\widehat{U}_{jk} + \Delta R_{ik}, \tag{2.99}$$

$$\|\Delta R_{ik}\| \lesssim k u_1 \|A_{ik}\| + \sum_{j=1}^{k-1} \max\left(\lambda_1\lambda_2 c\varepsilon \max(\beta_{ij}, \beta_{jk}),\ (\lambda_2^2 c + j)u_1\|D_{jj}\|\right). \tag{2.100}$$

$\widehat{R}_{ik}$ is then compressed into T_{ik} such that the part of T_{ik} stored in precision u_ℓ satisfies

$$T_{ik}^{(\ell)} = \widehat{R}_{ik}^{(\ell)} + E_{ik}^{(\ell)}, \quad \|E_{ik}^{(\ell)}\| \le \varepsilon \beta_{ik}/u_\ell, \tag{2.101}$$

and so overall, by Theorem 2.1,

$$T_{ik} = \widehat{R}_{ik} + E_{ik}, \quad \|E_{ik}\| \lesssim (2p-1)\varepsilon \beta_{ik}. \tag{2.102}$$

Finally, we solve (2.88) for L_{ik}, and by Theorem 2.5, the computed $\widehat{L}_{ik}$ satisfies

$$\widehat{L}_{ik}D_{kk}\widehat{U}_{kk} = T_{ik} + F_{ik}, \quad \|F_{ik}\| \lesssim pb\lambda_1 \varepsilon \beta_{ik} + bu_1\lambda_2^2\|D_{kk}\|. \tag{2.103}$$

Putting together (2.103), (2.102), (2.100), we obtain

$$\widehat{L}_{ik}D_{kk}\widehat{U}_{kk} = A_{ik} - \sum_{j=1}^{k-1} \widehat{L}_{ij}D_{jj}\widehat{U}_{jk} + \Delta R_{ik} + E_{ik} + F_{ik}, \tag{2.104}$$

and so

$$A_{ik} = \sum_{j=1}^{k} \widehat{L}_{ij} D_{jj} \widehat{U}_{jk} + \Delta A_{ik}, \tag{2.105}$$

$$\|\Delta A_{ik}\| \lesssim k u_1 \|A_{ik}\| + \sum_{j=1}^{k} \max\left(\lambda_1 \lambda_2 c\varepsilon \max(\beta_{ij}, \beta_{jk}),\ (\lambda_2^2 c + j) u_1 \|D_{jj}\|\right). \tag{2.106}$$

With the choice $\beta_{ij} = \beta_{jk} = \|A\|$ for $j = 1\colon k$, and since $k \le q$, we obtain

$$\|\Delta A_{ik}\| \lesssim \lambda_1 \lambda_2 c q \varepsilon \|A\| + q u_1 \|A_{ik}\| + q(\lambda_2^2 c + q)\rho u_1 \|A\|, \tag{2.107}$$

where we have used $\|D_{jj}\| \le \rho \|A\|$, where ρ denotes the growth factor. This concludes the case $i < k$. Bounds analogous to (2.107) hold for $i = k$ and $i > k$, and so overall we have

$$A = \widehat{L} D \widehat{U} + \Delta A, \quad \|\Delta A\| \lesssim q^2\left(\lambda_1 \lambda_2 c\varepsilon + (\lambda_2^2 c + q)\rho u_1\right)\|A\|. \tag{2.108}$$

We summarize this analysis in the next theorem.

Theorem 2.6 (Mixed precision BLR LU factorization)**.** *Let $A \in \mathbb{R}^{n\times n}$ be a BLR matrix partitioned into q^2 blocks of order b. If the BLR LU factorization of A in p precisions described by Algorithms 2.1–2.4 runs to completion, the computed LU factors satisfy*

$$A = \widehat{L} D \widehat{U} + \Delta A, \quad \|\Delta A\| \lesssim q^2\left(\lambda_1 \lambda_2 c\varepsilon + (\lambda_2^2 c + q)\rho u_1\right)\|A\|, \tag{2.109}$$

where λ_1 and λ_2 are defined by (2.86)–(2.87), ρ is the growth factor, and $c = p^2 b + (p + 1)r^{3/2} + p$.

Theorem 2.6 therefore proves the backward stability of the mixed precision BLR LU factorization: the computed LU factors give an exact LU decomposition of a perturbed matrix, where the norm of the perturbation $\|\Delta A\|$ is of order ε. The precise value of the constants $q^2\lambda_1\lambda_2 c$ and $q^2(\lambda_2^2 c + q)\rho$ in (2.109) is not of great importance but, as a check, we compare it against the uniform precision bound of Higham and Mary (2021, Thm. 4.3)

$$A = \widehat{L}\widehat{U} + \Delta A, \quad \|\Delta A\| \lesssim q\varepsilon\|A\| + (b + 2r^{3/2} + q)u_1 \|\widehat{L}\|\|\widehat{U}\|. \tag{2.110}$$

Since $\|\widehat{L}\|\|\widehat{U}\| \lesssim n^2\rho\|A\|$ and, with partial pivoting, $\lambda_2 \le b$, we see that both (2.109) and (2.110) grow as $O(n^2(b + q)\rho)$.

After Theorem 2.6, not much additional effort is needed to prove the backward stability of the solution of linear systems $Ax = v$ by mixed precision BLR LU factorization. We note that mixed precision arithmetic can also be used in the solution of the triangular systems with the LU factors, but in the interest of space, we omit these details.

Table 2.1: List of matrices used in the experiments of chapter 2. We use their Schur complement corresponding to the root separator in their multifrontal factorization, whose order n is given in the second column.

Matrix	n	b	Application	symmetry
nd24k	8k	128	2D/3D problem	SPD
audikw_1	4k	128	Structural problem	SPD
perf009d	2k	64	From EDF (code_aster): elastic computation of a pump with internal pressure (safety device in a nuclear power plant)	SYM
Transport	5k	256	3D finite element flow and transport	UNSYM
P64	4k	128	Poisson equation (3D, mesh size=64)	SPD
nlpkkt80	14k	256	3D PDE-constrained optimization problem	SYM
Fault_639	8k	128	Contact mechanics for a faulted gas reservoir	SPD
Geo_1438	13k	256	Geomechanical model of earth crust	SPD
Serena	16k	256	Gas reservoir simulation for CO2 sequestration	SPD
Cube_Coup_dt0	21k	256	3D coupled consolidation problem (3D cube)	SYM

2.5 Experimental results

2.5.1 Experimental setting

We have written a MATLAB code that implements a mixed precision variant of Algorithm 2.1 that uses Algorithms 2.2–2.4. Our implementation can use any number of arbitrary precisions, where the lower precisions are simulated using the `chop` function of Higham and Pranesh (2019). To compress the blocks, we use a mixed precision truncated QR decomposition with column pivoting—we omit a detailed description of this algorithm, which we plan to investigate more in depth in future work.

For our experiments, we use the matrices listed in Table 2.1. These matrices are all obtained as the Schur complement of larger sparse matrices (specifically, the root separators of their multifrontal factorization) arising in various applications: P64 comes from the discretization of a Poisson equation, perf009d comes from a structural mechanics problem from EDF (French electricity supplier), the others come from the SuiteSparse collection (Davis and Hu, 2011).

To confirm experimentally the numerical stability of the algorithms, and to assess the impact of mixed precision arithmetic on their accuracy, we measure backward errors. Rather than measuring the backward error for the LU factorization, which is expensive to compute, we use the computed LU factors to solve a linear system $Ax = v$, where x is the vector of ones (and v is computed as Ax), and we use the computed solution $\widehat{x}$ to measure the backward error

$$\frac{\|A\widehat{x} - v\|}{\|A\|\|\widehat{x}\|} \tag{2.111}$$

given by the Rigal–Gaches theorem (Higham (2002, Thm 7.1), Rigal and Gaches (1967)).

To evaluate the potential of using mixed precision for the BLR LU factorization, we

will focus on two performance metrics: the storage cost for the factorized matrix and the expected time cost of the factorization. Both these costs depend on the relative performance of each arithmetic. It is easy to measure the storage cost since, for each arithmetic, it is proportional to the number of bits used: an fp32 number requires half the storage of an fp64 one, and a bfloat16 number requires a quarter of the storage. Hence, we have

$$\text{storage cost} = (\#\text{entries in fp64}) + 0.5 \times (\#\text{entries in fp32}) + 0.25 \times (\#\text{entries in bf16}). \tag{2.112}$$

The time cost will be estimated based on the number of flops performed in each arithmetic. This is a more complex issue because the relative speed of each arithmetic strongly depends on the hardware, the matrix, and several other factors. A practical high-performance implementation of the mixed precision BLR factorization is outside the scope of this chapter but, as a rough indicator, we make the assumption that the speed of each arithmetic is also proportional to the number of bits. Hence, we use the cost model

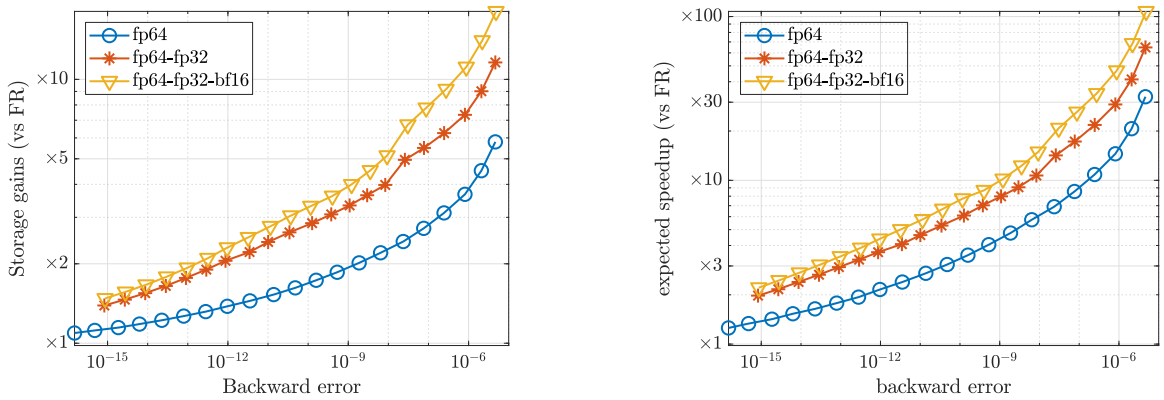
$$\text{expected time cost} = (\#\text{flops in fp64}) + 0.5 \times (\#\text{flops in fp32}) + 0.25 \times (\#\text{flops in bf16}). \tag{2.113}$$

2.5.2 Performance–accuracy tradeoff

The analytical error bounds obtained in section 2.4 show that the use of mixed precision arithmetic should only increase the backward error by a small constant. In this first experiment, we check experimentally (i) that the error increase is indeed small, and (ii) whether the expected time and storage gains obtained by the use of mixed precision justify this error increase, that is, whether the mixed precision variant achieves a better performance–accuracy tradeoff than the uniform precision variant. Indeed, since the mixed precision variant achieves a slightly larger error, to be completely fair, we should compare it against the uniform precision variant with a correspondingly larger ε.

To answer this question, we perform the following experiment in Figure 2.3: for a given matrix, taking several values of ε, we plot the gains in storage and expected time (2.112) and (2.113) as a function of the backward error (2.111). We compare three variants of the BLR factorization: the standard uniform precision variant run entirely with fp64 arithmetic, a two-precision variant using both fp64 and fp32, and a three-precision variant using bfloat16 as well. The figure shows that the two-precision variant achieves a much better performance–accuracy tradeoff than the uniform precision one, and that the three-precision variant further improves this tradeoff. Indeed, using lower precisions slightly increases the error, but the experiment shows that this increase is largely compensated by the cost reductions. Indeed, the closer a variant is to the top left corner of the plots, the better its tradeoff is: for a given accuracy, it is less costly than the other variants, or, equivalently, for a given cost, it achieves an improved accuracy.

In light of this experiment, and to avoid hand-tuning ε for every variant and every



(a) Storage. (b) Expected time.

Figure 2.3: Storage (2.112) and expected time (2.113) for three variants of the BLR LU factorization of matrix perf009d, given as a factor of gain compared with the full-rank factorization, and as a function of the backward error (2.111). Each point corresponds to a run performed with a certain value of ε, which ranges from 10^{-16} to 10^{-6}

matrix, in the remainder of our experiments we directly compare the variants with the same value of ε.

2.5.3 Results on real-life matrices

In this section we experiment on the Schur complement of real-life matrices listed in Table 2.1. Figure 2.4 compares the backward error (2.111) achieved by the BLR factorization for three values of ε: 10^{-12}, 10^{-9}, and 10^{-6}. For $\varepsilon = 10^{-6}$, we compare the uniform fp32 precision BLR factorization with the two-precision one using fp32 and bfloat16; for $\varepsilon = 10^{-9}$ and 10^{-12}, we compare the uniform fp64 precision factorization with both a two-precision variant (using fp64 and fp32) and a three-precision one (also using bfloat16). The figure shows that the use of mixed precision arithmetic does not significantly impact the backward error, leading to an increase of at most an order of magnitude in the worst case (and very often much less than that).

Figure 2.5 shows the associated storage and time gains expected from the use of mixed precision arithmetic. For each matrix, each bar corresponds to a different value of ε. For $\varepsilon = 10^{-12}$ and 10^{-9}, we focus on the gains achieved by the three-precision variant. The y-axis (height of the bars) indicates the number of entries (or number of flops) required by the mixed precision variant as a percentage of the double precision variant. For storage, this percentage would be at least 100% if we could ignore some minor numerical perturbations, and can be larger than that because of the difference between conditions (2.26) and (2.27). Indeed, as explained in section 2.2, there are some blocks that satisfy (2.27) but do not satisfy (2.26): that is, we allow the mixed precision variant to store more entries, because we expect this increase to pay off thanks to the use of lower precisions. The same property stands for the flops, but some operations are also skipped (see section 2.4.2): therefore, the percentage for the flops can also differ from 100%, either being larger or smaller.

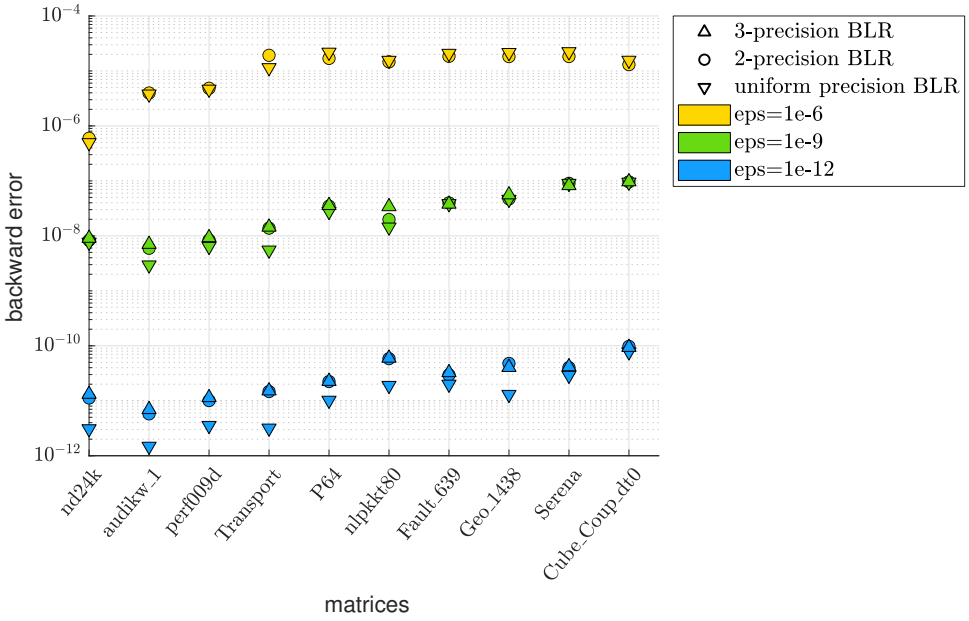


Figure 2.4: Backward error (2.111) for the uniform and mixed precision BLR factorizations, for $\varepsilon = 10^{-12}$, 10^{-9}, and 10^{-6}.

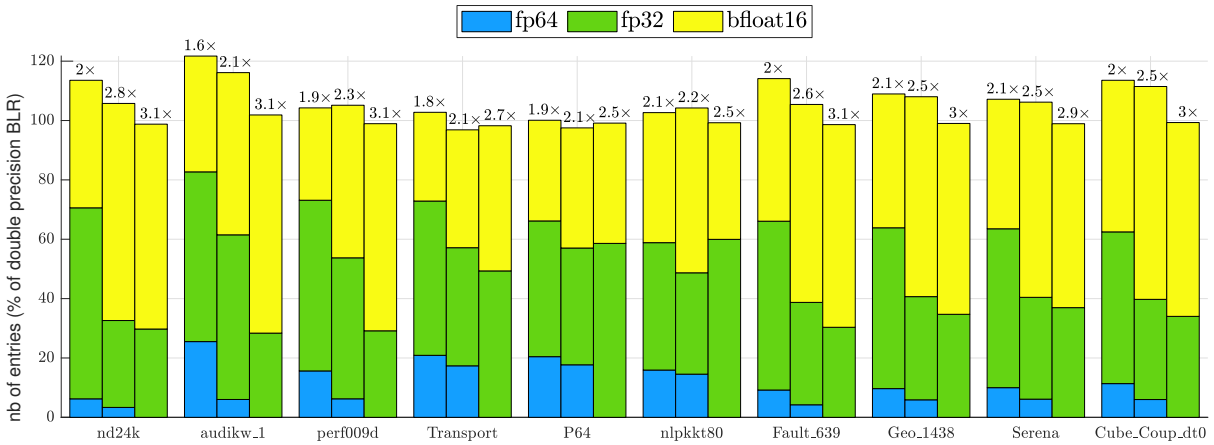
The colors breakdown in Figure 2.5 shows the proportion of entries that are stored in each precision, and the proportion of flops that are performed in each precision. For $\varepsilon = 10^{-6}$, note that the mixed precision algorithm recovers the fact that we do not need any entries or flops in fp64 arithmetic, since $\varepsilon > u_s$. For all matrices there is a significant fraction of the entries and flops that can be switched to lower precisions, even for $\varepsilon = 10^{-12}$. This is a very positive result that confirms that BLR matrices are amenable to the use of mixed precision arithmetic and that the proposed mixed precision BLR representation can achieve very significant gains with respect to the one in double precision.

The number on top of each bar in Figure 2.5 indicates the resulting gains in storage and expected time achieved by the mixed precision variant compared with the double precision variant. The figure shows very significant reductions, of up to a factor 2.8× in storage and 3.5× in expected time for $\varepsilon = 10^{-9}$.

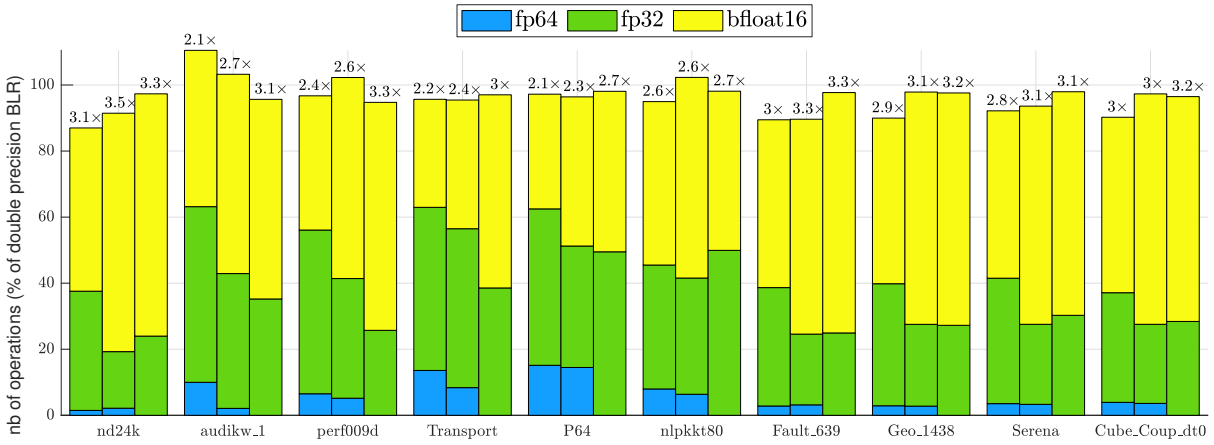
Finally, in Figure 2.6 we perform a similar experiment as in Figure 2.5 for matrices of increasing size belonging to the same Poisson problem class. This experiment highlights an important and valuable property of the mixed precision BLR factorization: the storage and expected time gains increase with the problem size, as a larger and larger fraction of the entries and flops can be safely switched to lower precisions.

2.6 Conclusion

We have introduced a novel approach to exploit mixed precision arithmetic for low-rank approximations. Given a prescribed accuracy ε, we have proved in Theorem 2.1 that singular vectors associated with sufficiently small singular values can be stored in precisions with unit roundoff larger than ε while preserving an overall accuracy of order ε. This approach is not only applicable to low-rank matrices built with a singular value decompo-



(a) Storage.

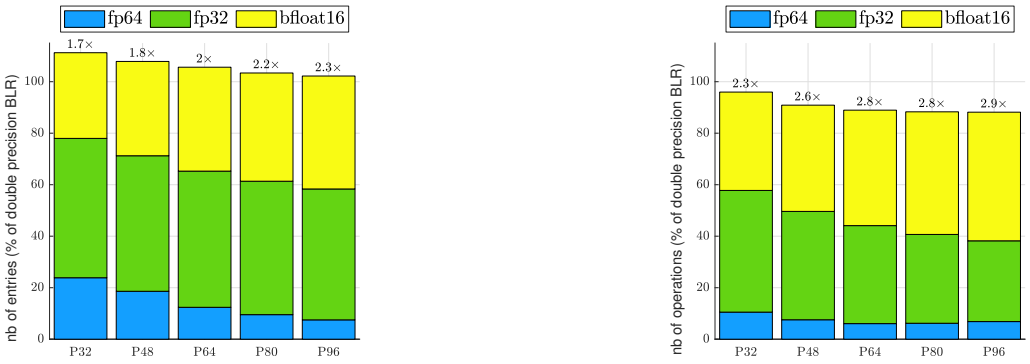


(b) Expected time.

Figure 2.5: For each matrix, the 3 bars correspond to $\varepsilon = 10^{-12}$, $\varepsilon = 10^{-9}$ and $\varepsilon = 10^{-6}$ respectively. The y-axis shows the number of entries and flops required by the mixed precision variant with respect to the double precision variant (which can be slightly different from 100% because of the different conditions to represent a block under low-rank form (2.26) and (2.27)). The color breakdown gives the proportion of entries and flops in each precision. The number above each bar indicates the resulting factor of gain in storage (2.112) and expected time (2.113) achieved by the mixed precision variant compared with the double precision one.

sition, but also to many other low-rank decompositions, in particular rank-revealing QR (Lemma 2.2).

We have applied this approach to block low-rank (BLR) matrices, for which this new mixed precision low-rank approximation presents a high potential. We have adapted the existing uniform precision BLR LU factorization algorithm (Algorithm 2.1) to exploit the mixed precision representation of the blocks. We carried out the rounding error analysis of this new algorithm and obtained two key results. First, we proved in Theorem 2.6 that the use of mixed precision arithmetic does not compromise the numerical stability of BLR LU factorization recently proven by Higham and Mary (2021). Second, our analysis helps us determine what accuracy is needed for each floating-point operation. The resulting mixed precision BLR algorithms are summarized in Algorithms 2.2, 2.3, and 2.4. Interestingly enough, our analysis suggests that some operations may even be



(a) Storage.

(b) Expected time.

Figure 2.6: Proportion of entries and flops in each precision, and, on top of the bars, the resulting factors of gain in storage and expected time achieved by the mixed precision BLR factorization for Poisson matrices of increasing size. We have set $\varepsilon = 10^{-12}$ and $b = 64$.

skipped (see section 2.4.2).

We have evaluated the potential of this mixed precision BLR LU factorization on a range of matrices coming from real-life problems from industrial and academic applications. We have shown that a large fraction of the entries and flops can be safely switched to lower precisions. For $\varepsilon = 10^{-9}$, by mixing fp64, fp32, and bfloat16 arithmetics, we obtain reductions in storage of up to 2.8× with respect to double precision BLR. Moreover, assuming fp32 and bfloat16 flops are, respectively, twice and four times faster than fp64 ones, we estimate the expected time gains, predicting reductions of up to 3.5× with respect to double precision BLR. We emphasize that these gains are not achieved at the expense of accuracy: for the same accuracy, the mixed precision variant is less expensive than the uniform precision one, or, equivalently, for a fixed storage or work budget, the mixed precision variant is more accurate (Figure 2.3).

Given the very promising results obtained on large dense matrices with this new mixed precision BLR approach, we will now try to adapt it to the multifrontal method, and to develop its high-performance implementation within the sparse direct solver MUMPS (Amestoy et al., 2019a), which already exploits BLR compression.

Chapter 3

The multifrontal method in mixed precision

In this chapter we want to improve the performance of the multifrontal method. We adapt it in order to take advantage of mixed precision, based on the algorithms and concepts presented in chapter 2. First, we will consider the use of mixed-precision BLR (MPBLR) as a storage format only. Its aim is to further reduce the cost of storing a BLR matrix. We include it in the multifrontal method in order to reduce its memory consumption. In a second section, we perform computations in mixed precision exploiting the MPBLR compression. By doing this we hope to accelerate the different steps of the multifrontal method.

We implemented our algorithms in the MUMPS multifrontal solver. Some of the solver functionalities presented in this chapter are not public yet, and may be added in a future version. All the experiments were performed on the Olympe supercomputer of the CALMIP center (project P0989). Unless specified otherwise, experiments are performed on 2 computational nodes, using 4 MPI processes, and 18 OpenMP threads per process.

3.1 Mixed precision aiming for storage and communication reductions

3.1.1 Using custom precision formats

We first present an alternative MPBLR format aiming at optimizing the storage cost of a matrix. In the next subsection we will integrate it in the multifrontal method.

As seen in chapter 1, a floating-point number is a number of the form

$$f = (-1)^s \times \sum_{i=0}^{t} m_i 2^{-i} \times 2^{(e-bias)}$$

where s is the sign bit, the integer e is the exponent, and $m = m_0 m_1 \cdots m_t$ is the mantissa. $m_0 = 1$ is an implicit bit, and is not actually stored, according to the description of the

binary interchange floating-point formats in the IEEE 754 standard (see IEEE Computer Society, 2008). In this case, the unit roundoff is $u = 2^{-t}$. If the representation of the exponent stays the same, a format with fewer mantissa bits, less costly in terms of storage, will have a greater unit roundoff. As a consequence, in order to maximize the storage gains, we will try to use a unit roundoff as big as possible.

It is worth mentioning that, according to the description of the interchange floating-point formats, the bits are stored in the following order: s, e, m. In particular, the least significant bits of m are stored last.

In chapter 2 we saw that, given a set of formats whose unit roundoffs are $u_1 \cdots u_p$, we can find a criterion indicating in which precision format we should store the columns X_k and Y_k of a low-rank approximation $XY^T \approx B$. Such a criterion may vary depending on the type of low-rank approximation. For example, we may choose the largest value of u_i (i.e., the lowest precision) satisfying $\|X_k Y_k^T\| \leq \varepsilon/u_i$. This condition ensures that the error is bounded by a small multiple of ε, as shown in section 2.3.2.

Another way to consider this is that column k should be stored in any precision format u above a certain target accuracy $u_{target}^{(k)} = \varepsilon/\|X_k Y_k^T\|$. Thus it is understandable that, if we add new precision formats, we will sometimes be able to switch the column to a lower precision format, less costly, while the resulting error bound will be closer to ε (but still below it). The error bound of each column would be equal to ε if we had a large number of available precision formats, enough so that u would vary almost continuously. Such a set of precision formats would minimize the cost for storing an MPBLR matrix (under the constraint of respecting a certain error bound).

We did not try to implement such a continuum of precision formats. However, we understand that having only 2 or 3 precision formats like in chapter 2 may not be entirely satisfactory: we wish to add a few more precision formats to the set, in order to make a trade-off between these two extremes. In this section, MPBLR compression is only used for storage: therefore, in order to define a new precision format, one only needs functions doing the conversions between the working precision and the new format. Such conversions may be implemented in software, which means that it is possible to add virtually any format.

We choose to obtain new custom formats by keeping only the t' most significant bits of the mantissa. Conveniently, the discarded bits are all located at the right-hand side of the format, making it easier to truncate them. The remaining bits correspond to a new floating-point format, with the same exponent range but fewer mantissa bits and a higher roundoff $u' = 2^{-t'}$. The rounding mode of this conversion is towards zero.

In order to obtain an implementation that is easy and somewhat efficient, we choose to only allow byte manipulations, instead of bit manipulations: the new formats are obtained by removing the last bytes of the mantissa. The conversion back to high precision consists in padding the missing bits with zeros. As a consequence of these choices, we obtain new formats on 56, 48, 40 and 24 bits respectively, that we name fp56, fp48, fp40 and fp24.

name	sign & exponent bits	mantissa bits
fp64	12	52
fp56	12	44
fp48	12	36
fp40	12	28
fp32	9	23
fp24	9	15
bf16	9	7

Table 3.1: A set of seven custom floating-point formats

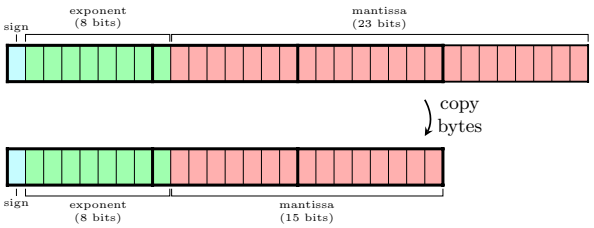


Figure 3.1: Conversion from fp32 to fp24, performed by copying the first bytes.

We also add a 16-bits format, which turns out to correspond to the definition of bloat16. We also reuse the standard formats fp64 and fp32. As a result, we obtain a set of seven precision formats, which is described in Table 3.1 As a matter of fact, the very same set of 7 floating-point formats has already been proposed several times in the literature, in particular by Anderson et al. (2017), Mukunoki and Imamura (2016) and Graillat et al. (2022).

An example of such a custom floating-point format would be fp24. Its number of exponent bits is the same as fp32: indeed, we deem the range of fp32 to be sufficient, as this hypothesis was already made several times in this thesis. Therefore, when converting a number from fp64 to fp24, it has to be converted to fp32 first. The conversion from fp32 to fp24, shown in Figure 3.1, is performed by discarding the last byte, which corresponds to the 8 least significant bits. The conversion back to fp32 is obtained by padding the last byte with zero. We made a first implementation of these byte manipulations in Fortran.

An interesting perspective would be to increase the number of formats even more, by truncating the mantissa at the bit level instead of the byte level. The sizes of the available formats would take all values between 10 bits (mantissa of size 1), and 64 bits, thus approximating a continuum of floating-point formats. The gain would be especially noticeable for very low precision formats. For example, we may want to convert the last non-discarded columns from bfloat16 to "fp10". For these columns, the gain of switching the truncation operation from the byte level to the bit level would be the most beneficial, reaching an additional storage reduction of $\frac{6}{16} \approx 38\%$. Handling such formats would probably be complex and costly due to data misalignment, but it might be useful when wanting to obtain a near-optimal MPBLR compression, with no regard for the overhead.

When handling very low precision formats, the cost for storing the exponent bits begins to matter. A good illustration of this fact is the format fp10, whose exponent bits

occupy 80% of the storage space. For bfloat16, this proportion is already 50%. However, we notice that the range of the exponents in a given format is rather narrow, in particular for the lowest precision formats. This is because we use low precision for the columns having the smallest coefficients: all coefficients of a column stored in precision i tend to be smaller than ε/u_i (though this may depend on the kind of low-rank approximation being used). Therefore, another perspective would be to perform a lossless compression of the exponent. For example, we could use a limited number of bits to represent the exponents, by removing the leading bits whose values are zero. In order to know the number of bits to be discarded, we would need to check beforehand the maximal value of the exponent in a column.

3.1.2 Multifrontal method with mixed-precision storage

We want to take advantage of the storage-focused MPBLR format described previously. The main idea is the following: whenever the LU factors are not being used in a computation, we would like them to be stored in mixed precision. Indeed, the storage of the LU factors contributes to a large part of the memory peak of the multifrontal method in practice. However, they are not needed between the end of the factorization of a front and the triangular solution (last step of the multifrontal method).

The main modifications added to Algorithm 1.4 are the following:

- While computing the low-rank compression, we also compute the ranks of the mixed-precision parts $r_1, \cdots, r_p$.
- At the end of the factorization of a front, the low-rank blocks are copied from uniform precision to the mixed-precision low-rank storage format. The original blocks in uniform precision are not needed anymore and may be discarded.
- Before it is used in the triangular solve step, each low-rank block is temporarily converted from mixed precision to uniform precision. The computations remain in uniform precision.

3.1.3 Block-admissibility conditions

In a BLR compression, we use a condition for choosing whether a block should be low-rank or full-rank: this is the block-admissibility condition, already mentioned in section 2.2 (see Equation 2.26). This choice may be based on the "distance" between the groups of variables corresponding to the rows and the columns. However there is also a more pragmatic alternative, which is used in MUMPS: a block is considered as low-rank if and only if the compression leads to a storage reduction. That is to say, the low-rank admissibility condition for a block of size (m, n) is:

$$r(m+n) \leq mn. \tag{3.1}$$

Conveniently, this condition for reducing the storage is also the condition for reducing the number of operations.

However, the situation is a bit different when using BLR compression for computations and MPBLR for storing the factors. The condition for reducing the number of operations stays the same, whereas the condition for reducing storage becomes the following:

$$(m+n)\sum_{k=1}^{p} c_k r_k \leq mn, \tag{3.2}$$

where c_k corresponds to the cost of storing a floating-point number in precision u_k instead of u_1. For example, if we use two precisions fp64 and fp32, then Equation 3.2 takes the form $(m+n)(r_1+0.5r_2) \leq mn$.

However trivial this aspect may seem, it has proven to be important in practice, as a noticeable fraction of the blocks satisfy the mixed-precision condition (Equation 3.2), but not the uniform-precision condition (Equation 3.1). That is to say, they would be full-rank in BLR but are low-rank in MPBLR.

On the other hand, switching to the block-admissibility condition for mixed precision (Equation 3.2) requires additional operations, which should lead to an overhead. Indeed, some blocks that were considered as full-rank are now compressed, and this compression requires additional operations. Moreover, the maximal values for the ranks have been increased: therefore, more operations may be needed before knowing whether a block is low-rank or not, at least if this choice is done by trying to compress the block. Finally, when using the block-admissibility suited to mixed precision, some blocks are not suited for computations: the ranks are too high, which is not optimal. They should have been considered as full-rank in order to minimize the number operations, as shown in Equation 3.1. By forcefully considering them as low-rank, we increase the number of operations, which should lead once again to an overhead. This is what is done in the current implementation in MUMPS, but it could still be improved in the future.

3.1.4 Implementation in MUMPS

We implemented our algorithm within the sparse solver MUMPS: we can now choose to use an MPBLR compression as a storage format. We added several options in order to customize its use. In particular:

- mpblr_thld: This variable allows the user to modify the thresholds for switching between two precision formats. In chapter 2, we had considered that such a threshold would be ε/u_i for precision i. We now use a modified threshold, divided by mpblr_thld. Unless otherwise specified, we set mpblr_thld $= 10$.

  As a result, the theoretical error bounds obtained when converting a low-rank block to mixed precision (with $p = 2$ formats) is roughly $(1+2(p-1)/\text{mpblr\_thld})\varepsilon = 1.2\varepsilon$, instead of 3ε obtained in Equation 2.22. This setting was deemed to be conservative

enough with regard to the uniform precision, whose error bound is ε: in most applications the numerical error introduced by the use of mixed precision should not be significant. If it is, then an easy correction would be to increase the value of mpblr_thld.

- custom_prec: Allows the user to choose between using only the precision formats supported by the hardware, or a set of custom precision formats. In the current state of our implementation, the first choice results in using fp64 and fp32. Otherwise, we use a subset of our 7 precision formats described in Table 3.1. We should not need any format whose precision is higher than the working precision. Therefore, if MUMPS is compiled in fp32, we will only use 3 precision formats: fp32, fp24 and bf16.

- mixed_adm: Allows to switch between the usual block-admissibility condition suited to uniform precision (see Equation 3.1) and a condition minimizing the storage in mixed precision (see Equation 3.2), at the cost of a few additional operations.

3.1.5 Storage gains and time overhead

In our implementation, we handle two BLR formats at once: one that is used for computations in the working precision, and an MPBLR format that is used for storage. Performing data copies between the two formats, as well as the allocations/deallocations, may cause some overhead. Moreover, handling custom precision formats may have a relatively high cost, due to the fact that all conversions are implemented at the software level instead of the hardware level. Finally, we may want to use the appropriate block-admissibility condition: this choice results in additional storage reduction, at the cost of an extra overhead. Therefore, choosing of an MPBLR variant results in a trade-off between memory and computation time: despite reducing the memory needed for the factorization even further, the use of more complex storage formats should induce an overhead.

In Table 3.2 we compare the execution time of different steps of MUMPS, as well as the memory peak, when using different levels of compression. The full-rank variant (FR) has no compression, contrary to BLR. MPBLR(2) and MPBLR(7) use sets of 2 and 7 precision formats respectively, and only differ from BLR by adding data conversions (i.e., they keep the same block-admissibility condition as in double precision). We also compare those two variants to their respective optimizations using the block-admissibility conditions suited to mixed precision: MPBLR(2)* and MPBLR(7)*.

As expected, the size of the factors is reduced when increasing the number of precision formats: we obtained reductions of the factor size of up to 24% by using two hardware-supported precision formats, and up to 32% when using software-implemented custom precision formats. Such a compression of the factors induces a reduction of the memory peak. On the other hand, we confirm that increasing the number of precision formats

generally causes an overhead, albeit small.

When adding mixed-precision admissibility, the compression becomes a bit more efficient, obtaining storage reductions of up to 27% 38% with sets of 2 and 7 precisions respectively. In general, the time spent in factorization and in solve tends to increase again. In later experiments using MPBLR compressions, we will use this mixed-precision admissibility unless specified otherwise.

matrix	variant	LU factors (GBytes)	Memory peak (GBytes)	Backward error	Time (s) factorization	solve
Poisson200	FR	289†	OOM	–	–	–
($\varepsilon = 10^{-10}$)	BLR	71	220	7.0E-8	395	0.69
	MPBLR(2)	54	211	8.9E-8	448	0.79
	MPBLR(7)	48	208	7.5E-8	445	0.90
	MPBLR(2)*	52	209	7.2E-8	439	0.81
	MPBLR(7)*	44	205	5.9E-8	400	0.97
thmgaz	BLR	103	132	3.8E-14	61	1.7
($\varepsilon = 10^{-10}$)	MPBLR(2)	88	127	3.9E-14	62	1.8
	MPBLR(7)	84	126	6.9E-14	63	1.8
	MPBLR(2)*	80	120	5.2E-14	68	1.9
	MPBLR(7)*	67	111	5.5E-14	68	2.1
Geoazur160	FR	182	253	2.5E-4	465	1.52
($\varepsilon = 10^{-4}$)	BLR	81	153	6.7E-3	308	0.46
	MPBLR(7)	61	138	2.1E-2	323	0.84
	MPBLR(7)*	61	138	2.5E-2	317	0.84

Table 3.2: A comparison of the storage reductions and time overheads for different variants. FR uses no compression and BLR stands for the standard BLR compression in uniform precision. MPBLR(2) and MPBLR(7) use sets of 2 and 7 precisions respectively, with the block-admissibility for uniform precision (Equation 3.1). The variants MPBLR(2)* and MPBLR(7)* differ by the use of the block-admissibility condition suited for mixed precision (Equation 3.2). For matrix Geoazur160, the working precision is fp32 and any precision format higher than this will not be used. Therefore, it makes no sense to use variant MPBLR(2), equivalent to uniform precision. When using variant MPBLR(7), only the 3 lowest precision formats of the set are used.
†: estimation
OOM : out of memory
Backward error: $\frac{\|A\widehat{x}-b\|_\infty}{\|A\|_\infty\|\widehat{x}\|_\infty}$

We observed that using a BLR storage format different from the working format causes an overhead with our implementation. However, such overheads may be reduced, or may not even happen at all with an implementation focused on performance. In fact, the data movements have been reduced by doing this: at the very least, a memory-bound algorithm such as the BLR triangular solve should be able to turn such a reduction of the data movements into actual time gains, under the assumption that the conversion functions are sufficiently well tuned (which is not the case for our naive implementation, targeting storage gains only). Indeed, as explained in section 1.4.6 , we could hope to obtain, for a memory-bound algorithm such as this one, a speedup factor equal to the

factor of storage gain from mixed precision at most.

A side-effect of using mixed precision is that the backward error could be increased a bit. In fact, we saw in section 2.2 that using mixed precision increased the error bounds of a BLR compression by a certain factor. For a number of precision formats $p = 7$, we should expect the error bounds to be multiplied by approximately $1 + 2(p - 1)/\text{mpblr\_thd} = 2.2$ (a less pessimistic bound would be that, the error bound is multiplied by $1 + 2\sqrt{p-1}/\text{mpblr\_thd} \approx 1.49$, as a consequence of Equation 2.23). Therefore, we could expect to observe a similar increase in the backward error of a sparse linear system. However, we did not observe such a behavior: like in the dense case (see section 2.5), the error stays very similar when adding mixed precision, and we are still far from such an increase.

3.1.6 Compressing contribution blocks in mixed precision

There is another possibility to reduce the memory consumption of the multifrontal factorization. The basic idea is similar to our motivation to use MPBLR as a storage format: when it is kept unused for a long time, data could be compressed as much as possible, in order to reduce the memory peak, even at the cost of some additional operations. Therefore, we allow the compression of the block of non-eliminated variables at the end of a front (the so called Contribution Block, also referred to as CB), by using a BLR compression instead of keeping them uncompressed: in fact, the off-diagonal blocks of the CB have low numerical ranks, similar to the factors. As a consequence, the memory peak will be reduced (as well as the volume of the communication). Such a feature is being implemented in the multifrontal solver MUMPS, and is referred to as option "compressCB".

However, contrary to the use of MPBLR, the time complexity of the compression operations performed here is by no means small, leading in general to a significant time overhead. Moreover, those compressions are used for storage gains only, and they do not contribute to reducing the number of operations like the compressions of the factors. Therefore, such an option should only be used when the memory consumption is critical and a certain overhead is acceptable.

We observed a reduction of the memory peak of up to of 24% when using option CompressCB, as illustrated in Table 3.3.

In MUMPS, we added the possibility of switching the BLR compression of the contribution block to mixed precision. Thanks to this, the memory peak is further reduced by up to 15%, which makes a total reduction of up to 35% when considering the combined gains of both options (variants "BLR" and "MPBLR*+MP-CCB*" from Table 3.3). The overhead caused by the copy/conversion operations should be negligible compared to the cost of the extra low-rank compressions that are already added with option CompressCB.

As a bonus, the MPI communication volume has been reduced even more, given the

fact that a possibly huge part of the MPI communications now concern mixed-precision data.

matrix	variant	Memory peak (GB)	Backward error	Time for factorization (s)
Poisson200 ($\varepsilon = 10^{-10}$)	BLR	220	6.6E-8	442
	MPBLR*	208	7.6E-8	447
	BLR+CCB	168	6.2E-8	127
	MPBLR*+MP-CCB*	144	6.9E-8	128
thmgaz ($\varepsilon = 10^{-10}$)	BLR	132	5.3E-14	62.0
	MPBLR*	125	6.8E-14	62.3
	BLR+CCB	122	3.7E-14	69.3
	MPBLR*+MP-CCB*	104	3.0E-14	69.3

Table 3.3: Impact of option "compressCB" (CCB), and its mixed-precision variant using 7 precision formats (MP-CCB) on the memory peak and the factorization time. The notation * refers to the use of the block-admissibility condition suited for mixed precision (Equation 3.2).

3.1.7 Reducing the communication volume

Distributed memory parallelism in MUMPS

As explained in section 1.1.3, the multifrontal method can be parallelized by exploiting two kinds of parallelism, referred to as tree parallelism and node parallelism. In MUMPS, distributed-memory parallelism is used in order to take advantage of both sources of parallelism. This is illustrated by Figure 3.2: for the node parallelism, each front is split between several processes. Each process is mapped on a group of rows, on which it will perform all update operations. The master process is associated with the fully summed rows, while the other rows are split between the worker processes.

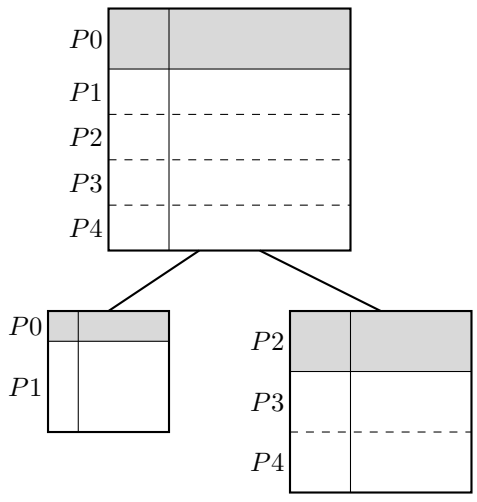


Figure 3.2: Illustration of tree and node parallelism. The shaded part of each front represents its fully summed rows.

Sending the LU factors in mixed precision

We consider the case of an LU factorization, whose communication pattern is simpler than an LDL^T factorization. We consider a node whose rows are split between several MPI processes (node parallelism), and a worker process that owns a set of rows I. Its main task is to perform update operations on its rows, in right-looking:

$$A_{ij} \leftarrow A_{ij} - L_{ik}U_{kj}, \text{ for } i \in I, \text{ for } j > k.$$

The blocks L_{ik}, $i \in I$ needed for this operation have been computed locally, and do not need to be sent by another process. However, the blocks U_{kj} have been computed on the master node and need to be sent. Therefore, once the master process has updated a block-row of U, it sends it to the other processes, so that they may perform the corresponding updates.

When using BLR compression in MUMPS, the volume of communication is reduced by sending the blocks U_{kj} in a low-rank form. We added the possibility of sending these blocks in mixed precision, in order to reduce the volume of communication.

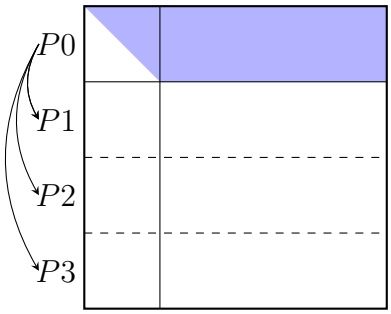


Figure 3.3: A node split between several MPI processes. The blue part represents the content of the messages, that is to say the fully summed U factors.

Experimental results

When using this MPBLR compression of the communications on U, as well as option compressCB in mixed precision, most communications are performed in mixed precision. As shown in Table 3.4, the results are quite promising: we can expect a reduction in the communication volume by a factor of 1.9 by adding mixed precision. When combining these gains with the ones from option compressCB, we realize that the reduction of the communication volume is quite substantial, up to a factor 13.

No time gain has been observed yet by switching the MPI communications to mixed precision. However, we can reasonably hope that another implementation, better tuned, would be able to achieve time gains thanks to the communication being in mixed precision, in the case of a large problem treated on a huge number of processes. In fact, Table 3.3 shows that reducing the communication volume may sometimes lead to actual time gains on the appropriate problem: on matrix Poisson200, using option compressCB reduces the factorization time by a factor 3.5, despite performing a greater number of operations.

matrix	variant	Communication volume (GBytes)		
		LU	CB	Total
Poisson200	BLR	13.8	203	217
	MPBLR*	6.7	202	209
	BLR+CCB	13.7	21	34
	MPBLR*+MP-CCB*	6.7	10	17
thmgaz	BLR	3.9	8.6	12.5
	MPBLR	2.0	8.6	10.6
	BLR+CCB	3.9	3.9	7.9
	MPBLR*+MP-CCB*	2.0	2.1	4.2

Table 3.4: Communication volumes during an LU factorization with the same variants of MUMPS as in Table 3.3. The runs are done on 2 computational nodes: 8 MPI processes and 9 OpenMP threads per process. The contributions to the communication volume are the U factors as well as the contribution bloc (CB).

3.2 Mixed precision aiming for time gains

In the previous section we presented an algorithm in which the low-rank blocks are stored using a MPBLR compression. During the triangular solve phase, these blocks were converted back to the working precision (e.g., double precision), and then used in this format for the computations.

However we know from our theoretical analysis in chapter 2 that this might not be optimal: if the precision formats used for storage are also available on hardware (e.g.: fp64 and fp32), then some of the computations can directly be performed on the MPBLR structure, without the need to convert it back to the working precision. This is the case because a low-precision part can often be used in the precision format that was used for storing it. By doing this, the accuracy of the solution is preserved, while taking advantage of the higher speed of low-precision computations.

We will first discuss algorithmic issues when switching computations to mixed precision for the product between a low-rank block and a matrix. Then we will apply it to the triangular solve, in order to obtain time gains. Finally, we will generalize this kernel so that it can be used in the factorization as well.

3.2.1 Main techniques

LR×FR product

One of the main low-rank kernels is the multiplication between a low-rank block XY^T and a full-rank matrix M, and we will refer to it as LR×FR kernel. As explained in chapter 1, this operation can be accelerated when using low-rank approximations: if we choose to perform the multiplications in the right order, i.e. $X \times (Y^T \times M)$, then the complexity is reduced compared to the FR version, from b^3 to $2rb^2$ if M is a square matrix of size (b, b).

Algorithm 3.1 LR×FR product: $X(Y^TM)$.

Input: a low-rank block XY^T and a full-rank matrix M
Output: $Z = XY^TM$

1: $W \leftarrow Y^TM$
2: $Z \leftarrow XW$

We now want to further reduce the time complexity by performing the operations in mixed precision, based on the algorithm presented in chapter 2. If XY^T is stored in the mixed-precision form $[X_1 \cdots X_p][Y_1 \cdots Y_p]^T$, then the operations $X_k \times (Y_k^T \times M)$ can be performed in precision k. This is illustrated in Figure 3.4, in the case where the chosen precision formats are fp64 and fp32. The full details are given in Algorithm 3.2, which can handle an arbitrary number p of precision formats (this will prove helpful for supporting half precision in the future). A minor optimization has been taken into account by looping on the precision formats from the lowest one to the highest: by doing this, we can manage to switch some additions to lower precision formats, in case $p \geq 3$.

Another minor optimization would be to skip any precision format whose part is empty, making sure that none of these formats require any operation. The implementation is relatively easy: we basically need to loop on the precision formats whose parts are not empty.

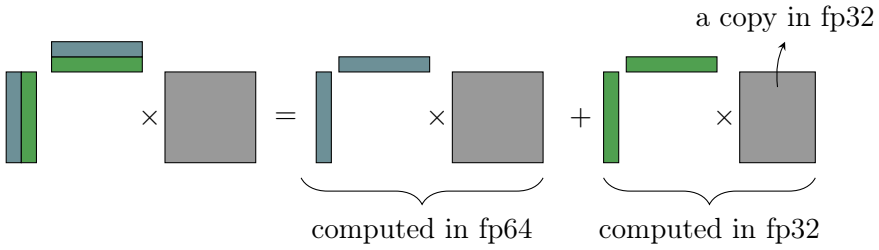


Figure 3.4: Illustration of a low-rank product $X(Y^TM)$ performed in mixed precision.

Algorithm 3.2 LR×FR product in mixed precision

Input: a mixed precision low-rank block $XY^T = [X_1 \cdots X_p][Y_1 \cdots Y_p]^T$ and p copies of M in each precision $M_1, \ldots, M_p$.
Output: $Z = XY^TM$

1: $Z_p \leftarrow 0$
2: **for** $k = p$ **to** 1 **do**
3: $W_k \leftarrow Y_k^T M_k$ /* performed in precision k */
4: $Z_k \leftarrow Z_k + X_k W_k$ /* performed in precision k */
5: **if** $k > 1$ **then**
6: Cast Z_k to precision $k-1$ and store the result in Z_{k-1}.
7: **end if**
8: **end for**
9: $Z \leftarrow Z_1$

Avoid conversions by duplicating a variable in every precision format

Note that Algorithm 3.2 requires on input the copies of the full-rank block M in every precision format. Depending on the context, we may choose to cast this block on the fly, just before entering this kernel. However, most of the time we should prefer to keep those copies of a full-rank block, despite the temporary extra memory cost. Indeed, those copies will be reused many times.

A similar choice is available for the matrix Z on output: it may be duplicated in order to reduce the number of conversions. More specifically, its contributions $Z_k = X_k Y_k^T M$, for $k \in [1, p]$ can either be cast on the fly, or be reused in mixed precision. In Algorithm 3.2, we have chosen to do these conversions on the fly, as soon as Z_k has been computed. This choice is meaningful if only one update at a time is performed, as in a right-looking algorithm.

However, we often want to perform many updates in a row on the block Z, i.e. we may want to perform the following operations: $Z \leftarrow Z - \sum_{j=1}^{q} X^{(j)} Y^{(j)T} M^{(j)}$. This is typically the case with a left-looking algorithm. In this case, it is still possible to reuse Algorithm 3.2 and do the conversions on the fly, one after the other: we obtain Algorithm 3.3. However, this algorithm is a bit naive, and we may try to better take advantage of this sum.

Another possibility (see Algorithm 3.4) is to accumulate the contributions of each precision format u_k in a separate buffer Z_k, i.e. perform $Z_k = \sum_{j=1}^{q} X_k^{(j)} Y_k^{(j)T} M^{(j)}$. By doing this, most conversions are avoided. The only ones left are performed on each subsum Z_k, when doing the operation $Z \leftarrow Z - \sum_{k=1}^{p} Z_k$. Therefore, the number of conversions has been reduced by a factor q overall. The huge majority of the operations performed are now BLAS-like matrix operations, instead of having 1 conversion every 3 matrix operations: we can expect this to be more efficient.

It is technically possible to implement right-looking algorithms based on Algorithm 3.4: we would still have to compute similar sums, but a large number of them at once, with bad spatial and temporal localities. Also, the size of the temporary buffers Z_k would not be small like before. Indeed, we perform a large number of such sums at the same time, and we have to duplicate all the temporary sums. In the worst case (e.g., right-looking factorization), the whole front has to be copied p times. Due to this extra memory cost, Algorithm 3.4 may not be worth using in right-looking.

Algorithm 3.3 Computation of a sum of LR×FR products

Input: a set of mixed precision low-rank blocks $X^{(j)}Y^{(j)T} = [X_1^{(j)} \cdots X_p^{(j)}][Y_1^{(j)} \cdots Y_p^{(j)}]^T$, for $j \in [1; q]$ and p copies of $M^{(j)}$ in each precision $M_1^{(j)}, \ldots, M_p^{(j)}$, for $j \in [1; q]$.
Output: $Z = \sum_{j=1}^{q} X^{(j)} Y^{(j)T} M^{(j)}$

1: $Z \leftarrow 0$
2: **for all** j **do**
3: $\quad Z \leftarrow Z + \text{prod}(X^{(j)}, Y^{(j)}, M^{(j)})$ /* using Algorithm 3.2 */
4: **end for**

Algorithm 3.4 Computation of a sum of LR×FR products
(accumulation of updates in mixed precision, minimizing the number of conversions)

Input: a set of mixed precision low-rank blocks $X^{(j)}Y^{(j)T}=[X_1^{(j)} \cdots X_p^{(j)}][Y_1^{(j)} \cdots Y_p^{(j)}]^T$, for $j \in [1;q]$ and p copies of $M^{(j)}$ in each precision $M_1^{(j)}, \ldots, M_p^{(j)}$, for each $j \in [1;q]$].
Output: $Z = \sum_{j=1}^{q} X^{(j)}Y^{(j)T}M^{(j)}$

$Z_p \leftarrow 0$
for $k = p$ **to** 1 **do**
  for all j **do**
    $W_k^{(j)} \leftarrow Y_k^{(j)T} M_k^{(j)}$ /* performed in precision k */
    $Z_k \leftarrow Z_k + X_k^{(j)} W_k^{(j)}$ /* performed in precision k */
  end for
  Cast Z_k to precision $k-1$ and store the result in Z_{k-1}.
end for
$Z \leftarrow Z_1$

Combining MPBLR formats for storage and computations

If we want to obtain better speedups, then the set of precision formats used for storage should be the same as the formats used for computations. Indeed, by doing this we remove the need to perform costly copies and conversions on the data, as emphasized at the beginning of the section.

However, in the case where reducing the memory footprint is the main priority, it is in general beneficial to use a custom set of precision formats for storing the factors, and still do the computations using all hardware-supported precision formats. Indeed, we may hope that each computation will be quicker than in double precision. As for casting the custom precision formats to the computation formats, it should not be more costly than casting them to double precision (in both cases, the conversion has to be implemented at the software level). We could also expect to reduce the data movements, which could make the conversion quicker.

As a consequence, we will try to combine the use of the two MPBLR formats at once in order to take advantage of both the optimal storage reduction and computational time gains. In all that follows we will refer to an MPBLR format as MPBLR(sto) if it is used for storage, and we will try to refer to its low-rank blocks as $X^{(sto)}Y^{(sto)T}$ instead of XY^T. Similarly, we will refer to an MPBLR format used for computations as MPBLR(calc), and we will call refer to its low-rank blocks as $X^{(calc)}Y^{(calc)T}$.

When the sets of precision formats used for storage and for computations are identical, we do not need to perform actual data copies between the two formats, because they will refer to the same data structure.

3.2.2 Application to the triangular solve

Algorithms

We apply the use of mixed precision for computations to the triangular solution phase of the multifrontal method. We will consider both the forward elimination and the backward elimination. Both steps are quite similar, and we could use the same techniques for both of them. However, in the current MUMPS implementation, the forward solve is performed in right-looking, and the backward solve is performed in left-looking.

We use Algorithms 1.5 and 1.6 for these two steps. The function *prod* (at lines 4 and 3 respectively) refers to the product of a block of the factors with a block of the right-hand side (RHS). If the block of the factors is represented as low-rank, then this operation can be seen as a product LR×FR, and it can be performed using Algorithm 3.1.

We now want to switch some of the computations of the forward elimination to mixed precision, in order to obtain Algorithm 3.5. Therefore, we now use mixed precision for the LR×FR products (function *prod*), as in Algorithm 3.2. However, this kernel needs each block B_j to be copied in every precision format: we do so at line 3. We only need to keep copies of one block at a time, which means that the extra memory cost caused by these copies will be negligible.

We also add mixed precision in the left-looking algorithm doing the backward elimination, and we obtain Algorithm 3.6. In left-looking, the updates of a block can be considered as doing the operation $B_i \leftarrow B_i - \sum_j U_{ij} B_j$. It requires a sum of LR×FR products (via function *sum_prods*), and it may be computed using one of the algorithms presented previously.

In a first implementation, we used Algorithm 3.3 in order to compute such a sum: the terms are computed independently, with a large number of conversions needed. Another alternative would be to use Algorithm 3.4 for *sum_prods*, summing together contributions in each precision format.

In both cases we choose to keep copies of each block of the RHS in all precision formats, so that we do not need to cast it on-the-fly before using it. The number of conversions is reduced, but the memory is increased if we are using a large number of right-hand sides.

Algorithm 3.5 Forward elimination (Right-looking algorithm, optionally with mixed precision)

1: **for** $j = 1$ **to** q_{fs} **do**
2: $\quad B_j \leftarrow L_{jj}^{-1} B_j$
3: $\quad$ Copy B_j in precisions $u_2, \cdots, u_p$
4: $\quad$ **for** $i = j + 1$ **to** q **do**
5: $\quad\quad B_i \leftarrow B_i - prod(L_{i,j}, B_j)$
6: $\quad$ **end for**
7: **end for**

Algorithm 3.6 Backward elimination (Left-looking algorithm, with mixed precision)

1: Copy the whole RHS B in precisions $u_2, \cdots, u_p$
2: **for** $i = q_{\text{fs}}$ **to** 1 **do**
3: $\quad B_i \leftarrow B_i - sum\_prods(U_{i1}, \cdots, U_{iq}, B_j)$
4: $\quad B_i \leftarrow U_{ii}^{-1} B_i$
5: **end for**

Results

We evaluate the performance of the mixed-precision solve on matrix Queen_4147 from Table 4.3.We report in Tables 3.5 and 3.6 the time performance of the forward elimination and backward elimination respectively for different solve variants, for three values of n_{rhs}.

We begin by analyzing the single RHS case ($n_{\text{rhs}} = 1$). This is rather critical for EDF applications since the solve phase (whose performance can be critical when MUMPS is used as a preconditioner) often involves a unique RHS.

We first consider the MPBLR variants presented in the previous section, referred to as sto2* and sto7* here. Having the mixed-precision block-admissibility activated, these two variants minimize the memory cost of the factorization given their respective sets of precisions (2 and 7 precisions), which makes them interesting when reducing the memory is a major concern. However, the time spent in both forward and backward substitutions have been increased drastically, as we had already seen in Table 3.2. Using 7 precisions formats, this time is increased by a factor 2.75.

However, we may want to accelerate the computations of the solve by performing them in mixed precision instead of double precision. This is what we do in the next two solve variants. When using 7 precisions for storage, we successfully accelerated the computations by adding mixed-precision computations, as can be seen in column sto7*+calc of both tables: the time for forward elimination has been reduced by 20%, and the time for backward elimination by 21%. However, both steps tend to be still slower than doing everything in double precision (column BLR).

If we allow to slightly compromise the storage reductions by using a set of 2 precisions instead of 7, then we obtain rather impressive time gains. The time for forward elimination has been reduced by 56% and the time for backward by 56% when doing computations in mixed precision (sto2*+calc compared to sto2*). Indeed, we do not need to convert and copy the MPBLR matrix from one format to another using this variant: the same data structure is kept for both storage and computation, which proves to be more efficient. In terms of time performance, we even outperform the double-precision variant.

Finally, we analyze the multiple RHS case ($n_{\text{rhs}} \gg 1$). Tables 3.5 and 3.6 show that the gain achieved with mixed precision is not as good as with a single RHS. In certain cases, it may even be slightly slower than double precision, especially for the backward elimination. The most likely explanation is that in the multiple RHS case, the data movements associated with the RHS are much more costly and become the bottleneck of the BLR triangular solve. Thus, reducing the size of the LU factors does not have an effect as important as in the single RHS case. In fact, this observation is even true when comparing the performance of double precision BLR solve with that of the FR solve: since the RHS is not compressed, the effect of BLR compression on the time performance is also much less significant with multiple RHS. For example, on matrix Queen_4147 the double precision BLR solve is 1.6× faster than the FR one with $n_{\text{rhs}} = 1$, whereas it is only 1.2× faster with $n_{\text{rhs}} = 250$.

matrix	n_{rhs}	time for forward elimination (s)					
		FR	BLR	sto2*	sto7*	sto2*+calc	sto7*+calc
Queen_4147	1	0.41	0.25	0.50	0.69	0.22	0.55
($\varepsilon = 10^{-10}$)	30	1.40	0.67	0.94	1.2	0.65	1.04
	250	5.40	4.37	4.95	5.3	4.47	5.05

Table 3.5: Times for forward elimination for the following variants:

- FR: full-rank
- BLR: in double precision
- sto2* : using MPBLR with 2 precisions for storage and double precision for computations in solve
- sto7* : using MPBLR with 7 precisions for storage and double precision for computations
- sto2*+calc : using MPBLR with 2 precisions for storage and for computations
- sto7*+calc : using MPBLR with 7 precisions for storage and MPBLR with 2 precisions for computations

The admissibility condition is chosen so that the storage would be minimal.

matrix	n_{rhs}	time for backward elimination (s)					
		FR	BLR	sto2*	sto7*	sto2*+calc	sto7*+calc
Queen_4147	1	0.71	0.46	0.93	1.26	0.41	0.99
($\varepsilon = 10^{-10}$)	30	1.22	0.91	1.42	1.77	0.99	1.59
	250	5.24	5.45	6.38	6.82	7.09	7.73

Table 3.6: Times for backward elimination for the same variants as in Table 3.5

In the next chapter, this observation motivates us to rethink the BLR solve algorithm in the context of many right-hand sides.

Block-admissibility condition

In most previous experiments presented earlier in this chapter, we have used the mixed-precision admissibility condition when handling mixed precision, be it for storage or computations (with the exception of Table 3.2). In fact, this condition somewhat optimizes the cost for storing data using a given set of precision formats, while introducing a certain overhead.

We analyze the complexity of a LR×FR product involving a block of the RHS. The matrix sizes involved are $m \times r$ and $r \times n$ for the low-rank parts, and $n \times n_{\text{rhs}}$ for the RHS block. Around $2(m+n)r_i n_{\text{rhs}}$ operations are performed in precision number i. Having defined c'_i the time cost for performing an operation in precision format i (compared to doing it in precision 1), and $\tilde{r}' = \sum_{i=1}^{p} c'_i r_i$, the weighted number of flops needed for the LR×FR product in mixed precision are $\tilde{f}' = 2(m+n)\tilde{r}' n_{\text{rhs}}$. After comparing it to

matrix	time for forward elimination (s)		
	MPBLR(calc)	MPBLR(calc)*	BLR
Queen_4147 ($\varepsilon = 10^{-10}$)	0.229	0.217	0.248

Table 3.7: Impact of the mixed-precision block-admissibility condition on the time spent in the forward solve with 1 RHS. MPBLR(calc)* uses the block-admissibility condition suited to mixed precision, whereas MPBLR(calc) does not.

matrix	time for backward elimination (s)		
	MPBLR(calc)	MPBLR(calc)*	BLR
Queen_4147 ($\varepsilon = 10^{-10}$)	0.336	0.408	0.461

Table 3.8: Impact of the mixed-precision block-admissibility condition on the time spent in the backward solve with 1 RHS. MPBLR(calc)* uses the block-admissibility condition suited to mixed precision, whereas MPBLR(calc) does not.

the complexity of the FR×FR operation, we reach the condition for a block to be worth storing as low-rank in terms of time complexity:

$$(m+n)\tilde{r}' < mn \tag{3.3}$$

This condition is very similar to the admissibility condition for mixed-precision storage (Equation 3.2), except for the fact that the coefficients c'_i refer to the gain in speed from format u_i compared to u_1, instead of the gain in storage. However, it is a rather reasonable approximation to consider that $\forall i, c_i = c'_i$, i.e., the time spent in a computation using a certain precision format is proportional to the space needed to store it. In particular, single precision is supposed to be twice as fast as double precision in our case, which means that $c'_2 = 0.5c'_1$ and both mixed-precision admissibility conditions are identical: there is no need to choose.

We now want to assess whether or not these theoretical time complexities are close to the reality. More precisely, we want to confirm whether or not the resulting choice of a mixed-precision block-admissibility condition is the right one in practice. Therefore, we compare the variants MPBLR(calc) (uniform-precision admissibility) and MPBLR(calc)* (mixed-precision admissibility) in Tables 3.7 and 3.8, using a single RHS.

For the forward solve, it is faster to use the admissibility for mixed precision: we obtain a time reduction of 5%. This does not seem to be the case for the backward solve (yet), and further work may be needed.

3.2.3 Towards an application to the factorization

After having successfully accelerated the BLR triangular solution by performing computations in mixed precision, we now want to do so for the factorization as well. We will see how to adapt the algorithms presented in chapter 2 to the implementation of the factorization of a frontal matrix. In particular, we will reuse and generalize the techniques

presented in section 3.2.1.

Given an algorithm performing the factorization of a frontal matrix, we want to modify a certain number of its kernels by switching them to mixed precision. Those kernels are rather independent from each other, so it is possible to only switch some of them when modifying a uniform-precision implementation. We can therefore obtain a series of implementations in which mixed precision has been added gradually. One advantage of this method is that it allows for a progressive implementation, in which we can check that everything works as intended after the modification of each kernel. Another advantage is that we can make sure that switching each of those kernels introduces some time gains, and quantify them. Indeed, there is no guarantee that all kernels are worth switching to mixed precision, especially for the most complex ones such as LR×LR, which has the lowest granularity. In the worst-case scenario, it might be beneficial to use an implementation in which some kernels are kept in uniform precision, if it proves to be faster in some cases.

Factorization kernels

We may add mixed precision in the following kernels, independently:

- The LR×FR and FR×LR products
- The Solve step (so-called TRSM): as argued in chapter 2, the step $X \leftarrow L^{-1}X$ may be switched to mixed precision: if $X = [X_1 \cdots X_p]$, we perform each operation $X_i \leftarrow L^{-1}X_i$ in precision i.
- The LR×LR products

When performing a product between a LR block and a FR block, we could reuse the algorithms presented in section 3.2.1. However, we may also want the same kernel to be able to perform a FR×LR product since it is very similar. Therefore, we are inclined to describe a more generic approach, where we want to perform an operation $A \times B$, where only one matrix among A and B is low-rank. We decompose this kernel into several steps:

- Do the intermediary product in order to get a mixed-precision low-rank approximation $Z \times W$ of the result, via the function *computeW*. One of these two matrices is the result of the product between the full-rank block and the closest low-rank part of the other block.
- Compute the outer-product $Z \times W$.

Due to implementation details chosen for MUMPS, we sometimes have to deal with the transpose of a block instead of the original block. Therefore, while describing a generic implementation of a mixed-precision product between a LR block and a FR block, we might as well allow any of the two blocks to be transposed.

As for the LR×LR kernel, its implementation details may vary. For example, in the so-called inner product we may compute $W_{ij} = Y_{Ai}^T X_{Bj}$ in precision $max(i, j)$.

In all cases, we need additional conversions before performing LR×LR products on a block : the part X_i should be copied in each lower precision format $k > i$.

Similarly to the case of a LR×FR product, we notice that a LR×LR product may be split into two steps: (i) a function *computeW*, which includes the inner-product and the middle-product, returns a low-rank approximation of the result; and then (ii) an outer-product, which is similar in every way to the one performed in the LR×FR kernel.

Right-looking and left-looking factorizations

The factorization may be done in right-looking. In the implementation of MUMPS, this particular case happens when a front is split between several MPI processes: as explained in section 3.1.7, a worker process will perform the updates of its blocks in right-looking, treating a block-row as soon as it is received from the master process.

We may use some of the kernels in mixed precision as explained above. Note that doing so requires the low-rank blocks to have been copied in mixed precision beforehand. If the LR×FR and FR×LR products are performed in mixed precision, then the FR blocks should also be copied in all other precision formats as well. If the TRSM step is performed in mixed precision, then the blocks L_{kk} and U_{kk} of the factors should be copied to all precision formats. In case LR×LR products are performed, additional conversions have to be performed: X_i and Y_i should be copied to all precision formats lower than i.

It seems worthwhile to perform all these copies and conversions beforehand: indeed, each block will be reused many times.

The same can be applied to a left-looking factorization, but there are a few additional possible optimizations:

- When computing a sum of outer-products in mixed precision, we may once again reuse Algorithm 3.4. This way, all contributions in precision k will be summed together, thus reducing the number of conversions. This is very similar to what was done in the case of the solution phase, except for the fact that we now add the contributions of LR×FR, FR×LR and LR×LR products at once.

- A very promising perspective would be to adapt LUA (low-rank updates accumulation) to the use of mixed precision. As explained in section 1.2.3, it is possible to consider a sum of outer-products $\sum_j Z^{(j)} W^{(j)}$ as a product of two larger matrices $[Z^{(1)} \cdots Z^{(q)}] \times [W^{(1)}; \cdots; W^{(q)}]$. Although the number of operations performed is left unchanged, the granularity of the matrix operations is improved by doing this, resulting in time reductions. We could adapt the principle behind Algorithm 3.4 in order to compute all the contributions in precision k as one larger matrix product, performed in precision k: $\sum_j Z_k^{(j)} W_k^{(j)}$ would be replaced by $[Z_k^{(1)} \cdots Z_k^{(q)}] \times [W_k^{(1)}; \cdots; W_k^{(q)}]$.

By doing LUA in mixed precision, we would improve the very small granularity induced by the use of mixed precision. In particular, the ranks of the parts in precision u_1 are often small. Some may be empty and are skipped, but there also are many such blocks having a rank r_1 of 1 or 2: the time performance of such matrix products is rather poor. By performing low-rank updates accumulations on each precision format, this aspect should be improved.

Yet another perspective would be to compute the inner-products using LUA as well. Indeed, this kernel is the one whose granularity was the most reduced by the use mixed precision, each matrix product having been split into 4.

Dissociating storage and computation formats

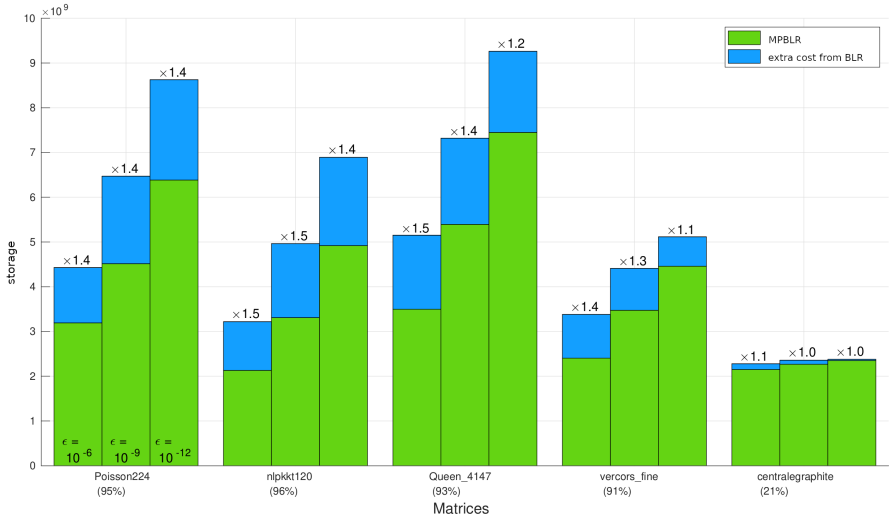
Contrary to the case of the solution phase, combining mixed precision for computations and storage is rather easy, and its overhead should be negligible. We just need to convert the MPBLR(calc) format to an MPBLR(sto) once at the end of the factorization of a frontal matrix. If the precision formats used for computations and for storage are identical, there is no need to perform actual data copies: we can once again reuse the same data structure.

3.3 Conclusion

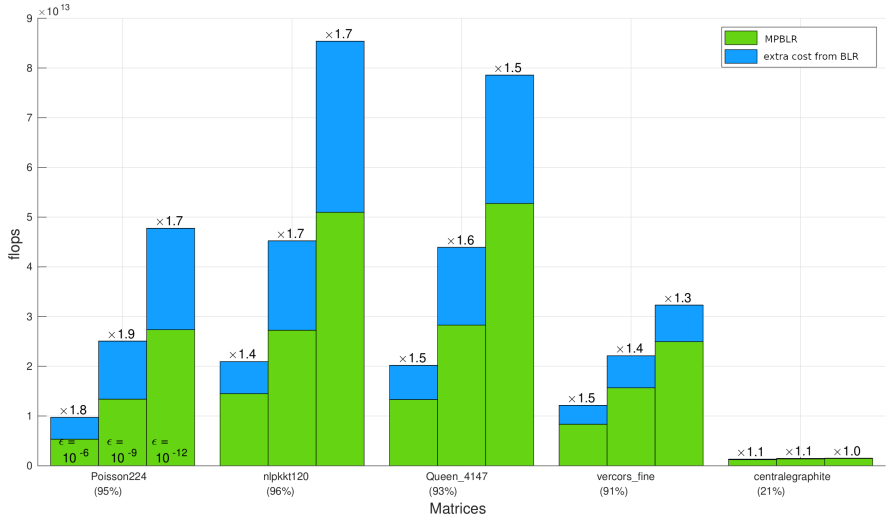
In this chapter, we adapted the multifrontal method in order to benefit from the theoretical performance gains from chapter 2. We first considered the BLR compression in mixed precision (MPBLR) as a storage format only, in order to reduce the size of the factors and the memory peak of the factorization. We succeeded in obtaining reductions of the LU factor size up to 38%, without impacting too much the factorization time nor the error of the solution. We also obtained reductions in the volume of communications up to 50%, which can be a critical aspect when solving very large linear systems on massively parallel architectures.

We also described how to adapt the different steps of the multifrontal method in order to obtain speedups from the use of mixed-precision computations. We implemented the modifications of the triangular solve step. We obtained time reductions up to 12% compared to the standard BLR algorithm, and 20% compared to the mixed-precision version aiming for storage gains only, while keeping the same storage gains as the latter.

In future work we will turn to the implementation of computations in mixed precision during the BLR factorization phase of the multifrontal method, as described in section 3.2, and to obtain actual time gains. A preliminary study of the number of flops eligible for mixed precision seems to indicate that its potential for acceleration is higher than for storage. For example, as illustrated in figure 3.5, the potential factor of acceleration from the use of mixed precision is 1.9× for matrix Poisson224 with $\varepsilon = 10^{-9}$, whereas the LU storage reduction is only 1.4×.



(a) Size needed for storing the LU factors.



(b) Expected time for the factorization. Similarly to section 2.5, such a value is based on the weighted flops, i.e. take the hypothesis that flops in single precision are twice as fast as flops in double precision.

Figure 3.5: Estimations of the size of the LU factors and the time spent in the multifrontal factorization, for several matrices obtained from industrial applications. The green bars correspond to the complexity of the MPBLR variant using 2 precision formats (fp64+fp32), while the blue bars correspond to the complexity added by not using mixed precision. Each group of 3 bars corresponds to values of ε of 10^{-6}, 10^{-9} and 10^{-12} respectively. The percentage below the matrix name indicates which percentage of the entries belong to BLR fronts: the higher the value, the more efficient BLR compression is on this matrix. More importantly, the number above each bar (e.g. $\times 1.8$) indicates the factor of gain expected for the mixed-precision variant, compared to using uniform precision.

Chapter 4

Hybrid algorithm for solve

4.1 Introduction

This chapter is concerned with the performance of the triangular solve phase, which is critical in several contexts. Indeed, the BLR factorization is often used as a preconditioner for iterative solvers (Amestoy et al., 2023c; Higham and Mary, 2019), which require several iterations and thus solves. Moreover, even in a pure direct solver context, some of the real-life applications where BLR solvers have been the most successful are in the field of geosciences (Operto et al., 2023; Amestoy et al., 2016; Shantsev et al., 2017; Mary, 2017), where we need to solve a system

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

with many right-hand sides (RHS): $\mathcal{B} \in \mathbb{R}^{n \times n_{\text{rhs}}}$ is a (possibly sparse) matrix with n_{rhs} columns, where n_{rhs} is typically in the thousands or tens of thousands. In this case, the triangular solves $\mathcal{L}\mathcal{Y} = \mathcal{B}$ and $\mathcal{U}\mathcal{X} = \mathcal{Y}$ are the bottleneck of the computation.

This work started from the observation, made in section 3.2.2, that the time gains from using computations in mixed precision is underwhelming in the case of multiple RHS (see Table 3.5). While trying to understand what we initially thought to be a performance issue specific to mixed precision, we realized that a something similar arises in uniform precision. This is illustrated in Table 4.1, which provides the time for the forward solve $\mathcal{L}\mathcal{Y} = \mathcal{B}$ both for the BLR and FR solvers (i.e. with and without compression), depending on the number of RHS, for a few problems of interest. For a single RHS ($n_{\text{rhs}} = 1$), the BLR compression reduces the solve time by significant factors for all problems. However, with multiple RHS ($n_{\text{rhs}} = 250$), the speedup achieved thanks to BLR compression becomes much smaller for all problems (for example, for the Poisson120 problem, the 3.9× speedup with $n_{\text{rhs}} = 1$ becomes only a 1.7× speedup with $n_{\text{rhs}} = 250$).

Therefore, this motivated us to rethink the BLR solve algorithms. The key observation is that with BLR compression, the performance of the triangular solve is memory bound, even for multiple RHS. Therefore, the performance of the BLR solve is mainly determined by its communication costs. Crucially, while BLR compression reduces the size of the LU factors and therefore the cost of accessing them, it does not reduce the size of the RHS,

Table 4.1: Some motivating examples: forward solve time (s) for the so-called full-rank (FR, with no compression) and BLR solvers on 2×18 cores.

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

which are uncompressed. The consequence is that when there are many RHS, the cost of accessing them is likely to dominate the cost of accessing the LU factors, thereby reducing (or even canceling) the performance benefits of the BLR compression.

The main contribution of this chapter is to overcome this limitation by proposing new hybrid algorithms that reduce the number of accesses to the RHS and therefore the total volume of communications. We did not consider the use of mixed precision in this chapter, and all our experiments are therefore performed in uniform precision.

The rest of this chapter is organized as follows. After recalling some preliminaries on the existing BLR solve variants in section 4.2, we describe in section 4.3 several novel variants of the BLR solve that reduce its communication costs. We confirm that these new variants are indeed communication-avoiding by performing a theoretical communication volume analysis in section 4.4. To quickly analyze the performance of these new variants and assess their potential, we first develop a simplified prototype code and present our results on synthetic data in section 4.5. Based on these results, we implement a selected subset of the most promising variants in the MUMPS solver and test their performance on a range of real-life applications in section 4.6.

Throughout the chapter, we will discuss the case of the forward solve $\mathcal{LY} = \mathcal{B}$ without loss of generality. The algorithms and ideas proposed in this chapter also apply to the backward solve $\mathcal{UX} = \mathcal{Y}$. For clarity of notation, we rename the forward solve $\mathcal{LX} = \mathcal{B}$ hereinafter. In our experiments with the MUMPS solver, we will measure the time spent in the computations for the forward solve. The entire triangular solution phase of the solver also consists of the backward solve and some additional non-computational parts (data copies, etc.), whose cost represents a fixed overhead that is independent of the algorithm variants considered in this chapter.

4.2 Preliminaries and notations

4.2.1 Notations

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

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

We summarize below the notations used for a given front L and its corresponding part of the solution X, which are illustrated in Figure 4.1. Note that these notations differ a bit from those used in chapter 3.

- $\mathcal{A}$, $\mathcal{X}$, $\mathcal{B}$, $\mathcal{L}, \mathcal{U}$: the matrix, solution, RHS, and LU factors associated with the global sparse problem;
- A, X, B, L, U, the matrix, solution, RHS, and LU factors associated with a given frontal matrix;
- m, the number of rows of L and X;
- n, the number of columns of L;
- n_{rhs}, the number of columns of X (the number of right-hand sides);
- b, the block size;
- r, the rank of the low-rank blocks;
- $L_{ij} \in \mathbb{R}^{b\times b}$, the (i,j)th block of L, and $U_{ij} V_{ij}^T$, its low-rank representation;

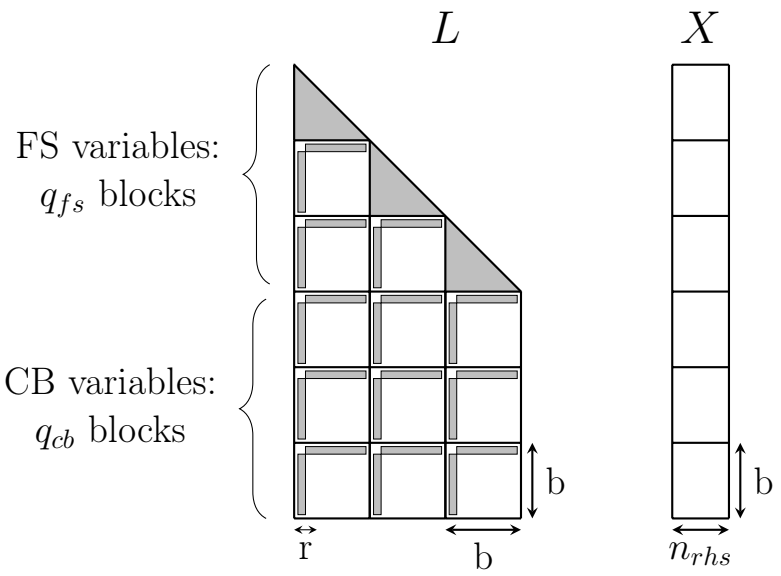


Figure 4.1: A frontal BLR matrix L and its right-hand side X. For convenience we repeat this figure, which was already present in section 1.2.3.

- $X_i \in \mathbb{R}^{b \times n_{\text{rhs}}}$, the ith block of X;
- $q_{\text{fs}} = n/b$: the number of block rows in the FS part, also the number of block columns;
- $q_{\text{cb}} = (m-n)/b$: the number of block rows in the CB part;
- $q = q_{\text{fs}} + q_{\text{cb}} = m/b$, the total number of block rows.
- We also define $q = q_{\text{fs}}(q_{\text{fs}}-1)/2 + q_{\text{fs}}q_{\text{cb}}$, the total number of off-diagonal blocks in L;
- $FS = [1\colon q_{\text{fs}}]$, the set of block indices for FS variables; and
- $CB = [q_{\text{fs}}+1\colon q]$, the set of block indices for CB variables.

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

$$X_i \leftarrow L_{ii}^{-1}(X_i - \sum_{j<i} L_{ij}X_j) \tag{4.1}$$

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

$$X_i \leftarrow X_i - \sum_{j \le q_{\text{fs}}} L_{ij}X_j \tag{4.2}$$

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

Computations (4.1) and (4.2) thus involve two types of tasks:

- update(i,j) for $i \in FS \cup CB$ and $j \in FS$ verifying $i > j$: $X_i \leftarrow X_i - U_{ij}(V_{ij}^T X_j)$, which can be performed only after trsm(j) has been completed; and

- trsm(i) for $i \in FS$: $X_i \leftarrow L_{ii}^{-1} X_i$, which can be performed only after update($i, 1$), ..., update($i, i-1$) have all been completed.

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

4.2.2 Right-looking and left-looking variants

Algorithm 4.1 describes the right-looking (RL) variant, which performs the update(i, j) tasks as soon as they are ready to be performed (eager approach): as soon as trsm(j) has been completed, all the update(i, j) for $i > j$ are immediately performed, as illustrated in Figure 4.2.

Algorithm 4.1 Right-looking variant.

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

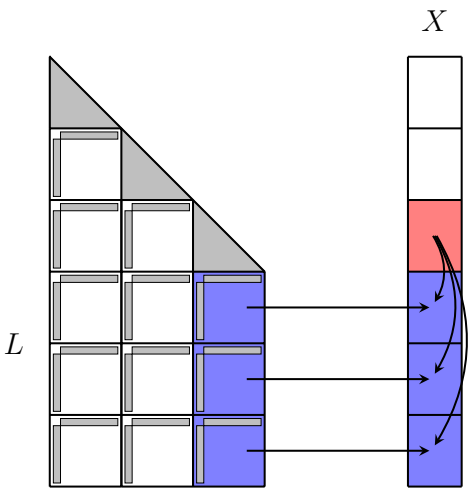


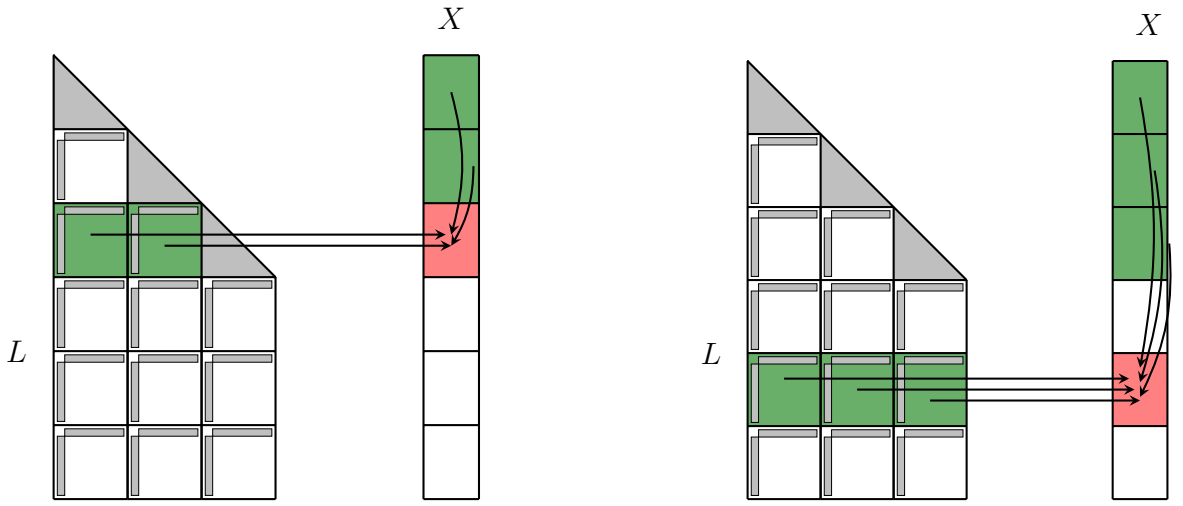
Figure 4.2: Step $j = 3$ of Algorithm 4.1 (right-looking variant).

Conversely, Algorithm 4.2 describes the left-looking (LL) variant, which performs the update(i, j) tasks as late as possible (lazy approach): for a given i, the update(i, j) are delayed until they are all ready to be performed together, as illustrated in Figure 4.3.

The RL and LL variants perform the same computations in two different orders. In a sequential context, it is not clear whether one variant can be expected to yield better performance than the other. However, in a parallel context, a major difference appears. The RL variant can be parallelized efficiently by executing the loop on block rows (line 3 of Algorithm 4.1) in parallel, since all update(i, j) tasks are independent for a fixed j. This approach is usually efficient because it does not lead to any conflict and the size

Algorithm 4.2 Left-looking variant.

```
1: for i ∈ FS do
2:    for j < i do
3:       X_i ← X_i − U_ij(V_ij^T X_j)
4:    end for
5:    X_i ← L_ii^{-1} X_i
6: end for
7: for i ∈ CB do
8:    for j ∈ FS do
9:       X_i ← X_i − U_ij(V_ij^T X_j)
10:   end for
11: end for
```



(a) Step $i = 3$ of Algorithm 4.2: Update of an FS block.

(b) Step $i = 5$ of Algorithm 4.2: Update of a CB block.

Figure 4.3: Left-looking variant.

of the loop, $q - j$, is large enough to expose a high amount of concurrency. In contrast, the parallelization of the LL variant is more difficult. The CB part of the update can be efficiently parallelized by executing the loop on the block-rows (line 7 of Algorithm 4.2) in parallel, since the updates for different block-rows are independent and the size of the loop, q_{cb}, is large enough. However, for the FS part of the computation, the only loop that can be executed in parallel is the loop on the block columns (line 2 of Algorithm 4.2), which presents two difficulties. First, it requires a reduction operation to avoid conflicts since all update(i, j) tasks for a fixed i modify the same block X_i. Second, the loop is only of size $i - 1$, with $i \leq q_{\text{fs}}$, so that there is very little concurrency in the first steps of the loop, and even for the later steps because q_{fs} is typically much smaller than q_{cb}.

4.2.3 Parallelism in multifrontal solve

In the multifrontal solution, we must carry out several partial solves with frontal matrices following a bottom-up traversal of a tree. As a result, two types of parallelism can be exploited, as mentioned in section 1.1.3.

- Node parallelism consists in processing a given front in parallel, by parallelizing the partial solve as described above for the RL and LL variants. The amount of work required by one partial solve is usually sufficient to be efficiently parallelized only for the largest fronts, which are at the top of the tree.

- Tree parallelism consists in processing multiple fronts on different branches concurrently, using only one process per front. This allows for efficiently parallelizing the bottom of the tree, which consists of many independent fronts of small size.

Since node and tree parallelism are more efficient for the top and bottom of the tree, respectively, the best approach is to combine both types of parallelism. One possibility to do so is to exploit tree parallelism for the bottom layers of the tree, and switch to node parallelism after a given layer (so-called the "$\mathcal{L}_0$" layer) is reached. This $\mathcal{L}_0$ approach is implemented in MUMPS (L'Excellent and Sid-Lakhdar, 2014).

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

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

4.3 New hybrid variants of the BLR triangular solve

4.3.1 A novel hybrid variant

As mentioned in section 4.1, the performance of the BLR solve is underwhelming when dealing with many RHS. Based on the description of the RL and LL variants, we can see that one common weakness of both variants is that they require multiple accesses to the entire RHS. Indeed, at each step the RL variant only reads one block of the RHS, but needs to write all the bottom part of the RHS. Conversely, at each step the LL variant only writes one block, but needs to read all the top part of the RHS. We will precisely measure the volume of communications in the next section, but it is clear that when the number of RHS n_{rhs} is large, the accesses to the RHS can represent a much larger volume than the accesses to the BLR factors, and thus constitute the bottleneck of the computation.

Motivated by this observation, we propose in this section a novel BLR solve variant that is based on a hybrid scheme that only needs to read and write one block of the RHS per step. The main idea is to perform the read operations following a right-looking

scheme, and the write operations following a left-looking scheme. To do so, we divide the update(i, j) task $X_i \leftarrow X_i - U_{ij}(V_{ij}^T X_j)$ into two separate subtasks:

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

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

This new hybrid variant is outlined in Algorithm 4.3 and illustrated in Figure 4.4.

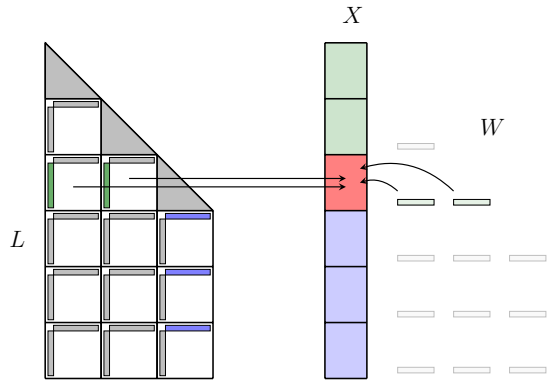
Algorithm 4.3 Hybrid variant.

1: **for** $k \in FS$ **do**
2: **for** $j < k$ **do**
3: $X_k \leftarrow X_k - U_{kj} W_{kj}$
4: **end for**
5: $X_k \leftarrow L_{kk}^{-1} X_k$
6: **for** $i > k$ **do**
7: $W_{ik} = V_{ik}^T X_k$
8: **end for**
9: **end for**
10: **for** $k \in CB$ **do**
11: **for** $j \in FS$ **do**
12: $X_k \leftarrow X_k - U_{kj} W_{kj}$
13: **end for**
14: **end for**

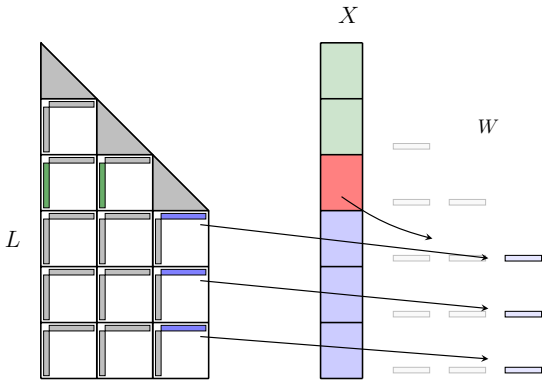
4.3.2 Parallelism-driven hybrid variant

One issue with the hybrid variant of Algorithm 4.3 is that it suffers from the same problems as the left-looking variant in a parallel setting: the updateU tasks corresponding to FS blocks of the RHS are performed following a left-looking pattern (loop on line 2 of Algorithm 4.3), which is of small size and requires a reduction.

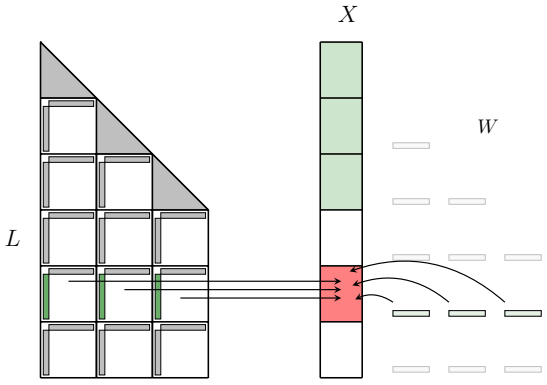
To overcome this issue, we propose in Algorithm 4.4 a modified hybrid variant that is more amenable to a parallel execution. The idea is to use the hybrid scheme for the CB part of the computation only, which can be efficiently parallelized with a parallel loop on the block rows (line 11 of Algorithm 4.4), and to keep the efficiently parallelizable right-looking scheme for the FS part (with a parallel loop on line 3). Since the FS part is typically much smaller than the CB part, we can still expect to retain most of the benefits associated with the use of the hybrid scheme.



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



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



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

Figure 4.4: Hybrid variant.

Algorithm 4.4 Parallelism-driven hybrid variant.

```
1:  for k ∈ FS do
2:     X_k ← L_kk^{-1} X_k
3:     for i > k do
4:        if i ∈ FS then
5:           X_i ← X_i − U_ik (V_ik^T X_k)
6:        else
7:           W_ik = V_ik^T X_k
8:        end if
9:     end for
10: end for
11: for k ∈ CB do
12:    for j ∈ FS do
13:       X_k ← X_k − U_kj W_kj
14:    end for
15: end for
```

4.3.3 Low-rank updates accumulation

The hybrid variant described above minimizes the number of accesses to the RHS in order to reduce the volume of communications. It also increases arithmetic intensity of

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

As explained in section 1.2.3, LUA can be used to improve the performance of the BLR factorization. We now discuss how to adapt this LUA technique to the BLR solve. The situation is more complicated due to the presence of the RHS, which is not in low-rank form. In fact, we will see that LUA cannot be fully used in either the RL or LL variants of the BLR solve. The hybrid variant allows us to take full advantage of LUA, as we now explain.

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

$$X_k \leftarrow X_k - \sum_{j=1}^{J} U_{kj}W_{kj} = X_k - \left[U_{k1} \cdots U_{kJ}\right] \left[W_{k1}^T \cdots W_{kJ}^T\right]^T, \tag{4.3}$$

with $J = k - 1$ for the FS updates and $J = q_{\text{fs}}$ for the CB updates. The sum in (4.3) can be efficiently evaluated using only one matrix–matrix product, instead of J smaller ones. In order to do so, all U_{kj}, $j \leq J$, must be allocated contiguously in memory, as well as all W_{kj}, $j \leq J$.

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

$$\left[W_{i_0k}^T \cdots W_{qk}^T\right]^T = \left[V_{i_0k} \cdots V_{qk}\right]^T X_k \tag{4.4}$$

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

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

4.4 Communication volume analysis

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

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

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

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

4.4.1 Analysis

Right-looking variant At each step $j \in FS$, the RL variant (Algorithm 4.1) requires the following transfers:

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

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

- $2qrb + q_{\text{fs}}b^2$ RO transfers,

- $qbn_{\text{rhs}} + q_{\text{fs}}bn_{\text{rhs}}$ RW transfers,

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

Left-looking variant At each step $i \in FS$, the LL variant (Algorithm 4.2) requires the following transfers:

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

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

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

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

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

Hybrid variant At each step $k \in FS$, the hybrid variant (Algorithm 4.3) requires the following transfers:

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

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

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

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

- $2qrb + qrn_{\text{rhs}} + q_{\text{fs}}b^2$ RO transfers,
- qrn_{rhs} WO transfers,
- qbn_{rhs} RW transfers.

Parallelism-driven hybrid variant Finally, we compute the communication volume of the parallelism-driven hybrid variant (Algorithm 4.4), which is obtained by combining the RL volume for the FS part of the computation with the hybrid volume for the CB part. At each step $k \in FS$, the parallelism-driven hybrid variant requires the following transfers:

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

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

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

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

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

4.4.2 Discussion

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

We now seek to determine when the hybrid variant requires less communications than the LL one. To do so, we must make an assumption on the relative cost of RO, WO, and RW transfers. Under the simplifying assumption that RO and WO transfers are equally

costly, and neglecting lower order terms in the expression of the communication volume, we obtain a ratio between the LL volume and the hybrid volume approximately equal to

$$\frac{2qrb + 2qrn_{\text{rhs}}}{2qrb + qbn_{\text{rhs}}} = \frac{1 + n_{\text{rhs}}/b}{1 + n_{\text{rhs}}/2r}. \tag{4.5}$$

Thus the condition for the hybrid variant to require less communications than the LL one is $2r \leq b$, which we can expect to be always satisfied, since it corresponds to the condition for a block to be low-rank (if $r > b/2$, the block requires less storage if represented as a full-rank block: see Equation 2.26). We can thus conclude that the hybrid variant should always communicate less than the LL one.

A more important question is when can we expect the hybrid variant to communicate *much* less than the LL one: that is, when is ratio (4.5) much less than 1? In the regime where $n_{\text{rhs}}/2r$ is small (just one or few RHS), the ratio (4.5) is close to 1. The hybrid variant therefore does not significantly reduce the volume of communications for small numbers of RHS. However, in the regime where $n_{\text{rhs}}/2r \gg 1$, the ratio (4.5) is approximately equal to $2r/n_{\text{rhs}} + 2r/b$. This shows that (4.5) is much less than 1 if $\min(n_{\text{rhs}}, b) \gg 2r$: that is, the hybrid variant could lead to significant gains in case there is a large number of RHS and the blocks are very low-rank.

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

We summarize in Table 4.2 the dominant terms in the total communication volume of the different variants.

Table 4.2: Summary of the communication volume analysis: dominant terms for each variant.

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

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

4.5 Performance analysis based on a simplified prototype

4.5.1 Experimental setting

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

The prototype implements the four BLR solve variants: RL, LL, hybrid, and parallelism-driven hybrid. The prototype also allows for the optional use of LUA for the updateU and/or updateV tasks. We observe the actual time spent in the computation, which allow us to compare different variants.

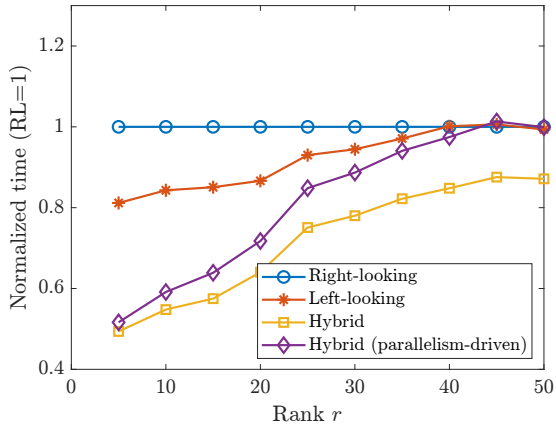
As explained in section 4.2.3, the frontal BLR solve algorithms are needed for large fronts at the top of the tree, where node and tree parallelism are both exploited, but also for smaller fronts at the bottom of tree, where only tree parallelism is exploited. Therefore, our goal is to use the prototype to analyze the performance of the BLR solve in two types of configurations: the first configuration uses tree parallelism only by running 36 instances in parallel (with MPI), each using a single thread; the second configuration uses both node and tree parallelism by running two instances in parallel (with MPI), each of them being parallelized with 18 OpenMP threads.

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

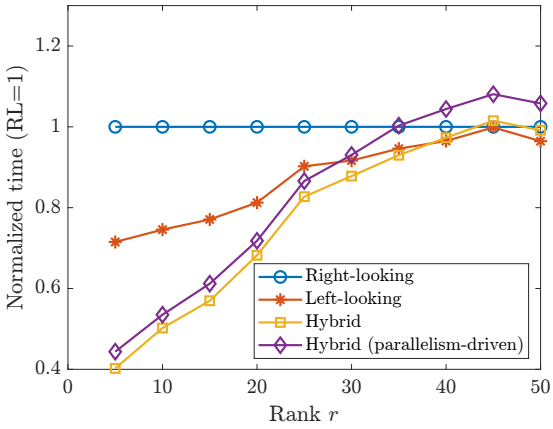
4.5.2 Performance analysis of hybrid variants

We report our performance results in Figures 4.5 and 4.6 for the tree and node parallelism configurations, respectively. For all experiments we set $n_{\text{rhs}} = 250$ and $q_{\text{fs}} = q/5$. For the node+tree parallelism experiments, we use large fronts (BLR block size $b = 500$ and $q = 100$ or 200, leading to a front size of $50,000$ or $100,000$). For the tree parallelism experiments, we use smaller fronts (BLR block size $b = 250$ and $q = 30$ or 50, leading to a front size of $7,500$ or $12,500$).

These results show that, as could be hoped, the hybrid variants can be faster than the RL and LL variants in many cases. The gains are the most significant when the rank r is small, and when the problem size is large, with speedups reaching a factor 2.5× in the best case. The parallelism-driven hybrid variant is not as fast as the standard hybrid one in the configuration with tree parallelism only, but is significantly superior, as expected, in the configuration with node parallelism. Indeed, as explained in section 4.3.2, the left-looking loop at line 2 of Algorithm 4.3 involves a reduction operation on a loop of small

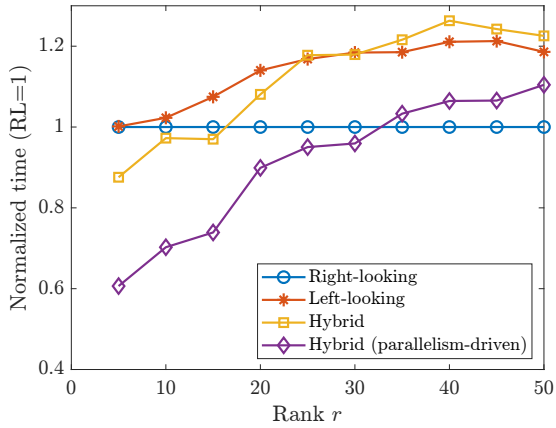


(a) $q = 30$

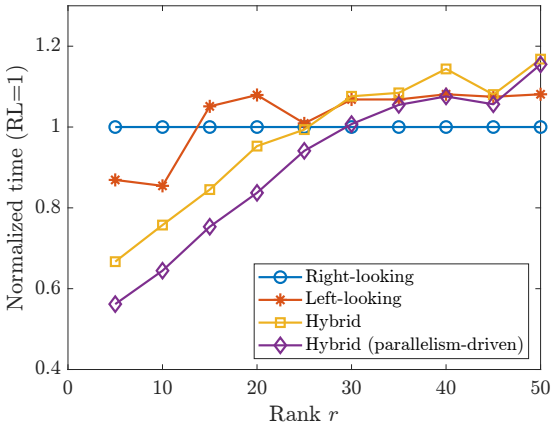


(b) $q = 50$

Figure 4.5: Time spent in solve with tree parallelism (36 instances with 1 thread per instance), $b = 250$, $n_{\text{rhs}} = 250$, and $q_{\text{fs}} = q/5$.



(a) $q = 100$



(b) $q = 200$

Figure 4.6: Time spent in solve with node+tree parallelism (2 instances with 18 threads per instance), $b = 500$, $n_{\text{rhs}} = 250$, and $q_{\text{fs}} = q/5$.

size, whose parallelization on 18 threads is not very efficient. This is also confirmed in Figure 4.6 by the observation that for larger values of q the performance of left-looking, hybrid and parallelism-driven hybrid variants gets closer.

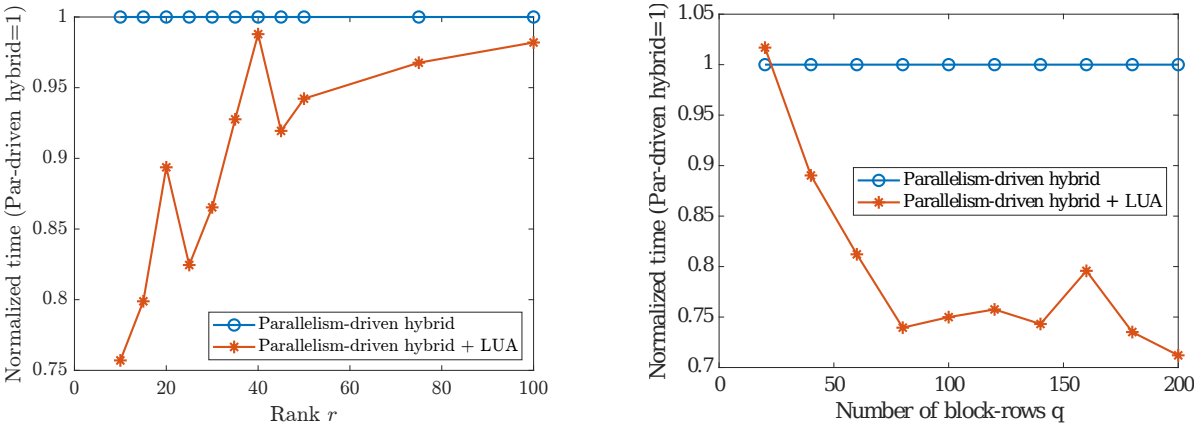
4.5.3 Performance analysis of LUA

We finally analyze the time gains obtained when combining the hybrid variant with the LUA approach described in section 4.3.3. We report the performance with and without LUA in Figure 4.7. We use a node+tree parallelism configuration and therefore take the parallelism-driven hybrid variant as baseline. The figure consists of two plots, one where the number of block-rows q is fixed and the rank r varies, and the other where r is fixed and q varies. These two plots illustrate two opposite trends:

- As r increases, the benefits of using LUA diminish, since the granularity of the low-rank updates without accumulation is already large enough to achieve good performance.

- As q increases, the benefits of using LUA increase, since there are more blocks to accumulate together, which leads to a better granularity.

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



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

Figure 4.7: Time gains obtained with LUA, with node+tree parallelism (2 instances with 18 threads per instance), $b = 500$, $n_{\text{rhs}} = 250$, and $q_{\text{fs}} = q/5$.

4.6 Results on real-life applications with the MUMPS solver

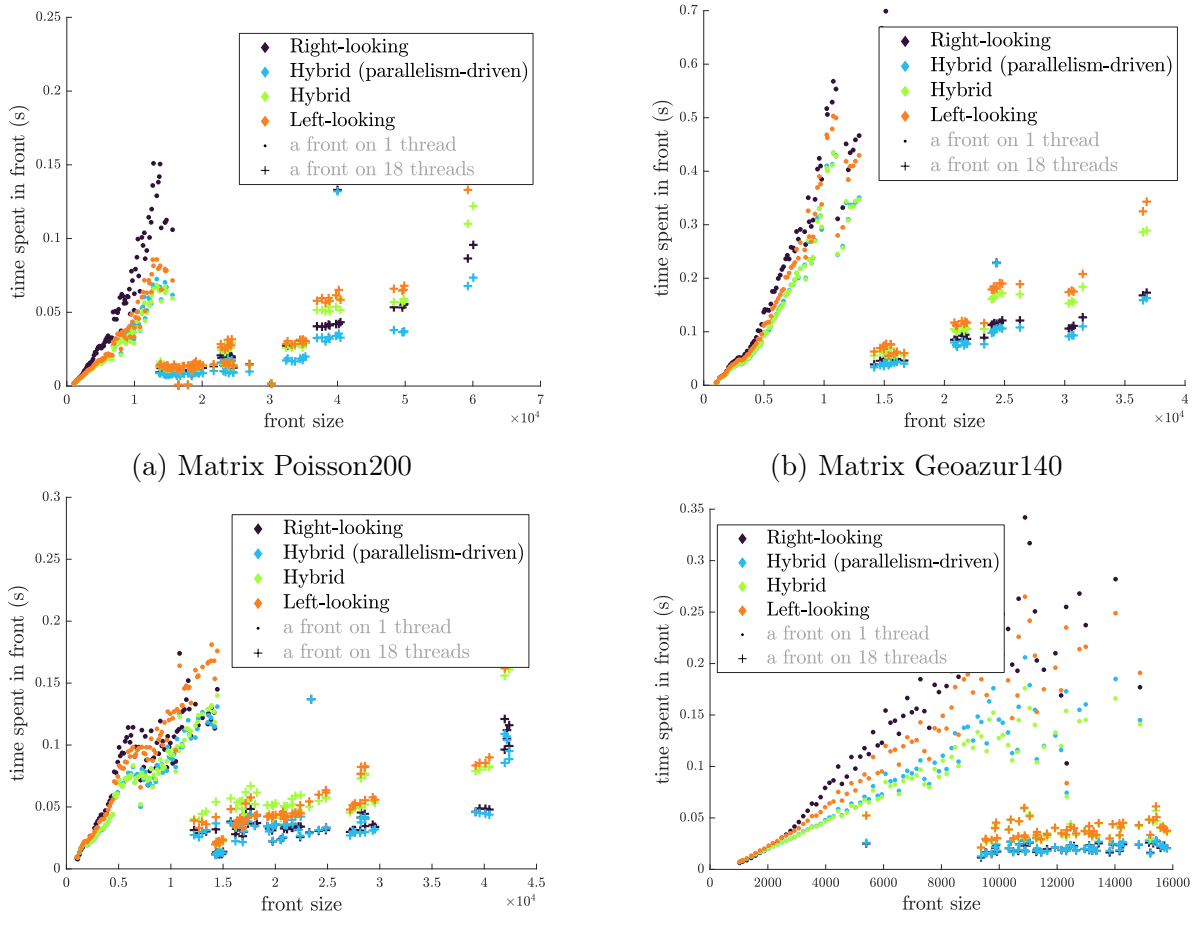
Based on the encouraging results obtained with our prototype in the previous section, we have implemented a subset of variants directly in the MUMPS solver. In addition to the existing RL variant (used so far by MUMPS), we have implemented the LL, hybrid, and parallelism-driven hybrid variants. For now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

We now present some experimental results obtained with these new variants of the MUMPS BLR solve on a range of real-life matrices, listed in Table 4.3. All the experiments have been performed on one node of the Olympe supercomputer previously described, using 2 MPI processes and 18 threads per MPI process, except for the tests with the Poisson200 and thmgaz matrices, which do not fit on a single node, and for which 4×18 threads were used. The BLR ϵ controls the accuracy of the BLR factorization: the low-rank blocks $L_{ij} \approx U_{ij}V_{ij}^T$ are truncated such that $\|L_{ij} - U_{ij}V_{ij}^T\| \leq \epsilon$. We use double precision real arithmetic ("d") for all problems except the Geoazur ones, for which we use single precision complex arithmetic ("c").

First, in Figure 4.8, we plot the time spent in the forward solve of MUMPS for each BLR front. This leads to plots where we can distinguish two distinct groups of points. Those corresponding to the smaller fronts in the lower part of the tree where tree parallelism is used, are each processed using 1 thread. Those corresponding to the larger fronts higher in the tree, where node parallelism is used, are processed using 18 threads. The

Table 4.3: List of sparse matrices used in the experiments of chapter 4.

Matrix	Order	BLR ϵ	Arithmetic	BLR compression (factor size, % of FR)	source	symmetry
Poisson120	1.7M	10^{-6}	d	27%		SPD
Poisson160	4.1M	10^{-6}	d	22%		SPD
Poisson200	8.0M	10^{-6}	d	19%		SPD
Geoazur100	1.6M	10^{-4}	c	52%	S.Operto	UNSYM
Geoazur140	3.8M	10^{-4}	c	47%	S.Operto	UNSYM
atmosmodl	1.5M	10^{-6}	d	30%	SuiteSparse	UNSYM
Geo_1438	1.4M	10^{-6}	d	50%	SuiteSparse	SPD
Queen_4147	4.1M	10^{-6}	d	28%	SuiteSparse	SPD
Serena	1.4M	10^{-6}	d	40%	SuiteSparse	SPD
Transport	1.6M	10^{-6}	d	42%	SuiteSparse	UNSYM
thmgaz	5.0M	10^{-6}	d	30%	EDF (code_aster)	UNSYM



(a) Matrix Poisson200

(b) Matrix Geoazur140

(c) Matrix Queen_4147

(d) Matrix Thmgaz

Figure 4.8: Time spent in each BLR front, in the forward solve of MUMPS.

figure shows that for the larger fronts with node parallelism, the parallelism-driven hybrid variant is the fastest, whereas for the smaller fronts with tree parallelism, the hybrid variant seems to be the best choice for most fronts. This confirms the trends observed with the prototype experiments in the previous section.

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

Table 4.4: Time spent in the forward solve in MUMPS; $n_{\text{rhs}} = 250$ and 2×18 cores are used for all problems except Poisson200 (4×18).

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

To assess the impact of these per-front gains on the total time, we report in Table 4.4 the cumulative time spent in all fronts. This includes the time spent in the full-rank fronts (the very smallest fronts at the bottom of the tree, where BLR compression is not exploited), which are not shown in Figure 4.8. The table compares the standard RL variant to the optimized hybrid variant (which uses the parallelism-driven algorithm only on parallel fronts). The results confirm that the hybrid variant can achieve noticeable time reductions overall.

4.7 Conclusion

The performance of BLR sparse triangular solve, which is critical in several applications, is underwhelming when there are many RHS. This is explained by the fact that the computational bottleneck is the memory access to the RHS, which are dense and far heavier than the compressed BLR LU factors. To overcome this limitation, we have proposed novel hybrid algorithms that combine right-looking and left-looking communication patterns to minimize the number of accesses to the RHS. We have carried out a communication volume analysis that proves that these new variants are indeed communication-avoiding. Based on a performance analysis on synthetic data using a simplified prototype of the BLR triangular solve, we have selected a subset of the most promising hybrid variants and implemented them in the widely used MUMPS solver. Using this implementation, we have confirmed the potential of these new variants on several real-life applications, obtaining time reductions up to 20%.

Chapter 5

Conclusion

Summary

In this thesis, we have presented, analyzed, and evaluated several techniques aiming at further improving the performance of dense and sparse direct solvers, on top of using a BLR compression. In particular, we have proposed a new variant of BLR compression in which low-precision floating-point formats are used for storing the least significant columns of a low-rank approximation.

We described such a format in chapter 2 and applied it to reduce both the time and storage complexities of the LU factorization of a BLR matrix. We performed a backward error analysis of the algorithm that demonstrates its stability, in addition to guiding the choice of precision format for each computation. Similarly to the case of introducing BLR compressions, the theoretical gains of adding mixed precision are entirely dependent on the numerical properties of the matrix. We will observe such gains under the assumption that, for most off-diagonal blocks, the singular values decrease rapidly enough. Our simulations showed that our method has a huge potential on sufficiently large BLR matrices : when using 3 precision formats, we observed a theoretical gain of a factor 1.5× to 2× on the storage cost and 2× to 2.5× on the time complexity of the algorithm, while the error stays of the same order of magnitude.

In chapter 3, we adapted the multifrontal method in order to benefit from the theoretical performance gains from chapter 2. Indeed, the multifrontal method relies on a kernel that performs a partial LU factorization of a BLR matrix, sharing many similarities with the factorization of a dense BLR matrix (see Algorithm 2.1). We first considered the BLR compression in mixed precision (MPBLR) as a storage format only, in order to reduce the size of the factors and the memory peak of the factorization. We succeeded in obtaining reductions of the LU factor size up to 38%, without impacting too much the error of the solution. We also obtained reductions of the volume of communications up to 50%, which can be a critical aspect when solving very large linear systems on massively parallel architectures.

We also described how to adapt the different steps of the multifrontal method in

order to obtain speedups from the use of mixed-precision computations. We implemented the modifications of the triangular solve step and obtained time reductions up to 12% compared to the standard BLR algorithm, while keeping the previous storage reductions.

In the case of multiple right-hand sides, our experiments revealed a performance issue of the BLR triangular solve algorithm. Therefore, in chapter 4 we presented a new variant of this BLR algorithm that has a better data locality, as highlighted by a communication volume analysis. After implementing this variant, we were successful in reducing the computation time of this phase by up to 20%.

Software output

The experiments on new variants of the multifrontal method presented in this thesis were all based on developments made within the MUMPS solver. Some of these developments have been made available as features in advance in the consortium version of MUMPS, available to its industrial partners. None of them have been added to the public version yet.

The options allowing the use of mixed-precision BLR as a storage format were introduced in the consortium version v5.5c in April 2022. The use of computations in mixed precision for accelerating the solve phase without impacting the storage gains was added to the consortium version v5.6c in April 2023. Some of the partners of MUMPS are already taking advantage of these new capabilities. In fact, an option has been added for EDF users of code_aster, allowing them to reduce the memory consumption.

The option allowing to send LU factors in mixed precision in order to further reduce the communication volume (see section 3.1.7) will probably be added in a future release. The hybrid algorithms for accelerating the solve phase in multiple RHS (chapter 4) might one day become an experimental option of MUMPS as well.

Publication output

The contributions presented in chapter 2 have been published in *IMA Journal of Numerical Analysis* (Amestoy et al., 2023b). These contributions and their application to the multifrontal method (chapter 3) were the subject of a talk at the conference CANUM 2022 and a poster at conference Sparse Days 2022.

The MPBLR storage format of MUMPS has also been used to reduce the memory footprint of applications in geophysics. It has contributed to a publication in *The Leading Edge* (Operto et al., 2023). The reduction of the memory footprint of applications from Florian Faucher has also led to a talk by J.Y.L'Excellent at ECCOMAS Congress 2022.

The contributions from chapter 4 have been submitted to the journal *SIAM Journal on Matrix Analysis and Applications* (Amestoy et al., 2023a). They were also presented at conference SIAM CSE 2023.

Perspectives

We now briefly discuss a few of the remaining challenges that could be the object of future work.

The most immediate perspective is to implement the computations in mixed precision during the BLR factorization phase of the multifrontal method, as described in section 3.2, and to obtain actual time gains. Such developments are ongoing in the MUMPS solver.

The BLR triangular solve phase could still be accelerated, by trying to combine different methods presented in this thesis. The use of mixed-precision BLR could be combined with the so-called hybrid variant presented in chapter 4, in order to further reduce the data movements in case there are several RHS. Indeed, performing computations in mixed precision generally requires extra accesses to the RHS, which can be mitigated by the use of the hybrid algorithm. Both approaches could also be combined with the so-called LUA variant of the solve. This technique, presented in section 4.3.3, consists in regrouping small matrix-matrix products into bigger ones, thus improving the data locality and making the multicore parallelization easier, by leaving everything to a well-optimized BLAS function. We have already obtained some mitigated time gains on a separate prototype of LUA, although we think it should be possible to obtain better gains with a finely tuned implementation. We believe that mixed-precision variants could greatly benefit from LUA, given the fact that the granularity of the computations involved can be critically low: for some blocks, the parts in the highest precision may even contain only 1 or 2 columns, and it should be beneficial to stack several of them together using LUA. Only half of the operations can benefit from this increase of the granularity, but this aspect can be improved by using our hybrid algorithm.

The use of custom precision formats could still be the object of future research and developments. First, the implementation and integration of the conversion from mixed precision to uniform precision could be improved, by putting an emphasis on the time spent in the conversion, and not only the storage reductions. Indeed, we observed slight time overheads when using mixed precision as a storage format, and larger overheads when using our set of 7 custom precision formats. However, a correctly tuned implementation should be able to remove those overheads. One might even hope that they would turn into speedups for the triangular solve phase, since the data movement is reduced and the algorithm is memory-bound. Indeed, such speedups have been obtained in the literature for other memory-bound problems, as mentioned in section 1.4.6. Second, one may think of using more complex custom precision formats in order to reduce the storage cost of an MPBLR compression even further, as mentioned in section 3.1.1. If we allow operations more complex than byte copies, we can choose the size of the mantissa down to the bit level, thus further saving storage. The number of bits of the exponent could also be slightly reduced in order for the range of representable numbers not to be mostly wasted on unused exponents.

The use of low precision is particularly well suited to GPUs: those architectures tend

to support computations in more floating-point formats than CPUs. Support for fp16 and/or bfloat16 are becoming the norm, and the latest GPU architecture from NVIDIA, Hopper, even supports 8-bit floating-point formats. As a consequence, it would make sense to try adapting our mixed-precision BLR algorithms to GPUs. However, combining BLR computations and GPU may prove to be rather challenging. So far, the granularity of the computations involved was deemed to be too low to try to combine the use of BLR compression with GPU accelerators in MUMPS. Much work would be needed, including rethinking the size of the low-rank blocks, and using low-rank updates accumulation (LUA) as much as possible. We note that, given the high computational efficiency of GPUs, the algorithms tend to be more easily memory-bound, which would be an argument for trying to reduce the data movement by using advanced storage formats such as the MPBLR compression.

Finally, most of the work presented in this thesis is not exclusive to the MUMPS solver, nor even to the multifrontal method. Indeed, the use of MPBLR compression could also be applied, at least partially, to the supernodal method. It should also be interesting to try to apply some of our work to low-rank formats other than BLR, such as HODLR or $\mathcal{H}$-matrices: one might think of switching each low-rank approximation to mixed precision. Finally, we mostly considered the low-rank approximation of a block based on a truncated SVD or truncated QR decomposition in this thesis. However, other compression kernels may be considered as well, as long as we keep the fundamental property of adding a high-accuracy estimation of a block and a small perturbation, the latter being potentially eligible for low precision.

Scientific presentations

As speaker:

- Rencontres Arithmétique de l'Informatique Mathématique (RAIM), 28 May 2021
- MUMPS Consortium meeting, 24 June 2021
- Seminar at EDF organized by Olivier Boiteau : Solveurs linéaires HPC pour les études industrielles, 18 November 2021.
- Workshop on mixed precision computing, 30 May 2022, LIP6.
- CANUM, 14 June 2022, Évian-les-Bains.
- Sparse Days, Saint-Girons, 21 June 2022 (poster).
- MUMPS Consortium meeting, 29 June 2022
- *Reducing communications and memory costs of parallel Block Low-Rank solvers*, SIAM Conference on Computational Science and Engineering (CSE23), 3 March 2023, Amsterdam.
- MUMPS Users Days, 22 June 2023

As a co-author:

- *Mixed precision sparse direct solver applied to 3D wave propagation*, P. Amestoy, A. Buttari, F. Faucher, M. Gerest, J.-Y. L'Excellent, T. Mary, ECCOMAS Congress, 5-9 June 2022, Oslo.
- Inauguration of Delft Supercomputer: Sparse Linear Solvers: key tools for High-Performance Computing and Simulation, P.R. Amestoy, J.-Y. L'Excellent, and C. Puglisi , A. Buttari, M. Gerest, T. Mary, Art of scientific computing, 30 September 2022, Prinsenhof

Bibliography

S. Abdulah, H. Ltaief, Y. Sun, M. G. Genton, and D. E. Keyes. Geostatistical modeling and prediction using mixed precision tile cholesky factorization. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 152–162, Dec. 2019. doi: 10.1109/HiPC.2019.00028.

P. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L'Excellent, and C. Weisbecker. Improving multifrontal methods by means of Block Low-Rank representations. *SIAM J. Sci. Comput.*, 37(3):A1451–A1474, 2015. doi: 10.1137/120903476.

P. Amestoy, O. Boiteau, A. Buttari, M. Gerest, F. Jézéquel, J.-Y. L'Excellent, and T. Mary. Communication avoiding Block Low-Rank parallel multifrontal triangular solve with many right-hand sides, Apr. 2023a. To appear in IMA J. Numer. Anal. Preprint available at https://hal.science/hal-04082415.

P. Amestoy, O. Boiteau, A. Buttari, M. Gerest, F. Jézéquel, J.-Y. L'Excellent, and T. Mary. Mixed precision low-rank approximations and their application to Block Low-Rank LU factorization. *IMA J. Numer. Anal.*, 43(4):2198–2227, July 2023b. ISSN 0272-4979. doi: 10.1093/imanum/drac037. URL https://doi.org/10.1093/imanum/drac037.

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

P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Anal. Appl.*, 23 (1):15–41, 2001.

P. R. Amestoy, R. Brossier, A. Buttari, J.-Y. L'Excellent, T. Mary, L. Métivier, A. Miniussi, and S. Operto. Fast 3D frequency-domain full waveform inversion with a parallel Block Low-Rank multifrontal direct solver: application to OBC data from the North Sea. *Geophysics*, 81(6):R363–R383, 2016. doi: 10.1190/geo2016-0052.1.

P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary. On the complexity of the Block Low-Rank multifrontal factorization. *SIAM J. Sci. Comput.*, 39(4):A1710–A1740, 2017. doi: 10.1137/16M1077192.

P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary. Performance and scalability of the Block Low-Rank multifrontal factorization on multicore architectures. *ACM Trans. Math. Software*, 45(1):2:1–2:26, 2019a. doi: 10.1145/3242094.

P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary. Bridging the gap between flat and hierarchical low-rank matrix formats: the multilevel Block Low-Rank format. *SIAM J. Sci. Comput.*, 41(3):A1414–A1442, 2019b. doi: 10.1137/18M1182760.

A. Anderson, S. Muralidharan, and D. Gregg. Efficient multibyte floating point data formats using vectorization. *IEEE Transactions on Computers*, 66(12):2081–2096, 2017. doi: 10.1109/TC.2017.2716355.

M. Bebendorf. *Hierarchical Matrices: A Means to Efficiently Solve Elliptic Boundary Value Problems*. Lecture notes in Computational Science and Engineering (LNCSE). Springer-Verlag, 2008. ISBN 3540771468.

P. Blanchard, N. J. Higham, F. Lopez, T. Mary, and S. Pranesh. Mixed precision block fused multiply-add: Error analysis and application to GPU Tensor Cores. *SIAM J. Sci. Comput.*, 42(3):C124–C141, 2020. doi: 10.1137/19M1289546.

E. Carson and N. J. Higham. A new analysis of iterative refinement and its application to accurate solution of ill-conditioned sparse linear systems. *SIAM J. Sci. Comput.*, 39 (6):A2834–A2856, 2017. doi: 10.1137/17M1122918.

E. Carson and N. J. Higham. Accelerating the solution of linear systems by iterative refinement in three precisions. *SIAM J. Sci. Comput.*, 40(2):A817–A847, 2018. doi: 10.1137/17M1140819.

T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Software*, 38(1):1:1–1:25, Dec. 2011. ISSN 0098-3500. doi: 10.1145/2049662.2049663. URL http://doi.acm.org/10.1145/2049662.2049663.

N. Doucet, H. Ltaief, D. Gratadour, and D. Keyes. Mixed-precision tomographic reconstructor computations on hardware accelerators. In *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 31–38, Nov. 2019. doi: 10.1109/IA349570.2019.00011.

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

C. Eckard and G. Young. The approximation of one matrix by another of lower rank. *Psychometrica*, 1:211–218, 1936.

R. D. Fierro and P. C. Hansen. Low-rank revealing utv decompositions. *Numerical Algorithms*, 15(1):37–55, 1997.

P. Ghysels, X. S. Li, F.-H. Rouet, S. Williams, and A. Napov. An efficient multicore implementation of a novel HSS-structured multifrontal solver using randomized sampling. *SIAM J. Sci. Comput.*, 38(5):S358–S384, 2016. doi: 10.1137/15M1010117.

S. Graillat, F. Jézéquel, T. Mary, and R. Molina. Adaptive precision sparse matrix-vector product and its application to Krylov solvers, Sept. 2022. To appear in SIAM J. Sci. Comput. Preprint available at https://hal.science/hal-03561193.

T. Grützmacher, H. Anzt, and E. S. Quintana-Ortí. Using Ginkgo's memory accessor for improving the accuracy of memory-bound low precision BLAS. *Software: Practice and Experience*, 53(1):81–98, 2023. doi: https://doi.org/10.1002/spe.3041. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.3041.

W. Hackbusch. *Hierarchical Matrices : Algorithms and Analysis*, volume 49 of *Springer series in computational mathematics*. Springer, Berlin, 2015. doi: 10.1007/978-3-662-47324-5.

A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham. Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC18 (Dallas, TX), pages 47:1–47:11, Piscataway, NJ, USA, 2018. IEEE. doi: 10.1109/SC.2018.00050.

P. Hénon, P. Ramet, and J. Roman. PaStiX: A high-performance parallel direct solver for sparse symmetric definite systems. *Parallel Computing*, 28(2):301–321, Jan. 2002.

N. J. Higham. Optimization by direct search in matrix computations. *SIAM J. Matrix Anal. Appl.*, 14(2):317–333, Apr. 1993. doi: 10.1137/0614023.

N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002. ISBN 0-89871-521-0. doi: 10.1137/1.9780898718027.

N. J. Higham and T. Mary. A new preconditioner that exploits low-rank approximations to factorization error. *SIAM J. Sci. Comput.*, 41(1):A59–A82, 2019. doi: 10.1137/18M1182802.

N. J. Higham and T. Mary. Solving Block Low-Rank linear systems by LU factorization is numerically stable. *IMA J. Numer. Anal.*, 42(2):951–980, 04 2021. ISSN 0272-4979. doi: 10.1093/imanum/drab020. URL https://doi.org/10.1093/imanum/drab020.

N. J. Higham and S. Pranesh. Simulating low precision floating-point arithmetic. *SIAM J. Sci. Comput.*, 41(5):C585–C602, 2019. doi: 10.1137/19M1251308.

N. J. Higham, S. Pranesh, and M. Zounon. Squeezing a matrix into half precision, with an application to solving linear systems. *SIAM J. Sci. Comput.*, 41(4):A2536–A2551, 2019. doi: 10.1137/18M1229511.

IEEE Computer Society. IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008. doi: 10.1109/IEEESTD.2008.4610935.

J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra. Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems). In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006. doi: 10.1109/SC.2006.30.

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

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

D. Mukunoki and T. Imamura. Reduced-precision floating-point formats on gpus for high performance and energy efficient computation. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 144–145, 2016. doi: 10.1109/CLUSTER.2016.77.

R. Ooi, T. Iwashita, T. Fukaya, A. Ida, and R. Yokota. Effect of mixed precision computing on H-matrix vector multiplication in BEM analysis. *HPCAsia2020*, 2020a.

R. Ooi, T. Iwashita, T. Fukaya, A. Ida, and R. Yokota. Effect of mixed precision computing on H-matrix vector multiplication in BEM analysis. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. ACM Press, New York, Jan. 2020b. doi: 10.1145/3368474.3368479.

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

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

G. Pichon, E. Darve, M. Faverge, P. Ramet, and J. Roman. Sparse supernodal solver using Block Low-Rank compression: Design, performance and analysis. *Journal of Computational Science*, 27:255–270, 2018. ISSN 1877-7503. doi: https://doi.org/10.1016/j.jocs.2018.06.007. URL http://www.sciencedirect.com/science/article/pii/S1877750317314497.

J. L. Rigal and J. Gaches. On the compatibility of a given solution with the data of a linear system. *J. Assoc. Comput. Mach.*, 14(3):543–548, July 1967. ISSN 0004-5411. doi: 10.1145/321406.321416. URL http://doi.acm.org/10.1145/321406.321416.

F.-H. Rouet, X. S. Li, P. Ghysels, and A. Napov. A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization. *ACM Trans. Math. Software*, 42(4):27:1–27:35, June 2016. ISSN 0098-3500. doi: 10.1145/2930660. URL http://doi.acm.org/10.1145/2930660.

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

R. D. Skeel. Scaling for numerical stability in gaussian elimination. *J. Assoc. Comput. Mach.*, 26(3):494–526, July 1979. doi: 10.1145/322139.322148.

C. Weisbecker. *Improving multifrontal solvers by means of algebraic Block Low-Rank representations*. PhD thesis, Institut National Polytechnique de Toulouse, Oct. 2013. URL http://ethesis.inp-toulouse.fr/archive/00002471/.

Matthieu GEREST **8 novembre 2023**

Sujet : Utilisation de compression Block Low-Rank en précision mixte pour améliorer les performances d'un solveur linéaire creux direct

Résumé : EDF effectue des simulations numériques dans différents domaines de la physique. Plusieurs de ses codes de calcul font appel au logiciel MUMPS pour traiter de façon générique, robuste et performante l'étape de résolution de systèmes linéaires creux, qui est très coûteuse. Dans cette thèse, nous explorons plusieurs pistes d'amélioration d'une fonctionnalité existante de MUMPS, la compression Block Low-Rank (BLR). En combinant plusieurs arithmétiques en virgule flottante (précision mixte), il est possible de réduire les complexités en temps et en mémoire, tout en obtenant un résultat suffisamment précis. Notre démarche, guidée par une analyse d'erreur, permet dans un premier temps de réduire la complexité d'une factorisation LU de matrice dense, sans pour autant impacter l'erreur commise de façon significative. Notre méthode est ensuite adaptée au cas d'une factorisation de matrices creuses avec MUMPS. Une première implémentation utilise notre compression BLR en précision mixte comme format de stockage, et permet ainsi de réduire la consommation mémoire de MUMPS. Une seconde implémentation permet de combiner ces gains en mémoire avec des gains en temps lors de la phase de résolution de systèmes triangulaires, grâce à des calculs effectués en précision faible. Enfin, d'autres techniques sont étudiées pour améliorer la localité mémoire de cette phase, dans le cas de seconds membres multiples. Elles conduisent elles aussi à une réduction du temps de calcul de MUMPS.

Mots clés : matrices creuses, solveurs linéaires directs, approximations de rang faible, arithmétique à virgule flottante, analyse d'erreur d'arrondi, calcul haute performance

Subject : Using Block Low-Rank compression in mixed precision for sparse direct linear solvers

Abstract: EDF performs numerical simulations in different domains of physics. Several of its software use the library MUMPS in order to perform the costly step of solving sparse linear systems in a way that is generic, robust and efficient. The goal of this work is to develop new techniques for improving the performance gains of an existing functionality of MUMPS, the Block Low-Rank (BLR) compression. By combining several formats of floating-point numbers (mixed precision), it is possible to reduce the time and memory complexities, without compromising the accuracy of the result. Based on an error analysis, we design new variants of the LU factorization of dense matrices. We then adapt this work to the case of a sparse matrix factorization with MUMPS. A first implementation uses our mixed-precision BLR compression as a storage format, thus reducing the memory consumption of MUMPS. A second implementation allows to combine these memory gains with time gains during the resolution of triangular systems. Finally, we study new techniques for improving the data locality of the BLR triangular solve with multiple right-hand sides, and obtain time reductions within MUMPS.

Keywords : sparse matrices, direct methods for linear systems, low-rank approximations, floating-point arithmetic, rounding error analysis, high-performance computing