RT/APO/12/6
(Submitted)

Improving multifrontal methods by means of block low-rank representations

Patrick Amestoy[1], Cleve Ashcraft[2], Olivier Boiteau[3],
Alfredo Buttari[4], Jean-Yves L'Excellent[5], Clément Weisbecker[6]

ABSTRACT

Matrices coming from elliptic Partial Differential Equations (PDEs) have been shown to have a low-rank property: well defined off-diagonal blocks of their Schur complements can be approximated by low-rank products. Given a suitable ordering of the matrix which gives to the blocks a geometrical meaning, such approximations can be computed using an SVD or a rank-revealing QR factorization. The resulting representation offers a substantial reduction of the memory requirement and gives efficient ways to perform many of the basic dense algebra operations.

Several strategies have been proposed to exploit this property. We propose a low-rank format called Block Low-Rank (BLR), and explain how it can be used to reduce the memory footprint and the complexity of direct solvers for sparse matrices based on the multifrontal method. We present experimental results that show how the BLR format delivers gains that are comparable to those obtained with hierarchical formats such as Hierarchical matrices ($\mathcal{H}$ matrices) and Hierarchically Semi-Separable (HSS matrices) but provides much greater flexibility and ease of use which are essential in the context of a general purpose, algebraic solver.

This work was granted access to the HPC resources of CALMIP under the allocation 2012-p0989 and is funded by EDF under collaboration contract 8611-AAP-5910070284.

Keywords: Sparse direct methods, multifrontal method, low-rank approximations, elliptic PDEs.

AMS(MOS) subject classifications: 05C50, 65F05, 65F50

---

[1] INPT(ENSEEIHT)-IRIT, Université de Toulouse, France (patrick.amestoy@enseeiht.fr

[2] Livermore Software Technology Corporation, Livermore, CA, United States (cleve@lstc.com)

[3] EDF Recherche et Développement, Clamart, France (olivier.boiteau@edf.fr)

[4] CNRS-IRIT, Toulouse, France (alfredo.buttari@enseeiht.fr)

[5] INRIA-LIP, Lyon, France (jean-yves.l.excellent@ens-lyon.fr)

[6] INPT(ENSEEIHT)-IRIT, Université de Toulouse, France (clement.weisbecker@enseeiht.fr)

INPT(ENSEEIHT)-IRIT UMR 5505, Université de Toulouse
2, rue Camichel
31071 Toulouse Cedex 7, France

December 21, 2012

Also available as Inria report RR-8199.

1 Introduction

We are interested in efficiently computing the solution of large sparse linear systems which arise from various applications such as mechanics and fluid dynamics. Modern applications commonly require the solution of linear systems stemming from the discretization of partial differential equations with several millions of unknowns. A sparse linear system is usually referred to as:

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

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

Two types of methods [17] are commonly used to solve (1). *Iterative methods* are cheap in memory consumption and provide better scalability in parallel environments but their effectiveness strongly depends on the numerical properties of the problem. *Direct methods* are more robust and reliable but costly (in terms of flops and memory). As the problems become larger and larger, time and memory complexity become critical in order to be able to tackle challenging applicative problems.

This work focuses on a well known direct approach called the multifrontal method [1]. The objective is to compute a Cholesky LL^T factorization of A:

$$A = LL^T\,, \tag{2}$$

where L is a lower triangular matrix. Without loss of generality, here and in the rest of the paper we only discuss symmetric, positive definite systems for the sake of simplicity. In practice and in the experimental section, the presented methods and algorithms have been applied to the cases of symmetric, indefinite (LDL^T factorization) and unsymmetric (LU factorization) matrices.

We aim to improve the multifrontal method by means of low-rank approximation techniques, which give the opportunity to reduce the memory requirements and the complexity of dense factorizations.

After an introduction to the multifrontal method in the context of nested dissection, we show how it is possible to exploit low-rank properties of matrices to decrease their storage requirements as well as the complexity of some of the basic algebraic operations they are involved in. Then, we present the relevant details of some low-rank formats proposed in the literature and propose a new one called Block Low-Rank (BLR). Through a brief experimental comparison, we show that all these formats have comparable efficiency on a set of test problems. We then show how the BLR format can be integrated within a general purpose linear multifrontal solver. Lastly, we present and discuss experimental results in order to show the efficiency of our implementation of a low-rank sequential multifrontal solver. The study is carried out on a set of large-scale artificial and real life applicative problems.

2 The multifrontal method

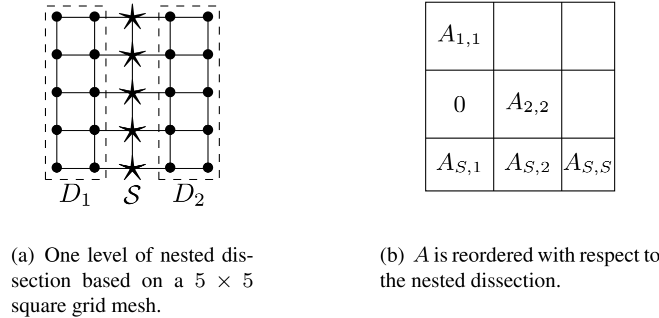
The multifrontal method was first introduced by Duff and Reid [13, 14] in 1983 and, since then, has been the object of numerous studies and the method of choice for several, high-performance, software packages such as `MUMPS` [4], `UMFPACK` [11], `WSMP` [21], `HSL` [18], `PSPASES` [20] and `DSCPACK` [27].

This section provides a brief presentation of the multifrontal algorithm based on the nested dissection method which better suits the context of our work and allows for an easier understanding of how low-rank approximation techniques can be used within sparse, multifrontal solvers; more general and formal presentations are available in the literature [1, 12]. We assume that the reader is familiar with the basics of graph theory.

It is well known that, despite their good reliability and numerical robustness, sparse direct solvers pose heavy requirements in terms of computational and memory resources. This is mostly due to the fact that the factors of a sparse matrix are, in general, much denser and this, in turn, is due to the appearance of *fill-in*, i.e., new nonzero coefficients introduced by the factorization process. Many methods and algorithms have been proposed in order to reduce the amount of fill-in and, among them, one of the most commonly used is the *nested dissection* method [15].

Given a sparse, symmetric matrix A of size n, its structure can be represented with an *adjacency graph*, i.e., a graph $\mathcal{G}(V, E)$ containing n vertices (one for each unknown in A) and edges (i, j) for all $a_{ij} \neq 0$. The fill-in can then be easily modeled using this graph. Specifically, eliminating a variable of A amounts to removing the associated vertex from $\mathcal{G}$ along with all incident edges and adding new edges that connect the

neighbors of the eliminated vertex that were not already connected to each other; these newly introduced edges represent fill-in coefficients. Now assume that a vertex separator S of $\mathcal{G}$ is computed, i.e., a subset of vertices which, if removed, splits the graph into two subgraphs D_1 and D_2 and assume that A is permuted in such a way that all the variables in D_1 are eliminated first, all those in D_2 are eliminated second and all those in S are eliminated last. Because all the neighbors of vertices in D_1 are either in D_1 or in S, no fill-in will be generated inside the submatrix that connects the variables in D_1 to those in D_2. This is illustrated in Figure 1 for the simple case of a matrix from a 5-point stencil operator.



(a) One level of nested dissection based on a 5×5 square grid mesh.

(b) A is reordered with respect to the nested dissection.

Figure 1: One level of nested dissection.

Because fill-in occurring inside the diagonal blocks $A_{1,1}$ and $A_{2,2}$ may still be excessive, this procedure can be recursively and independently applied to the two subgraphs corresponding to D_1 and D_2 until the fill-in inside diagonal blocks can be considered negligible or not worth being reduced. This procedure essentially describes the nested dissection method although modern nested dissection ordering tools [24, 26] employ much more sophisticated algorithms.

Applying the nested dissection algorithm to a graph generates a *separators tree* which implicitly defines a fill-in reducing permutation of the input matrix. This is shown in Figure 2, where, for the leaves of the tree, we assumed $S_i = D_i$ which is equivalent to saying that S_i is a separator that splits D_i into two empty subdomains.

The separators tree can also be regarded as an *elimination* or *assembly tree* – a data structure introduced by Schreiber [28] which is at the base of all modern sparse factorization algorithms. This tree defines the dependencies between the variables of a matrix and thus, implicitly, the order in which they have to be eliminated. Specifically, the elimination tree states that the elimination of the variables associated with a node (or with the corresponding separator, in our case) only affects variables associated with ancestor nodes, that is, nodes along the path that connects the eliminated node to the root of the tree. Therefore, all the pivotal orders defined by topological traversals of the tree are equivalent in the sense that they produce the same amount of fill-in.

The multifrontal method [1, 13, 14] achieves the factorization of a sparse matrix through a sequence of operations on relatively small dense matrices called *frontal matrices* or, simply, *fronts*. One of these matrices is associated with each node of the tree and is formed by coefficients related to the variables associated with that node as well as their neighbors. Note that the neighbors include those variables that have become such due to fill-in introduced by the elimination of previous variables.

The multifrontal factorization consists in a topological order (i.e., bottom-up) traversal of the tree where,

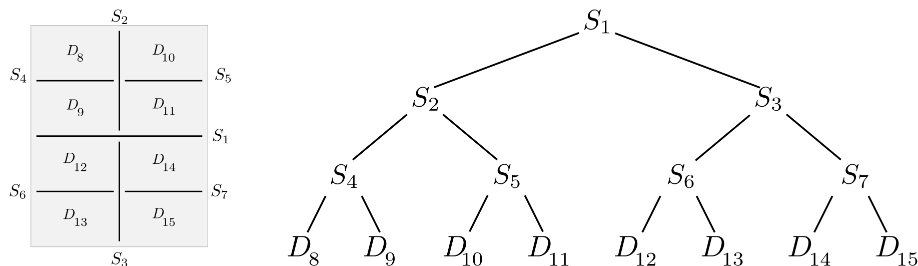
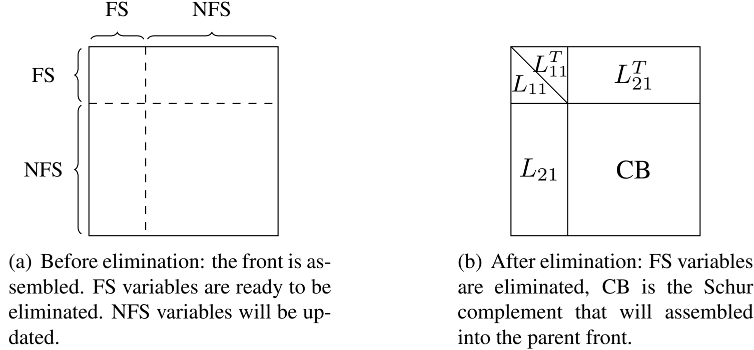


Figure 2: A general nested dissection and its corresponding separator tree.

each time a node is visited, two operations are performed:

- **assembly**: the frontal matrix is formed by summing the coefficients in the rows and columns of the variables associated with the tree node with coefficients produced by the factorization of its children.
- **factorization**: once the frontal matrix is formed, a partial Cholesky factorization is performed on it in order to eliminate the variables associated with the tree node. The result of this operation is a set of rows of the global L factor and a Schur complement, also commonly referred to as *contribution block* (CB), containing coefficients that will be assembled into the parent node.



(a) Before elimination: the front is assembled. FS variables are ready to be eliminated. NFS variables will be updated.

(b) After elimination: FS variables are eliminated, CB is the Schur complement that will assembled into the parent front.

Figure 3: Structure of a front F.

As shown in Figure 3, the set of variables in a frontal matrix can be split into two subsets:

- **fully-summed (FS) variables**: these are the variables associated with the tree node or, equivalently, with the corresponding separator. They are called fully-summed because the rows and columns in the front are up to date with respect to previously eliminated variables, that is, those that have been eliminated at descendant nodes.
- **non fully-summed (NFS) variables**: this subset contains the neighbors of the fully-summed variables and is formed by pieces of separators belonging to ancestor nodes which form a border around the fully-summed variables. For the frontal matrix at node S_7 of the tree in Figure 2, this border is formed by half of the S_1 separator and the entire S_3 separator; variables in this border have become neighbors of those in S_7 due to fill-in coefficients introduced by the eliminations of variables in S_{14} and S_{15}.

Once a frontal matrix is factorized, the resulting factors are stored apart while the contribution block is held in a temporary memory area which is then freed when the parent front is processed. For this reason the global amount of memory needed to run the multifrontal factorization is, in general, bigger than the size of the resulting factors. The temporary memory area is managed as a stack which is very convenient if the tree is traversed in postorder. In the rest of this paper we will refer to this area simply as *CB stack*.

This paper describes a technique based on the usage of low-rank approximations to represent frontal matrices and to perform all the operations they are involved in. This allows to reduce both the memory consumption as well as the total volume of computations of the multifrontal method.

3 Low-rank approximations

In this section, we present how low-rank properties of matrices can be revealed and exploited. Several matrix representations are described and discussed.

3.1 Low-rank property

A low-rank matrix can be represented in a form which decreases its memory requirements and the complexity of basic linear algebra operations it is involved in, such as matrix-matrix products and triangular solves. This is formalized by Definition 1.

Definition 1 ([7] Low-rank matrix) *Let A be a matrix of size $m \times n$. Let k_ε be the approximated numerical rank of A at accuracy ε. A is said to be a low-rank matrix if there exist three matrices U of size $m \times k_\varepsilon$, V of size $n \times k_\varepsilon$ and E of size $m \times n$ such that :*

$$A = U \cdot V^T + E \,,$$

where $\|E\|_2 \leq \varepsilon$ and $k_\varepsilon(m+n) < mn$.

The numerical rank at precision ε, k_ε, can be computed together with U and V with a Singular Value Decomposition (SVD), in which case U and V are unitary, or, less precisely but much faster, with a Rank-Revealing QR (RRQR) factorization, in which case only U is unitary. Low-rank approximation techniques are based upon the idea to ignore E and simply represent A as the product of U and V^T. Thus, $A = U \cdot V^T$ is said to be the *low-rank form* of A and it is an approximation whose precision can be controlled through the parameter ε called *low-rank threshold*. In the rest of this paper, the subscript ε in k_ε will be omitted for the sake of clarity. Definition 1 simply states that k has to be small enough to guarantee that the low-rank form of A takes less memory that the standard form. Analogous criteria can be defined in the case where the objective is to reduce the complexity of operations involving A. Table 1 shows the requirements on the rank k in order to reduce the operation count of the matrix-matrix product and of the triangular system solve, when the resulting matrix is stored in dense form.

Table 1: Cost of dense and low-rank basic linear algebra operations. A, L, $A_1 = U_1V_1^T$ and $A_2 = U_2V_2^T$ are $n \times n$, L is lower triangular, U_1, V_1, U_2 and V_2 are $n \times k$. The bracketing is critical to exploit the smaller dimension of each matrix.

operation type		dense	low-rank	rank requirement
Cholesky factorization	LL^T	$n^3/3$	—	—
RRQR compression	$U_1V_1^T$	$6kn^2 - 6k^2n + \frac{10}{3}k^3$	—	—
triangular solve	$U_1(V_1^TL^{-T})$	n^3	$3kn^2$	$k < n/3$
matrix-matrix product	$U_1(V_1^TU_2)V_2^T$	$2n^3$	$2kn^2 + 4k^2n$	$k < n/2$

In practice, matrices coming from application problems are not *low-rank*, which means that they cannot be directly written in low-rank form. However, Börm [8] and Bebendorf [7] show that low-rank approximations can be performed on submatrices defined by an appropriately chosen partitioning of matrix indices. Assuming $I = \{1, \ldots, n\}$ is the set of row (and column) indices of A, a set of indices $\sigma \subset I$ is called a cluster. Then, a *clustering* of I is a disjoint union of clusters which equals I. $b = \sigma \times \tau \subset I \times I$ is called a *block cluster* based on clusters σ and τ. A *block clustering* of $I \times I$ is then defined as a disjoint union of block clusters which equals $I \times I$. Let $b = \sigma \times \tau$ be a block cluster, $A_b = A_{\sigma\tau}$ is the subblock of A with row indices σ and column indices τ.

A considerable reduction of both the memory footprint and of the operations complexity can be achieved if the block clustering defines blocks whose singular values have a rapid decay (for instance, exponential) in which case each block can be accurately represented by a low-rank product $U \cdot V^T$. Clearly, this condition cannot be directly used in practice to define the matrix block clustering. In many practical cases it is, however, possible to exploit the knowledge of the mathematical problem or the geometrical properties of the domain where the problem is defined in order to define an *admissibility condition*, i.e., a heuristic rule that can be cheaply checked to establish whether or not a block is (likely to be) low-rank or that can be used to guide the block clustering computation. For instance, in the case of matrices deriving from discretized elliptic PDEs one such admissibility condition is presented in Definition 2.

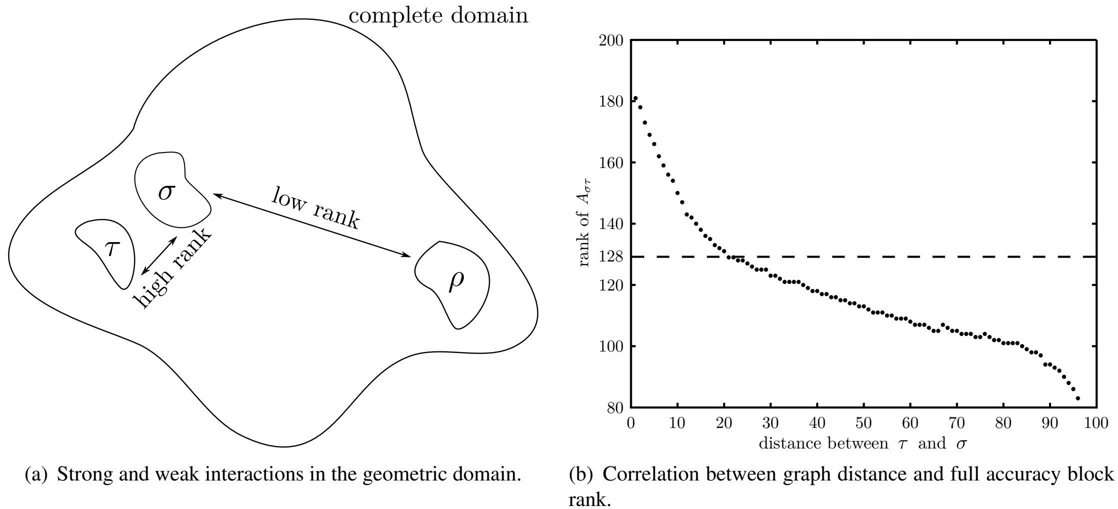
Definition 2 ([7, 8] Admissibility condition for elliptic PDEs) *Let $b = \sigma \times \tau$ be a block cluster. b is admissible if*

$$diam(\sigma) + diam(\tau) \leq 2\eta \, dist(\sigma, \tau) \,,$$

where diam() and dist() are classical geometric diameter and distance, respectively, and η is a problem dependent parameter.

This admissibility condition follows the intuition that variable sets that are far away in the domain are likely to have weak interactions which translates into the fact that the corresponding block has a low rank;

this idea is depicted in Figure 4(a). Figure 4(b) shows that the rank of a block $A_{\sigma\tau}$ is a decreasing function of the geometric distance between clusters σ and τ. This experiment has been done on a top-level separator of a 3D 128^3 wave propagation problem called `Geoazur128` (and further described in Table 3), with square clusters of dimension 16×16, so that each subblock has size 256. It shows that depending on the distance between clusters, there is potential for compression which can be exploited. The dashed line at $y = 128$ shows the cutoff point where it pays to store the subblock using a low-rank representation.



(a) Strong and weak interactions in the geometric domain.

(b) Correlation between graph distance and full accuracy block rank.

Figure 4: Two illustrations of the admissibility condition for elliptic PDEs.

An admissibility condition being given, it is possible to define an *admissible block clustering*, i.e., a block clustering with admissible blocks, which can be used to efficiently represent a dense matrix by means of low-rank approximations with an accuracy defined by the low-rank threshold ε in Definition 3.1.

It has to be noted that, in practice, admissibility conditions may be even simpler than the one presented above and may also take into account aspects that, for example, pertain to the efficiency of basic linear algebra kernels on the U and V matrices of the low-rank form; further details about admissibility conditions are given in Section 4.

3.2 Matrix representations

In order to exploit low-rank blocks arising from admissible block clusterings, several matrix representations have been proposed in the literature. In this section, we very briefly describe Hierarchical matrices [7] ($\mathcal{H}$-matrices) and Hierarchically Semi-Separable matrices [35] (HSS matrices) because they have been used in the context of a multifrontal solver. The Hierarchically Block-Separable (HBS) format introduced by Gillman *et al.* [16] is highly relevant too but will not be discussed in details as it is essentially equivalent to HSS. All these structures can be used to approximate fronts and thus globally reduce the memory consumption as well as the flop count of the multifrontal process.

3.2.1 Hierarchical matrices ($\mathcal{H}$-matrices)

The $\mathcal{H}$-matrix [7, 8] format, where $\mathcal{H}$ stands for Hierarchical, is historically the first low-rank format for dense matrices. This format is based on an admissible block clustering which is obtained by a recursive subdivision of $I \times I$. Algorithm 1 shows how the $\mathcal{H}$-matrix format of a matrix A is built.

The result of this recursive procedure is a *block cluster tree* built in a top-down fashion. Note that a set of siblings of the tree defines a block clustering of the block cluster associated with their parent node and that the final admissible clustering is defined by the leaves of the tree.

Figure 5 shows an example of a $\mathcal{H}$-matrix with the corresponding block cluster tree. In this case the block clustering has been defined by splitting the set of matrix indices as $I = I_7 = I_3 \cup I_6 = \{I_1 \cup I_2\} \cup \{I_4 \cup I_5\}$.

Algorithm 1 $\mathcal{H}$-matrix construction

Input A, a matrix defined on row and column indices I.
Output $\widehat{A}$, the $\mathcal{H}$-matrix form of A.

1: initialize `list` with $[I \times I]$
2: **while** `list` is not empty **do**
3: remove b from `list`
4: **if** b is admissible **then**
5: $\widehat{A}_b \leftarrow$ low-rank form of A_b
6: **else if** b is large enough to split **then**
7: $\begin{pmatrix} b_1 & b_2 \\ b_3 & b_4 \end{pmatrix} = b$
8: add b_1, b_2, b_3 and b_4 to `list`
9: **else**
10: $\widehat{A}_b \leftarrow A_b$
11: **end if**
12: **end while**

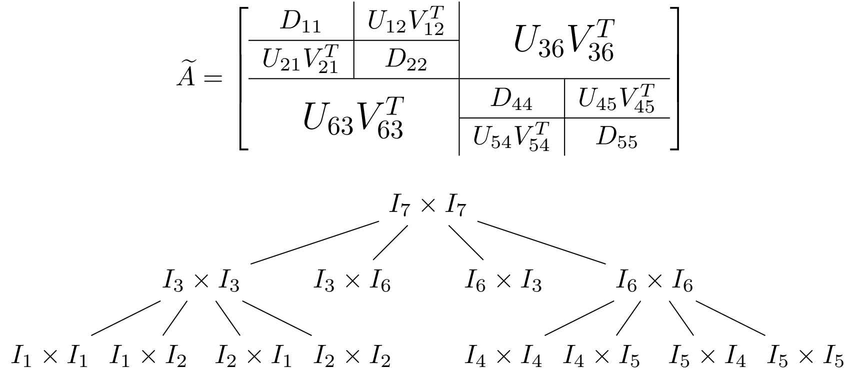


Figure 5: $\mathcal{H}$-matrix structure and associated block cluster tree

In a more general case, the structure of a $\mathcal{H}$-matrix is not necessarily as regular as in the provided example which means that the diagonal blocks may not have the same size. For more details about $\mathcal{H}$-matrices, please refer to Bebendorf [7], Börm [8] or Hackbusch [22].

3.2.2 Hierarchically Semi-Separable matrices (HSS matrices)

HSS matrices are also based on a hierarchical blocking defined by a cluster tree which is built by recursively splitting the set of matrix indices.

Definition 3 ([33] HSS matrix) *Assume A is an $n \times n$ dense matrix, and $I = \{1, 2, \ldots, n\}$. I is recursively split into smaller pieces following a binary tree T with k nodes, denoted by $j = 1, 2, \ldots, k \equiv$ root(T). Let $t_j \subset I$ be a cluster associated with each node j of T. (T is used to manage the recursive partition of A.) We say A is an* HSS form *with the corresponding postordered* HSS tree *T if :*

1. *T is a full binary tree in its postordering, or, each node j is either a leaf or a non-leaf node with two children j_1 and j_2 which satisfy $j_1 < j_2 < j$;*
2. *The index sets satisfy $t_{j_1} \cup t_{j_2} = t_j$ and $t_{j_1} \cap t_{j_2} = \emptyset$ for each non-leaf node j, with $t_k \equiv I$;*
3. *For each node j, there exist matrices $D_i, U_i, V_i, R_i, W_i, B_i$ (called* HSS generators*), which satisfy the following recursions for each non-leaf node j:*

$$D_j \equiv A|_{t_j \times t_j} = \begin{pmatrix} D_{j_1} & U_{j_1} B_{j_1} V_{j_2}^T \\ U_{j_2} B_{j_2} V_{j_1}^T & D_{j_2} \end{pmatrix}, U_j = \begin{pmatrix} U_{j_1} R_{j_1} \\ U_{j_2} R_{j_2} \end{pmatrix}, V_j = \begin{pmatrix} V_{j_1} W_{j_1} \\ V_{j_2} W_{j_2} \end{pmatrix},$$

where U_k, V_k, R_k, W_k and B_k are not needed (since $D_k \equiv A$ is the entire diagonal block without a corresponding off-diagonal block). Due to the recursion, only the D_j, U_j, V_j generators associated with a leaf node j of T are stored.

An example of a HSS matrix is given in Figure 6.

$$\widetilde{A} = \left[\begin{array}{c|c|cc} D_1 & U_1 B_1 V_2^T & U_1 R_1 B_3 W_4^T V_4^T & U_1 R_1 B_3 W_5^T V_5^T \\ \hline U_2 B_2 V_1^T & D_2 & U_2 R_2 B_3 W_4^T V_4^T & U_2 R_2 B_3 W_5^T V_5^T \\ \hline U_4 R_4 B_6 W_1^T V_1^T & U_4 R_4 B_6 W_2^T V_2^T & D_4 & \multicolumn{1}{|c}{U_4 B_4 V_5^T} \\ \cline{3-4} U_5 R_5 B_6 W_1^T V_1^T & U_5 R_5 B_6 W_2^T V_2^T & U_5 B_5 V_4^T & \multicolumn{1}{|c}{D_5} \end{array}\right]$$

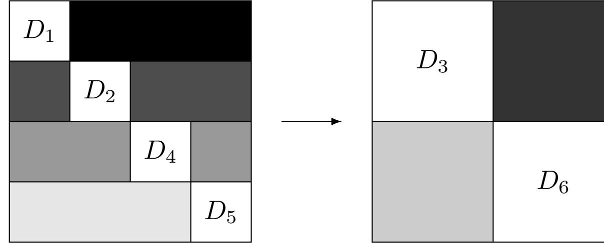


Figure 6: Structure of a HSS matrix and its basic construction scheme

The construction of an HSS matrix is achieved through a topological order traversal of the T tree, also referred to as *HSS tree*. Each time a node j is visited, the block-row and block-column corresponding to the related cluster (i.e., $A_{t_j,:} = A_{t_j,(I \setminus t_j)}$ and $A_{:,t_j} = A_{(I \setminus t_j),t_j}$, respectively) are compressed into a low-rank form. Note that, apart from the leaves, the compressed block-row or block-column is formed by combining the result of previous compressions. The bottom of Figure 6 shows this process when T has four leaves (seven nodes, total).

Although it should be possible to have a different blocking for the upper and lower triangular parts of the matrix as for the $\mathcal{H}$-matrices, we have not found any example of this case in the literature.

For a more detailed presentation and study of HSS matrices, please refer to Xia *et al.* [33, 35, 34] and Wang *et al.* [30, 32].

3.2.3 Block Low-Rank matrices (BLR matrices)

We propose a new structure based on a non-hierarchical blocking of the matrix.

Definition 4 (Block Low-Rank matrix (BLR)) *Given an admissibility condition, let P be an admissible block clustering with p^2 clusters. Let A be an $n \times n$ matrix.*

$$\widetilde{A} = \begin{bmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{p1} & B_{p2} & \cdots & B_{pp} \end{bmatrix}$$

is called a "Block Low-Rank matrix" if $\exists (\sigma, \tau) \in \{1, 2, \ldots, p\}^2$, there exists $k_{\sigma\tau}$ such that $B_{\sigma\tau}$ is a low-rank block with rank $k_{\sigma\tau}$.

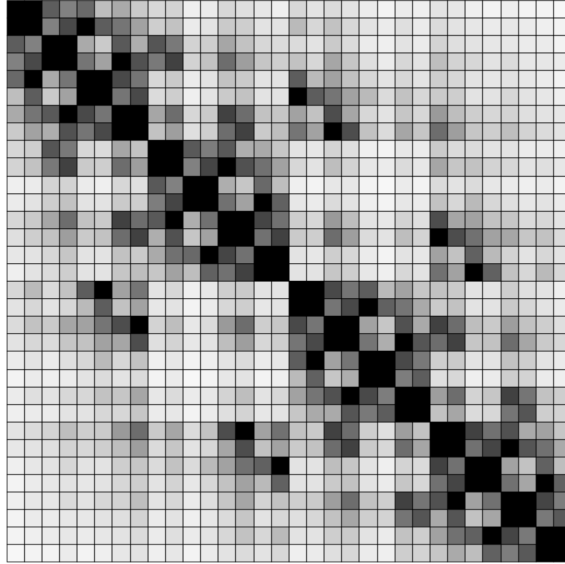
The structure of a BLR matrix is not hierarchical: a flat block matrix structure is used, as illustrated in Figure 7. Based on the same block clusters as in example Figures (5) and (6), we show in Equation (3) an example of a 4×4 BLR matrix. Remember that U and V are not necessarily unitary.

$$\widetilde{A} = \left[\begin{array}{c|c|c|c} D_1 & U_{12} V_{12}^T & U_{13} V_{13}^T & U_{14} V_{14}^T \\ \hline U_{21} V_{21}^T & D_2 & U_{23} V_{23}^T & U_{24} V_{24}^T \\ \hline U_{31} V_{31}^T & U_{32} V_{32}^T & D_3 & U_{34} V_{34}^T \\ \hline U_{41} V_{41}^T & U_{42} V_{42}^T & U_{43} V_{43}^T & D_4 \end{array}\right]. \tag{3}$$

Figure 7 shows the global structure of the BLR representation of a dense Schur complement of order 128×128 corresponding to the top level separator of a $128 \times 128 \times 128$ `Laplacian` problem, with a low-rank threshold set to 10^{-14}. The numbering scheme illustrated in 7(a) is recursive although this is only required for $\mathcal{H}$ and HSS matrices.

27	28	31	32
25	26	29	30
19	20	23	24
17	18	21	22
11	12	15	16
9	10	13	14
3	4	7	8
1	2	5	6

(a) Numbering scheme of the graph associated with the Schur complement. The numbers give the ordering of the 32 blocks of $32 \times 16 = 512$ variables within the BLR structure. Any other numbering of the clusters would give equivalent results.



(b) Structure of a BLR matrix. The darkness of a block is proportional to its storage requirement (the lighter a block is, the smaller is the memory needed to store it). Each block in the matrix is of size 512×512.

Figure 7: Illustration of a BLR matrix of a dense Schur complement of a $128 \times 128 \times 128$ `Laplacian` problem with a low-rank threshold ϵ set up to 10^{-14}. The corresponding clustering of its 128×128 plan graph into $4 \times 8 = 32$ blocks is also given.

BLR matrices can be viewed as a particular case of $\mathcal{H}$-matrices where all the subblocks have been subdivided identically, i.e., where all the branches of the block cluster tree have the same depth.

3.2.4 Comparative study

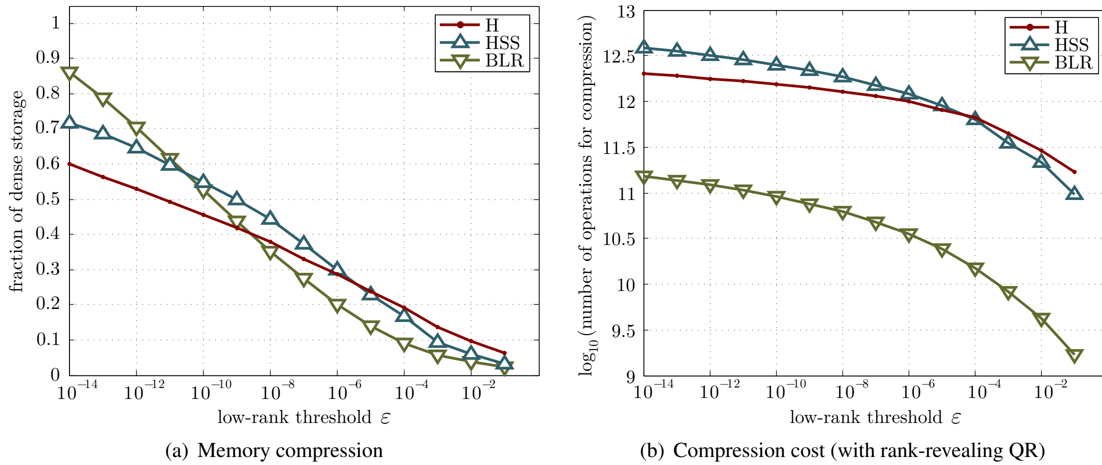
A preliminary comparative study of the three formats described above (both in terms of memory and computational costs) is presented in this section in order to validate the potential of our BLR format in the context of the development of algebraic methods for exploiting low-rank approximations within a general purpose, multifrontal solver. The $\mathcal{H}$, HSS and BLR formats are used for compressing the Schur complement associated with a separator of a cubic mesh (or graph) of size 128; the separator is a 128×128 surface lying in the middle of the cubic domain and the corresponding Schur complement is a dense matrix of order 16384. This is assessed for two different problems: the `Geoazur` problem (see Section 6.1) and a `Laplacian` operator discretized with a 11-point stencil. For all three formats, the clustering was defined by a 8×8 recursive checkerboard partitioning of the separator into blocks of size 256, using the same approach as in the 4×8 case from Figure 7(a); this clustering is essentially what is referred to as the *weak admissibility condition* in Börm [8, Remark 3.17]. Other block sizes have been experimented and give comparable results for both problems. These results are in compliance with what was observed in Wang *et al.* [31].

In terms of memory compression, Figures 8(a) and 9(a) show that the three formats appear to be roughly equivalent at low precision: the truncation is so agressive that almost no information is stored anymore. In the case of the `Geoazur` problem, the three formats provide comparable gains with BLR being slightly worse than the other two at high accuracy and better when the approximation threshold is bigger than 10^{-10}.

For the `Laplacian` problem, BLR is consistently better than the hierarchical formats although all three provide, in general, considerable gains. These preliminary results suggest that the BLR format delivers gains that are comparable to those obtained with the hierarchical ones on the test problems.

Figures 8(b) and 9(b) show the cost of computing the $\mathcal{H}$, HSS and BLR formats starting from a standard dense matrix. This is computed as the cost of the partial RRQR factorizations; because for $\mathcal{H}$ and HSS matrices these operations are performed on much larger blocks, the conversion to BLR format is much cheaper with respect to the cases of hierarchical formats and also with respect to the cost of an LU factorization of the same front. The cost of a full rank factorization is indeed 10 (`Geoazur128` problem) or 100 (`Laplacian` problem) times larger than the BLR compression cost at full accuracy ($\varepsilon = 10^{-14}$). Moreover, because a truncated rank-revealing QR factorization is used to compress the blocks, the compression cost decreases when the accuracy decreases. This property is extremely useful as it allows to switch from full-rank to low-rank and *vice versa* at an acceptable cost compared to the dense LU factorization cost.

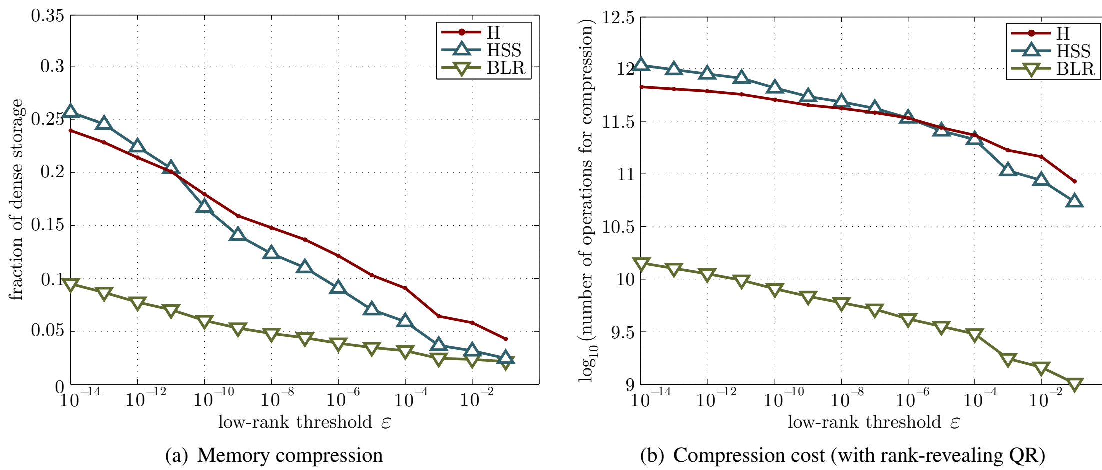
A deep and detailed comparison of low-rank formats, both theoretical and experimental, is out of the scope of this paper; nonetheless, it is the subject of ongoing research work and collaborations.



(a) Memory compression

(b) Compression cost (with rank-revealing QR)

Figure 8: Comparison between $\mathcal{H}$, HSS and BLR formats on a 128^3 `Geozur` problem. The matrix considered is the factorized Schur complement associated with the top level separator of a perfect hand generated nested dissection tree. The clustering is hand-computed and geometrical.



(a) Memory compression

(b) Compression cost (with rank-revealing QR)

Figure 9: Comparison between $\mathcal{H}$, HSS and BLR formats on a 128^3 `Laplacian` problem. The matrix considered is the factorized Schur complement associated with the top level separator of a perfect hand-computed nested dissection tree. The clustering is hand-computed and geometrical.

3.3 Motivation to use BLR format in a multifrontal solver

$\mathcal{H}$ and HSS matrices have been successfully used to accelerate multifrontal solvers and to reduce their memory consumption [33]. These approaches, however, either rely on the knowledge of mathematical properties of the problem or on the geometric properties of the domain on which it is defined (i.e., the discretization mesh) or lack some of the features that are essential for a general purpose, sparse, direct solver like, for instance, robust threshold pivoting or dynamic load balancing in a parallel environment. Our objective is, instead, to exploit low-rank approximations within a general purpose, multifrontal solver which aims at providing robust numerical features in a parallel framework and where no knowledge of the problem can be assumed except the matrix itself. In such a context, hierarchical structures are likely to be too hard to handle and may severely limit the necessary flexibility:

- In a parallel multifrontal solver, frontal matrices may be statically or dynamically partitioned, in an irregular way in order to achieve a good load and memory balance; hierarchical formats may pose heavy constraints or may be complex to handle in this case.

- The HSS format achieves good compression rates only if each node of the HSS tree defines an admissible cluster. This means that the diagonal blocks cannot be permuted in an arbitrary way but have to appear along the diagonal in a specific order which depends on the geometry of the separator. This information may not be available or too complex to extrapolate in a general purpose, algebraic solver.

- In HSS matrices the compressions are achieved through SVD or RRQR decompositions on block-rows and block-columns, i.e., strongly over or under-determined submatrices. This has a twofold disadvantage. First it is based on the assumption that all the columns in a block-row (or, equivalently, rows in a block-column) lie in a small space which has been proven only for some classes of problems and may not be true in general [9]. Second, due to the shape of the data, these operations are often inefficient and difficult to parallelize; this has led researchers to consider the use of communication-avoiding RRQR factorizations or the use of randomization techniques.

- It is not natural, and does not seem to have been done in practical implementations, to use the $\mathcal{H}$ or HSS formats on the L_{21} part of a frontal matrix (see Figure 3(b)). The approach presented by Xia [33], for instance, handles this submatrix as a whole low-rank block which may result in sub-optimal compression and which again requires inefficient operations to compute the low-rank form.

- The assembly of a frontal matrix is very difficult to achieve if the contribution blocks are stored in a hierarchical, low-rank format. For this reason, intermediate full-rank representations are used in practice [33, 30, 34]. This, however, comes at a very significant cost (see Figures 8(b) and 9(b)).

The analysis presented in Section 3.2.4, although restricted to a limited number of problems, suggests that a simpler format such as BLR delivers benefits comparable to the $\mathcal{H}$ and HSS ones; BLR, though, represents a more suitable candidate for exploiting low-rank techniques within a general-purpose, multifrontal solver as it presents several advantages over hierarchical formats:

- The matrix blocking is flat, i.e., not hierarchical, and no relative order is imposed between the blocks of the clustering. This is a very valuable feature because it allows to compute the index clustering more easily and because it delivers much greater flexibility for distributing the data in a parallel environment.

- The size of blocks is homogeneous and is such that the SVD or RRQR operations done for computing the low-rank forms can be efficiently executed concurrently with sequential code (for example, the LAPACK `_GESVD` or `_GEQP3` routines). This property is extremely valuable in a distributed memory environment where reducing the volume of communications is important.

- The L_{21} submatrix can be easily represented in BLR format with a block clustering induced by those of the L_{11} and L_{22} submatrices.

- Pivoting techniques seem hard and inefficient to use when factorizing a matrix stored in a hierarchical format; this has led researchers to employ different types of factorizations, e.g. a ULV factorization [10, 35]. BLR format is more naturally suited for applying partial threshold pivoting techniques within a standard LU factorization.

- The assembly of frontal matrices is relatively easy if the contribution blocks are in BLR format. We also have the option of switching to an intermediate full-rank format because compression is relatively cheap (see Figures 8(b) and 9(b)).

Motivated by the previous observations, we focus in this paper on the BLR format for exploiting, in an algebraic setting, low-rank techniques. The remainder of this paper describes how block clustering is performed and the main algorithmic issues of the low-rank factorization phase.

4 Block clustering

We describe in this section how block clusterings can be computed in order to obtain efficient BLR representations of fronts.

4.1 Admissibility condition

The admissibility condition presented in Definition 2 requires geometric information in order to properly compute diameters and distances. In the context of an algebraic solver, this is not conceivable because the only available information is the matrix.

Thus, another condition must be used and a natural idea is to use the graph of the matrix. In Börm [8] and Grasedyck [19], other admissibility conditions have been studied. They only need the graph of the matrix $\mathcal{G}$ and are thus called *black box* methods. Definition 5 details this graph version of Definition 2.

Definition 5 ([8, 19] Black box admissibility condition) *Let $b = \sigma \times \tau$ a block cluster where σ and τ are sets of graph nodes. b is admissible if*

$$diam_{\mathcal{G}}(\sigma) + diam_{\mathcal{G}}(\tau) \leq 2\eta \, dist_{\mathcal{G}}(\sigma, \tau) \, ,$$

where $diam_{\mathcal{G}}$ and $dist_{\mathcal{G}}$ are classical graph diameter and distance in $\mathcal{G}$, respectively, and η a problem dependent parameter.

This admissibility condition may still be unpractical because, first, it is not clear how to choose the η parameter in an algebraic context and, second, because computing diameters and distances of clusters of a graph may be costly. This condition, however, can be further simplified and complemented with other practical considerations in order to define a clustering strategy suited for the BLR format:

1. For efficiency reasons, the size of the clusters should be chosen between a minimum value that ensures a good efficiency of the BLAS operations performed later with U and V matrices and a maximum value that allows to perform the SVDs or RRQRs on the blocks with sequential routines as well as an easy and flexible distribution of blocks in a parallel environment.

2. The distance between any two clusters σ and τ has to be greater than zero ($dist_{\mathcal{G}}(\sigma, \tau) > 0$) which amounts to saying that all the clusters are disjoint. Note that this point only is equivalent to the *weak admissibility condition* proposed by Börm [8, Remark 3.17].

3. For a given size of clusters (and consequently a given number of clusters in $\mathcal{G}$) the diameter of each cluster should be as small as possible in order to group within a cluster only those variables that are likely to strongly interact with each other. For example, in the case where $\mathcal{G}$ is a flat surface, it is better off to define clusters by partitioning $\mathcal{G}$ in a checkerboard fashion rather than cutting it into slices.

Once the size of the clusters is chosen, the objectives 2 and 3 can be easily achieved by feeding the graph $\mathcal{G}$ to any modern graph partitioning tool such as METIS [24] or SCOTCH [26], as discussed in the next two sections which show how to compute the clustering of the fully assembled and non-fully assembled variables in a frontal matrix. Note that, in practice, partitioning tools take the number of partitions as input and not their size; however the size of the resulting clusters will differ only slightly because partitioning methods commonly try to balance the weight of the partitions. Because of the simple nature of the BLR format which does not require a relative order between clusters, simpler and cheaper partitioning or clustering techniques may be employed instead of complex tools such as METIS or SCOTCH. Experimental results in Section 6 show that the cost induced by the separators clustering using METIS is however acceptable.

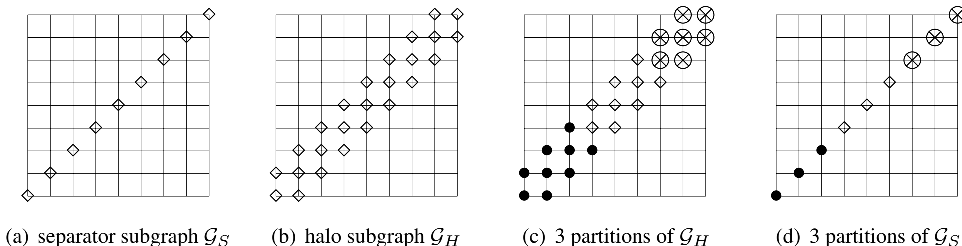
4.2 Separator clustering

Remember that two sets of variables are present in a front. An admissible partition of both sets must be computed. We consider $\mathcal{G}_S$, the graph induced by the nodes of the separator $\mathcal{S}$ corresponding to front F. A graph partitioning tool such as METIS or SCOTCH with a given partition size will compute BLR admissible blocks. However, the connectedness of $\mathcal{G}_S$ cannot be guaranteed. Figure 10(a) indeed shows the worst case of disconnected separator.

In this case, any partitioner cannot do anything else than computing ten groups of size one because only singletons would be given in input. This leads to a blocking made of 1×1 blocks, which are not compressible. In this extreme case, the clustering computed by the partitioning tool is completely useless.

Although in reality this would rarely occur, it is quite commonly the case where a separator is formed by multiple connected components that are close to each other in the global graph $\mathcal{G}$. This may lead to a sub-optimal clustering because variables that strongly interact due to their adjacency will end up in different clusters.

This problem may be overcome by reconnecting $\mathcal{G}_S$ in a way that takes into account the geometry and shape of the separator: variables close to each other in the original graph $\mathcal{G}$ have to be still close to each other in the reconnected $\mathcal{G}_S$. We describe an approach that achieves this objective by extending the subgraph induced by each separator with a halo formed by a relatively small number of level sets. The graph partitioning tool is therefore run on this extended graph and the resulting partitioning projected back on the original subgraph $\mathcal{G}_S$. Figure 10 shows how this is done on the example above using just one level set. For a limited number of level sets, the extended graph preserves the shape of the separator, keeps the cost of computing the clustering limited and allows to compute clusterings that better comply with the strategy presented in the previous section. In practice with complex physical domains, more layers may be needed to effectively reconnect the separators. On regular grids, we observed that two layers are enough to reconnect the separator and to obtain good performance.



(a) separator subgraph $\mathcal{G}_S$ (b) halo subgraph $\mathcal{G}_H$ (c) 3 partitions of $\mathcal{G}_H$ (d) 3 partitions of $\mathcal{G}_S$

Figure 10: Halo-based partitioning of $\mathcal{G}_S$ with depth 1. $\mathcal{G}_H$ is partitioned using METIS.

4.3 Explicit border clustering

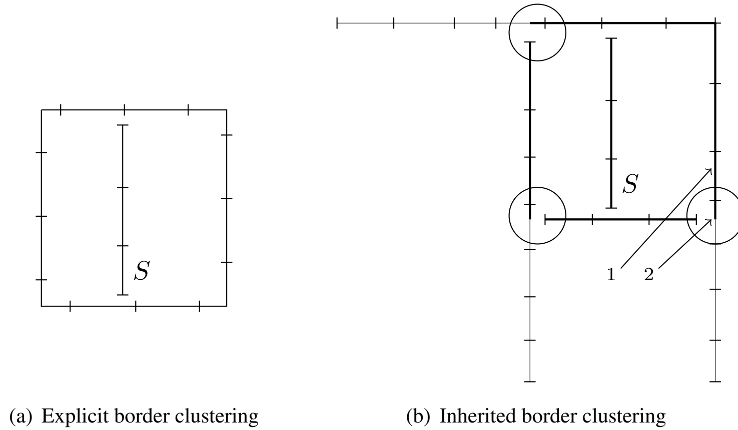
The previous section shows how the halo method can be employed to compute clusterings of the separators. This method can obviously be used for clustering any set of variables and, therefore, also those in the non fully-assembled part of the front that form a border around the corresponding separator (see Section 2). Figure 11(a) illustrates the way the clusters are computed using the halo method for both the separator and the border.

Although this approach will lead to an efficient clustering of the non fully-summed variables, it may suffer performance issues due to its high cost.

4.4 Inherited border clustering

As one can observe on Figure 11(b), one particular variable belongs to one separator which, in turn, may be (partially) included in multiple borders. Inversely, as Figure 2 illustrates, a border can be viewed as a union of parts of separators lying on the path that connects the related node to the root of the elimination tree. As a consequence, clustering the variables of all the separators is sufficient to obtain, by induction, a clustering of the borders.

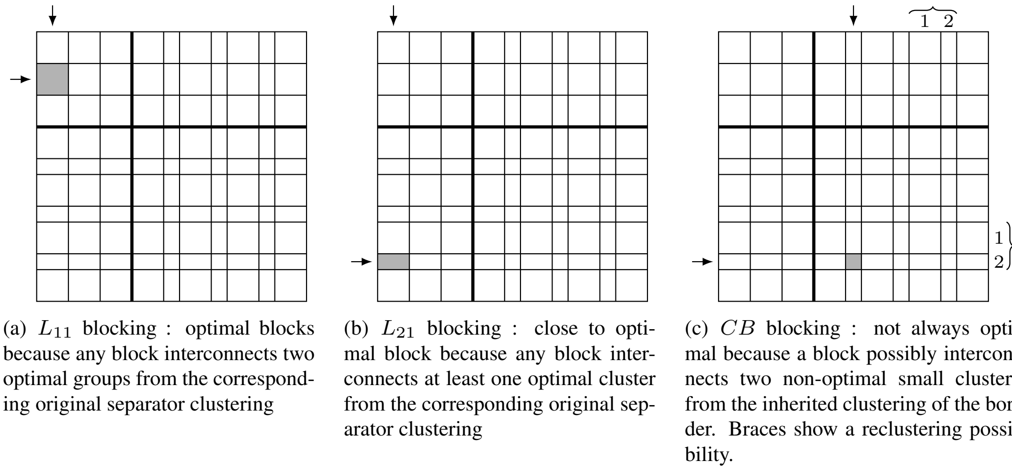
A top-down traversal of the separator tree is performed and the variables of each separator are clustered with the halo method proposed in Section 4.2. Then, the clustering of a given border is inherited from the



(a) Explicit border clustering (b) Inherited border clustering

Figure 11: Two different ways to perform the clustering of a border

clustering of all the separators it is made of, as shows Figure 11(b). Depending on where the separators intersect, small clusters, which are not admissible, may indeed be formed; as a result, the CB may include blocks which are too small to be effectively compressed and to achieve a good BLAS efficiency for the related operations. Note that this problem also affects the blocking of the L_{21} submatrix but to a lower extent because the effect is damped by the good clustering computed for the separator (see Figure 12).



(a) L_{11} blocking : optimal blocks because any block interconnects two optimal groups from the corresponding original separator clustering

(b) L_{21} blocking : close to optimal block because any block interconnects at least one optimal cluster from the corresponding original separator clustering

(c) CB blocking : not always optimal because a block possibly interconnects two non-optimal small clusters from the inherited clustering of the border. Braces show a reclustering possibility.

Figure 12: Relation between inherited clustering and front blocking. The small clusters correspond to clusters located in corners in Figure 11(b). The large clusters correspond to optimal clusters which are integrally kept in the current front. Note that not all the large clusters of Figure 11(b) are represented here.

To avoid having small blocks, reclustering strategies are being investigated. Figure 12(c) shows a basic strategy which merges neighbor clusters together in order to improve the BLAS efficiency as well as the compression gains.

It has to be noted that the inherited clustering also provides another convenient property: the blocking of a frontal matrix is compatible with the blocking of its parent front. This translates into the fact that one block of its Schur complement will be assembled into exactly one block of the parent front. This may considerably ease the assembly of frontal matrices especially in the parallel case where frontal matrices are distributed.

We will report in Section 6.3 on the effects of explicit and inherited clustering strategies and show related experimental results.

5 Block Low-Rank multifrontal method

This section describes how the BLR format can be integrated within a multifrontal solver in order to reduce its complexity (and, thus, its time to completion) as well as its memory footprint for solving problems with low-rank properties.

The previous sections describe how the BLR format possesses properties and features that are extremely favorable in a parallel fully functional general multifrontal solver. However, presenting and analysing such a solver is out of the scope of this work. The main objective of this article is, instead, to evaluate the potential of the BLR format and to show that, when used within a multifrontal solver, it is a valid alternative to other approaches such as those based on hierarchical formats.

BLR matrices can be efficiently employed to improve the multifrontal method. In particular, we will show how the fronts factorization (Section 5.2), where most of the computational time of the global execution is spent, can benefit from the BLR representation of the fronts. Clearly, using the BLR format also affects the way the fronts are assembled. This is described in Section 5.1

5.1 Assembly

The assembly phase of the BLR multifrontal method depends on how the BLR factorization (see Section 5.2) is done.

Because of their nature, assembly operations are relatively inefficient [6] and their cost can significantly affect the speed of the overall multifrontal factorization. In the experimental code we developed, we decided not to perform the assembly operations in BLR format in order to avoid an excessive amount of indirect addressing which may seriously impact performance. An analogous choice was also made in related work on the usage of HSS matrices within multifrontal solvers [33].

Frontal matrices are, thus, assembled in full-rank form and compressed progressively, by panels, as the partial front factorization proceeds, as shown in the next section. This, however, does not mean that contribution blocks are not compressed; despite the fact that compressing contribution blocks does not help reducing the overall number of floating point operations, storing them in low-rank form has a twofold advantage:

1. It allows to reduce the peak of active memory. In a sequential, multifrontal method, at any moment, the active memory is defined as the sum of the size of the CB stack(described in Section 2) plus the size of the front currently being factorized. Stacking the contribution blocks in low-rank form reduces the active memory. Although this makes the relative weight of the current front in the active memory higher, it has to be noted that if the partial front factorization is done in a left-looking fashion, frontal matrices can be assembled panelwise which means that in the active memory only one panel at a time is stored in full-rank. We have not implemented this feature yet, and, as discussed in the next section, the front factorization is still performed in a right-looking fashion.

2. In a parallel environment, it reduces the communications volume. In a parallel solver, frontal matrices are generally mapped onto different nodes which means that assembly operations involve transferring contribution blocks from one node to another. By storing contribution blocks in low-rank form, the total volume of communications can be reduced.

Because fronts are assembled in full-rank form, if a contribution block is in low-rank form, it has to be decompressed before the associated assembly takes place. This only increases the overall number of floating point operations by a small amount because the cost of converting to and from the BLR format is contained, as shown in Figures 8 and 9. Because it is not necessary to systematically compress the contribution blocks, in the experimental results of Section 6 the gains provided by this operation and the associated cost are always reported separately.

5.2 Factorization

Once a front has been assembled, a partial factorization is performed in order to compute the corresponding part of the global factors. These computations have a large influence on the global performance of the software. Moreover, the storage of these factor parts becomes larger and larger as matrix size grows. It is thus critical to improve this phase. We assume that clusterings $\mathcal{C}_{FS}$ for the FS variables and $\mathcal{C}_{NFS}$ for the NFS variables have been computed, which give a blocking of the front. In the full-rank standard case, four

fundamental tasks must be performed in order to compute the blocked incomplete factorization of the front F :

1.	factor (F)	$L_{\rho,\rho}L_{\rho,\rho}^T = F_{\rho,\rho}$	for $\rho \in \mathcal{C}_{FS}$.
2.	solve (S)	$L_{\tau,\rho} = F_{\tau,\rho}L_{\rho,\rho}^{-T}$	for $\rho \in \mathcal{C}_{FS}$, $\tau \in \mathcal{C}_{FS} \sqcup \mathcal{C}_{NFS}$, where $\tau > \rho$ ($\tau > \rho$ if $F_{\tau,\tau}$ appears after $F_{\rho,\rho}$ on the diagonal).
3.	internal update (U)	$F_{\tau,\sigma} = F_{\tau,\sigma} - L_{\tau,\rho}L_{\sigma,\rho}^T$	for $\rho, \sigma \in \mathcal{C}_{FS}$, $\tau \in \mathcal{C}_{FS} \sqcup \mathcal{C}_{NFS}$, where $\tau \geq \sigma > \rho$.
4.	external update (U)	$F_{\tau,\sigma} = F_{\tau,\sigma} - L_{\tau,\rho}L_{\sigma,\rho}^T$	for $\rho \in \mathcal{C}_{FS}$, $\tau, \sigma \in \mathcal{C}_{NFS}$, where $\tau \geq \sigma$.

We add a new task which corresponds to the compression of a block defined by the clusterings:

5.	compress (C)	$L_{\tau,\rho} \simeq U_{\tau,\alpha_1}V_{\rho,\alpha_2}^T$	for $\rho \in \mathcal{C}_{FS}$, $\tau \in \mathcal{C}_{FS} \sqcup \mathcal{C}_{NFS}$, where $\lvert\alpha_1\rvert = \lvert\alpha_2\rvert =$ numerical rank of $L_{\tau,\rho}$.

Due to the flexibility of the BLR format, several versions of the BLR factorization of a front can be implemented based on these five tasks, depending on the position of the compression. Figure 13 presents five different algorithms for the factorization of a front. Note that for the sake of simplicity, the order relations between σ, ρ and τ are now ignored. Similarly, the subscripts $_1$ and $_2$ in α_1 and α_2 will be omitted. Remember that U and V are both unitary only if a SVD is used for the compression. The first algorithm is the conventional full-rank partial factorization [17] and is called FSUU, which stands for *Factor, Solve, internal Update, external Update*. The other four are based on the BLR representation of the front and differ on when the compression is performed. As we move down the figure, from FSUUC to FCSUU, the computational cost decreases as we are performing the compression earlier and thus making more and more of an approximation to the factorization.

We decided to implement and study Algorithm FSCUU as it gives the best compromise between savings (for both memory and flops) and robustness of the solver:

- The number of operations needed for the solve task is much lower than for the internal and external updates so that we can focus on these two latter tasks.
- The approximation done in the factors is only due to approximations done at lower levels of the elimination tree. The solve task is done accurately.

The algorithms in Figure 13 target different objectives in the context of applicative solvers. For instance, FSUUC can be used to speed-up the solve phase based on a full off-line compression of the accurate factors, which, for example, may be beneficial in Newton type solvers where multiple solves have to be done based on the same factorization.

Note that a sixth task can be added which corresponds to the compression of the CB:

6.	external compress (C_e)	$F_{\tau,\sigma} \simeq U_{\tau,\alpha}V_{\sigma,\alpha}^T$	for $\tau, \sigma \in \mathcal{C}_{NFS}$.

This task can be performed at the end of any of the algorithms in Figure 13 in order to decrease the active memory size as well as the amount of communication necessary to assemble the parent front in a parallel context, as explained in the previous section.

5.3 Front selection

It is not necessary, nor efficient, to use low-rank approximations on all the frontal matrices. Indeed, to obtain good compression rates which overcome the compression costs, it is better to consider only larger fronts. How much work and memory is done and consumed in fronts larger than a given size can be easily assessed on a regular 9-point stencil by reusing some work and notations by Alan George [15] on nested dissection; the results of this analysis are shown in Figure 14. The elimination tree directly based on a complete nested dissection is used (i.e. a nested dissection where the smallest separators have size 1). However, in practice,

Figure 13: Standard and BLR factorizations of a front. The flexibility of the BLR format allows to define 4 different BLR factorization algorithms. When two different compressed blocks have to be used, variable β is used in addition to α.

FSUU : **no compression**

for $\rho \in \mathcal{C}_{FS}$ in ascending order
- F factor $F_{\rho,\rho} = L_{\rho,\rho} L_{\rho,\rho}^T$
- S solve $L_{\tau,\rho} = F_{\tau,\rho} L_{\rho,\rho}^{-T}$ **for** $\tau \in \mathcal{C}_{FS} \sqcup \mathcal{C}_{NFS}$
- U internal update $F_{\tau,\sigma} = F_{\tau,\sigma} - L_{\tau,\rho} L_{\sigma,\rho}^T$ **for** $\sigma \in \mathcal{C}_{FS}, \tau \in \mathcal{C}_{FS} \sqcup \mathcal{C}_{NFS}$
- U external update $F_{\tau,\sigma} = F_{\tau,\sigma} - L_{\tau,\rho} L_{\sigma,\rho}^T$ **for** $\tau, \sigma \in \mathcal{C}_{NFS}$

FSUUC : **compress after updates**

for $\rho \in \mathcal{C}_{FS}$ in ascending order
- F factor $F_{\rho,\rho} = L_{\rho,\rho} L_{\rho,\rho}^T$
- S solve $L_{\tau,\rho} = F_{\tau,\rho} L_{\rho,\rho}^{-T}$ **for** $\tau \in \mathcal{C}_{FS} \sqcup \mathcal{C}_{NFS}$
- U internal update $F_{\tau,\sigma} = F_{\tau,\sigma} - L_{\tau,\rho} L_{\sigma,\rho}^T$ **for** $\sigma \in \mathcal{C}_{FS}, \tau \in \mathcal{C}_{FS} \sqcup \mathcal{C}_{NFS}$
- U external update $F_{\tau,\sigma} = F_{\tau,\sigma} - L_{\tau,\rho} L_{\sigma,\rho}^T$ **for** $\tau, \sigma \in \mathcal{C}_{NFS}$
- C compress $L_{\tau,\rho} \simeq U_{\tau,\alpha} V_{\rho,\alpha}^T$ **for** $\tau \in \mathcal{C}_{FS} \sqcup \mathcal{C}_{NFS}$

FSUCU : **compress after internal and before external updates**

for $\rho \in \mathcal{C}_{FS}$ in ascending order
- F factor $F_{\rho,\rho} = L_{\rho,\rho} L_{\rho,\rho}^T$
- S solve $L_{\tau,\rho} = F_{\tau,\rho} L_{\rho,\rho}^{-T}$ **for** $\tau \in \mathcal{C}_{FS} \sqcup \mathcal{C}_{NFS}$
- U internal update $F_{\tau,\sigma} = F_{\tau,\sigma} - L_{\tau,\rho} L_{\sigma,\rho}^T$ **for** $\sigma \in \mathcal{C}_{FS}, \tau \in \mathcal{C}_{FS} \sqcup \mathcal{C}_{NFS}$
- C compress $L_{\tau,\rho} \simeq U_{\tau,\alpha} V_{\rho,\alpha}^T$ **for** $\tau \in \mathcal{C}_{FS} \sqcup \mathcal{C}_{NFS}$
- U external update $F_{\tau,\sigma} = F_{\tau,\sigma} - U_{\tau,\alpha} \left(V_{\rho,\alpha}^T V_{\rho,\beta} \right) U_{\sigma,\beta}^T$ **for** $\tau, \sigma \in \mathcal{C}_{NFS}$

FSCUU : **compress before updates**

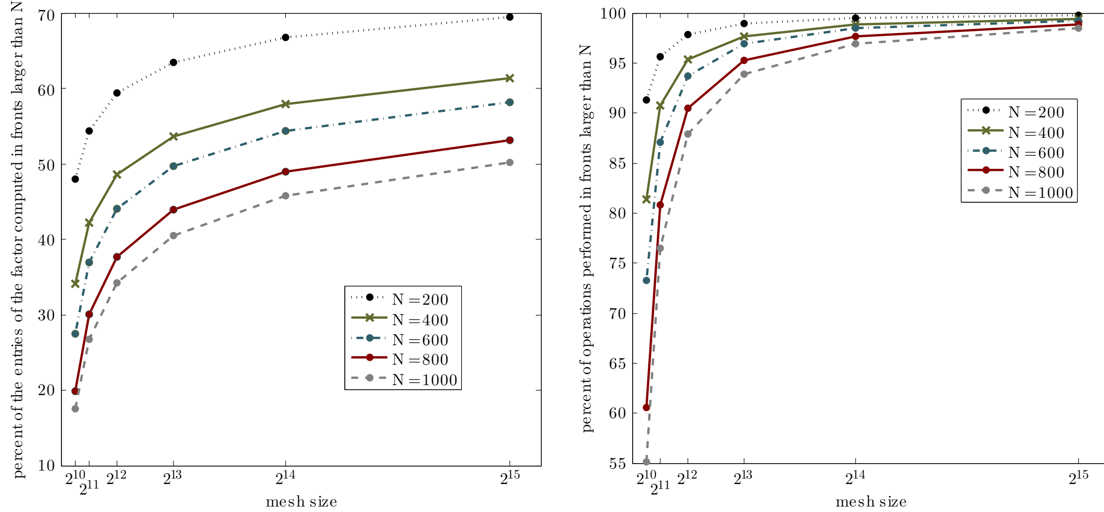
for $\rho \in \mathcal{C}_{FS}$ in ascending order
- F factor $F_{\rho,\rho} = L_{\rho,\rho} L_{\rho,\rho}^T$
- S solve $L_{\tau,\rho} = F_{\tau,\rho} L_{\rho,\rho}^{-T}$ **for** $\tau \in \mathcal{C}_{FS} \sqcup \mathcal{C}_{NFS}$
- C compress $L_{\tau,\rho} \simeq U_{\tau,\alpha} V_{\rho,\alpha}^T$ **for** $\tau \in \mathcal{C}_{FS} \sqcup \mathcal{C}_{NFS}$
- U internal update $F_{\tau,\sigma} = F_{\tau,\sigma} - U_{\tau,\alpha} \left(V_{\rho,\alpha}^T V_{\rho,\beta} \right) U_{\sigma,\beta}^T$ **for** $\sigma \in \mathcal{C}_{FS}, \tau \in \mathcal{C}_{FS} \sqcup \mathcal{C}_{NFS}$
- U external update $F_{\tau,\sigma} = F_{\tau,\sigma} - U_{\tau,\alpha} \left(V_{\rho,\alpha}^T V_{\rho,\beta} \right) U_{\sigma,\beta}^T$ **for** $\tau, \sigma \in \mathcal{C}_{NFS}$

FCSUU : **compress before solve**

for $\rho \in \mathcal{C}_{FS}$ in ascending order
- F factor $F_{\rho,\rho} = L_{\rho,\rho} L_{\rho,\rho}^T$
- C compress $F_{\tau,\rho} \simeq U_{\tau,\alpha} Z_{\rho,\alpha}^T$ **for** $\tau \in \mathcal{C}_{FS} \sqcup \mathcal{C}_{NFS}$
- S solve $L_{\tau,\rho} = U_{\tau,\alpha} (Z_{\rho,\alpha}^T L_{\rho,\rho}^{-T})$ **for** $\tau \in \mathcal{C}_{FS} \sqcup \mathcal{C}_{NFS}$
  i.e, $L_{\tau,\rho} = U_{\tau,\alpha} V_{\rho,\alpha}^T$ with $V_{\rho,\alpha}^T = Z_{\rho,\alpha}^T L_{\rho,\rho}^{-T}$
- U internal update $F_{\tau,\sigma} = F_{\tau,\sigma} - U_{\tau,\alpha} \left(V_{\rho,\alpha}^T V_{\rho,\beta} \right) U_{\sigma,\beta}^T$ **for** $\sigma \in \mathcal{C}_{FS}, \tau \in \mathcal{C}_{FS} \sqcup \mathcal{C}_{NFS}$
- U external update $F_{\tau,\sigma} = F_{\tau,\sigma} - U_{\tau,\alpha} \left(V_{\rho,\alpha}^T V_{\rho,\beta} \right) U_{\sigma,\beta}^T$ **for** $\tau, \sigma \in \mathcal{C}_{NFS}$

the nested dissection is stopped before the separators reach size 1 and the corresponding elimination tree is post-processed with different techniques, such as *amalgamation*, which aim at merging fronts in order to increase efficiency. The resulting tree is called the *assembly* tree and contains basically less fronts but larger ones. For this reason, in a practical context, the graphs shown in Figure 14(a) would be translated upwards. This remark should be taken into account when reading the analysis below.

Figures 14(a) and 14(b) show that most of the factor entries are computed and almost all of the floating point operations are done within fronts of relatively large size. This is particularly interesting considering that these larger fronts only account for a very small fraction of the total number of fronts in the elimination tree, as shown in Table 2.



(a) Proportion of entries of the factors computed in the top levels of the multifrontal tree.

(b) Proportion of the total number of operations performed for the partial factorization of the top levels of the multifrontal tree.

Figure 14: Proportions of entries of the factors and of number of operations in the top levels of the multifrontal tree. The problem studied is a 2D Laplacian on a 9-point stencil. The corresponding matrix size is the square of the mesh size. Note that the largest fronts we look at in these plots represent very few fronts compared to the total number of fronts, as shows Table 2.

Table 2: Proportion of fronts larger than N with respect to the total number of fronts in the elimination tree, for different mesh sizes.

	mesh size					
	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}
$N = 200$	1.61‰	1.68‰	1.71‰	1.73‰	1.74‰	1.74‰
$N = 400$	0.37‰	0.40‰	0.42‰	0.42‰	0.43‰	0.43‰
$N = 600$	0.17‰	0.20‰	0.21‰	0.22‰	0.23‰	0.23‰
$N = 800$	0.07‰	0.09‰	0.10‰	0.11‰	0.11‰	0.11‰
$N = 1000$	0.05‰	0.06‰	0.06‰	0.06‰	0.06‰	0.06‰

This shows that there is no need to compress small fronts for two reasons:

1. It would be too much work for a small gain (in small fronts we can only target a few entries and a few computations, and moreover the SVD compression has a cost) ;

2. It is more critical to focus on the top of the tree because the corresponding large fronts represent most of the work to be done in the multifrontal process, in terms of computations and memory.

6 Experiments

This section presents experimental results on the set of problems described in Section 6.1. Our methods have been developed and incorporated in the general purpose symmetric and unsymmetric sparse linear solver `MUMPS` [2] (in complex and real arithmetic) but can be applied to any multifrontal solver.

- For each test matrix, we study a linear system whose right-hand side is a vector of ones. To measure the effectiveness of our approach, several metrics are presented in Section 6.2.
- The global ordering used for `MUMPS` is METIS [24], although experiments have shown comparable results with other global orderings such as AMD [3].
- Only fronts larger than 100 are candidate for compression. The halo depth used during the clustering phase is 2. Inherited clustering with cluster size set up to 200 is used, except in Section 6.3 where the clustering strategy is studied.
- The compression is achieved through a truncated rank-revealing QR factorization (modified from Lapack's `CGEQP3`) with an absolute low-rank threshold. We are currently investigating the efficiency of a relative dropping in terms of compression, stability and accuracy.
- Results are obtained with a sequential code and only from the factorization phase. For the solve phase, the flops reduction is equal to the factor memory reduction. The supercomputer used for these experiments is a SMP Altiv UV (ccNUMA architecture) equipped with 48 WESTMERE EX octo-core processors @2.67GHz and 3TB of RAM.

6.1 Set of problems

The experiments presented throughout this paper have been run on a set of problems coming from different physics applications. These matrices will be used all along this section to illustrate each aspect of our work and are described in Table 3. They correspond to four important classes of applications within the field of elliptic PDEs.

Name	Prop.	Arith.	N ($\times 10^6$)	NZ ($\times 10^9$)	factors storage	flops ($\times 10^{12}$)	CSR (full rank)	application field
`Curl5000`	2D/sym.	D	50	0.2	27 GB	5	2×10^{-15}	electromag.
`Geoazur128`	3D/unsym.	Z	2	55	46 GB	62	2×10^{-12}	wave prop.
`TH_RAFF7`	3D/sym	D	8	118	138 GB	100	8×10^{-15}	thermal
`ME_RAFF12`	2D/sym.	D	134	1	215 GB	200	4×10^{-15}	mechanical

Table 3: Set of problems used for the experimentations. They are finite-difference or finite elements methods simulations. CSR = Componentwise Scaled Residual = $\max_i \frac{|\mathbf{b} - \mathbf{A}\bar{\mathbf{x}}|_i}{(|\mathbf{b}| + |\mathbf{A}|\,|\bar{\mathbf{x}}|)_i}$. D=double precision real, Z=double precision complex.

`Curl5000` is an academic problem and corresponds to the curl-curl operator, which is widely used in computational electromagnetism. `Geoazur128` matrix is a complex-valued impedance matrix resulting from the finite-difference discretization of the heterogeneous Helmholtz equation which is the second-order visco-acoustic time-harmonic wave equation for pressure p. The aim is the modeling of visco-acoustic wave propagation in a 3D visco-acoustic medium parameterized by wavespeed, density and quality factor. For more details about these matrices, see Operto [25]. The last two matrices are thermal and mechanical simulations from the French Electricity Company (EDF). For matrix `ME_RAFF12`, we will also study different mesh refinements of the model to analyse the impact on the LR properties of the matrix. Finally, various sizes of a 3D `Laplacian` problem defined on an 11-point stencil are also used to demonstrate the potential of a BLR based factorization.

6.2 Metrics

The effectiveness of the proposed techniques will be measured with different metrics mostly related to the reduction of memory and flops:

- The factor compression $|L|$ is defined as the ratio of the number of entries needed to store the factor computed with the BLR approach over the number of entries needed to store the regular factor.
- The maximum size of CB stack compression $|CB|$ is the ratio of the maximum size of the CB stack with the BLR approach over the maximum size of the CB stack with a regular full rank approach. For a definition of the *CB stack*, please refer to Section 2.
- For each task (or tasks combination) defined in Section 5.2, we will indicate either the corresponding absolute low-rank flop count (execution time), either the corresponding flops (execution time) compression as a percentage of the full-rank factorization (`FR` facto) flop count (execution time).

  For instance, a column called "F+S" shows absolute data or percentages related to flops or execution time (it is always indicated) related to the "Factorization" and "Solve" tasks of the algorithm.

6.3 Clustering strategies

Table 4 reports experimental results related to the clustering strategies. As the problem size grows, the explicit border clustering time becomes excessively time consuming. Experiments confirm that the inherited clustering is 2 to 8 times faster than the explicit border clustering, with a comparable efficiency. The overhead due to this specific phase is very low compared to the full rank analysis time. In pratice, a basic reclustering strategy (as explained in Section 4) can be performed in a negligible time. It improves slightly the compression rates and substantially the BLAS operations, which was our target.

Table 4: Comparison of the efficiency of the explicit and inherited clustering strategies. Each *inh* column gives results with the clustering done during the analyis. Each *exp* column gives results with the clustering done during the factorization. The *clustering time* is the total time spent for the clustering of all the fronts. The low-rank threshold is 10^{-8}. The *full rank analysis time* is the time spent in the analysis phase for a regular full rank multifrontal factorization (excluding clustering time).

	memory				flops		time		
	$\|L\|$		$\|CB\|$		F+S+C+U+U		clustering		full rank analysis
clustering	inh	exp	inh	exp	inh	exp	inh	exp	
`Curl5000`	63.7%	62.4%	7.0%	5.5%	10.9%	11.1%	13 s	27 s	897 s
`Geoazur128`	79.0%	77.0%	47.0%	45.0%	60.8%	59.1%	5 s	42 s	62 s
`TH_RAFF7`	34.1%	30.7%	17.5%	16.2%	7.2%	6.6%	34 s	206 s	387 s
`ME_RAFF12`	52.9%	51.1%	4.8%	4.1%	6.1%	6.1%	121 s	239 s	1971 s

6.4 Influence of the cluster size

By definition of the BLR admissibility condition described in Section 4.1, a target block size has to be chosen in order to define suitable admissible blocks. For a general algebraic multifrontal solver, having such a parameter can be a constraint if it has to be tuned finely to ensure a good efficiency of the method for a given problem. Although block sizes of 100 or 150 are too small to achieve good compression rates, results presented in Table 5 show that the dependence between compression and block size is usually quite small, so that a variation in the block size does not have a critical impact on the global efficiency of the method.

This property is very desirable because it will ease the unsymmetric (and symmetric indefinite) pivoting, which can in some cases modify dynamically the number of fully summed variables within a front. From a more general point of view, block sizes can also be modified by reclustering (see Section 4), or because of out-of-core and parallelism, which are already strong algorithmic constraints in a general purpose sparse solver. Thus, the flexibility in the block sizes is very desirable to suppress a new constraint.

Table 5: Influence of the block size used for the BLR compression on the efficiency of the BLR multifrontal method. Study for Geoazur128 with basic reclustering and low-rank threshold set to 10^{-8}.

	memory		flops		
block size	$\|L\|$	$\|CB\|$	F+S+C+U+U	F+S+U+U	C
100	88%	55%	75%	4.6E+13	9.8E+11
150	83%	50%	68%	4.1E+13	1.5E+12
200	79%	46%	63%	3.7E+13	1.9E+12
250	76%	45%	60%	3.4E+13	2.4E+12
300	75%	44%	58%	3.3E+13	2.7E+12
350	74%	44%	56%	3.2E+13	3.1E+12
400	73%	45%	56%	3.1E+13	3.4E+12
450	73%	44%	56%	3.1E+13	3.7E+12
500	73%	44%	57%	3.1E+13	4.4E+12

6.5 Memory, flops and accuracy

In this section we present global results related to memory, flops and accuracy with different low-rank thresholds. Figure 15 shows the cost and memory occupancy of the BLR based solver relative to the full-rank one; for instance, for problem TH_RAFF7 with $\varepsilon = 10^{-14}$, the number of operations needed for the factorization is divided by 5, the maximum size of CB stack by 4 and the factor memory by 2. In the figure, also the accuracy of the computed solution is reported, measured in terms of Componentwise Scaled Residual [29, 23, 5], defined as $CSR = \max_i \frac{|\mathbf{b} - \mathbf{A}\bar{\mathbf{x}}|_i}{(|\mathbf{b}| + |\mathbf{A}|\,|\bar{\mathbf{x}}|)_i}$.

The accuracy of the solution follows the low-rank threshold used for the compression. No propagation is observed and ε gives to the user a good control over the numerical behaviour of the solver.

As far as memory compression is concerned, the method shows good efficiency at high to middle accuracy. At low accuracy, the results are even better. With the ME_RAFF12 and TH_RAFF7 matrices, good compression rates are also obtained at full accuracy. At full accuracy, the compression of the maximum size of CB stack ($|CB|$) can be quite interesting even when we do not have a global gain in terms of factor compression ($|L|$) and flops reduction (F+S+C+U+U flops), see Figure 15(a). This is probably due to the fact that the full-rank maximum size of CB stack is reached on large fronts on which the compression of the CB block is quite large. It is an interesting feature which could be exploited to limit the memory requirement for the factorization or to compress the multifrontal CB stack on demand.

In terms of computational cost compression, gains are almost always higher than those due to memory compression for two reasons:

- as shows Figure 14, for a given front size, the top level fronts concentrate a higher flops ratio than entries in factor ratio (with respect to the global flops and entries in factor).
- operations on blocks have a cubic complexity whereas memory has a square complexity.

For matrix Curl5000, with $\varepsilon = 10^{-14}$ and $\varepsilon = 10^{-12}$, we do slightly more operations than in a usual multifrontal solver because of the additional cost for QR factorizations performed to reveal uncompressable subblocks ($|L|$ = 100%).

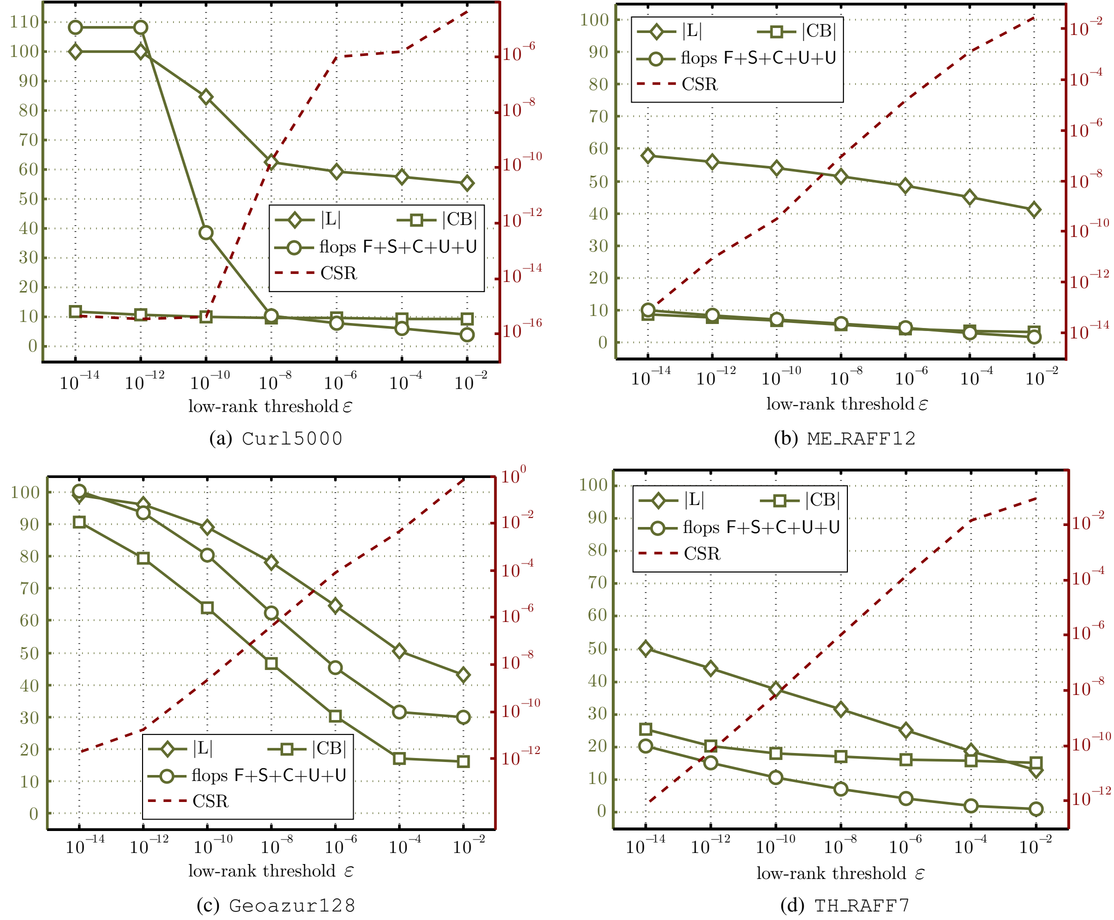


Figure 15: Global results of the BLR methods implemented within MUMPS. The solver is fully algebraic. Compression results are shown for the memory needed to store the factors and the contribution blocks, and for the number of operations needed to perform the factorization (left-hand side vertical axis). Results related to the accuracy of the computed solution (right-hand side vertical axis) are also given. For the exact definition of the metrics used in these graphs, please refer to Section 6.2.

6.6 Scalability with respect to the problem size

The efficiency of low-rank approximation techniques on multifrontal factorizations can strongly depend on the size of the problem. Table 6 illustrates this idea on different mesh refinements of the applicative problem class ME_RAFF$\star$. The efficiency of the low-rank methods increases when refining the mesh.

Table 6: Overview of the influence of the mesh refinement degree on the memory and flops reductions. The low-rank threshold is set up to 10^{-8}.

		memory		flops
Refinement	N	$\lvert L\rvert$	$\lvert CB\rvert$	F+S+C+U+U
ME_RAFF8	$2,101,258$	71%	27%	32%
ME_RAFF11	$33,570,826$	59%	11%	12%
ME_RAFF12	$134,250,506$	52%	7%	6%

In order to further investigate this behavior, a full set of tests were conducted on matrices of increasing size from the Laplacian operator discretized with a 3D 11-point stencil. Table 7 shows how the effec-

tiveness of low-rank techniques obtained with the BLR format increases with the mesh size, in terms of memory, flops and time reductions.

Table 7: Scalability of the BLR factorization with respect to the mesh size M. Results are given for a matrix coming from the `Laplacian` operator discretized with a 3D 11-point stencil. The low-rank threshold is set up to 10^{-14} so that a full accuracy is kept. *FR facto* is a full rank factorization obtained with `MUMPS`. The percentages are given with respect to *FR facto*. C_e can be viewed as the price to pay to obtain the memory reduction $|CB|$. See Section 6.2 for more details about the metrics.

	flops					memory	
					low-rank CB		
M	FR facto	F+S+C+U+U		C	C_e	$\|CB\|$	$\|L\|$
32	1.2E+10	1.1E+10	94.0%	17.5%	22.4%	85.6%	94.5%
64	7.8E+11	3.0E+11	38.2%	5.3%	11.8%	49.1%	70.0%
96	9.1E+12	1.6E+12	17.9%	2.0%	6.2%	26.5%	52.5%
128	5.2E+13	5.0E+12	9.7%	0.9%	3.2%	17.4%	41.9%
160	2.1E+14	1.2E+13	5.7%	0.5%	1.7%	12.3%	33.5%
192	6.4E+14	2.5E+13	3.9%	0.3%	1.1%	8.8%	28.2%
224	1.6E+15	4.5E+13	2.8%	0.2%	0.8%	7.5%	24.5%
256	3.6E+15	8.1E+13	2.2%	0.1%	0.5%	6.2%	21.2%

	time (in s.)				
M	FR facto	F+S+C+U+U		C	C_e
32	1.6	2.1	131.3%	50.0%	43.8%
64	99.3	55.2	55.6%	15.1%	23.8%
96	1133.2	327.0	28.9%	6.0%	12.7%
128	6351.4	1148.0	18.1%	2.9%	6.6%
160	25742.8	3415.6	13.3%	1.6%	3.9%
192	78049.0*	8607.0	11.0%	1.0%	2.6%
224	195120.0*	19496.3	9.9%	0.8%	1.8%
256	439020.0*	40221.7	9.2%	0.7%	1.2%

For a mesh size of 160, a factor of almost 20 is obtained in terms of flops reduction. The flop counts reveal an $O(N^{4/3})$ complexity, which is comparable to what is obtained in 3D with an HSS solver [31, 33].

The overhead due to the compression of CBs appears to be globally very small, which is critical to decrease the active memory consumption. The same behavior is observed in terms of memory: for mesh size of 160, the size of the factors has been reduced by a factor of 3 and the maximum size of CB stack by a factor of almost 10.

The second part of Table 7 shows how the flops reduction is translated into time reduction. Results show that a large part of the flops reduction is converted into time reduction. When the flops compression rate is important, the benefits of BLAS 3 operations are partly lost because of the very low rank of the blocks, which explains why the efficiency decreases slightly. However, for large enough mesh sizes, very interesting time reductions are obtained (a factor of 7 is already obtained for mesh size 160), that are increasing with the size of the problem.

7 Conclusion

A new format for low-rank multifrontal solvers has been presented. More flexible than other low-rank matrix structures, the BLR format shows a high efficiency on applicative problems, both in terms of memory and flops reduction. The low-rank threshold ε gives a full control on the global accuracy of the solver and no error propagation has been observed. We showed that this format is very suitable for the algorithmic requirements of a robust, general multifrontal solver, and presents some distinctive features over hierarchical

formats such as HSS and $\mathcal{H}$ that make its usage more immediate and, likely, more effective. By means of an experimental comparison, we showed that the gains achieved with the BLR format are comparable to those obtained with hierarchical formats. Through this comparison we also showed that the cost of constructing the low-rank representation is much lower in the case of the BLR format with respect to the HSS and $\mathcal{H}$ approaches; this provides an additional property which is very favorable in the context of a sparse, multifrontal solver involving operations with complex data access patterns. We proposed a method for computing the blocking of frontal matrices for the BLR format; thanks to the properties of BLR, this technique does not require any knowledge of the geometry of the problem and can be run at a cost which is marginal relative to the cost of the analysis phase of a multifrontal solver.

In order to evaluate the potential of the proposed techniques, we integrated the BLR format and the blocking technique in the sequential version of the MUMPS solver and we ran experiments on a number of problems from real world applications as well classical, textbook problems. The experimental results show that considerable gains can be achieved already at very accurate approximation levels, in which case the final solution backward error is comparable to the one obtained with a standard, full-rank solver. Experimental results also show that the BLR format is tolerant with respect to variations in the size of the blocks – a property that may be used to accommodate the BLR format of frontal matrices to data distribution in a parallel environment as well as to classical pivoting techniques.

The work presented in this document lays the foundations of an ongoing research effort which aims at exploiting the BLR format within a parallel, fully featured multifrontal solver.

8 Acknowledgement

We wish to thank Xiaoye S. Li (LBNL) and Artem Napov (LBNL) for the helpful discussions on low-rank techniques and for the experimental HSS code used in Section 3.

References

[1] P. R. Amestoy, A. Buttari, I. S. Duff, A. Guermouche, J.-Y. L'Excellent, and B. Uçar. The multifrontal method. In David Padua, editor, *Encyclopedia of Parallel Computing*. Springer, http://www.springerlink.com, 2010.

[2] P. R. Amestoy, A. Buttari, I. S. Duff, A. Guermouche, J.-Y. L'Excellent, and B. Uçar. MUMPS (MUltifrontal Massively Parallel Solver). In David Padua, editor, *Encyclopedia of Parallel Computing*. Springer, http://www.springerlink.com, 2010.

[3] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, 1996.

[4] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. MUMPS: a general purpose distributed memory sparse solver. In A. H. Gebremedhin, F. Manne, R. Moe, and T. Sørevik, editors, *Proceedings of PARA2000, the Fifth International Workshop on Applied Parallel Computing, Bergen, June 18-21*, pages 122–131. Springer-Verlag, 2000. Lecture Notes in Computer Science 1947.

[5] M. Arioli, J. Demmel, and I. S. Duff. Solving sparse linear systems with sparse backward error. *SIMAX*, 10:165–190, 1989.

[6] C. Ashcraft and R. Grimes. The influence of relaxed supernode partitions on the multifrontal method. *ACM-TOMS*, 15:291–309, 1989.

[7] M. Bebendorf. *Hierarchical Matrices: A Means to Efficiently Solve Elliptic Boundary Value Problems (Lecture Notes in Computational Science and Engineering)*. Springer, 1 edition, 2008.

[8] S. Börm. *Efficient Numerical Methods for Non-local Operators*. European Mathematical Society, 2010.

[9] S. Chandrasekaran, P. Dewilde, M. Gu, and N. Somasunderam. On the numerical rank of the off-diagonal blocks of Schur complements of discretized elliptic PDEs. *SIAM Journal on Matrix Analysis and Applications*, 31(5):2261–2290, 2010.

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

[11] T. A. Davis. Algorithm 832: `UMFPACK V4.3` — an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30(2):196–199, 2004.

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

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

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

[15] A. George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, pages 345–363, 1973.

[16] A. Gillman, P. Young, and P.-G. Martinsson. A direct solver with $\mathcal{O}(N)$ complexity for integral equations on one-dimensional domains. *Frontiers of Mathematics in China*, 7:217–247, 2012. 10.1007/s11464-012-0188-3.

[17] G. Golub and C. Van Loan. *Matrix computations*. Johns Hopkins University Press, 3 edition, 1996.

[18] N. I. M. Gould and J. A. Scott. A numerical evaluation of HSL packages for the direct solution of large sparse, symmetric linear systems of equations. *ACM Transactions on Mathematical Software*, 30(3):300–325, 2004.

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

[20] A. Gupta, F. Gustavson, M. Joshi, G. Karypis, and V. Kumar. `PSPASES`: An efficient and scalable parallel sparse direct solver. *Kluwer International Series in Engineering and Computer Science*, 515, 1999.

[21] A. Gupta and M. Joshi. `WSMP`: A high-performance shared- and distributed-memory parallel sparse linear equation solver, 2001.

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

[23] D. J. Higham and N. J. Higham. Componentwise perturbation theory for linear systems with multiple right-hand sides. *Linear algebra and its applications*, 174:111–129, 1992.

[24] G. Karypis and V. Kumar. `MeTiS`*: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 4.0*. University of Minnesota, Army HPC Research Center, Minneapolis, 1998.

[25] S. Operto, J. Virieux, P. R. Amestoy, J.-Y. L'Excellent, L. Giraud, and H. Ben Hadj Ali. 3D finite-difference frequency-domain modeling of visco-acoustic wave propagation using a massively parallel direct solver: A feasibility study. *Geophysics*, 72(5):195–211, 2007.

[26] F. Pellegrini. `Scotch` and `libscotch 5.0` User's guide. Technical Report, LaBRI, Université Bordeaux I, 2007.

[27] P. Raghavan. `DSCPACK`: Domain-separator codes for the parallel solution of sparse linear systems. Technical Report CSE-02-004, Department of Computer Science and Engineering, The Pennsylvania State University, 2002.

[28] R. Schreiber. A new implementation of sparse Gaussian elimination. *ACM Transactions on Mathematical Software*, 8(3):256–276, 1982.

[29] R. D. Skeel. Scaling for numerical stability in Gaussian elimination. *Journal of the ACM (JACM)*, 26(3):494–526, 1979.

[30] S. Wang, M. V. de Hoop, and J. Xia. On 3D modeling of seismic wave propagation via a structured parallel multifrontal direct Helmholtz solver. *Geophysical Prospecting*, 59(5):857–873, 2011.

[31] S. Wang, M. V. de Hoop, J. Xia, and X. S. Li. Massively parallel structured multifrontal solver for time-harmonic elastic waves in 3D anisotropic media. *Geophysical Journal International*, 2012.

[32] S. Wang, X. S. Li, J. Xia, Y. Situ, and M. V. De Hoop. Efficient scalable algorithms for hierarchically semiseparable matrices. *Submitted SIAM Journal on Scientific Computing*, 2012.

[33] J. Xia. Efficient structured multifrontal factorization for general large sparse matrices. *Submitted to SIAM Journal on Scientific Computing*, 2012.

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

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