Supernodes ordering to enhance Block Low-Rank compression in sparse direct solvers

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

Supernodes ordering to enhance Block Low-Rank compression in sparse direct solvers

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

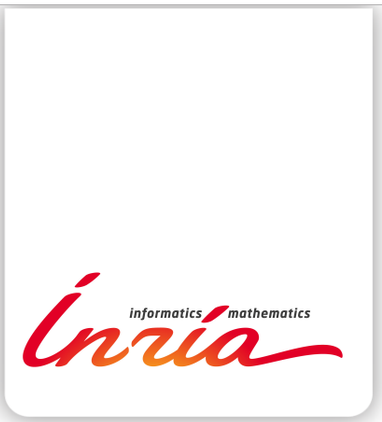
RESEARCH REPORT

N° 9238

December 2018

Project-Team HiePACS

ISSN 0249-6399 ISRN INRIA/RR--9238--FR+ENG



Supernodes ordering to enhance Block Low-Rank compression in sparse direct solvers

Grégoire Pichon[*†‡], Eric Darve[§], Mathieu Faverge[¶*‡], Pierre Ramet [†*‡], Jean Roman [*¶‡]

Project-Team HiePACS

Research Report n° 9238 — December 2018 — 28 pages

Abstract: Solving sparse linear systems appears in many scientific applications, and sparse direct linear solvers are widely used for their robustness. Still, both time and memory complexities limit the use of direct methods to solve larger problems, while the amount of memory available per computational units is decreasing in modern architectures. In order to tackle this problem, low-rank compression techniques have been introduced in direct solvers to compress large dense blocks appearing in the symbolic factorization. In this paper, we consider the Block Low-Rank (BLR) compression format and address the problem of clustering unknowns that come from separators issued from the nested dissection process. We show that methods considering only intra-separators connectivity (i.e., k-way or recursive bisection) as well as methods managing only interaction between separators have limitations. We propose a new strategy that considers interactions between multiple levels of the elimination tree of the nested dissection. This strategy tries to both reduce the number of off-diagonal blocks in the symbolic structure and increase the compression ratio of the large separators. We demonstrate how this new method enhances the BLR strategies in the sparse direct supernodal solver PaStiX.

Key-words: Sparse linear solver, low-rank compression, ordering, clustering

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

RESEARCH CENTRE BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Renumérotation des supernoeuds pour optimiser la compression de rang faible dans les solveurs directs creux

Résumé : La résolution de systèmes linéaires creux est utilisée dans de nombreuses applications scientifiques, et les solveurs directs creux sont réputés pour leur robustesse. Néanmoins, les complexités en temps et en mémoire limitent l'utilisation de ces méthodes pour résoudre des problèmes de très grande taille. Afin de s'attaquer à ce problème, des techniques de compression de rang faible ont été introduites dans les solveurs directs pour compresser les blocs denses apparaissant lors de l'étape de factorisation symbolique. Dans cette étude, nous considérons le format de compression Bloc Low-Rank (BLR) et nous abordons le problème du regroupement 'clustering' des inconnues dans les séparateurs issus de la méthode de dissection emboitée. Nous mettons en évidence les limitations des méthodes ne prenant en compte que les connectivités internes à un séparateur (c.-à-d. k-way ou dissection récursive) ainsi que des méthodes qui n'optimisent que les interactions entre séparateurs. Nous proposons une nouvelle stratégie qui prend en compte les interactions entre plusieurs niveaux de l'arbre d'élimination de la dissection emboitée. Cette stratégie essaye de réduire à la fois le nombre de blocs extra-diagonaux dans la structure symbolique et d'augmenter le taux de compression des séparateurs les plus gros. Nous démontrons que cette nouvelle méthode permet d'améliorer la compression BLR dans le solveur supernodal direct creux PaStiX.

Mots-clés : Solveur linéaire creux, compression de rang faible, renumérotation, regroupement

1 Introduction

Many scientific applications use numerical models that require to solve linear systems of the form $Ax = b$, where the matrix A is sparse and large. A classic approach to solve these problems is to factorize the matrix into a product of triangular matrices before solving triangular systems. This direct approach has some limitations, because both memory requirements and time-to-solution grow more than linearly with the problem size.

In order to reduce the complexities of direct sparse linear solvers, many recent works have investigated the low-rank representations of dense blocks appearing during the sparse matrix factorization. By compressing those blocks through many possible compression formats such as Block Low-Rank (BLR) [1–3], Hierarchical ($\mathcal{H}$) [4–6], $\mathcal{H}^2$ [7], Hierarchically Semi-Separable (HSS) [8–11], Hierarchical Off-Diagonal Low-Rank (HODLR) [12,13], ..., both memory and time complexities can be reduced.

The common approach to solve a sparse system is divided into four main steps: 1) ordering of the unknowns, 2) block-symbolic factorization, 3) numerical block-factorization, and 4) triangular system solves. The ordering of the unknowns aims at minimizing the fill-in to reduce both the memory consumption and the number of operations to factorize the matrix, and to provide sufficient parallelism. It is usually performed with the nested dissection algorithm [14] and partitioning libraries such as Metis [15] or Scotch [16]. When performing nested dissection on a graph G, one wants to express G as $G_A \cup G_B \cup G_C$, where G_C is named the separator and such that there is no path between G_A and G_B vertices, except going through G_C vertices. The objective is to compute G_C as minimal as possible and such that $|G_A| \simeq |G_B|$. This process will be recursively applied on G_A and G_B, until reaching small enough subgraphs to apply local ordering strategies. From this process, each separator or underlying subpart is named a supernode. One default of the current ordering strategies for compression techniques is that the resulting ordering is not fit to compress correctly the supernodes.

Given this partitioning, the second step, the block-symbolic factorization, predicts the structure of the factorized matrix (L), in such a way that the data structure that will hold the factorized matrix can be allocated before any numerical operations. The objective of this step is also to represent the matrix as a set of blocks instead of scalars, to allow the use of efficient BLAS Level 3 operations [17]. Given this block-structure, numerical operations—factorization and solves—can be performed more efficiently. One main challenge of sparse direct solvers is to correctly manage the sparsity pattern to obtain reasonable blocking sizes and to reach good level of efficiency on modern architectures. It becomes an even harder problem when introducing low-rank compression techniques that may degrade existing blocking strategies.

For extending low-rank compression techniques that were designed for dense matrices to the sparse case, most algebraic sparse low-rank solvers use a common strategy: from the block-symbolic factorization, diagonal and off-diagonal blocks that are large enough are represented in a flat (BLR) or hierarchical ($\mathcal{H}$, $\mathcal{H}^2$, HSS, HODLR) format.

The low-rank clustering consists into splitting unknowns of a separator among clusters to exhibit the flat or hierarchical representation of a dense block. More precisely, its objective is to form clusters that are well separated such that most of the interactions are low-rank. As separators issued from the nested dissection can be reordered without modifying the fill-in, a suitable reordering within separators can be computed to better cluster unknowns. For the dense case, the clustering has to maximize compressibility, while in the sparse case it also impacts the granularity of the data structures, which makes the clustering of sparse matrices a challenging problem. In addition, if finding a clustering can be straightforward in a geometric context, it is a more challenging problem in a purely algebraic context (where only the adjacency graph of the

matrix is known).

As mentioned in [18], one cannot classify vertices of a separator among sets receiving exactly the same contributions. It would result in clusters of size $O(1)$ by considering all interactions in the elimination tree. Indeed, for a graph split into $A \cup B \cup C$ with C the separator, subparts A and B are ordered independently, which increases the number of possible sets of contributions a vertex can receive. For instance, let us consider only one level of children (two children), and the corresponding projections $A_A \cup B_A \cup C_A$ (respectively $A_B \cup B_B \cup C_B$) for the subpart A (respectively B). As, by definition of the nested dissection process, C is connected to both A and B subparts, vertices belonging to the separator can be classified into nine different parts (combination of each kind of vertex for each subpart). For a large number of children, this number grows quickly, so it is impossible to classify unknowns among clusters with this approach.

In order to highlight the issue of clustering separators on a simple example, let us consider a regular cube and two levels of nested dissection, as presented in Figure 1. Note that it is a perfect nested dissection, in the sense that separators are as small as possible and split vertices among perfectly balanced subparts. The gray plane corresponds to the graph of the first separator and the green and red planes correspond to second-level separators. On the left, the graph of the gray separator is illustrated and vertices that are directly connected to vertices of second-level separators are colored in green and red. All throughout the paper, we named those vertices traces.

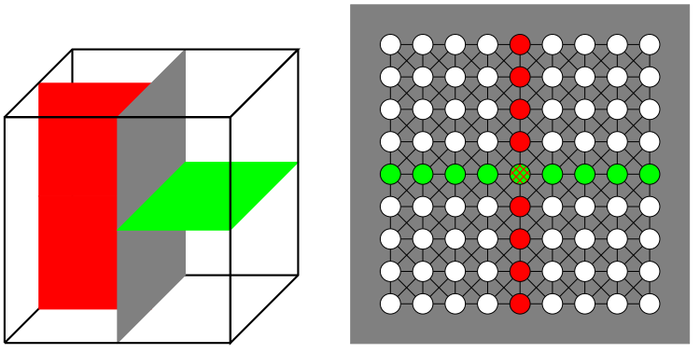


Figure 1: Two levels of nested dissection on a regular cube (on left) and traces (green and red) of second-level separators on the first separator (on right).

A classical approach used in MUMPS [1, 2] and STRUMPACK [8, 9] multifrontal solvers is to use k-way partitioning in fronts to obtain the clustering. The main issue of such an approach is that it correctly considers interactions within a separator but not from outside the separator. In practice, if we consider the sparse matrix corresponding to the cube in Figure 1, the approach considers only vertices of the gray separator and orders those vertices without considering upcoming contributions. One can note that a k-way partitioning will divide vertices corresponding to interaction with direct children (red and green traces) among several clusters, while they receive exactly the same contributions. When using algebraic partitioning tools such as METIS or SCOTCH, separators interactions are even more irregular.

To better define the objectives of a suitable clustering strategy, let us consider the block matrix:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \tag{1}$$

where the set of unknowns belonging to the last separator corresponds to A_{22} and the remaining unknowns to A_{11}. The blocks A_{21} and A_{12} correspond to the interaction between A_{11} and A_{22}.

Given this representation, operations are divided into:

1. *POTRF(A_{11})* to factorize A_{11},
2. *TRSM(A_{11},A_{21})* to solve the off-diagonal blocks A_{21} and A_{12},
3. *HERK(A_{21},A_{22})* to perform the updates that will contribute to A_{22} and
4. and *POTRF(A_{22})* to factorize A_{22}.

The objective of a good clustering strategy for A_{22} is to reduce the cost of factorizing a separator, *i.e. POTRF(A_{22})* in our example, but also to exhibit an efficient coupling to reduce the cost of *HERK(A_{21},A_{22})*.

The issue with existing clustering strategies is that they do not consider both intra (A_{22}) and inter (A_{12} and A_{21}) separators properties. For instance, k-way partitioning takes into account only intra-separator properties in A_{22}, but does not consider A_{21}. On the opposite, the reordering strategy presented in [18] orders correctly the coupling parts A_{12} and A_{21}, but fails to exhibit a suitable low-rank structure for A_{22}.

In this paper, we study the impact of clustering techniques on the PaStiX solver that takes advantage of BLR compression [3] and propose a new heuristic to couple assets of existing methods. In Section 2, we describe the clustering operation and present assets and drawbacks of existing heuristics (k-way and reordering). In Section 3, we propose a new heuristic and evaluate its impact with respect to existing strategies in Section 4 for the sparse supernodal BLR solver PaStiX. Finally, we discuss limitations of the new heuristic and future works in Section 5.

2 Low-rank clustering problem

We recall that permuting vertices within a separator does not impact the fill-in since diagonal blocks are considered as dense blocks. Then, when permuting unknowns within a separator, both the memory consumption and the number of operations are kept untouched for full-rank arithmetic, while it can impact low-rank compressibility. Thus, the objective is to perform a clustering of unknowns that 1) enhances the compression rates and 2) maintains efficient sparse structures, by permuting unknowns within a separator. We expect to couple strategies that were designed to obtain efficient sparse data structures with low-rank clustering strategies originally introduced for dense matrices. In a geometric context, the objective is to form as many large admissible blocks (according to some admissibility criterion depending on the diameters and distances between clusters) as possible, while in a fully algebraic context it is more challenging since the distances between points are unknown.

In Section 2.1, we recall the problem of the clustering. We discuss existing strategies in Section 2.2, before explaining on a simple example the limitations of the existing strategies in Section 2.3.

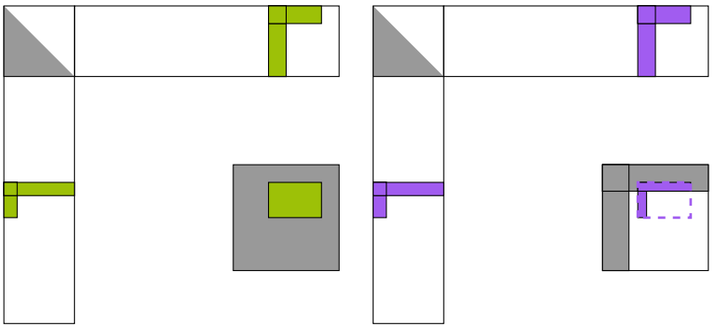
2.1 Problem of the low-rank clustering

Both hierarchical ($\mathcal{H}$, $\mathcal{H}^2$, HSS, HODLR) and flat (BLR) compression techniques require a suitable clustering of unknowns that achieves two conditions: 1) form compact clusters in the sense that unknowns belonging to a same cluster are close together in the graph and 2) ensure that clusters have as few neighbors as possible between them, such that most clusters are well-separated and their interactions are thus low-rank.

Most sparse direct solvers using low-rank compression follow the multifrontal method, the only solvers—to the best of our knowledge—using the supernodal method are [13] in a geometric context with fixed ranks and our solver, presented in [3]. In the multifrontal method, the commonly used approach is to consider the graph made of fully-summed variables of a front and to perform a partitioning of this graph to obtain the low-rank clustering. For hierarchical strategies, this partitioning is performed recursively while in the BLR case, where no hierarchy is required, a k-way partitioning is usually performed.

In the supernodal approach, one can use a similar method to order the unknowns within separators, as it exactly corresponds to the fully summed variables of the fronts in the multifrontal method. In this context, the main drawback of the k-way partitioning is that it considers only intra-separator interactions (A_{22}) and not the interactions between the separators.

In Figure 2, we schematically describe the difference between fully-structured and non-fully structured approaches to perform low-rank update operations. In Figure 2(a), a low-rank update is added to a dense block by forming explicitly the dense update. The block receiving the contribution is then compressed later, when its supernode will be eliminated. On the other hand, in Figure 2(b), a low-rank update is added to a low-rank matrix. This operation requires a more complex operation, and need to perform re-compression. This comes to a cost depending on the target dimensions (size and rank), as opposed to the previous approach where the complexity depends of the contribution size. Thus, the sparsity pattern of the inter-separators blocks (A_{12}) impacts in both cases the granularity of the updates, but it also impacts the number of updates which might have a much larger impact on the factorization time in the context of fully-structured updates.



(a) Non-fully-structured update. (b) Fully-structured update.

Figure 2: Illustration of the different update strategies when using low-rank representation for A and B matrices in the $C = C - AB$ update. Non fully-structured updates, on the left, use a full-rank representation of C, while fully-structured updates, on the right, use a low-rank representation of C.

In a non-fully-structured approach, where updates are applied to full-rank blocks, k-way partitioning may be sufficient since a larger number of updates, may reduce their performance (smaller blocks), but do not impact the overall number of flops to perform. However, in a fully-structured approach, where low-rank updates are performed, one can expect that a clustering strategy that avoids scattering updates among too many blocks will benefit the solver by reducing the flop overhead of the updates. Finally, for both *Minimal Memory* (fully-structured updates) and *Just-In-Time* (non fully-structured updates) strategies presented in [3], reducing the number of low-rank blocks will increase the updates granularity, and thus should enhance the compression

rates of the coupling parts.

2.2 Related work: low-rank clustering strategies

Several techniques were designed to perform the clustering of dense matrices for flat or hierarchical compression schemes. A commonly used approach consists in building clusters to respect a given admissibility condition. For dense matrices, one can require clusters to have a small diameter and a few neighbors to increase the number of interactions that will be considered as compressible. In this section, we present existing techniques for dense and sparse clustering.

In [19], Rebrova et al. built cluster trees from the geometry of specific problems. Such an approach cannot directly be extended to the algebraic case, which is the focus of this paper.

In [20], Bebendorf presented block-admissibility conditions and a spectral bisection strategy to cluster unknowns of a sparse matrix is introduced. The authors also mention nested dissection to perform the low-rank clustering of a sparse matrix. As in our study we are interested in clustering separators, using such an approach would be similar to use graph partitioning techniques on the subgraph induced by the separator.

In [21], Yu et al. presented a method to cluster unknowns of a dense matrix without the knowledge of the geometry or properties of underlying equations. The authors use the fact that any SPD matrix corresponds to a Gram matrix of vectors in an unknown Gram space [22]. Each entry of the matrix can be seen as an inner product, which allows to define distances among points. The authors use sampling to avoid computing all distances as it would be too costly for a dense matrix, and then split unknowns recursively to obtain a balanced cluster tree using those distances.

In practice, k-way partitioning or recursive bisection are the most commonly used approaches to perform clustering of fronts or separators. It is the strategy adopted in both MUMPS [1,2] and STRUMPACK [8,9] solvers. However, those graph methods are dedicated to connected graph, which is not necessarily the case for a front or a separator. In practice, those subgraphs issued for the original larger graph are reconnected using halo vertices at distance 1 or 2 to obtain fully connected graphs. Then, k-way methods available in partitioning tools such as METIS or SCOTCH are used to perform the clustering.

The techniques used to reduce the number of off-diagonal blocks appearing in the block-symbolic structure (as presented in [18]) can also be used to enhance the sparsity pattern and thus reduce the number of low-rank updates. Such approaches allow clustering vertices that will receive similar contributions together, but those vertices are not necessarily close in the subgraph of the separator receiving contributions, which can degrade compressibility within the separator. When such a reordering strategy is used, vertices are clustered after the reordering process. Since vertices receiving similar contributions are ordered consecutively, one can expect that splitting the set of vertices into clusters of equal sizes will provide a suitable ordering. In practice, a smarter split using non fixed sizes up to a given tolerance has been introduced in Lacoste's thesis [23] to reduce the number of off-diagonal blocks split when clustering the unknowns of a given separator.

To summarize, clustering techniques can be classified into four classes: 1) use the geometry and/or the kernel function of the underlying equations; 2) introduce algebraically a distance for a dense matrix and use graph partitioning methods; 3) apply graph partitioning techniques for sparse matrices and 4) focus on sparse pattern without considering distances in the graph. Clustering techniques corresponding to 1) and 2) are out-of-scope of this paper since they concern dense matrices or are not algebraic. The remaining of the paper will focus on the last two strategies only.

2.3 Example with advantages and drawbacks for k-way and reordering

Both k-way and reordering approaches may not provide a suitable clustering of unknowns for supernodal solvers and when using low-rank assembly in general.

Let us illustrate the problem with a plane separator, by considering a 7-point stencil of size $8 \times 8 \times 8$ for which the first separator is a surface of size 8×8. In Figure 3(a), we present the block-symbolic factorization obtained by clustering unknowns of the last separator with a k-way partitioning into four parts. In the upper part of the figure, a zoom presents the number of external contributions received by each block. The clustering of the last separator is presented in the graph of the separator, where vertices belonging to a same cluster are marked with the same color.

In Figure 3(b), we present the block-symbolic factorization obtained by performing reordering on the last separator. From this block-symbolic structure, we perform the four parts clustering of the last separator with smart splitting [23], which gives parts of size 16, 18, 14 and 16. It avoids cutting too many off-diagonal blocks among different clusters, by computing an average block size and performing the actual split in an interval around the mean value to minimize the number of blocks affected by the cut. Similarly to the previous case, in the upper part of the figure, a zoom presents the number of external contributions received by each block after clustering. The figure also presents the graph of this last separator, where vertices belonging to a same part are marked with the same color.

One can note that both k-way and reordering are not optimal to obtain good data structures. The k-way clustering may provide good compression rates within the separator (A_{22}), however induces more off-diagonal updates. Furthermore, splitting the off-diagonal blocks in smaller contributions may make them incompressible. The reordering strategy reduces the number of off-diagonal blocks (A_{21} and A_{12}), as highlighted with a reduced number of contributions on last separator. However, vertices belonging to a same cluster are not close in the graph, which will degrade A_{22} compressibility.

3 Pre-selection heuristic

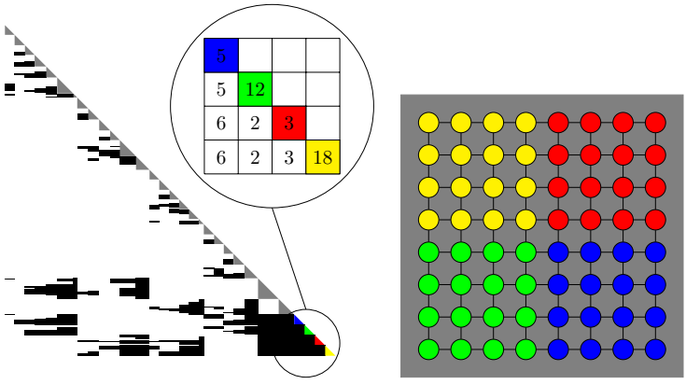
We propose a new heuristic to perform the clustering of the unknowns in order to respect two conditions:

1. Minimize the rank of the interactions between clusters. This condition turns into maximizing the number of well-separated clusters, which can be performed by exhibiting clusters with a small diameter and only a few neighbors;

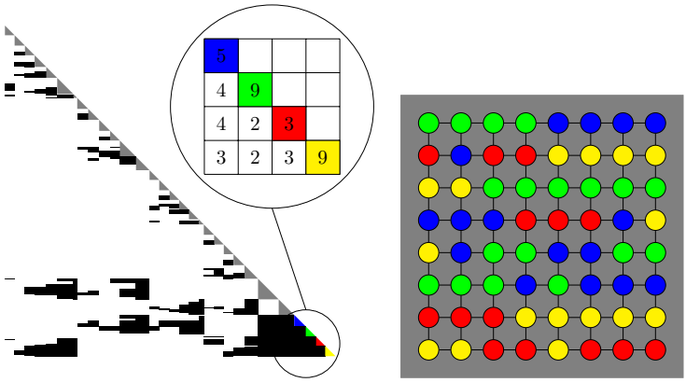
2. Minimize the number of contributions coming from children.

In addition, we expect to correctly identify interactions that are not well separated and that will lead to incompressible blocks to avoid performing needless low-rank compression. In a sense, the main idea is to identify the important unknowns of the supernodes as it is studied in [24]. The main difference resides in the fact that we want to identify that information from the adjacency graph only, while they discover it empirically and numerically during the computation with the values of the matrix.

In Figure 1, we defined the concept of traces, which correspond to the vertices of a separator that are directly connected to children which are close in the elimination tree. In Section 3.1, we present how traces are introduced to cluster vertices, before detailing the overall strategy in Section 3.2. Finally, we discuss some implementation detail in Section 3.3.



(a) K-way partitioning.



(b) Reordering strategy.

Figure 3: Illustration of the clustering obtained through the k-way partitioning, on top, and the reordering heuristic, on bottom, for the top-level separator of size 8×8 of a $8 \times 8 \times 8$ regular grid. The symbolic factorizations, on the left, show the evolution of the off-diagonal blocks in number and size with a focus on the number of external contributions applied to each block of the matrix associated with the last separator (A_{22}). The meshes, on the right, show the distribution of the unknowns into the clusters on the graph of the separators.

3.1 Using traces to pre-select and cluster vertices

The strategy to enhance supernodes clustering is to consider how children will contribute to a given separator. The objective is to order unknowns of a separator accordingly to the set of contributions it receives from the closest children in the elimination tree. Only closest children are considered, otherwise there are only few vertices receiving the same set of contributions. In addition, this strategy tries to isolate (pre-select) some vertices that represent strong connections, and that may not be compressible.

In practice, let us consider a separator and its closest children in the elimination tree. In order to cluster vertices of the separators depending on which contributions they receive, we consider traces of children on their ancestor. It was illustrated in Figure 1 for two levels of nested dissection, where green and red traces correspond to vertices of the separator that are directly connected to at least one children separator. In Figure 4, we present a separator with two red traces which correspond to interactions with direct children and four green traces that correspond to interactions with grand children in the elimination tree. From those traces, vertices belonging to a same connected subpart in the separator will receive the same set of contributions from the next two levels of children in the elimination tree. Naturally, the contributions coming from deeper children in the elimination tree will not be necessarily identical.

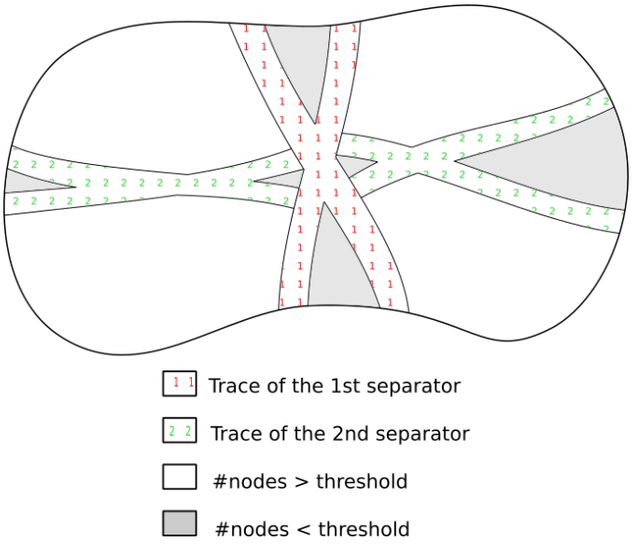


Figure 4: Two levels of traces on a generic separator.

Vertices of a separator can be split into two categories: vertices belonging to one or more traces and vertices that are not connected to the closest children in the elimination tree. Vertices belonging to traces are targeted as being the important ones, and they are named pre-selected vertices in the rest of the paper. Each connected subpart made of vertices that were not pre-selected forms a cluster, since those vertices will receive the same contributions (the sparsity pattern will be identical) from children whose traces were considered.

Beyond the problem of forming suitable clusters, pre-selecting vertices intends to isolate some special vertices that represent strong interactions and thus are not compressible. For this reason, we do not try to compress intra-separator blocks corresponding to pre-selected vertices. More precisely, if the block A_{22} is split into A_{ss} for pre-selected vertices and A_{kk} for the rest of vertices,

we obtain:

$$A_{22} = \begin{pmatrix} A_{kk} & A_{ks} \\ A_{sk} & A_{ss} \end{pmatrix} \tag{2}$$

From this representation, compressible blocks are only in A_{kk}, which corresponds to interaction between non-pre-selected vertices.

Some other blocks may be non compressible. For instance, off-diagonal blocks that are just above or below the main diagonal include some vertices that are non compressible, so we do not compress those blocks. In addition, off-diagonal blocks that represent contributions between neighbors in the k-way partitioning include strong (distance-1) connections and may be not compressible. In our implementation, we do not manage those blocks differently than others because it may degrade the overall compression rate.

3.2 Overall approach: compute pre-selected vertices and manage underlying subparts

The objective is to cluster unknowns to increase compressibility. In practice, we rely on traces to pre-select some vertices that will increase the distance between blocks, and thus the compressibility of those interactions. Traces are used to cluster vertices at a coarse level and k-way is used to refine those clusters to obtain suitable blocking sizes. The approach consists of computing pre-selected vertices before extracting distinct connected components in the set of vertices that do not belong to traces.

First of all, connected components that contain too few vertices to form a compressible cluster are merged together in order to form supernodes which size is larger than the minimum compressible size. The threshold used, as presented in Figure 4, is simply the minimum size used to compress a supernode. For larger connected components, the number of vertices can be too large to obtain reasonable clusters, which is necessary to reduce the size of dense diagonal blocks. For this reason, those subgraphs are clustered one-by-one using k-way partitioning.

To improve the projection process, the maximum number of pre-selected vertices must be controlled. For a separator of size n, and considering a constant number of children projections, the number of pre-selected vertices should not exceed $\Theta(\sqrt{n})$, which is the size of a separator for the separator being reordered and also the size of the traces of direct children. It ensures that only a few number of blocks are not compressed.

In Figure 5, we present the clustering of the last separator of a $80 \times 80 \times 80$ Laplacian matrix. Six traces are considered, two for the first level and four for the second level. From pre-selection, four large clusters were exhibited since we consider only two levels of nested dissection. Depending on their size, each large cluster was again split into five or six clusters using a k-way partitioning.

Using traces to cluster vertices seems to be straightforward for a regular 2D or 3D graph, as presented in Figure 5 for a relatively large case. However, such an approach is not appropriate for enhancing ordering of geometries where one dimension is larger than others. For instance, for a 2.5D graph where two dimensions are relatively large and the last one is much smaller, the first separators will be parallel plans. Thus, there is no connection between a separator and its direct children and obviously no vertex will be pre-selected. Such a pre-selecting algorithm is designed to enhance the clustering of graphs with a good aspect ratio.

3.3 Implementation details

The objective is to pre-process each separator before the block-symbolic factorization to form clusters and pre-select some non compressible vertices. In order to do so, each separator which

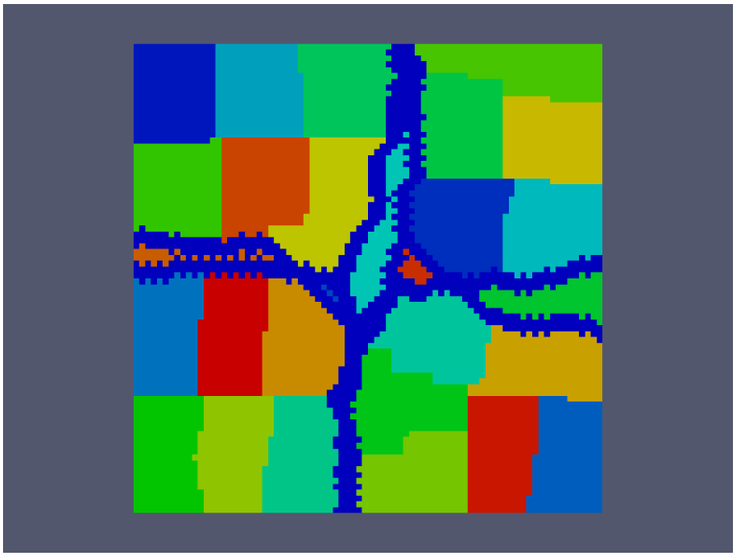


Figure 5: Two levels of trace for the last separator of a $80 \times 80 \times 80$ Laplacian matrix.

is large enough is pre-processed with the heuristic relying on traces, and a k-way partitioning is performed if traces are not working, *i.e.*, if zero vertices were pre-selected.

First, separators computed by partitioning tools are not necessarily connected. Thus, reconnection of separators can be performed using paths of length 1 or 2 in the original graph. Then, we apply a clustering technique: either k-way, reordering, or the new heuristic introduced just before. Finally, for each cluster, the reordering is still performed to reduce the number of off-diagonal blocks contributing to the given cluster and thus the number of contributions. The reordering cost is even reduced, as the number of vertices considered in each block being reordered is much smaller and impacts the complexity by a quadratic factor [18]. Thus, it is less expensive to perform reordering on clusters instead of performing reordering on the full separator.

3.3.1 Compute pre-selected vertices

We briefly described how traces are defined to obtain blue vertices in Figure 5. In practice, several parameters are considered to select vertices that are non-compressible and isolate suitable clusters.

The first parameter, named *levels of projections* (l), corresponds to the distance in the elimination tree for which children are considered. If this parameter is set to $l \geq 1$, the number of children considered will be 2^l (using nested dissection). This parameter has a large impact on the number of selected vertices, since increasing the number of children considered will increase the number of traces and by the same effect the number of distinct connected components. If too many children are considered, the connected components resulting from traces will be too small for being compressible. Note that we do not consider children that were not issued from the nested dissection process, for instance children that were obtained thanks to minimum-fill, but this type of ordering will only appear at the bottom of the elimination tree.

The second parameter, named *halo distance for projections* (d), corresponds to the distance from which a vertex from the separator being reordered and a vertex from children are considered as connected. In practice, for each vertex of the current separator, we are looking at his

neighborhood at a distance d to see if the vertex has to be selected or not.

The third and last parameter, named *width of projections* (w), corresponds to the width of traces. After vertices of the separator have been selected thanks to *levels of projections* and *halo distance for projections* parameters, this third parameter will increase the width of bands to ensure a good separability.

Note that *halo distance for projections* and *width of projections* parameters are quite close in the sense that they both increase the width of selected vertices, but from a different point of view. Those parameters also increase the distances between clusters that were separated thanks to traces and then the compressibility of interaction blocks between those clusters.

3.3.2 Control the number of pre-selected vertices

To control the number of pre-selected vertices, we introduced different parameters, depending on the blocking size b and the size of the separator n. First of all, a separator is clustered based on traces if its size is larger than $16b$. We expect to form four large connected components for a 2D separator of a 3D graph, as it would happen for a regular Laplacian with a constant number of traces. In addition, we want to form at least four k-way clusters in each connected component, which gives the 16 factor. K-way partitioning is always performed to obtain blocks of the maximum authorized blocking size, b.

Then, the number of children (and thus traces) considered is adapted level by level given a maximum limit. The objective is to select less than $\Theta(\sqrt{n})$ vertices such that the remaining number of vertices is larger than $4b$. Thus, we pre-select vertices level by level until reaching the maximum authorized number of pre-selected vertices and such that only the closest children in the elimination tree, given a maximum depth, are considered.

3.3.3 Graph algorithms

We now describe the different graph algorithms that are used to compute the clustering. Let us consider the graph of a separator $C = (V_C, E_C)$ made of V_C vertices and E_C edges for the complexity analysis.

OrderSupernode(C**,** l**,** d**,** w **)** This is the main routine (see Algorithm. 1) that orders unknowns of a separator C.

ConnectSupernodeHalo(C **)** This routine isolates the graph of the separator C being reordered. Connections through the original graph at distance 1 are turned into direct connections to obtain a connected graph and to better apply next partitioning algorithms. Reconnection at distance 2 was also used in MUMPS or STRUMPACK but it was shown in [25] that distance 1 is sufficient for most graphs. In terms of complexity, this routine requires to explore, for each vertex of a separator, its neighborhood at distance 1. Given a bounded-degree graph where the larger degree of a vertex is $\Delta(C)$, the complexity of this routine is bounded by $\Theta(|V_C| \times \Delta(C))$.

ComputeTraces(C**,** l, d, w **)** This routine considers vertices of the separator C being reordered and search for direct connections with children from next l levels in the original graph. Connections that are issued from paths of length d can be computed with $\Theta(|V_C| \times \Delta(C)^d)$ operations. Finally, within the graph of the separator, some vertices at distance w from already pre-selected vertices are also marked as pre-selected.

Algorithm 1 OrderSupernode(C, l, d, w): order unknowns within the separator C.

```
1: ConnectSupernodeHalo( C )
2: ComputeTraces( C, l, d, w )
3: IsolateConnectedComponents( C )
4: For each connected component C_i Do
5:     If |C_i| < threshold Then
6:         Merge C_i into small components vertices
7:     Else
8:         K-way( blocksize )
9:         For each k-way part K_j Do
10:            Reordering( K_j )
11:        End For
12:    End If
13: End For
14: Reordering(small components vertices)
15: Reordering(pre-selected vertices)
```

IsolateConnectedComponents(C **)** This routine isolates each connected components of the separator C. It can be used either to isolate distinct components for a non-connected separator (for instance when partitioning a tore) or after traces have been computed to correctly identify each subpart. This routine performs a Breadth-First Search (BFS) of the graph until each vertex has been visited once. When a BFS stops while all vertices have not been visited, a new connected component is created. This algorithm is linear in both the number of vertices and edges, its complexity is in $\Theta(|V_C| + |E_C|)$.

K-way(*blocksize* **)** K-way partitioning consists in partitioning a graph into a defined number of parts, such that each part has the same number of vertices and the number of edges (named cut) between parts is as low as possible. Note that k-way from METIS or SCOTCH try to minimize the overall edge-cut and not to balance edge-cut among different parts. For this routine, we directly call a SCOTCH strategy with an unbalance factor set to 5%. The complexity of this routine used in a multilevel framework is in $\Theta(k|E_C|)$, where k is the number of parts in the k-way partitioning.

Reordering($part_i$ **)** For each subpart, reorder vertices to enhance the sparsity pattern. A matrix of distances between the set of contributions for each unknown is computed and vertices are ordered using a traveling salesman algorithm. The algorithm and complexity study are presented in [18].

The complexity of the algorithm depends not only on the size and the connectivity of the separator graph, *i.e.*, average degree of nodes, but also on the parameters that are used. For instance, ComputeTraces(C, l, d, w) can be costly if d, *halo distance for projections* parameter, is too large. In Section 4.3.3, we study the cost of clustering strategies and show that there is few or no overhead with the use of projections.

4 Experiments

In this section, we study the behavior of the different clustering strategies: k-way partitioning, named *K-way*, the reordering strategy presented in [18] together with smart splitting, named

Reordering, and the newly introduced heuristic relying on traces, named *Projections*. In Section 4.1, we recall the behavior of the BLR supernodal solver PaStiX [3]. In Section 4.2, we describe the parameters used in the solver to manage blocking size and control pre-selecting vertices. We study the behavior of the newly introduced heuristic with respect to *K-way* and *Reordering* strategies on a large set of matrices in Section 4.3. In Section 4.4, we detail results for a smaller set of matrices, to better describe the behavior of all heuristics.

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

The PaStiX version used for our experiments is available on the public git repository[2] as the tag `clustering`. The multi-threaded version used is the static scheduling version presented in [23]. Note that for low-rank strategies, we never perform LL^t factorization because compression can destroy the positive-definite property. In the case where the matrix is SPD, we use LDL^t factorization for low-rank strategies. In addition, we consider that all problems have a symmetric pattern given by the pattern of $A + A^t$.

4.1 PaStiX BLR solver

In this section, we summarize [3] to emphasize the parts of the solver that will be impacted by the clustering of unknowns.

From the original block structure, adapting the solver to block low-rank compression mainly relies on the replacement of the dense operations with the equivalent low-rank operations. Still, different variants of the final algorithm can be obtained by changing *when and how* the low-rank compression is applied. We introduced two scenarios in [3]: *Minimal Memory*, which compresses the blocks before any other operations, and *Just-In-Time*, which compresses the blocks after they received all their contributions.

As each off-diagonal blocks in the refined partition is compressed individually, reducing the number of off-diagonal both enhances both memory consumption and data locality.

4.1.1 *Minimal Memory* strategy

The *Minimal Memory* strategy starts by compressing the original matrix A block-by-block without allocating the full matrix as it is done in the full-rank and *Just-In-Time* strategies. Then, each classic dense operation on a low-rank block is replaced by a similar kernel operating on low-rank forms, even for the usual matrix-matrix multiplication (*GEMM*) kernel that is replaced by an equivalent low-rank kernel operating on three low-rank matrices.

The main asset of this approach is that large blocks are never allocated in a dense fashion, and the memory peak of the solver is reduced. However, performing low-rank assemblies can be expensive as large low rank matrices receive many small contributions (cf. Figure 2(b)). Indeed, the cost of updating a low-rank matrix directly depends on its size and rank even if the contribution is much smaller.

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

Kind		Matrix	Arith.	Fact.	N	NNZ_A	TFlops	Memory (GB)
2d/3d		PFlow_742	d	LL^t	742793	18940627	1.4	4.3
		Bump_2911	d	LL^t	2911419	65320659	204.9	78.3
Computational fluid dynamics		StocF-1465	d	LL^t	1465137	11235263	3.6	8.7
		atmosmodl	d	LU	1489752	10319760	10.1	16.7
		atmosmodd	d	LU	1270432	8814880	12.1	16.3
	*	atmosmodj	d	LU	1270432	8814880	12.1	16.3
		RM07R	d	LU	381689	37464962	15.7	16.0
Dna electrophoresis		cage13	d	LU	445315	7479343	356.2	76.3
Electromagnetics		dielFilterV3clx	z	LU	420408	16653308	1.3	5.2
		fem_hifreq_circuit	z	LU	491100	20239237	1.6	6.0
		dielFilterV2clx	z	LU	607232	12958252	2.1	7.0
Magnetohydrodynamics		matr5	d	LU	485597	24233141	8.4	10.5
Materials		3Dspectralwave2	z	LDL^h	292008	7307376	6.5	6.3
Model reduction		boneS10	d	LL^t	914898	28191660	0.3	2.5
		CurlCurl_3	d	LDL^t	1219574	7382096	3.8	6.5
		bone010	d	LL^t	986703	36326514	4.4	9.4
		CurlCurl_4	d	LDL^t	2380515	14448191	13.7	15.7
Optimization		nlpkkt80	d	LDL^t	1062400	14883536	27.3	17.9
Structural		ldoor	d	LL^t	952203	23737339	0.1	1.2
		inline_1	d	LL^t	503712	18660027	0.1	1.5
		Flan_1565	d	LL^t	1564794	59485419	3.7	12.3
		ML_Geer	d	LU	1504002	110879972	4.2	17.2
	*	audikw_1	d	LL^t	943695	39297771	5.5	9.5
		Fault_639	d	LL^t	638802	14626683	7.7	9.0
	*	Hook_1498	d	LL^t	1498023	31207734	8.6	12.7
		Transport	d	LU	1602111	23500731	10.2	20.8
		Emilia_923	d	LL^t	923136	20964171	12.7	13.5
	*	Geo_1438	d	LL^t	1437960	32297325	18.0	20.1
	*	Serena	d	LL^t	1391349	32961525	28.6	21.7
		Long_Coup_dt0	d	LDL^t	1470152	44279572	47.1	31.9
		Cube_Coup_dt0	d	LDL^t	2164760	64685452	87.2	51.6
		Queen_4147	d	LL^t	4147110	166823197	251.8	110.0

Table 1: Set of real-life matrices issued from The SuiteSparse Matrix Collection [26], sorted by family and number of operations. The set of five matrices used in Section 4.4 is highlighted with stars.

4.1.2 *Just-In-Time* strategy

This second scenario delays the compression of each supernode after all contributions have been accumulated. The algorithm is thus really close to the previous one with the only difference being in the update kernel, which performs full-rank assemblies.

The main asset of this approach is the time-to-solution reduction, since low-rank products are less expensive than dense products. However, as the matrix is compressed during the factorization, all blocks are allocated in a dense fashion at the beginning and there is no room for reducing the memory peak. Note that both approaches still led to a similar factor sizes at the end of the factorization.

4.1.3 Expected impact

One can expect that using pre-selected vertices will reduce the time-to-solution of both strategies while memory consumption will be impacted by the fact that fewer blocks are set as compressible, but probably with better compressibility rates.

For the *Minimal Memory* strategy, such an approach should reduce the overhead introduced by updating many times matrices that will become full-rank (rank is too large) during the factorization, and one can expect large factorization time reduction. In addition, pre-selection is a compromise between the number of off-diagonal blocks (for which reordering is the best) and the compressibility of blocks (for which k-way seems better). For the *Just-In-Time* strategy, it will be probably more difficult to observe time-to-solution gain, as the only room for improvement is to not try to compress non compressible blocks, which does not represent a large part of computations.

4.2 Parameters and tuning of the solver

We use a large set of parameters to correctly tune our solver. All experiments are performed using 24 threads. Some parameters presented in [3] impact the solver by itself and not clustering strategies, studying their impact is out-of-scope of this paper.

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

In experiments, blocks that are larger than 256 are split into blocks of size within the range $128 - 256$ to create more parallelism while keeping sizes that are large enough to reach good efficiency. The same 128 criteria is used to define the minimal width of the column blocks that are compressible. An additional limit on the minimal height to compress an off-diagonal block is set to 20.

In order to obtain partitions with supernodes of similar width, the number of parts using k-way partitioning is defined to obtain clusters of size 256. The k-way partitioning method is the one from Scotch. As we will see in next experiments, the number of supernodes in the refined partition is almost invariant with the clustering method used.

Pre-selection is applied on separators which are large enough for being split. After pre-selecting vertices with traces, components of size lower than 128 are merged together, otherwise the corresponding blocks would be too small for being compressed. We use the newly introduced heuristic with $ComputeTraces(3, 1, 1)$, which provided in average the best results.

Finally, we switch to the *K-way* strategy if the number of pre-selected vertices is lower than $\alpha\sqrt{n}$, where α is set to 50, to correctly manage the number of pre-selected vertices as it was presented in Section 3.3.2.

4.3 Behavior on a large set of matrices

The objective of this section is to study the behavior of the three clustering techniques on a large set made of 33 matrices, made of the 32 matrices presented in Table 1, as well as on a Laplacian of size $120 \times 120 \times 120$.

In Figure 6 (respectively Figure 7), we present the performance profile for factorization (respectively for memory consumption) when using *K-way*, *Reordering* or *Projections* heuristics for both *Minimal Memory* and *Just-In-Time* factorizations using a 10^{-8} and a 10^{-12} tolerance. For each clustering heuristic, the percentage with respect to the optimal heuristic is computed (x axis) and accumulated for each matrix (y axis). It means that, on average, the best heuristics are curves that remain close to $x = 1$. The objective of those figures is to give a general trend on a relatively large set of matrices.

4.3.1 Impact on factorization time

In Figure 6, one can observe that using the *K-way* strategy allows to reduce the factorization time with respect to the use of the *Reordering* strategy. When using either the *Minimal Memory* or the *Just-In-Time* strategy, *K-way* improves the factorization time by 10% for a 10^{-8} tolerance and 15% for a 10^{-12} tolerance.

Now, if we consider the new *Projections* heuristic, we observe different behavior for both low-rank strategies. For the *Minimal Memory* strategy, the *Projections* heuristic allows to reduce the factorization time, as it was expected since it was designed to avoid updating blocks with a high rank. The gain is around 10% both for 10^{-8} and 10^{-12} tolerances. However, for the *Just-In-Time* strategy, there is almost no gain with respect to the *K-way* strategy. The burden on managing blocks with a high rank is less important for this strategy, as it was shown in [3]. For both low-rank strategies, the *Projections* heuristic outperforms the *K-way* strategy with a larger factor for a 10^{-12} tolerance, since ranks are higher than using a 10^{-8} tolerance.

4.3.2 Impact on memory consumption

In Figure 7, we observe that the *K-way* strategy is the most suitable method for memory consumption. Using the *Reordering* strategy increases the memory consumption with a factor of 10%, while the *Projections* heuristic increases this metric by only a factor of 5%. The results favor the *K-way* strategy, especially for a more relaxed tolerance, such as 10^{-8}, as ranks are smaller.

Our new heuristic has only slight impact on memory consumption, while several blocks are not compressed and managed in a full-rank fashion all throughout the factorization.

4.3.3 Impact for preprocessing statistics

In Figure 8(a), we present the impact of clustering heuristics on the blocking sizes. We take the *Reordering* strategy as a reference, but it can be at most at a factor 2 from the optimal (cf. [18]) and can eventually lead to a larger number of blocks than other clustering methods. We recall that for both *K-way* and *Projections* strategies, the same reordering strategy is still applied, but independently on each cluster of the separator and not the full separator. One can observe that both *K-way* and *Projections* strategies degrade the blocking sizes, since the number of off-diagonal blocks is larger. However, the average increase is of 5%, which should not impact much granularity. Note that for any clustering method, the number of supernodes in the refined partition is kept similar, by definition of our blocking sizes.

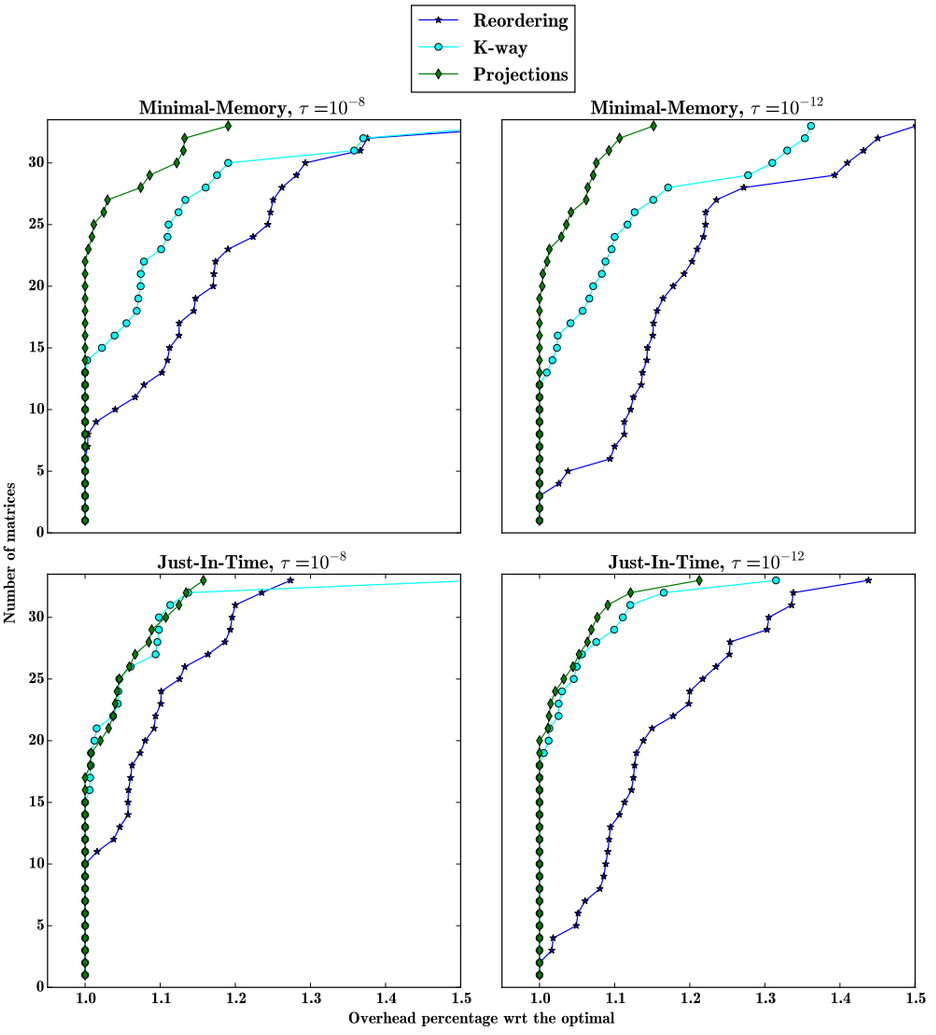


Figure 6: Performance profiles for the factorization time using three clustering strategies: *Reordering* (in blue star), *K-way* (in cyan circle), and *Projections* (in green diamond) on a set of 33 matrices for the *Minimal Memory* strategy on top, and the *Just-In-Time* strategy on bottom. On the left part, results with a 10^{-8} tolerance are presented and results with a 10^{-12} precision appear on the right.

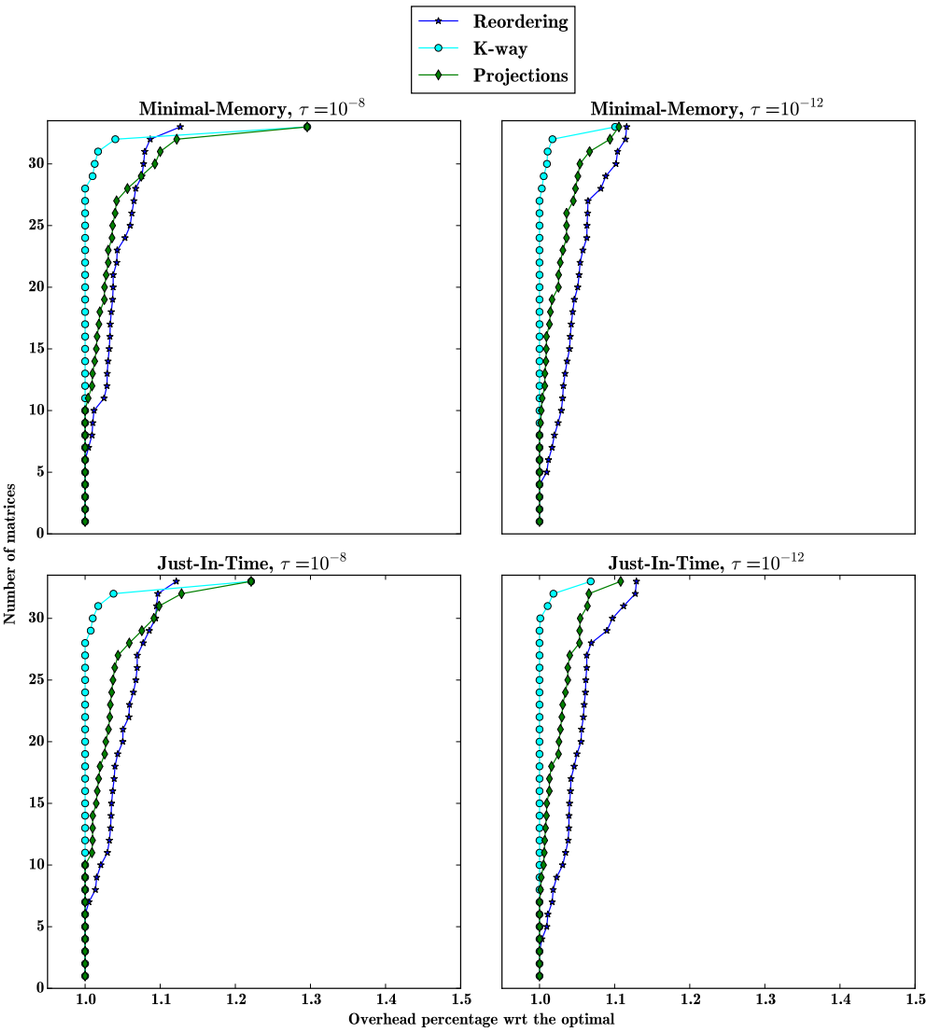
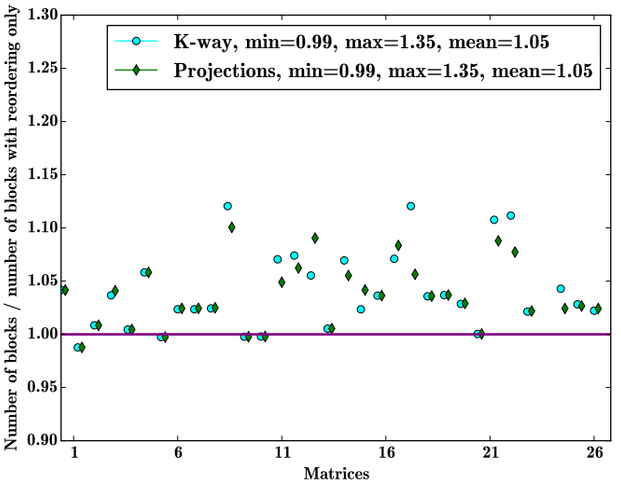
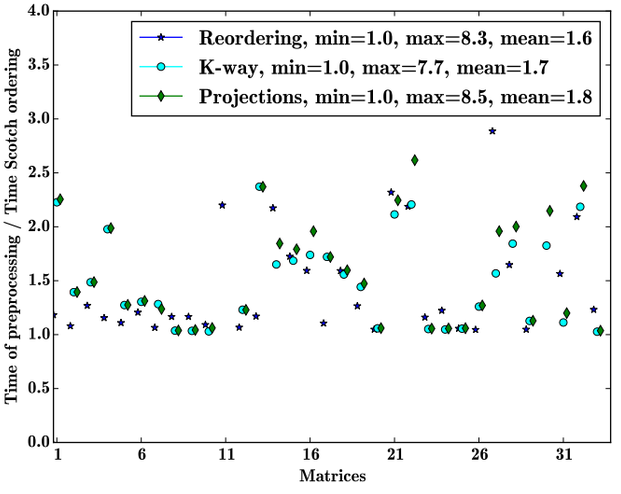


Figure 7: Performance profiles for the memory consumption time using three clustering strategies: *Reordering* (in blue star), *K-way* (in cyan circle), and *Projections* (in green diamond) on a set of 33 matrices for the *Minimal Memory* strategy on top, and the *Just-In-Time* strategy on bottom. On the left part, results with a 10^{-8} tolerance are presented and results with a 10^{-12} precision appear on the right.



(a) Impact on blocking sizes.



(b) Impact on preprocessing time.

Figure 8: Impact on preprocessing statistics for 33 matrices, for the number of blocks on top and for the preprocessing time on bottom. Three clustering strategies are studied: *Reordering* (in blue star), *K-way* (in cyan circle) and *Projections* (in green diamond). Those results are only structural and independent from the type of factorization or the tolerance used later.

In Figure 8(b), we analyze the preprocessing cost of the three clustering strategies. The metric presented is the cost of clustering (including pre or post reordering) with respect to the cost of performing ordering using Scotch. All methods are performed in sequential. One can note that all methods have a similar behavior, with on average an extra cost of a factor 0.7 with respect to the ordering stage. If both *K-way* and *Projections* strategies require extra computations to compute clusters, this extra cost is masked by the reduction of the reordering cost. Indeed, using those methods, reordering is only performed within clusters, which reduces a lot its complexity. In addition, it can be easily parallelized, since each separator is pre-processed independently.

4.4 Detail analysis

In this section, we detail the behavior of the three clustering strategies. For the sake of simplicity, we will compare heuristics on the last separator only and rely on blocks introduced by Equation (1) to study the compression over different parts of the matrix. Note that to ease the reading we refer to A_{21} as the blocks belonging to both A_{12} and A_{21} in Equation (1). We start by discussing the behavior of both the *K-way* and the *Reordering* strategies before detailing results for the *Projections* strategy.

The intuition given in Section 2.3 is that, on the one hand, the *K-way* strategy will favor compression of A_{22} by correctly clustering vertices of the separators thanks to distances and diameters consideration. On the other hand, the *Reordering* strategy is supposed to be suitable for compression of A_{21} since it reduces the number of off-diagonal blocks.

In Table 2, we present the number of operations and the memory consumption for six representative matrices issued from various applications, as presented in Table 1 (highlighted with stars) and a Laplacian of size $120 \times 120 \times 120$, with the full-rank, *Minimal Memory*, and *Just-In-Time* factorizations with a 10^{-8} tolerance. For low-rank strategies, we illustrate the memory consumption, the number of operations and the factorization time for the three considered clustering strategies. In this first analysis, we will only consider existing clustering methods and not the *Projections* heuristic.

The first observation is that, for all matrices, the *K-way* strategy leads to better factorization time with respect to the *Reordering* strategy. If we analyze how operations are split among different parts of the matrix, we observe that the *K-way* strategy does not only reduce the number of operations of A_{22}, but also the cost on the coupling part A_{21}, which is not intuitive. As k-way partitioning is only applied to the last separator to illustrate its behavior in a simpler case, there are only few differences for the number of operations corresponding to A_{11}. For this part, only the *TRSM*(A_{11}, A_{21}) kernel is impacted and it was shown in [3] that the corresponding operations do not represent a large percentage of the total number of operations.

This trend on the number of operations is reflected on the memory consumption, for which the *Reordering* strategy is worse than the *K-way* strategy not only for A_{22}, but also for A_{21} for all six matrices.

In order to better analyze the differences between the *K-way* and the *Reordering* strategies on the coupling part A_{21}, we introduce another metrics. In Table 3, we present, for the three clustering strategies, the distribution of off-diagonal blocks of A_{21} among three categories: 1) those which are compressed, 2) those which are compressible but have a high rank and thus are numerically incompressible and 3) those which are non-compressible as their height or their width is too small. This experiment was performed using a full-rank factorization followed by a compression of all blocks that are large enough using SVD with a 10^{-8} tolerance and without limiting the ranks to a maximum authorized rank. The objective is to illustrate the optimal compression rates attainable as well as the numerical properties of blocks which are incompressible (with a compression ratio lower than 1). We also integrate another metric, the number of

Matrix	Method	Strategy	HERK (A_{21}, A_{22}) (TFlops)	POTRF(A_{22}) (TFlops)	Total (TFlops)	Mem A_{11}	Mem A_{21}	Mem A_{22}	Fact(s)
lap120	Full-rank	-	3.71	0.95	14.44	8.66 GB	3.52 GB	805 MB	38.00
	Minimal-Memory	Reordering	0.21	0.04	2.63	3.77 GB	507 MB	151 MB	49.96
		K-way	0.16	0.02	2.62	3.77 GB	488 MB	123 MB	47.91
		Projections	0.07	0.25	2.78	3.77 GB	503 MB	445 MB	44.31
	Just-In-Time	Reordering	0.06	0.03	0.64	3.85 GB	527 MB	157 MB	13.77
		K-way	0.05	0.02	0.63	3.85 GB	504 MB	128 MB	13.58
		Projections	0.05	0.25	0.86	3.85 GB	521 MB	449 MB	13.95
atmosmodj	Full-rank	-	2.03	0.39	12.09	12.3 GB	3.42 GB	579 MB	35.69
	Minimal-Memory	Reordering	0.18	0.03	3.93	5.85 GB	651 MB	151 MB	69.72
		K-way	0.13	0.02	3.90	5.85 GB	599 MB	128 MB	65.03
		Projections	0.07	0.08	3.92	5.85 GB	658 MB	286 MB	62.64
	Just-In-Time	Reordering	0.04	0.02	0.83	5.99 GB	682 MB	159 MB	15.13
		K-way	0.04	0.02	0.80	5.99 GB	624 MB	133 MB	13.63
		Projections	0.04	0.08	0.88	5.99 GB	685 MB	289 MB	13.81
audi	Full-rank	-	0.32	0.02	5.23	8.62 GB	832 MB	52.6 MB	12.53
	Minimal-Memory	Reordering	0.08	0.01	3.82	5.98 GB	237 MB	31.6 MB	43.14
		K-way	0.05	0.00	3.78	5.98 GB	215 MB	25.4 MB	35.52
		Projections	0.05	0.00	3.78	5.98 GB	215 MB	25.4 MB	35.43
	Just-In-Time	Reordering	0.02	0.00	1.07	6.05 GB	243 MB	32.1 MB	9.17
		K-way	0.01	0.00	1.06	6.05 GB	221 MB	26.6 MB	7.97
		Projections	0.01	0.00	1.06	6.05 GB	221 MB	26.6 MB	7.95
Geo1438	Full-rank	-	3.23	0.70	18.40	15.7 GB	3.79 GB	658 MB	39.30
	Minimal-Memory	Reordering	0.78	0.18	9.22	10.6 GB	1.42 GB	266 MB	95.48
		K-way	0.68	0.14	9.00	10.6 GB	1.35 GB	242 MB	83.89
		Projections	0.40	0.32	8.97	10.6 GB	1.41 GB	460 MB	84.59
	Just-In-Time	Reordering	0.21	0.08	2.98	10.9 GB	1.49 GB	289 MB	22.69
		K-way	0.18	0.06	2.90	10.9 GB	1.41 GB	261 MB	22.41
		Projections	0.20	0.31	3.18	10.9 GB	1.48 GB	472 MB	22.66
Hook	Full-rank	-	0.94	0.10	8.82	10.8 GB	1.74 GB	183 MB	21.01
	Minimal-Memory	Reordering	0.17	0.02	4.75	6.95 GB	454 MB	74.4 MB	72.09
		K-way	0.12	0.01	4.68	6.95 GB	417 MB	61 MB	71.11
		Projections	0.07	0.04	4.66	6.95 GB	435 MB	118 MB	71.76
	Just-In-Time	Reordering	0.04	0.01	1.24	7.08 GB	475 MB	78.7 MB	14.01
		K-way	0.03	0.01	1.22	7.08 GB	436 MB	64.4 MB	13.55
		Projections	0.03	0.04	1.26	7.08 GB	455 MB	120 MB	13.33
Serena	Full-rank	-	5.67	1.18	29.04	15.8 GB	5.05 GB	944 MB	68.62
	Minimal-Memory	Reordering	0.73	0.20	8.97	9.4 GB	1.36 GB	311 MB	142.86
		K-way	0.66	0.15	8.86	9.4 GB	1.29 GB	271 MB	134.65
		Projections	0.44	0.36	8.82	9.4 GB	1.32 GB	532 MB	128.30
	Just-In-Time	Reordering	0.21	0.10	2.79	9.62 GB	1.43 GB	331 MB	32.18
		K-way	0.19	0.08	2.73	9.62 GB	1.35 GB	294 MB	31.52
		Projections	0.19	0.33	2.99	9.62 GB	1.38 GB	544 MB	30.59

Table 2: Number of operations and memory consumption for the factorization of six matrices with $\tau = 10^{-8}$ for the full-rank and both low-rank strategies. Three clustering strategies are studied: *Reordering*, *K-way* and *Projections*. To study the behavior on the last separator, we highlight three types of blocks: separator (A_{22}), its coupling (A_{21}) and the rest of the matrix (A_{11}), as presented in Equation (1).

compressed blocks that contain values from the original sparse matrix A, in order to evaluate how those values are split among blocks. Those interactions that already appear on the original matrix may be non-compressible.

Matrix	Strategy	Number of blocks				Compressible blocks containing values from A
		Total	Numerically compressible (ratio)	Numerically incompressible (ratio)	Non compressible, above threshold	
lap120	Reordering	46115	10137 (13.79)	0 (0)	35978	558
	K-way	51116	10809 (14.89)	1 (0.9)	40306	448
	Projections	50782	11259 (14.15)	2 (0.9)	39521	450
atmosmodj	Reordering	25653	4981 (10.33)	8 (0.9)	20664	306
	K-way	27486	4866 (12.11)	5 (0.9)	22615	244
	Projections	28678	5485 (10.88)	4 (0.9)	23189	234
audi	Reordering	4247	2187 (6.54)	74 (0.85)	1986	161
	K-way	4357	2048 (7.74)	59 (0.86)	2250	129
	Projections	4357	2048 (7.74)	59 (0.86)	2250	129
Geo1438	Reordering	17235	9955 (4.03)	314 (0.86)	6966	516
	K-way	18297	9958 (4.46)	292 (0.86)	8047	385
	Projections	18698	10454 (4.11)	331 (0.86)	7913	409
Hook	Reordering	11752	4617 (6.76)	56 (0.88)	7079	370
	K-way	12823	4835 (8.05)	35 (0.87)	7953	296
	Projections	13000	5185 (7.49)	46 (0.88)	7769	298
Serena	Reordering	25084	13250 (5.72)	206 (0.87)	11628	945
	K-way	27781	13537 (6.22)	180 (0.87)	14064	694
	Projections	27527	13731 (6.13)	178 (0.87)	13618	772

Table 3: Number of updates and compression rates for the coupling part A_{21} that represents interactions with the last separator. A full-rank factorization was performed and blocks were compressed afterwards using SVD with $\tau = 10^{-8}$ to illustrate the optimal compression rates attainable. Three clustering strategies are studied: *Reordering*, *K-way* and *Projections*. We distinguish compressible blocks, which sizes are large enough for compression and non compressible blocks. Among compressible blocks, numerically incompressible blocks are those whose ranks are too high for reducing memory consumption.

The main observation, which confirms the asset of the *Reordering* strategy, is that the total number of off-diagonal blocks is larger using the *K-way* strategy. However, the proportion of compressible blocks is quite similar and the compression rates obtained using the *K-way* strategy are better than the ones using the *Reordering* strategy. This trend can be linked with the number of compressible blocks that contain values from the original graph. As more blocks contain values from A with the *Reordering* strategy, it can explain the smaller compression rates. In practice, the *Reordering* strategy does not handle those blocks differently than others. However, as k-way partitioning clusters vertices that are close in the graph, it can order contiguously vertices that own edges connected with the same part of the original graph.

We now analyze low-level behavior of the *Projections* heuristic. The objective is to exhibit basic statistics about *how and when* compression rates are impacted.

In Table 3, one can see the *Projections* strategy has a behavior between the *K-way* and the *Reordering* strategies. Indeed, there are slightly less blocks, but more blocks that contain values from A. In addition, the compression rates for compressible blocks is slightly worse than the one of the *K-way* strategy but better than the one of the *Reordering* strategy.

In Table 2, we can observe the global behavior of the *Projections* strategy with respect to existing strategies. Firstly, one can note that for both *Minimal Memory* and *Just-In-Time* strategies, the *Projections* strategy allows reducing factorization time with respect to the use of *Reordering* strategy. Secondly, in terms of memory consumption, the consumption related to A_{22} is naturally increased since pre-selected blocks are managed in a full-rank fashion. For the coupling part A_{21}, memory consumption slightly increases with respect to the *K-way* strategy, but is better than the one of the *Reordering* strategy. Finally, the most relevant observation is the distribution of the number of operations. For both *Minimal Memory* and *Just-In-Time* strategies, the number of operations related to the factorization of A_{22} increases. However, as there is a gain on the factorization time even for the *Just-In-Time* strategy, this increase does not directly translate into a loss of time, since it concerns inefficient low-rank operations on high rank blocks. For the *Just-In-Time* strategy, the *Projections* strategy allows reducing the cost of *HERK(*A_{21}*,* A_{22}*)* corresponding to expensive low-rank updates between small and incompressible blocks. This gain directly turns into factorization time gain as those operations are not very efficient.

Some matrices do not take advantage of the *Projections* strategy, as it happens for the *audi* matrix. Indeed, with non regular geometries, the last separator may not be connected to its closest children, leading to zero pre-selected vertices. In such a case, the results obtained for the *Projections* strategy are identical with the *K-way* strategy, as it was shown in Table 2 and in Table 3.

5 Conclusion and future work

In this paper, we analyzed the behavior of existing clustering strategies (k-way partitioning and the reordering strategy introduced in [18]) and proposed a new heuristic to perform clustering and identify non-compressible contributions. We demonstrated that it can reduce time-to-solution with only a slight memory increase.

In the experiments, we analyzed the advantages of such an approach for both *Minimal Memory* and *Just-In-Time* strategies. We studied this new heuristic on a large set of matrices to exhibit the general trend. We showed a reduction of the time-to-solution by a factor of 10% for the *Minimal Memory* strategy, while the memory consumption slightly increases by a factor of less than 5%.

For future work, we plan to better analyze which blocks are not compressible, for instance considering distances between clusters in the k-way partitioning. Another possibility would be to

consider fill-in paths to cluster together unknowns that represent strong interactions, edges that exist in the original graph of A or edges corresponding to ILU(1) or ILU(2) factorizations, which are known to be numerically important for most matrices.

As geometric solvers generally take advantage of the properties of the problem to enhance low-rank compressibility, it seems interesting to try to exhibit more symmetric structures for the algebraic case. As discussed in [27], another approach can modify the nested dissection process, to obtain connections between separators that are more regular. For instance, considering a subgraph G as $G_A \cup G_B \cup G_C$ where G_C is the separator, both children separators can contribute similarly to G_C if there traces on G_C are aligned. If such an alignment is performed recursively, it will be easier to exhibit large separated connected components, and thus should increase compressibility.

Acknowledgments

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

References

[1] P. R. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L'Excellent, and C. Weisbecker, "Improving Multifrontal Methods by Means of Block Low-Rank Representations," *SIAM Journal on Scientific Computing*, vol. 37, no. 3, pp. A1451–A1474, 2015.

[2] 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. Softw.*, 2018, to appear.

[3] G. Pichon, E. Darve, M. Faverge, P. Ramet, and J. Roman, "Sparse supernodal solver using block low-rank compression: Design, performance and analysis," *International Journal of Computational Science and Engineering*, vol. 27, pp. 255 – 270, Jul. 2018.

[4] W. Hackbusch, "A Sparse Matrix Arithmetic Based on $\mathcal{H}$-Matrices. Part I: Introduction to $\mathcal{H}$-Matrices," *Computing*, vol. 62, no. 2, pp. 89–108, 1999.

[5] R. Kriemann, "H-LU factorization on many-core systems," *Computing and Visualization in Science*, vol. 16, no. 3, pp. 105–117, 2013.

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

[7] W. Hackbusch and S. Börm, "Data-sparse Approximation by Adaptive $\mathcal{H}^2$-Matrices," *Computing*, vol. 69, no. 1, pp. 1–35, 2002.

[8] 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 Journal on Scientific Computing*, vol. 38, no. 5, pp. S358–S384, 2016.

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

[10] P.-G. Martinsson, "A fast randomized algorithm for computing a hierarchically semiseparable representation of a matrix," *SIAM Journal on Matrix Analysis and Applications*, vol. 32, no. 4, pp. 1251–1274, 2011.

[11] J. L. Xia, "Randomized sparse direct solvers," *SIAM Journal on Matrix Analysis and Applications*, vol. 34, no. 1, pp. 197–227, 2013.

[12] A. Aminfar and E. Darve, "A fast, memory efficient and robust sparse preconditioner based on a multifrontal approach with applications to finite-element matrices," *International Journal for Numerical Methods in Engineering*, vol. 107, no. 6, pp. 520–540, 2016.

[13] J. N. Chadwick and D. S. Bindel, "An Efficient Solver for Sparse Linear Systems Based on Rank-Structured Cholesky Factorization," *CoRR*, vol. abs/1507.05593, 2015.

[14] A. George, "Nested dissection of a regular finite element mesh," *SIAM Journal on Numerical Analysis*, vol. 10, no. 2, pp. 345–363, 1973.

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

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

[17] J. Dongarra, J. D. Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 16, no. 1, pp. 1–17, 1990.

[18] G. Pichon, M. Faverge, P. Ramet, and J. Roman, "Reordering Strategy for Blocking Optimization in Sparse Linear Solvers," *SIAM Journal on Matrix Analysis and Applications*, vol. 38, no. 1, pp. 226 – 248, 2017.

[19] E. Rebrova, G. Chavez, Y. Liu, P. Ghysels, and X. S. Li, "A study of clustering techniques and hierarchical matrix formats for kernel ridge regression," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2018, Vancouver, BC, Canada, May 21-25, 2018*, 2018, pp. 883–892.

[20] M. Bebendorf, *Hierarchical matrices*. Springer, 2008.

[21] C. D. Yu, J. Levitt, S. Reiz, and G. Biros, "Geometry-oblivious fmm for compressing dense spd matrices," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 53.

[22] T. Hofmann, B. Schölkopf, and A. J. Smola, "Kernel methods in machine learning," *The annals of statistics*, pp. 1171–1220, 2008.

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

[24] K. L. Ho and L. Ying, "Hierarchical Interpolative Factorization for Elliptic Operators: Differential Equations," *Communications on Pure and Applied Mathematics*, vol. 8, no. 69, pp. 1415–1451, 2016.

[25] C. Weisbecker, "Improving multifrontal solvers by means of algebraic block low-rank representations," Ph.D. dissertation, Institut National Polytechnique de Toulouse-INPT, 2013.

[26] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[27] 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. dissertation, Université de Bordeaux, Talence, France, Nov. 2018.

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

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

ISSN 0249-6399