Deciding Non-Compressible Blocks in Sparse Direct Solvers using Incomplete Factorization

Esragul Korkmaz, Mathieu Faverge, Grégoire Pichon, Pierre Ramet

Deciding Non-Compressible Blocks in Sparse Direct Solvers using Incomplete Factorization

Esragul Korkmaz*, Mathieu Faverge*, Grégoire Pichon† and Pierre Ramet*
*Inria Bordeaux - Sud-Ouest, Bordeaux INP, CNRS, University of Bordeaux
{esragul.korkmaz | mathieu.faverge | pierre.ramet}@inria.fr
†Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP
gregoire.pichon@inria.fr

***Abstract*—Low-rank compression techniques are very promising for reducing memory footprint and execution time on a large spectrum of linear solvers. Sparse direct supernodal approaches are one of these techniques. However, despite providing a very good scalability and reducing the memory footprint, they suffer from an important flops overhead in their unstructured low-rank updates. As a consequence, the execution time is not improved as expected.**

In this paper, we study a solution to improve low-rank compression techniques in sparse supernodal solvers. The proposed method tackles the overprice of the low-rank updates by identifying the blocks that have poor compression rates. We show that the fill-in levels of the graph based block incomplete LU factorization can be used in a new context to identify most of these non-compressible blocks at low cost. This identification enables to postpone the low-rank compression step to trade small extra memory consumption for a better time to solution. The solution is validated within the PASTIX library with a large set of application matrices. It demonstrates sequential and multi-threaded speedup up to $8.5\times$, for small memory overhead of less than $1.49\times$ with respect to the original version.

***Index Terms*—sparse direct solvers, low-rank compression, ILU factorization**

I. INTRODUCTION

In many engineering and scientific applications, solving a large sparse linear system of the form $Ax = b$ is a mandatory but very time consuming step. Among the many various approaches to solve these systems, sparse direct solvers [1] are a robust and widely used solution. However, they are known to be both time and memory consuming. Recent studies tackle these issues by experimenting different data-sparse compression schemes trading a controlled precision loss for better memory and computation complexities. These techniques include but are not limited to (Multilevel-)Block Low-Rank format (BLR) [2], [3], [4], $\mathcal{H}$-Matrices [5], [6], $\mathcal{H}^2$ [7], Hierarchical Off-Diagonal Low-Rank (HODLR) [8], [9] or Hierarchically Semi-Separable (HSS) [10], [11]. These techniques can be classified into two categories. The first one considers the full problem and extracts the sparsity of the matrix from the low-rank representation. The second solution exploits the existing sparsity of the matrix structure and compresses the blocks that compose it independently. This paper focuses on the latter. More specifically, it targets block low-rank methods (BLR).

In this context, as in other linear algebra solvers, one of the most important operation is the block update. When using regular block sizes, as in dense or sparse multifrontal solvers, the cost of the low-rank update is usually small with respect to the full-rank version. On the other hand, when various block sizes are involved, as in sparse supernodal solvers, the cost of the low-rank update depends on the largest block involved. Therefore, the update operation may become more expensive than the full-rank one. However, supernodal approaches provide more parallelism and have less memory overhead than multifrontal methods. As a consequence, in this paper, we propose to improve supernodal methods by trading a small memory overhead for lower flops count and better time to solution. For that purpose, we implement our solution in the sparse direct solver PASTIX [12], [4], which supports the BLR compression scheme.

The PASTIX solver offers two opposite strategies: favor memory peak reduction over time to solution (*Minimal Memory*), or prefer time to solution by delaying the data compression (*Just-In-Time*). The former suffers from the costly update operations, while the latter avoids it without reducing the memory consumption compared to the full-rank solver. Identifying the potential compressibility of each block is a key problem to benefit from both strategies. In this work, we propose a new technique based on incomplete factorization to define levels of admissibility (compressibility) for the blocks.

The incomplete LU (ILU) factorization is a well-known method to get a general preconditioner for the iterative solvers. The idea of incomplete factorization relies on dropping some entries with a given criterion. Among the many existing criteria, the most used are the fill-in levels heuristic [13], which is a graph based solution, and the threshold heuristic, which relies on the numerical values [14]. Block versions of these algorithms have also been developed to increase efficiency and parallelism and are still widely studied [15], [16], [17]. In this paper, we propose to exploit the block fill-in levels heuristic to provide an intermediate solution that exploits strengths of both *Minimal Memory* and *Just-In-Time* strategies by delaying the compression on blocks that may have higher ranks. The algorithm is applied during the preprocessing stage and provides a trade-off between memory and flops. We show that this solution, while targeting the *Minimal Memory* strategy, also improves the *Just-In-Time* solution in terms of

both memory and time. Moreover, by exploiting the benefits of the two strategies, the new heuristic allows the user to tune the fill-in level criterion according to his specific time and memory needs. Accuracy of the PASTIX block low-rank factorization has been studied in [4], [12] and is out-of-scope of this paper. The proposed approach does not impact the precision, and leads to accuracy results similar to those of *Minimal Memory* and *Just-In-Time* strategies. It is important to emphasize that the heuristic presented in this paper aims to improve the number of flops and the time to solution within the parallel framework of the supernodal sparse direct solver PASTIX, while keeping the memory usage at a reasonable level.

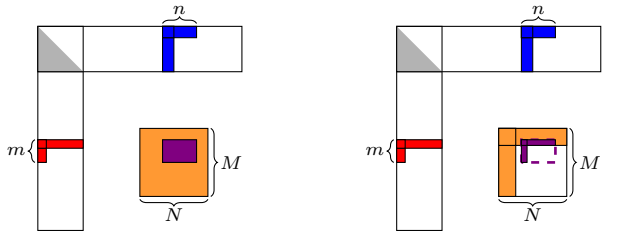
Section II sets the basis of this work by giving background information on the BLR implementation within the PASTIX solver and its limitations, as well as introducing the block ILU factorization. Section III presents the related work. Section IV details the new heuristic, which defines the non-compressible blocks to exploit the existing compression strategies in PASTIX. Section V analyses experiments on a large set of matrices. Finally, Section VI concludes on the results of this work and its perspectives.

II. BACKGROUND

This section provides background details on the PASTIX solver and its BLR implementation and recalls the general idea behind incomplete LU factorization.

A. Sparse supernodal direct solver using BLR compression

Sparse direct solvers generally follow four steps: 1) order the unknowns to reduce the fill-in that occurs during factorization; 2) perform a block-wise symbolic factorization to compute the structure of the final factorized matrix; 3) compute the actual numerical factorization based on the structure resulting from step 2; and 4) solve the triangular systems. In the remainder of the paper, we will focus only on the numerical factorization (step 3) and the matrix values initialization. We consider that steps 1 and 2 already exhibit a structure with large enough blocks to take advantage of BLAS Level 3 [18] operations whenever possible. Step 4 is not influenced by the work presented in this paper, and is thus not further discussed.



(a) Non-fully structured update (LR2FR)

(b) Fully structured update (LR2LR)

Fig. 1: Representation of the updates, $C -= AB$, for the *Just-In-Time* strategy on the left, and *Minimal Memory* on the right. A appears in red, B in blue, C in orange, and the impact of the contribution AB in purple.

BLR compression scheme has been introduced in the sparse supernodal solver PASTIX in [12], [4]. As already mentioned, the solver implements two strategies to target either lower memory cost (*Minimal Memory*), or lower time to solution (*Just-In-Time*). Both strategies differ depending on the update kernels, $C = C - AB$, that are involved in the numerical factorization. Figure 1 describes both options. On the left, the non-structured updates (LR2FR) lower the cost of the matrix product AB thanks to the low-rank representation. However, the updated C matrix is stored in full-rank to be able to perform a simple addition of the contribution at a cost of order mn, with m and n the dimensions of this contribution. On the right, although the low-rank structured update (LR2LR) is more complex, the C matrix is stored in low-rank to save memory. Here, while the contribution (AB) is also computed at a lower cost, the addition step into C requires a more complex low-rank to low-rank update with padding (zeroes are added to match the dimension of C). This update has a complexity of order MN, with M and N the dimensions of the updated matrix C. A more detailed complexity analysis is provided in [4].

The *Just-In-Time* strategy aims only at reducing the time to solution by exploiting the LR2FR update kernel. Indeed, LR2FR relies on high performance matrix-matrix multiplication kernels with smaller sizes than the ones of the full-rank implementation. On the other hand, the *Minimal Memory* strategy compresses the matrix at initialization exploiting the graph of the matrix to speedup the compression. Blocks of the factorized matrix which do not hold initial information are null and thus compressed at no cost. Other initial blocks compression can also be accelerated by automatically removing null columns and rows from the graph knowledge. This operation greatly improves the memory peak of the solver as the factorized matrix structure is never fully allocated. However, it forces the numerical factorization to rely on the LR2LR kernel. This kernel in the context of the supernodal method generates a flop overhead due to the padding operation. As a consequence, the factorization time may be highly impacted, unless the matrix is highly compressible. Thus, deciding which blocks to compress and when to do it is an important problem for supernodal solvers to reach good level of performance while benefiting from the lower memory consumption offered by low-rank techniques. Note that in sparse direct solvers, only blocks with sufficiently large sizes are considered to be admissible for low-rank compression. All small blocks are automatically defined as non-admissible due to their lower impact.

To decide which blocks to compress, [19] defined the admissibility condition. More specifically, the study proposes a strong and a weak admissibility conditions. The strong admissibility condition relies on the problem geometry and the definition of the diameter of a set of unknowns ($diam(\sigma)$), as well as the distance between two sets ($dist(\sigma, \tau)$). The interaction between two sets of unknowns is then considered admissible if the distance between the sets is sufficiently larger than both of the diameters of the sets ($max(diam(\sigma), diam(\tau)) \leq$

$\eta\, dist(\sigma, \tau)$). The least restrictive strong admissibility criterion considers that blocks are admissible only if their distance is larger than 0. Thus, all blocks except close neighbors are compressible. On the other hand, the weak criterion simply considers that all non-diagonal blocks are compressible, which is equivalent to the *Minimal Memory* strategy. This paper proposes to generalize the distance criterion used in the strong admissibility condition by computing *algebraic* distances of the blocks without any geometry knowledge requirement. The distance computation is performed similarly to block-wise ILU factorization.

B. Incomplete LU factorization

The incomplete LU (ILU) factorization is an approximated version of the LU factorization, where part of the information is dropped [20]. It has the form $A \approx LU = LU + R$. Here, the matrix R carries the negative values of the dropped elements. This method is usually used as a preconditioner for iterative methods [21].

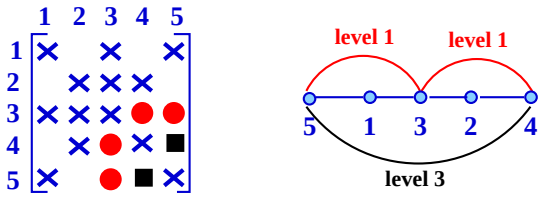


Fig. 2: An adjacency matrix (on the left) and its associated graph (on the right). Fill-in entries that may occur during the numerical factorization are represented in red (level 1) and black (level 3).

Multiple dropping heuristics have been studied through the years targeting either parallelism and efficiency of the implementation or a good numerical accuracy of the preconditioner. Among the dropping heuristics, there exist non-numerical solutions: using the position, or the fill-in levels [13], ILU(k), and numerical solutions: using a threshold [14] value, ILU(τ). Through the years, their block-based variant have been studied to improve their efficiency [15], [16], [17]. While we may consider low-rank solvers as a solution similar to ILU(τ), we exploit in this paper the block ILU(k) approach to enrich the block low-rank solver. The ILU method with the fill-in levels definition as we use is first suggested in 1981 [22] and improved by a graph-based definition through fill-path theorem [23] in [24]. Figure 2 illustrates the idea of the fill-in levels that is strongly related to the ordering of the unknowns. On the left, the matrix non-zeroes pattern is represented, where the blue crosses are the original entries. On the right, the graph associated to this matrix is shown. During the numerical factorization, some entries may become non-zeroes (fill-in). On Figure 2, these entries are represented in red and black, both on the matrix and as new edges on the graph. We can define the fill-in level as the length of the path connecting the two unknowns in the original graph. The path connecting 3 and 5 (and 3 and 4) in red goes only through 1 (resp. 2). Thus, the level of the fill-in between these two unknowns is 1. We can also see that 4 and 5 are connected at a level 3 (the path goes through 1, 3 and 2). As the fill-in level gets higher, the value of the new entry becomes smaller as it represents far interactions in the graph. That is, the fill-in levels can represent the generalized algebraic distance in the low-rank strong admissibility condition, without any knowledge of the geometry. Therefore, the ILU factorization can be implemented by dropping the values which have higher fill-in levels than a predefined maximum level. Similarly, this procedure can be applied in a block-wise fashion. Thus, considering the block fill-in levels as an admissibility criterion for low-rank compression, we can algebraically decide on which blocks the compression should be delayed to reduce the overhead of the fully structured updates.

III. Related work

Many recent studies have tackled the problem of reducing the memory consumption of linear solvers with low-rank compression.

In [19], the adaptation of hierarchical matrix techniques issued from the dense community to sparse matrices is studied. Here, by ignoring some structural zeroes, the opportunity for further memory savings is missed. Although low-rank updates are performed similarly to the *Minimal Memory* strategy, padding is not used as the zeroes are explicitly stored. This results in more efficient updates at the cost of a larger memory consumption.

Low-rank updates in the context of sparse supernodal solvers have already been studied in [9]. In this work, the authors considered fixed ranks for the blocks which must be known in advance. They demonstrated interesting memory savings, but with a slower factorization than the full-rank version.

The sparse multifrontal BLR solver MUMPS [25], [26] implements two close solutions: CUFS (Compress, Update, Factor, Solve) which is similar to our *Minimal Memory* scenario, and FCSU which is closer to the *Just-In-Time* scenario. However, as it is a multifrontal solver, these strategies are applied with tiled algorithms on the dense matrices of the fronts that appear during the factorization. Here, the memory saving is limited as fronts are allocated in full-rank before being compressed.

In [27], the preselection problem is approached from a different angle. The authors exploit performance models of the update kernel to decide whether or not to delay the compression of some of the blocks. In this work, this decision is taken at runtime during the numerical factorization and requires to generate correct models of the problem.

IV. Deciding the Non-Compressible Blocks

As mentioned in Section II, the *Minimal Memory* strategy suffers from the complexity overhead of the LR2LR update kernel. We propose to exploit ILU fill-in levels to identify, at low cost, the blocks with large ranks. These blocks will increase the cost of the update step while providing only a small memory reduction. Once identified, it is possible to

postpone the compression of these blocks as late as possible to replace the LR2LR kernels by LR2FR. This comes at the cost of a controlled memory overhead if the identification is correct. In Section II-B, we mentioned that as the ILU fill-in levels get larger, the magnitude of the entries gets smaller. Thus, blocks with large level values should have small ranks and should be kept compressed to save memory, while blocks with small level values should have high ranks.

Algorithm 1 Cholesky-based ILU fill-in levels initialization

```
1:  for all block A_ij in A do
2:      lvl(A_ij) = (A_ij ≠ 0) ? 0 : ∞
3:  end for
4:  for all column block A_*k in A do
5:      for all block A_ik in A_*k do
6:          for all block A_jk in A_*k (with j > i) do
7:              lvl(A_ij) = min(lvl(A_ij), lvl(A_ik) + lvl(A_jk) + 1)
8:          end for
9:      end for
10: end for
```

Algorithm 1 presents the main steps to compute the fill-in levels of the blocks. This algorithm performs the same loops as the numerical factorization focusing only on the fill-in level information and the symbolic structure of the factorized matrix L. Initially, all blocks are considered with level 0, if they are part of the original matrix A, or ∞ if they are created by fill-in (lines 1-3). Then, the main loop updates the levels according to the formula given in [20], which is adapted to the block-wise algorithm (Line 7). Note that, as PASTIX uses the symmetric pattern structure of $A + A^T$ even for LU factorization, Cholesky-based algorithms are presented. For strongly non-symmetric matrices, the fill-in levels of the blocks in L and U are computed separately for better identification. This very cheap algorithm is in fact fully integrated within the parallel matrix initialization to avoid an extra loop over the structure, and such that its cost is completely hidden to the user.

Now that the fill-in levels are computed, the numerical factorization can be adapted to exploit this information. Algorithm 2 presents the proposed algorithm with a generic parameter $maxlevel$, which allows to set the new admissibility criterion. First, lines $1-5$ compress the admissible blocks which have a fill-in level larger than $maxlevel$. These blocks have the smallest ranks and are compressed before starting the numerical factorization loop, which is at lines 6-17. Thus, they will be involved in LR2LR updates and are the most important ones to compress to reduce the memory footprint. On the other hand, the non-admissible blocks will be involved in LR2FR updates to reduce the flops count overhead while inducing a small memory overhead. These blocks are still compressed (lines 9-11), just after the factorization of the diagonal block, to reduce the cost of the following operations: solve and updates. In the remainder of the paper, we will refer to this BLR sparse factorization as $ILU(k)$, with k the maximum level of the admissibility criterion. Note that choosing the right

Algorithm 2 Cholesky BLR factorization with $maxlevel$ admissibility

```
1:  for all block A_ij in A do
2:      if lvl(A_ij) > maxlevel then
3:          Compress(A_ij)
4:      end if
5:  end for
6:  for all column block A_*k in A do
7:      Factorize(A_kk)
8:      for all block A_ik in A_*k do
9:          if lvl(A_ik) <= maxlevel then
10:             Compress(A_ik)
11:         end if
12:         Solve( A_kk, A_ik )
13:         for all block A_jk in A_*k (with j <= i) do
14:             Update( A_ik, A_jk, A_ij )
15:         end for
16:     end for
17: end for
```

k value for a given problem is important. As a matter of fact, the larger the number of non-admissible blocks, the higher the memory overhead.

It is important to observe that $ILU(-1)$ is the *Minimal Memory* scenario as all the admissible blocks are compressed during the initialization. On the opposite, $ILU(\infty)$ corresponds to the *Just-In-Time* scenario as all blocks are compressed only after all the updates were accumulated.

V. EXPERIMENTS

All the experiments are performed through the BLR supernodal direct solver PASTIX [4], using the `miriel` nodes of the *Plafrim*[1] supercomputer. Each `miriel` node is equipped with two INTEL `Xeon E5-2680v3` 12-cores running at 2.50 GHz and 128 GB of memory. For the multi-threaded experiments, we use 24 threads, one per core, with the default scheduler of the PASTIX library. The INTEL MKL `2020` is used for the BLAS kernels. The minimum block width and height criteria to allow compression are set to 128 and 20, respectively. In the experiments, we use only LDL^T and LU factorizations according to the input matrix features. For the SPD matrices, we avoid LL^T factorization as the positive definite property can be affected by the compression. In the following sections, all the experiments are run for a set of 31 real case matrices taken in the SuiteSparse Matrix Collection [28]. Data reported in the graphs are only related to the numerical factorization. The solve step is never considered as it is not impacted by this new algorithm. Interested reader can find an experimental backward error study on the *Minimal Memory* and *Just-In-Time* strategies in [12] that shows the numerical stability of the methods. The times shown are the average of 3 runs on each matrix.

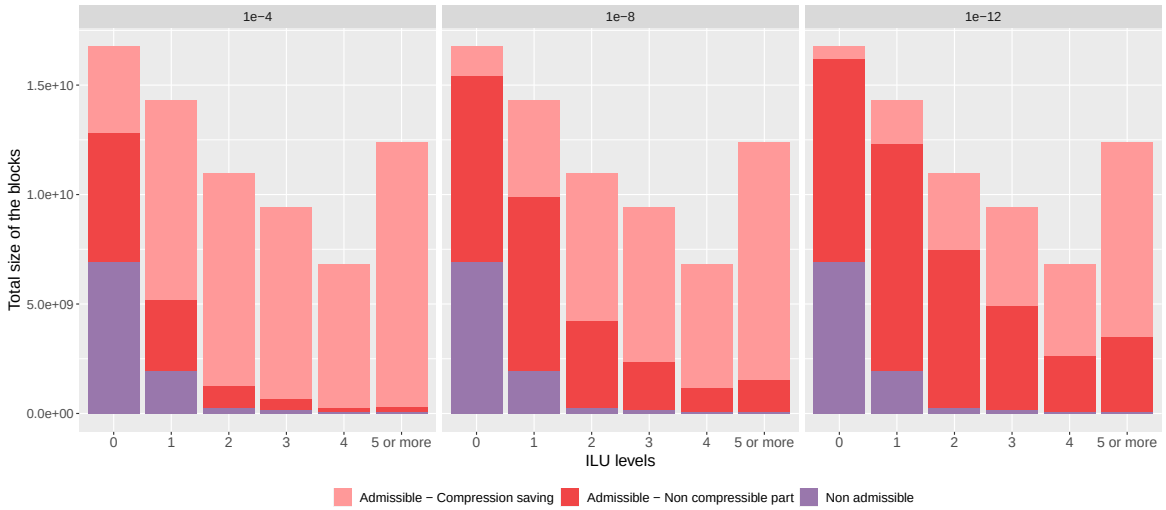


Fig. 3: Potential memory saving based on the tolerance criterion and fill-in levels. The bars report the cumulative memory of the 31 matrices. Purple represents the memory consumed by blocks below the size criteria, and red the memory of the admissible blocks for compression. Light red is the portion that can be saved when compressed.

A. *Compressibility statistics*

We first want to validate the hypothesis that using the fill-in levels is a good heuristic to classify the blocks based on their compressibility ratios. Figure 3 reports the accumulated memory consumption for all the 31 studied matrices in our experiments with three different tolerance criteria and by fill-in levels. The purple bars show the memory consumption of the structurally non-admissible blocks (too small to be compressed) and is stable for all precisions. The red part corresponds to the admissible blocks. The dark red is the memory footprint when they are compressed. It naturally increases with a higher precision. The light red shows the amount of memory that can be saved by compressing the blocks. One can observe that the lower the precision requirement, the higher the gain.

These results show that the compression ratio of the admissible blocks increases with the levels. It confirms the original hypothesis, that fill-in levels can help to better tune the admissibility criterion in order to save flops for a small memory overhead. Furthermore, this parameter needs tuning to adapt to the tolerance. One can see that for a tolerance of $1e^{-12}$, only levels greater than 2 offer more than 40% memory savings, while all levels at $1e^{-4}$ reach it. As a consequence, the fill-in level used to define the admissibility criterion will need to be adapted to both the tolerance and the maximum memory overhead defined by the user.

B. *Impact of the fill-in level heuristic on the sequential version*

This section discusses the sequential experiments. Figure 4 shows the memory peak, factorization flops and factorization time profiles obtained for different precisions. We study the impact of the first fill-in levels (0 to 4) with respect to *Minimal Memory* (-1) and *Just-In-Time* (∞). Each curve represents the number of matrices within a percentage overhead of the best solution for each metric and matrix.

First, as expected, the lower the fill-in level chosen for admissibility, the lower the memory peak of the solver. One can observe that the impact of the fill-in level increases as the precision decreases. This confirms the trend already observed on Figure 3. $ILU(\infty)$ consumes up to 6.6 times more memory at $1e^{-4}$, while it drops to 3.4 times at $1e^{-12}$. Additionally, at this high precision, high levels of fill-in are able to reach the best memory peak. This means that potential flops reduction is possible without negatively impacting the memory.

Second, when observing the flops count evolution, the results are naturally reversed. The higher levels of fill-in are better to generate less flops. One can observe that for low precision $(1e^{-4})$, a level of 0 is enough to reach the same flops count as the *Just-In-Time* scenario $(ILU(\infty))$. When increasing the precision, more levels need to be considered non-admissible to lower the flops count to its minimal value. Except some corner cases, levels 1 or 2 are enough for $1e^{-8}$, and respectively 3 or 4 for $1e^{-12}$.

Finally, the time profiles follow the same trend as the flops profiles, with larger differences between the $ILU(k)$ methods. This can be explained by the disparity of the LR2LR and LR2FR efficiency, as well as their variation in number that may increase the phenomena already observed on flops. Thus, one can observe that $ILU(0)$ is the best average solution at $1e^{-4}$. It even outperforms the *Just-In-Time* strategy by a factor up to $1.4\times$. To explain this performance, we recall that in the *Just-In-Time* strategy, many null-rank blocks are allocated and later compressed, while in the new $ILU(k)$ heuristic they may never be allocated. Indeed, they are originally null blocks and at low precision they may receive only null contributions. Thanks to these savings, $ILU(0)$ at this tolerance almost dou-

[1]https://www.plafrim.fr

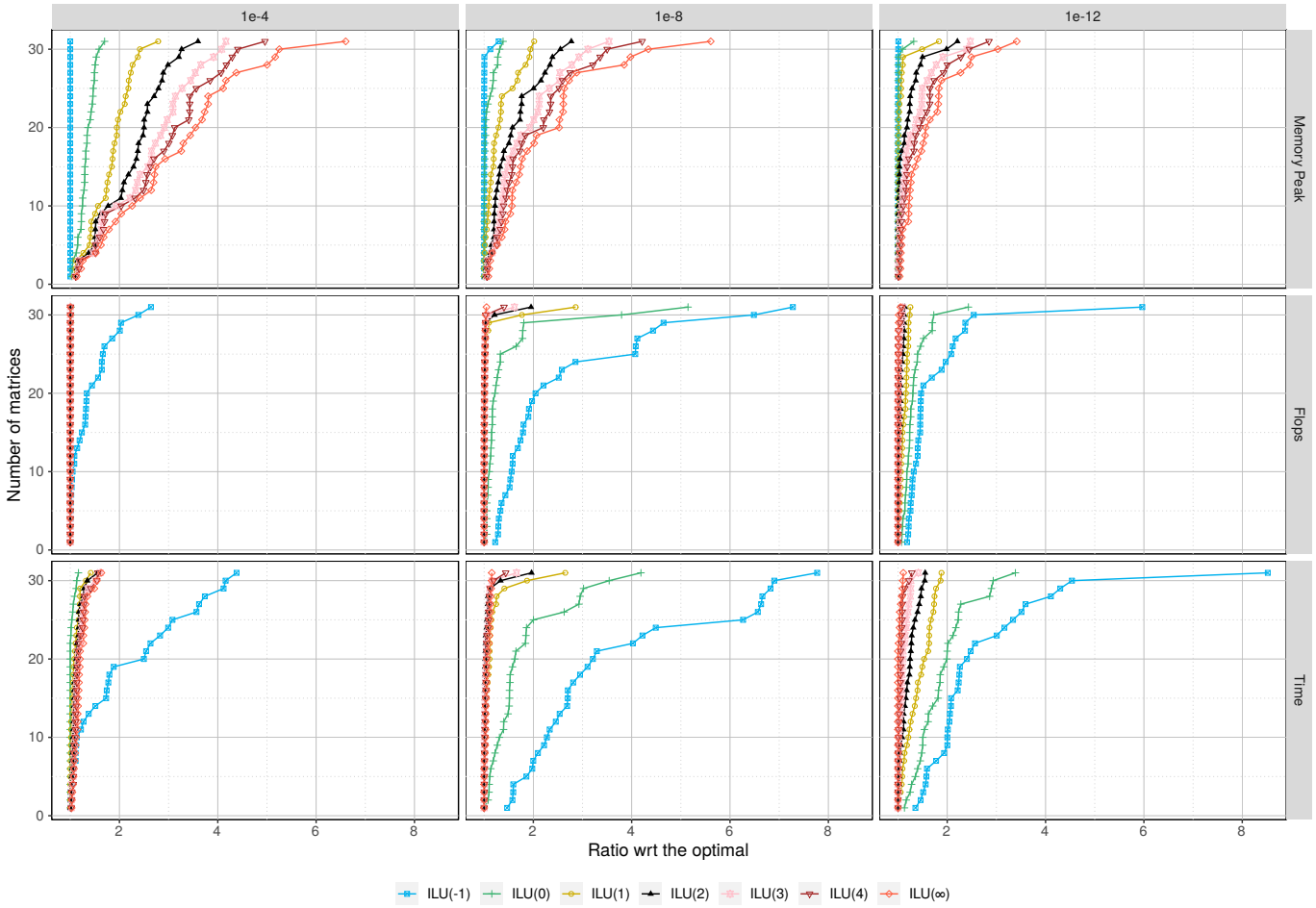


Fig. 4: Memory peak, factorization flops and factorization time profiles with different precisions for sequential runs. Each color stands for a different $ILU(k)$ level. The x-axis shows percentages with respect to the best method for each metric and matrix, while y-axis represents the matrix count in accumulated way.

bles the memory footprint in the worst case with respect to the *Minimal Memory* strategy, but it remains $0.23\times$ the memory consumption of the $ILU(\infty)$ and the full-rank versions.

The observations in higher precisions are similar to the lower precision, but with higher fill-in levels. At $1e^{-8}$, only the levels above 1 compete with the *Just-In-Time* strategy in terms of time, as well as providing a controlled memory overhead with respect to the best solution.

At $1e^{-12}$, the levels higher than 3 are required to get the best factorization time, which reduces the gain one can obtain on the memory footprint. However, solutions with level 1 is a good compromise at this precision. It is up to $8.5\times$ faster than $ILU(-1)$, with only up to $1.49\times$ more memory usage.

Figure 5 presents the detailed ratios of the memory peak of the different levels of admissibility with respect to the full-rank solution. On the left figure, the matrices are ordered by families and by increasing memory ratio of $ILU(-1)$ at a tolerance of $1e^{-12}$. As we can observe, the compression ratio of all matrices varies a lot with the matrices and the tolerances. It also reflects the fact that increasing the fill-in level does not necessarily means an extra memory usage in high precision problems as the dots are merged together. On the right figure, this trend is summarized with boxplots representing the average gain. One can observe that the *Just-In-Time* ($ILU(\infty)$) strategy has the same memory peak as the full-rank version, and that the lower the level, the lower the memory peak. One can also observe that increasing the level of admissibility at low precision seems to greatly increase the memory footprint with respect to the best one. However, the impact remains moderate for high precision and it can be afforded to highly reduce the time to solution, as observed in Figure 4.

The summary on the right of Figure 5 confirms the fact that the memory consumption increases with the higher fill-in levels and with the higher precision. The results at $1e^{-12}$ for the new heuristic at levels 1 and 2 show interesting results. As a matter of fact, they provide in average 25% memory improvement compared to the full-rank version. Note that, at this precision, even $ILU(\infty)$ does not manage to efficiently

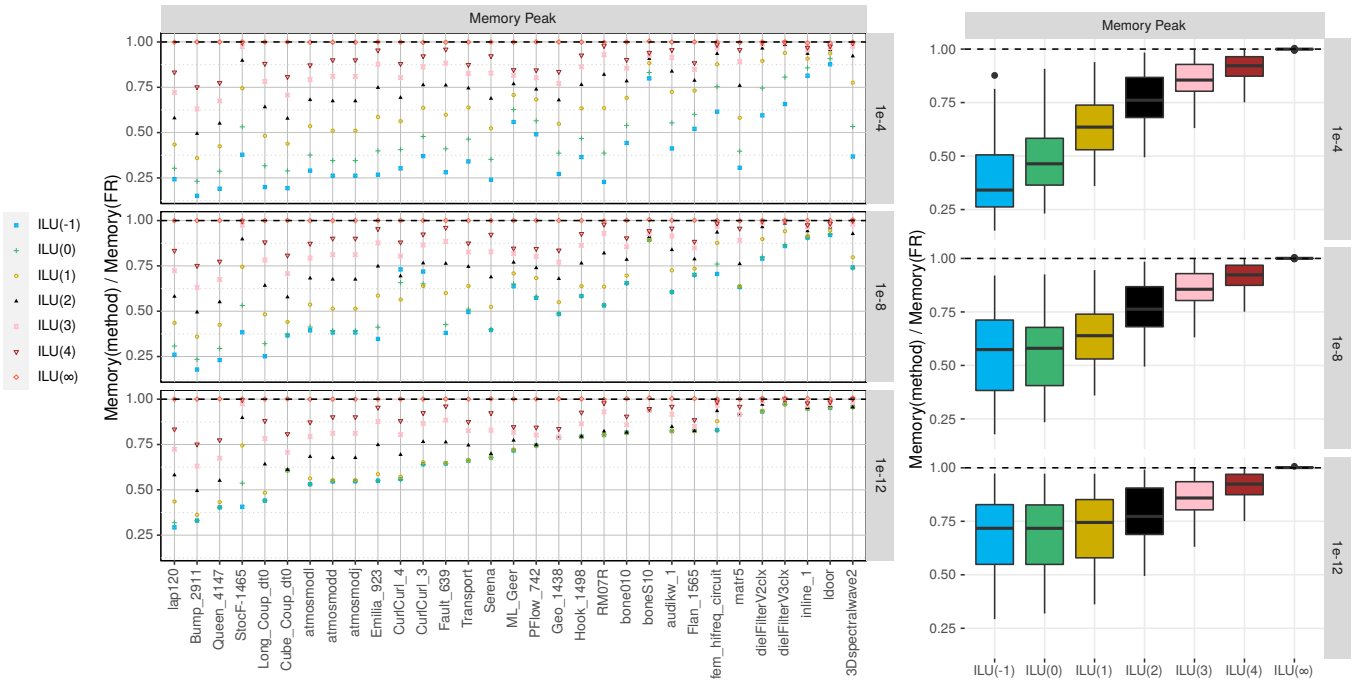


Fig. 5: Memory peak ratio of the $ILU(k)$ heuristic with respect to full-rank on the 31 test matrices. On the left, the detailed information is presented for each matrix and precision. On the right, the information is summarized with boxplots showing minimum, maximum, median, first quartile and third quartile for each fill-in level.

accelerate the full-rank version on the five right-most cases, which are poorly compressible. On the other matrices, in addition to this memory saving, levels 1 and 2 are also faster than the full-rank version.

To conclude, it is difficult to give a single level as the optimal solution. However, depending on the problem, as well as the precision and memory restrictions, the level can be tuned to provide a solution that outperforms the *Minimal Memory* strategy in terms of time, for a small controlled memory overhead. Moreover, it can even have a speedup compared to the *Just-In-Time* strategy, while reducing the memory footprint.

C. Impact of the fill-in level heuristic on the multi-threaded version

This section presents the results of the previous experiments in a multi-threaded environment with 24 threads. Figure 6 shows the time profiles of the multi-threaded numerical factorization on the set of 31 matrices. Memory peak and flops are not reported as they are identical to the sequential ones.

We can observe that the impact of the new heuristic is even greater in the multi-threaded environment. The shift in the memory usage induced by the heuristic improves the memory bandwidth in the multi-threaded context and allows to get better performance. The $ILU(k)$ heuristic performs better respectively with a level of 0, 2 and 4, for tolerances of $1e^{-4}$, $1e^{-8}$, and $1e^{-12}$. The proposed solution outperforms the *Just-In-Time* strategy as it can be especially seen at $1e^{-4}$. In this parallel context, the large memory reduction improves the memory contention of the threads, while it merely degrades the flops count with respect to the $ILU(\infty)$. $ILU(0)$, which initially stores only the blocks of the original matrix A, clearly outperforms the other versions at low precisions. When increasing the precision ($1e^{-8}$ or $1e^{-12}$), the extra flops count for small values of k degrades the performance. However, one can observe that $ILU(2)$ and $ILU(4)$ (at $1e^{-8}$ and $1e^{-12}$, respectively) are good competitors with $ILU(\infty)$ in terms of time, while they still induce memory savings as opposed to $ILU(\infty)$.

To better highlight the high gain obtained with our new heuristic, we compare it to the full-rank solver and to the previous low-rank strategies existing in the PaStiX solver. Figure 7 presents the time profiles of the multi-threaded numerical factorization with 24 threads on the left, as well as the memory peak at $1e^{-8}$ precision on the right. On Figure 7a, $ILU(best\,time)$ refers to the best factorization time obtained with the new heuristic, where the fill-in level is taken in the range $ILU(0)$ to $ILU(4)$ included. Similarly, in Figure 7b the $ILU(best\,memory)$ stands for the best memory consumption reachable with a fill-in level in the same range. Note that, on Figure 7b, the memory consumption of the full-rank solver and the $ILU(\infty)$ are mingled, as they are identical. The left figure shows that the new heuristics is up to 45 times faster than the full-rank solver, and is up to almost 2 times faster than the previous fastest implementation $ILU(\infty)$ except for

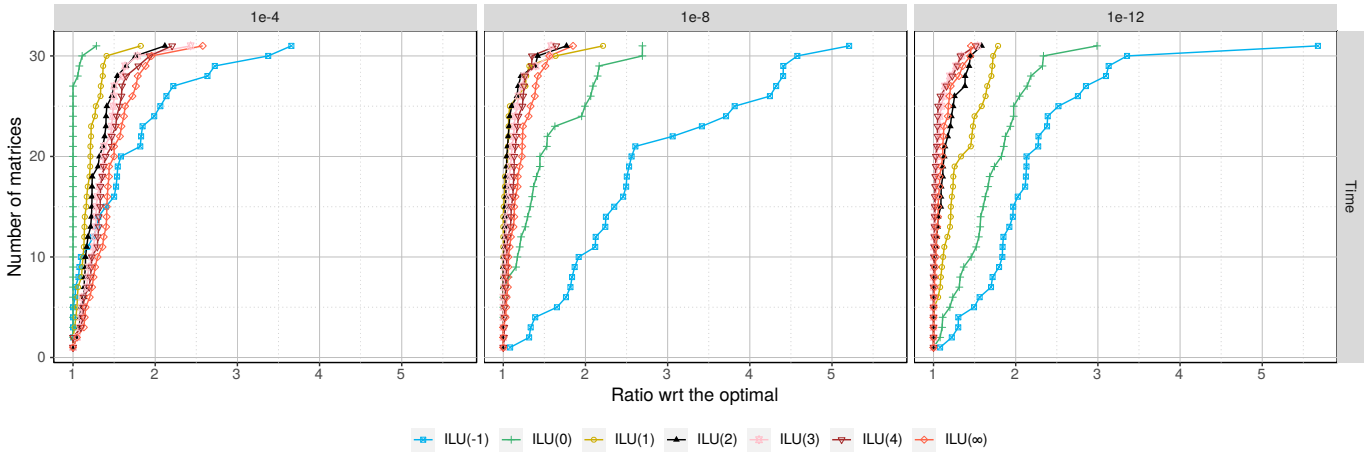
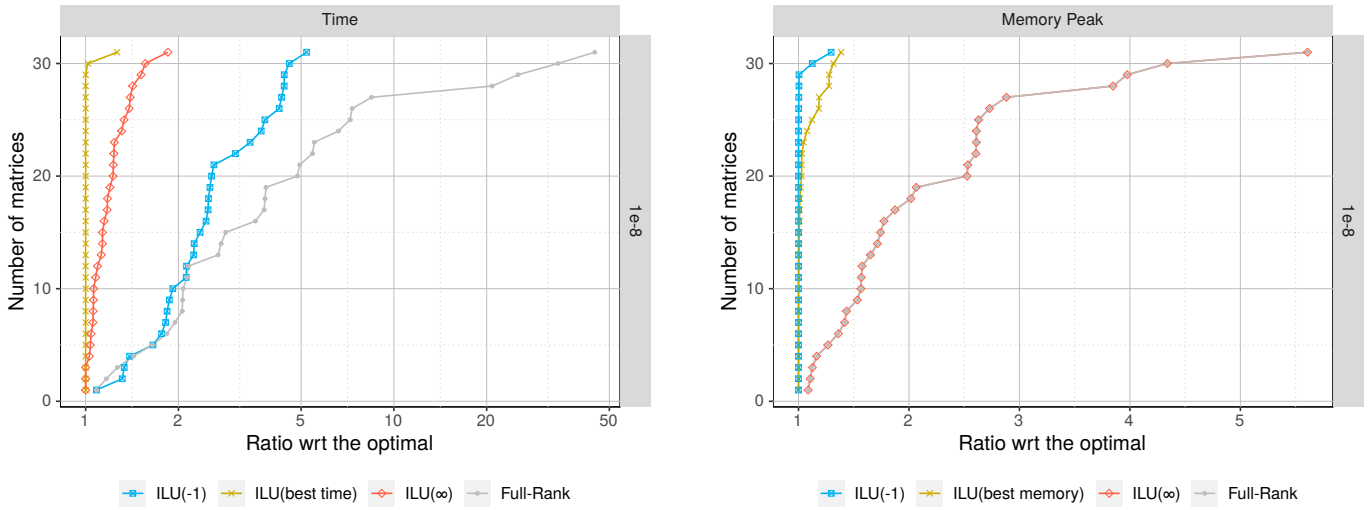


Fig. 6: Time profiles of the multi-threaded runs for different precisions. Each color shows a different $ILU(k)$ level result.



(a) Execution time of $ILU(k)$ heuristic compared to existing methods, where the level is chosen between 0 and 4 to obtain the best time.

(b) Memory peak of $ILU(k)$ heuristic compared to existing methods, where the level is chosen between 0 and 4 to obtain the smallest memory consumption.

Fig. 7: Best results achievable for time and memory with $ILU(k)$ heuristic, tuning the fill-in level between 0 and 4, at $1e^{-8}$ and using 24 threads.

the two `CurlCurl` matrices which would require higher fill-in levels to get better performance due to the high ranks of their contributions.

Overall, the proposed heuristic, if correctly tuned, is the fastest method for all matrices and outperforms the $ILU(\infty)$ solution while providing an effective memory footprint reduction. Furthermore, the memory footprint reduction is similar to the one obtained with the $ILU(-1)$ strategy in two thirds of the cases (using a fill-in level of 0 in most cases), and reaches at most a 39% overhead in the last set of matrices. One can notice that in two cases (the two `CurlCurl` matrices), the new heuristic outperforms the $ILU(-1)$ strategy thanks to a slight change in the order of allocation of the blocks. In these specific cases, the best memory peak is even obtained with a fill-in level of 1 that better delayed the workspace allocation to compress the blocks of the matrix.

D. Study of the Serena matrix in multi-threaded environment

Tolerance	$ILU(tuned)$
$1e^{-4}$	$ILU(0)$
$1e^{-6}$	$ILU(2)$
$1e^{-8}$	$ILU(2)$
$1e^{-10}$	$ILU(4)$
$1e^{-12}$	$ILU(4)$

TABLE I: Corresponding levels for $ILU(tuned)$ at each precision.

NbThreads	$1e^{-4}$			$1e^{-8}$			$1e^{-12}$			Full-Rank
	$ILU(-1)$	$ILU(0)$	$ILU(\infty)$	$ILU(-1)$	$ILU(2)$	$ILU(\infty)$	$ILU(-1)$	$ILU(4)$	$ILU(\infty)$	
1	26.2	21.1	26.5	330.7	109.7	111.5	1154.9	468.1	446.4	1025.9
4	7.4 (3.5)	6.6 (3.2)	9.7 (2.7)	91.8 (3.6)	32.8 (3.3)	35.7 (3.1)	318.2 (3.6)	129.2 (3.6)	124.7 (3.6)	279.3 (3.7)
6	5.5 (4.8)	4.8 (4.4)	6.8 (3.9)	64.5 (5.1)	24.2 (4.5)	27.6 (4.0)	230.6 (5.0)	92.8 (5.0)	91.7 (4.9)	203.2 (5.0)
12	3.5 (7.5)	2.5 (8.4)	4.7 (5.6)	35.1 (9.4)	13.8 (7.9)	16.7 (6.7)	131.7 (8.8)	52.8 (8.9)	55.4 (8.1)	137.1 (7.5)
24	4.4 (6.0)	2.2 (9.7)	6.4 (4.1)	35.1 (9.4)	11.2 (9.8)	14.4 (7.8)	104.9 (11.0)	37.7 (12.4)	40.3 (11.1)	98.4 (10.4)

TABLE II: Factorization times for the Serena matrix. Speedup with respect to the sequential runs are written inside parentheses.

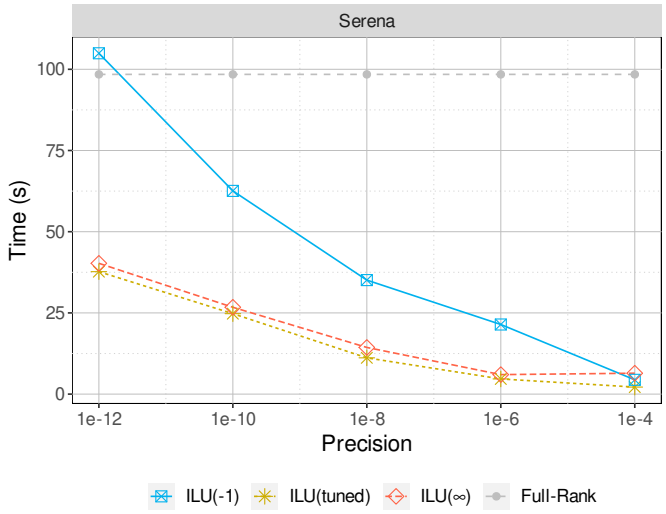


Fig. 8: Serena matrix time results of different precisions with 24 threads. The $ILU(tuned)$ corresponds to the smallest level, which runs faster than $ILU(\infty)$.

In this section we focus the study on one of the large matrices from the collection: the `Serena` matrix. The size of the matrix is $N = 1\,391\,349$ for $32\,961\,525$ of non zeroes and it requires 28.6 TFlops to be factorized in full-rank, which makes it a good average test case of the collection. Figure 8 presents a performance study of different compression tolerances with 24 threads. The ILU(tuned) corresponds to the use of levels chosen for each tolerance as reported in Table I. The full-rank result is shown in the figure as a reference for the speedup observation. In conclusion, the new heuristic outperforms the former two solutions on all precisions with a small fill-in level of 0 to 4, relatively to the precision. As shown previously, at these levels, this solution even provides an important memory gain as opposed to the fastest existing solution $ILU(\infty)$.

Table II reports the detailed factorization times of the Serena matrix with different number of threads and tolerances. The values inside parentheses present the speedup compared to the sequential run of each method at the corresponding precision. The $ILU(k)$ heuristic levels are chosen as in Table I. The level selection has been limited to the range 0 to 4 to ensure an useful memory saving compared to $ILU(\infty)$. That is why, with a small number of threads at $1e^{-12}$, our method cannot run faster than $ILU(\infty)$. However, the $ILU(k)$ heuristic benefits from a higher scalability, which allows it to quickly outperform other solutions as the number of threads increases. These results, while not reported in this article, are similar to the ones observed on the set of matrices used for the experiments. To conclude, the $ILU(k)$ heuristic, through correct tuning of the levels solves the original issue of the flops overhead of $ILU(-1)$. Furthermore, thanks to its better memory footprint, it provides a better scalability in parallel environments and outperforms the original fastest solutions.

VI. Conclusion

The behavior of sparse supernodal direct solvers using low-rank compression highly depends on when the compression is performed. On one hand, all admissible blocks can be compressed before the factorization (as it happens with the *Minimal Memory* / $ILU(-1)$ strategy). It allows high memory savings, but in the specific case of supernodal methods, it induces an expensive flops overhead during the low-rank updates. On the other hand, admissible blocks can be compressed after they have received all their updates (as for the *Just-In-Time* / $ILU(\infty)$ strategy). It reduces significantly the flops count as in dense solvers, and thus the execution time. However, the memory peak of allocating the matrix L still exists and it needs to be carefully controlled to delay the allocation to their first access.

In this paper, we proposed a new heuristic to estimate the compressibility of each block and constructed an algorithm that is a compromise between the two strategies mentioned previously. The new heuristic, named $ILU(k)$, identifies poorly compressible blocks similarly to the ILU method, which identifies the most important data. It relies on the block ILU fill-in levels to define an *algebraic* distance to compute low-rank admissibility of the blocks. The purpose of defining the admissibility is to propose an intermediate solution that accelerates the *Minimal Memory* solution, while it slightly increases the memory consumption. Moreover, it gives the chance to tune the levels according to the precision, the matrix properties, and the characteristics of the machine to better exploit the advantages of both the *Minimal Memory* and the *Just-In-Time* strategies.

The experiments that we conducted on a large set of 31 real matrices demonstrated that the $ILU(k)$ heuristic manages to identify efficiently the low-rank blocks. The solution proposed runs up to 5.2 times faster than *Minimal Memory* with only a 1.38 times increase of memory usage for high precision in both sequential and multi-threaded environments. Moreover, due to the elimination of the null blocks before the numerical

factorization, the $ILU(k)$ heuristic is also able to run 1.4 times faster than the *Just-In-Time* strategy in sequential, with a much lower memory consumption (0.23 times less). In the multi-threaded environment, it even goes up to 1.84 times faster and outperforms it for most of the cases. We showed that through correctly tuned fill-in levels, the $ILU(k)$ heuristic can be used as an improved version of the existing strategies. It improves the numerical factorization in terms of both memory and time, and it improves the scalability for parallel environments.

Despite the high gain of the $ILU(k)$ heuristic, it remains difficult to know beforehand which level will provide the best improvement. Section V has shown that only the first levels are worth consideration, and that a clear trend appears on the level to use depending on the tolerance. In future work, we would like to introduce tuning techniques to automatically infer the best value of k depending on the given tolerance, as well as the properties of the machine and the matrix used. An on-going work study this heuristic in distributed environments to evaluate its impact on the volume of communication, and validate the scalability improvement. In addition, we did not consider GPUs in this paper, as the PaStiX solver does not provide the adapted GPU kernels for the low-rank updates, and this will be part of future studies.

Acknowledgments

This work is supported by the Agence Nationale de la Recherche, under grant ANR-18-CE46-0006 (SaSHiMi). Experiments presented in this paper were carried out using the PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine (https://www.plafrim.fr/).

References

[1] I. S. Duff, A. M. Erisman, and J. K. Reid, "Direct methods for sparse matrices," *Clarendon Press*, 1986.

[2] 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.

[3] P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary, "Bridging the gap between flat and hierarchical low-rank matrix formats: the multilevel BLR format," *SIAM Journal on Scientific Computing*, vol. 41, no. 3, pp. A1414–A1442, May 2019.

[4] 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.

[5] 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.

[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, Paris, France, 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] 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.

[9] 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.

[10] 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.

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

[12] G. Pichon, "On the use of low-rank arithmetic to reduce the complexity of parallel sparse linear solvers based on direct factorization technique," Ph.D. dissertation, Universite de Bordeaux, Bordeaux, France, 2018.

[13] T. A. Davis and I. S. Duff, "An unsymmetric-pattern multifrontal method for sparse LU factorization," *SIAM Journal on Matrix Analysis and Applications*, vol. 18, no. 1, pp. 140–158, 1997.

[14] G. Karypis and V. Kumar, "Parallel threshold-based ilu factorization," *proceedings of the IEEE/ACM SC97 Conference*, 1997.

[15] A. Gupta, "Enhancing Performance and Robustness of ILU Preconditioners by Blocking and Selective Transposition," *SIAM Journal on Scientific Computing*, vol. 39, no. 1, pp. A303–A332, 2017. [Online]. Available: https://doi.org/10.1137/15M1053256

[16] M. Bollhöfer, O. Schenk, and F. Verbosio, "A high performance level-block approximate LU factorization preconditioner algorithm," *Applied Numerical Mathematics*, vol. 162, 01 2021.

[17] H. Anzt, T. Ribizel, G. Flegar, E. Chow, and J. Dongarra, "ParILUT - a parallel threshold ILU for GPUs," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 231–241.

[18] J. Dongarra, J. Du 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.

[19] W. Hackbusch, *Hierarchical Matrices: Algorithms and Analysis*. Springer, 12 2015, vol. 49.

[20] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. USA: Society for Industrial and Applied Mathematics, 2003.

[21] P. Hénon, P. Ramet, and J. Roman, "On finding approximate supernodes for an efficient block-ILU(k) factorization," *Parallel Computing*, vol. 34, no. 6, pp. 345–362, 2008, parallel Matrix Algorithms and Applications. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819107001330

[22] J. W. Watts, "A conjugate gradient truncated direct method for the iterative solution of the reservoir simulation pressure equation," *Society of Petroleum Engineers Journal*, vol. 21, pp. 345–353, 1981.

[23] D. J. Rose and R. E. Tarjan, "Algorithmic aspects of vertex elimination on directed graphs," *SIAM J. Appl. Math.*, vol. 34, no. 1, pp. 176–197, Jan. 1978.

[24] D. Hysom and A. Pothen, "A scalable parallel algorithm for incomplete factor preconditioning," *SIAM J. Sci. Comput.*, vol. 22, pp. 2194–2215, 2001.

[25] P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary, "On the complexity of the block low-rank multifrontal factorization," *SIAM Journal on Scientific Computing*, vol. 39, no. 4, pp. 1710–1740, 2017. [Online]. Available: https://oatao.univ-toulouse.fr/19066/

[26] T. Mary, "Block Low-Rank multifrontal solvers: complexity, performance, and scalability," Ph.D. dissertation, Toulouse University, Toulouse, France, Nov. 2017.

[27] L. Marchal, T. Marette, G. Pichon, and F. Vivien, "Trading Performance for Memory in Sparse Direct Solvers using Low-rank Compression," INRIA, Research Report RR-9368, Oct. 2020. [Online]. Available: https://hal.inria.fr/hal-02976233

[28] 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.