Trading Performance for Memory in Sparse Direct Solvers using Low-rank Compression

Loris Marchal, Thibault Marette, Grégoire Pichon, Frédéric Vivien

Trading Performance for Memory in Sparse Direct Solvers using Low-rank Compression

Loris Marchal, Thibault Marette, Grégoire Pichon, Frédéric Vivien

Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, F-69342, LYON Cedex 07, France

Abstract

Sparse direct solvers using Block Low-Rank compression have been proven efficient to solve problems arising in many real-life applications. Improving those solvers is crucial for being able to 1) solve larger problems and 2) speed up computations. A main characteristic of a sparse direct solver using low-rank compression is at what point in the algorithm the compression is performed. There are two distinct approaches: (1) all blocks are compressed before starting the factorization, which reduces the memory as much as possible, or (2) each block is compressed as late as possible, which usually leads to better speedup. Approach 1 reaches a very small memory footprint generally at the expense of a greater execution time. Approach 2 achieves a smaller execution time but requires more memory. The objective of this paper is to design a composite approach, to speedup computations while staying under a given memory limit. This should allow to solve large problems that cannot be solved with Approach 2 while reducing the execution time compared to Approach 1. We propose a *memory-aware* strategy where each block can be compressed either at the beginning or as late as possible. We first consider the problem of choosing when to compress each block, under the assumption that all information on blocks is perfectly known, i.e., memory requirement and execution time of a block when compressed or not. We show that this problem is a variant of the NP-complete Knapsack problem, and adapt an existing approximation algorithm for our problem. Unfortunately, the required information on blocks depends on numerical properties and in practice cannot be known in advance. We thus introduce models to estimate those values. Experiments on the PaStiX solver demonstrate that our new approach can achieve an excellent trade-off between memory consumption and computational cost. For instance on matrix Geo1438, Approach 2 uses three times as much memory as Approach 1 while being three times faster. Our new approach leads to an execution time only 30% larger than Approach 2 when given a memory 30% larger than the one needed by Approach 1.

Keywords: sparse direct solvers, low-rank compression, scheduling, memory constraints

1. Introduction

Many numerical applications such as computational fluid dynamics, electromagnetism, or structural mechanics use numerical models that require solving systems of the form $Ax = b$, where A is a sparse matrix of size n, meaning that the number of non-zero elements is in $\Theta(n)$. Sparse matrices appear for instance when discretizing Partial Differential Equations (PDEs) on 2D and 3D finite element or finite volume meshes: each point interacts only with its neighborhood. Solving large sparse linear systems is an expensive operation, thus enhancing this step is of interest for many real-life applications. Among the methods available to solve sparse linear systems, sparse direct solvers are widely used for their numerical robustness that allows to tackle most problems. However, both the memory footprint and the number of operations limit the use of sparse direct solvers for very large matrices. To circumvent this problem, a recent approach consists of compressing some blocks appearing during the factorization with low-rank compression techniques. The objective is then to reduce the computational cost and the memory footprint while loosing some numerical information in a controlled way. As many applications do not require to get a solution at the machine precision, this approach is now used in various contexts. Low-rank compression can thus appear as a perfect solution because it may both decrease computational cost and memory footprint. While this is true on parts of the computations (computing the updates), using low-rank compression can also increase the computational costs on other parts of the computations (applying the updates). In particular, when used to drastically reduce the memory footprint, low-rank compression usually ends up increasing the total processing time compared to versions that use temporary memory spaces.

There exist different formats to represent a matrix in a compressed form. Among them, Block Low-Rank (BLR) compression consists of splitting a matrix into regular blocks before compressing independently each block, for instance with Singular Value Decomposition (SVD) or Rank-Revealing QR (RRQR). This approach has been used for the Mumps [1, 2] and the PaStiX [3] sparse direct solvers and has been proven efficient for many real-life problems. The other type of representation relies on a recursive splitting of the matrix and has led to several formats: $\mathcal{H}$[4], $\mathcal{H}^2$[5], HSS[6], and HODLR [7] for instance. The approach commonly used in those solvers is to compress large dense blocks appearing during the factorization of a sparse matrix and not the sparse matrix itself. In this paper, we will focus on the BLR format.

If sparse direct solvers using BLR compression have demonstrated good results, allowing to solve large problems and/or to

significantly reduce the time-to-solution, there are still limitations that prevent solving large systems with a huge level of parallelism when working under some memory constraints. In the MUMPS solver, some large blocks, namely the fronts, are fully allocated (in dense format) before being compressed during the factorization. Thus, this reduces the potential memory gain as those blocks exist in their full-rank form before being compressed. In the PASTIX solver, two approaches have been developed, a first one that significantly reduces execution time without carefully managing the memory consumption and a second one that minimizes as much as possible the memory footprint but results in a larger execution time.

The objective of this paper is to propose a new memory-aware strategy for the PASTIX solver that does not minimize the memory consumption but keeps it under a given memory limit, utilizing as much as possible the available resources. The idea is to perform as many computationally efficient but memory expensive operations as possible while allowing some time overhead for some computationally expensive but memory thrifty operations, for cases for which executing all operations in a computationally efficient way would exceed the memory available on the machine.

The main contributions of this paper are the following:

- We propose a *memory-aware* strategy that allows each block to be compressed either as early as possible, or as late as possible, independently from the choice of other blocks.
- We show that the offline problem (with perfect information available) of choosing which blocks to compress at the beginning is a variant of the Knapsack problem. Furthermore, we adapt a 2-approximation algorithm for Knapsack into a 1.02 approximation algorithm for our problem (under some realistic hypotheses).
- We design models to estimate the information needed by the approximation algorithm, namely the size of the compressed blocks and their update times in the compressed and non-compressed formats.
- We provide a proof-of-concept implementation of a dynamic version of the proposed *memory-aware* strategy in the PASTIX solver. Our results on actual matrices demonstrate the large potential of the proposed approach, leading to excellent trade-offs between memory and performance.

We stress that the proof-of-concept implementation of the proposed strategy uses the sequential version of the PASTIX solver, even if the parallel version is clearly our final objective. There are two reasons for this limitation: (i) adapting the dynamic strategy to a parallel and distributed solver such as PASTIX would require non-trivial work to resolve synchronization problems, and (ii) the work presented here is needed to first assess the interest of our approach. Therefore, the present study is a necessary building block.

The rest of the paper is organized as follows. In Section 2, we present some background on sparse linear solvers and low-rank compression before detailing the problem and the objectives of the paper. In Section 3, we formalize the problem and prove that it is NP-hard before presenting a low complexity approximation algorithm. In Section 4, we present models to evaluate the cost and the memory footprint associated to each block. Combining the formalization of the problem and the predictive models, we present the results of the new strategy for the PASTIX solver in Section 5, before concluding this work in Section 6.

2. Background and Proposed Approach

Introducing low-rank compression in sparse direct solvers brings the problem of *when* to perform the compression of data blocks. In Section 2.1 we briefly present the different steps of sparse direct solvers. In Section 2.2, we describe how low-rank compression can be used to reduce the memory consumption and/or the time-to-solution. In Section 2.3, we present the idea driving this paper, that is, to reduce as much as possible time-to-solution while satisfying a memory constraint. In Section 2.4, we comment on the position of the literature with respect to this open problem.

2.1. Sparse Direct Solvers

This section intends to briefly present sparse direct solvers. For more details, see [8] for instance. In order to solve $Ax = b$, a sparse direct solver factorizes the matrix A into a product of triangular matrices, following the Gauss elimination. In the general case, when the matrix is not symmetric, one can compute $A = LU$, where L is a lower triangular matrix and U an upper triangular matrix. Then the system can be solved by performing triangular solves: we solve $Ly = b$ and then $Ux = y$.

Solving a sparse system with a sparse direct solver is usually divided into four main steps:

1. Ordering the unknowns to minimize the fill-in, i.e., null elements becoming non-zeroes during the factorization;
2. Computing the block symbolic factorization, that predicts the form of the factorized matrix before any numerical operations take place;
3. Factorizing the matrix;
4. Solving triangular systems.

The first step, ordering the unknowns, is usually performed using the nested dissection algorithm [9] through external partitioning tools such as METIS [10] or SCOTCH [11]. From this process, unknowns are grouped together into sets that will correspond to column blocks in the following block symbolic factorization.

The second step, building the block symbolic factorization, intends to represent the sparse matrix as a collection of dense blocks. A sparse matrix can be seen as a set of column blocks, each corresponding to a set of unknowns obtained from the nested dissection. It is composed of a dense diagonal block, representing the interactions between the unknowns of the column block, and several off-diagonal blocks (located either below on the same columns or on the right on the same rows as the

diagonal block), that represent the interactions with other column blocks. Figure 1 presents such a structure for a small matrix. In order to increase the level of parallelism, large diagonal blocks (and thus large column blocks) can be split into smaller blocks, increasing the overall number of column blocks. The objective of this block structure is to split the data among blocks that remain large enough to leverage efficient BLAS level 3 [12] operations.

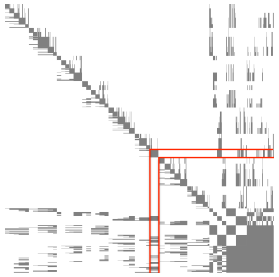


Figure 1: Symbolic factorization of a $10 \times 10 \times 10$ Laplacian. The red lines define the blocks grouped into a single column block: one diagonal block and several off-diagonal blocks.

Once this step is performed, all the blocks (either diagonal or off-diagonal) can be initialized with the original values of A, before performing the numerical factorization. The latter follows the Gauss elimination as for dense matrices, but the sparsity is carefully managed to operate only on blocks not entirely made of zeroes. The following operations take place to treat (or eliminate) each column block, one after the other:

1. **Factorize** the dense diagonal block;
2. Eliminate (**Solve**) off-diagonal blocks belonging to that column block;
3. **Update** the trailing sub-matrix (bottom right of the current column block) by performing a matrix-matrix product between each pair of off-diagonal blocks of the current column block and applying (summing) the contribution to the trailing sub-matrix. This step will be detailed in Section 2.2.

Finally, the triangular solves can be performed once the matrix is factorized.

2.2. Low-Rank Compression

Although sparse direct solvers are well-known for their numerical stability, their main limitation is their complexity, both for memory storage and number of operations. The storage corresponds to each gray block on Figure 1, while the time complexity mostly comes from the **Update** process, where a matrix-matrix product is performed between each pair of off-diagonal blocks belonging to the column block. In Table 2.2, we recall the complexity of sparse direct solvers with and without using low-rank compression ([2], Chap 4) for matrices issued from finite element meshes coming from simulations of 2D or 3D physical problems. Note that we consider here that the ranks depend on the matrix size, the low-rank complexities would be even lower with constant ranks.

Dimension	Operations		Memory	
	LR	FR	LR	FR
2D	$\Theta(n^{\frac{3}{2}})$	$\Theta(n^{\frac{3}{2}})$	$\Theta(n\ log(n))$	$\Theta(n\ log(n))$
3D	$\Theta(n^{\frac{10}{6}})$	$\Theta(n^2)$	$\Theta(n^{\frac{7}{6}} log(n))$	$\Theta(n^{\frac{4}{3}})$

Table 1: Complexities of sparse direct solvers with low-rank compression (LR) or without (FR)).

The BLR compression consists of compressing large off-diagonal blocks. Note that when compressing a block A of size $m \times n$, we obtain a low-rank form $u_A v_A^t$, where u_A is of size $m \times r_A$ and v_A of size $n \times r_A$; r_A is the rank of the matrix A and is usually much smaller than $\min(m, n)$. Diagonal blocks remain dense. Remember that large diagonal blocks have been split into smaller blocks during the block symbolic factorization so that diagonal blocks are small. In the scope of the PaStiX solver that we will use in this paper, blocks are split to obtain a size between 128 and 256. Thus, from the refined symbolic factorization, only small diagonal blocks remain. Thus, most of the blocks are off-diagonal blocks and then candidates for being compressed. Depending on *when* the low-rank compression is performed, the numerical factorization process is modified accordingly. We will now present two strategies introduced into the PaStiX [13] solver, presented in [3].

2.2.1. Minimal Memory – *Low-Rank Updates*

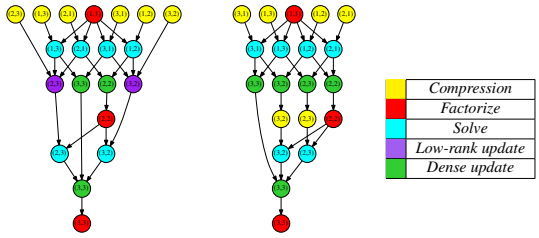
A first strategy, namely *Minimal Memory*, consists of compressing all large off-diagonal blocks before starting the numerical factorization. The main asset of this approach is that blocks are never allocated in their full-rank form, thus dramatically reducing the memory footprint of the solver. However, it raises the problem of updating low-rank blocks.

On Figure 2(b), we present a partial sparse matrix at the step of eliminating column block 1. An uncompressed matrix is represented by a full rectangle, like block $A_{(2,2)}$ and a compressed matrix by two small rectangles, like block $A_{(2,1)}$, to illustrate its storage as a product uv^t. The corresponding Direct Acyclic Graph (DAG) of tasks is presented on the left of Figure 2(a). The matrix on Figure 2(b) corresponds to the tasks shown on the third row of the DAG. In this strategy, blocks are compressed at the beginning of the factorization with the *Compression* kernel. Thus, instead of performing only full-rank updates, some low-rank matrices are updated with the *Low-rank update* kernel. For instance, block $A_{(3,2)}$ receives a contribution made of the product of blocks $A_{(3,1)}$ and $A_{(1,2)}$, corresponding to a *Low-rank update* task on the DAG on Figure 2(a). This operation is split into two parts: producing a low-rank contribution:

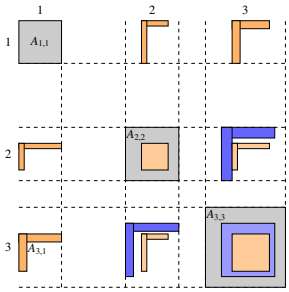
$$u_{AB} v_{AB}^t = (u_{A_{(3,2)}} v_{A_{(3,2)}}^t)(u_{A_{(3,1)}} v_{A_{(3,1)}}^t),$$

(see [3] for detail) and applying this contribution:

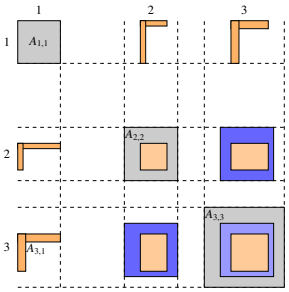
$$A_{(3,2)} - = u_{AB} v_{AB}^t.$$



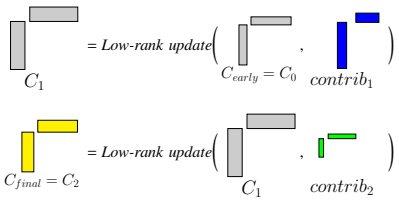
(a) DAGs: *Minimal Memory* on left, *Just-In-Time* on right.



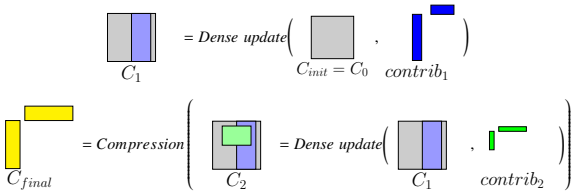
(b) Eliminating column block 1 with *Minimal Memory* strategy.



(c) Eliminating column block 1 with *Just-In-Time* strategy.



(d) Updates with *Minimal Memory* strategy.



(e) Updates with *Just-In-Time* strategy.

Figure 2: DAGs, state of the matrix when eliminating column block 1 and example of updating a block with two contributions for *Minimal Memory* and *Just-In-Time* strategies.

As depicted on Figure 2(d), updating a low-rank block with several low-rank contributions consists of utilizing dedicated kernels to maintain a low-rank structure. Producing the contribution $u_{AB}v_{AB}^t$ is faster than in the full-rank case, while applying this contribution can be expensive.

One can see that due to sparse properties, the target (that is, the block to be updated) may be much larger than some contributing blocks (for example, $contrib_2$ is much smaller than C_1 in Figure 2(d)). This is the drawback of this approach, as the complexity of updating a low-rank target depends on its size and not on the size of the smaller contribution. Note that the rank of the target block can grow with the updates, thus the size of the low-rank structure slowly increases before reaching its final size when receiving the last contribution. The rank may also decrease due to particular numerical properties (updates that cancel previous contributions), but this behavior is rarely observed in practice.

2.2.2. Just-In-Time – *Full-Rank Updates*

Another strategy introduced in the PaStiX solver consists of compressing blocks as late as possible, once they have received all their contributions and thus will not be updated anymore. This strategy is named *Just-In-Time*. Figure 2(c) presents the partial matrix obtained when eliminating the column block 1. One can see that off-diagonal blocks belonging to column block 2 are not yet compressed. The right part of Figure 2(a) presents the DAG of the corresponding operations. As the target blocks are now full-rank, the cost of applying the updates is as cheap as for the full-rank version of the solver. Thus, similarly to the *Minimal Memory* strategy, the cost is reduced when computing matrix-matrix products between low-rank matrices, while there is no extra overhead when updating the target matrix.

In the DAG of tasks, instead of compressing all large off-diagonal blocks before the beginning of the factorization, those blocks are compressed throughout the computation, as depicted with the *Compression* kernel that appears on lines 1 and 4 of the DAG. The *Low-rank update* kernel that appeared for the *Minimal Memory* strategy is not used anymore. As presented on Figure 2(e), an update consists now of forming the dense contribution and applying it directly to the target matrix. For instance, block $A_{(3,2)}$ receives a contribution made of the product of blocks $A_{(3,1)}$ and $A_{(1,2)}$. This product can be represented as a low-rank matrix $u_{AB}v_{AB}^t$, as in Section 2.2.1. The full-rank contribution is explicitly built before being applied to $A_{(3,2)}$. There is a huge gain when computing the matrix-matrix product between low-rank matrices, while there is no extra overhead when applying this product.

2.3. Towards a Mixed Strategy

The *Minimal Memory* strategy was introduced to consume as little memory as possible, while the *Just-In-Time* strategy is dedicated to reduce time-to-solution. If the matrix fits in memory, it is thus more interesting to use the *Just-In-Time* strategy, while it has been shown in [3] that the *Minimal Memory* strategy allows to solve problems that are too large to be solved with either the full-rank or the *Just-In-Time* strategy. In Table 2, we

Table 2: Factorization time and memory consumption for ten matrices solved at tolerance 10^{-8}, using *Just-In-Time* and *Minimal Memory* strategies.

Matrix	Strategy	Memory (GB)	Time(s)
atmosmodl	*Just-In-Time*	16.9	140.2
	Minimal Memory	5.75	1087.4
CurlCurl3	*Just-In-Time*	13.9	168.1
	Minimal Memory	8.93	1642.5
dielFilterV2real	*Just-In-Time*	9.07	87.4
	Minimal Memory	5.47	563.9
Flan1565	*Just-In-Time*	26.4	268.9
	Minimal Memory	15.7	920.7
Geo1438	*Just-In-Time*	43.2	591.0
	Minimal Memory	16.6	1579.3
Hook1498	*Just-In-Time*	27.2	409.6
	Minimal Memory	13.4	1838.0
PFlow742	*Just-In-Time*	9.19	51.6
	Minimal Memory	3.79	237.1
Serena	*Just-In-Time*	46.7	535.8
	Minimal Memory	14.7	1878.6
StocF1465	*Just-In-Time*	18.8	74.1
	Minimal Memory	5.44	121.2
3Dspectralwave	*Just-In-Time*	45.6	5961.0
	Minimal Memory	37.4	15236.1

highlight this behaviour on a set of ten representative matrices that will be used throughout this paper. For instance, for the `Geo1438` matrix, the memory consumption of the *Just-In-Time* strategy is roughly three times larger than the one of the *Minimal Memory* strategy, while the execution time is reduced by the same factor.

An intermediate approach which would better meet the needs of real-life users would be to use as much as possible the *Just-In-Time* strategy while moving to the *Minimal Memory* strategy if the memory consumption is too high. For instance, what could be the execution time for solving the `Geo1438` matrix (cf. Table 2) on a system with 25 GB ?

In practice, let us consider the block $A_{(3,2)}$ that appears on both Figure 2(b) and Figure 2(c). As presented on Figure 2(d) and Figure 2(e), the same contributions are applied on this block for both *Minimal Memory* and *Just-In-Time* strategies. In addition, after the block has received all its contributions, it is in a low-rank form (either it remains in that form, or it is compressed to reach it). Thus, the block **receives** and **produces** exactly the same information, no matter if the *Minimal Memory* or the *Just-In-Time* strategy is used. The inputs and the output are identical. Then it is possible, for a same factorization, to compute a subset of blocks using the *Minimal Memory* strategy and the remaining blocks with the *Just-In-Time* strategy. It is even possible to compress a block in "the middle" of the factorization, *i.e.*, applying first full-rank updates and then low-rank updates.

For instance, instead of using either the *Just-In-Time* strategy (as depicted on Figure 2(c)) or the *Minimal Memory* strategy (as depicted on Figure 2(b)), one could imagine to have $A_{(3,2)}$ compressed **early** before the beginning of the factorization and $A_{(2,3)}$ compressed in a **lazy** way as it happens with the *Just-In-Time* strategy.

The objective of this paper is to propose an intermediate strategy that chooses which blocks to execute following the *Just-In-Time* strategy and which ones to execute following the *Minimal Memory* strategy in order to respect a given memory constraint. Thus, the approach we study in this paper combines the assets of both existing strategies: allowing to solve very large problems while utilizing as many efficient operations as possible.

2.4. Related Work

Up to our knowledge, this work is the first attempt to combine both low-rank and full-rank updates in a sparse solver. Most solvers using low-rank compression are performing full-rank updates, which favor the reduction of execution time.

There are few work that propose an implementation of a low-rank solver using low-rank updates. In [14], such an approach is proposed, but the performance obtained is low, slower than a generic sparse direct solver. The work conducted for the PaStiX solver [3], in a supernodal context, manages low-rank updates between blocks of different sizes. The widely used sparse direct solver Mumps [1, 2] relies on the multifrontal factorization. In this solver, a block is not allocated before the start of the factorization but only when it is needed. Some large blocks, however, the fronts, are fully allocated in their full-rank form after they have received all their contributions and before being compressed. In order to reduce this memory cost, either the allocation could be performed panel by panel or low-rank updates should be used. Note that, in Mumps [1, 2], a variant named LUAR for Low-Rank Updates, Accumulation and Recompression is a trade-off between low-rank and full-rank updates. Instead of applying each update independently, updates are accumulated in an extra working space and recompressed all together before being applied. As we expect to use as low memory as possible in this paper, this approach is out-of-scope of our study. However, the models introduced to predict the costs associated to each block could help decide for which updates LUAR should be used.

The work in this article is of interest to the community as it provides an intermediate approach, using as much as possible efficient dense updates and moving to low-rank updates when necessary to stay under a given memory limit.

On a more general view, this study focuses on deriving space-time trade-offs using low-rank compression and scheduling. General space-time trade-offs have been pioneered through the use of pebble game models (see [15, Chapter 10]). Some of us have also studied how to efficiently schedule task trees [16] and task graphs [17] when the available memory is limited.

3. Modelization of the Problem

We now formalize the problem and describe how to decide which tasks should be computed with the *Minimal Memory* strategy and which ones should be performed with the *Just-In-Time* strategy. For the remainder of the paper, blocks following the *Minimal Memory* strategy will be said to be in *early* mode,

while blocks following the *Just-In-Time* strategy will be said to be in *lazy* mode.

In Section 3.1, we present the formalization of the problem. In Section 3.2, we demonstrate that this problem is equivalent to *Knapsack* and we explicit the approximation algorithm that will be used in the remainder of the paper.

3.1. Formalization: MakespanWithBoundedMemory

The objective here is to exhibit, among the off-diagonal blocks that are large enough to be compressed, two set of blocks, a first one to be performed in *early* mode and a second one to be performed in *lazy* mode, such that the total memory consumption of those blocks does not exceed a given bound and the time-to-solution is reduced as much as possible.

We consider a set $\mathcal{J}$ of n independent tasks, $J_1, \ldots, J_n$, where a task J_i represents an off-diagonal block. Each task has an execution time, corresponding to the sum of all elementary updates (as presented on Figure 2(d) and Figure 2(e)) that are applied to the block and a memory consumption, which is the memory storage of the block. Both this time and this memory depend on the mode chosen for this block:

- If executed in the *lazy* mode (as in the *Just-In-Time* strategy), its execution takes a time t_i and uses a memory (storage) S_i;
- Otherwise, in the *early* mode (as in the *Minimal Memory* strategy), its execution takes a time T_i and uses a memory (storage) s_i.

We make the assumption that executing a task in *lazy* mode takes less time and more memory than executing it in *early* mode. Therefore for any $i \in [1:n], S_i > s_i$ and $T_i > t_i$. Otherwise, if $S_i \leq s_i$ it is always better to execute the task in *lazy* mode and if $T_i \leq t_i$ it is always better to use the *early* mode.

mode / strategy	*lazy / Just-In-Time*	*early / Minimal Memory*
memory size	S_i	s_i
processing time	t_i	T_i

Table 3: Summary of key parameters.

All tasks are executed one after the other on a single processor; hence, the total processing time is the sum of all task execution times. In this model, we assume that all tasks are simultaneously present in memory, from the beginning of the execution of the first task until the completion of the very last task. Let $\mathcal{M}$ be the memory threshold, such that $\mathcal{M} \geq \sum s_i$, meaning that all tasks fit in memory if executed in the *early* mode. Note that we assume here that the ranks of blocks are kept invariant during the factorization, while it is not exactly the case as presented in Section 2.2. In Section 5.5, we will explain why it does not hurt our modelization.

Definition 1. *The MakespanWithBoundedMemory problem consists of choosing for each task if it will be executed in* early *or in* lazy *mode, while respecting the memory constraint and minimizing the total processing time.*

3.2. Equivalence to Knapsack and Heuristic to Solve the Problem

The *MakespanWithBoundedMemory* problem is actually exactly the *Knapsack* [18] problem.

Definition 2. *Knapsack problem: Let $\mathcal{I}$ be a set of n items. Each item has a value v_i and a weight w_i. The objective is to fit some of the items in a bag of weight capacity $\mathcal{W}$, while maximizing the value of the objects inside the bag.*

Theorem 1. *Knapsack and MakespanWithBoundedMemory are equivalent: any algorithm solving one problem can be used to solve the other.*

The intuition of this equivalence is the following. Consider that all tasks of *MakespanWithBoundedMemory* are initially in *early* mode, and we want to choose which ones to move to *lazy* mode. Chosen tasks will correspond to chosen items in *Knapsack*. We first want the memory consumption to stay below the threshold $\mathcal{M}$. For each task moved to *lazy* mode, we pay an extra memory of $S_i - s_i$ compared to the original storage cost ($\sum s_i$). This corresponds to the *Knapsack* constraint with weights $w_i = S_i - s_i$ and capacity $\mathcal{W} = \mathcal{M} - \sum s_i$. The objective is to minimize the execution time, and for each task moved to *lazy* mode, we earn a time $T_i - t_i$ compared to the initial time ($\sum T_i$). This corresponds to maximizing the sum of the values $v_i = T_i - t_i$ of the chosen items.

Proof 1. *Let us express MakespanWithBoundedMemory as an Integer Linear Program (ILP):*

We associate a variable $x_i \in \{0, 1\}$ to each $J_i \in [1:n]$. Let

- $x_i = 1$ *if the task J_i is executed in* lazy *mode,*
- $x_i = 0$ *if the task J_i is executed in* early *mode.*

Therefore, the ILP formulation is:

$$\textit{minimize} \quad \sum_{i=1}^{n}(x_i t_i) + \sum_{i=1}^{n}((1-x_i)T_i) \tag{1}$$

$$\textit{subject to} \quad \sum_{i=1}^{n}(x_i S_i) + \sum_{i=1}^{n}((1-x_i)s_i) \leq \mathcal{M} \tag{2}$$

$$\textit{and} \quad \forall i \in \{1,n\},\ x_i \in \{0,1\} \tag{3}$$

Moreover we have the following relations:

$$(1) \iff \textit{maximize} \sum_{i=1}^{n} x_i(T_i - t_i) - \sum_{i=1}^{n} T_i$$

$$\iff \textit{maximize} \sum_{i=1}^{n} x_i(T_i - t_i)$$

$$(2) \iff \sum_{i=1}^{n} x_i(S_i - s_i) \leq \mathcal{M} - \sum_{i=1}^{n} s_i$$

Thanks to these two equivalences, we just showed that the ILP is exactly a linear formulation of the Knapsack problem:

$$\begin{aligned} &\textit{maximize} && \sum_{i=1}^{n} x_i v_i \\ &\textit{subject to} && \sum_{i=1}^{n} x_i w_i \leq \mathcal{W} \\ &\textit{and} && \forall i \in \{1, n\}, x_i \in \{0, 1\} \end{aligned}$$

with the following transformation:

- $\forall i \in [1 : n], v_i = T_i - t_i$
- $\forall i \in [1 : n], w_i = S_i - s_i$
- $\mathcal{W} = \mathcal{M} - \sum_{i=1}^{n} s_i$

Therefore, Knapsack and MakespanWithBoundedMemory are equivalent.

The *Knapsack* problem is known to be NP-complete, however efficient approximation algorithms have been derived. The equivalence of our two problems does not a priori hold for the approximation ratios, especially since *MakespanWithBoundedMemory* is a minimization problem where *Knapsack* is a maximization problem. We prove in the following theorem that the simple greedy algorithm that selects tasks by non-increasing ratio of $\frac{T_i - t_i}{S_i - s_i}$ (Algorithm 1) is an approximation algorithm for our *MakespanWithBoundedMemory* problem. Note that this greedy algorithm does not compute an approximation for the *Knapsack* problem, but comparing this solution to the first non-selected item provides a 2-approximation for Knapsack [19].

Algorithm 1: Greedy approximation algorithm for *MakespanWithBoundedMemory*

Sort tasks by non-increasing $\frac{T_i - t_i}{S_i - s_i}$ values
Greedily add tasks to a set S while the sum of their weights $w_i = S_i - s_i$ does not exceed $\mathcal{M} - \sum_{i=1}^{n} s_i$

Theorem 2. *Algorithm 1 is a $(1 + 2\varepsilon\rho)$-approximation for the MakespanWithBoundedMemory problem, where ε is the ratio of the size of the largest (uncompressed) block to the size of the remaining memory when all blocks are allocated in their compressed form ($\varepsilon = \max_i S_i/(\mathcal{M} - \sum_j s_j)$) and ρ is the ratio between the computation time when all blocks are in the compressed form to the computation time when no blocks are compressed ($\rho = (\sum_i T_i)/(\sum_i t_i)$).*

Practical value of the approximation ratio. Note that ρ is of the same order as the ratio between the execution times of the *Minimal Memory* and *Just-In-Time* strategies, and thus bounded by $\rho \leq 10$ in practical cases [3]. As large blocks are split to obtain blocks for which both width and height are smaller than or equal to 256, we have $\max_i S_i \approx 0.5$ MB. If we made the assumption that *Minimal Memory* strategy can be enhanced, it means that some unused memory remains: we assume for instance that at least 10% of the memory is not used by *Minimal Memory* ($\mathcal{M} \geq 1.1 \times \sum_j s_j$) and that the overall memory required to store blocks is at least 5GB (this is true for all practical cases seen below). Then we have

$$\mathcal{M} - \sum_j s_j \geq 0.1 \sum_j s_j \geq 0.5GB$$

and $\varepsilon \leq 0.001$. In practice, these conservative assumptions make Algorithm 1 a 1.02-approximation algorithm.

Given the values of ε and ρ observed in practice, there is no use for the Fully Polynomial-Time Approximation Scheme for *Knapsack* to solve our problem, as the previous theorem proves that the greedy algorithm already provides a solution with very good quality, with only $O(n\ log(n))$ complexity. A lower bound on the achievable makespan can be computed using the classical greedy algorithm for the fractional *Knapsack* problem: it simply consists of using Algorithm 1 to construct the set S, and adding a fraction of first task not included in the solution to completely fill the memory. This allows us to verify that the solution of Algorithm 1 is very close to optimal in all the experiments.

Proof 2. *We start by recalling some generalities on the Knapsack problem, as presented above (adapted from [20]). The greedy algorithm for the Knapsack problem sorts items by non-increasing v_i/w_i and selects the maximum number of items in this order. Let V_{OPT} be the optimal solution of the Knapsack problem.*

Observation 1. Let k be the first item not selected by the greedy algorithm, then $v_1 + \cdots + v_k > V_{OPT}$. We consider the fractional version of the Knapsack problem, and denote its total value by $V_{OPT}^{frac} \geq V_{OPT}$. The adaptation of the greedy algorithm to the fractional problem computes an optimal solution, which includes (full) items $1, \ldots k-1$ as well as some fraction $\alpha < 1$ of item k. Thus:

$$V_{OPT}^{frac} = v_1 + \cdots + v_{k-1} + \alpha v_k \geq V_{OPT}$$

Thereby, adding $(1-\alpha)v_k > 0$ to this sum leads to a value larger than V_{OPT}, proving the result.

Observation 2. If for each item i we have $w_i \leq \varepsilon\mathcal{W}$, then Algorithm 1 is an $(1 - 2\varepsilon)$ approximation. To prove it, we again consider the first item k not selected by the greedy algorithm. Thanks to the ordering of the items, $v_i \geq w_i v_k / w_k$ for the selected items $i = 1 \ldots k-1$. We can deduce:

$$v_1 + \cdots + v_k \geq (w_1 + \cdots + w_k) v_k / w_k$$

Since item k has not been included in the greedy solution, we know that $w_1 + \cdots w_k > \mathcal{W}$. Besides, thanks to the assumption of this observation, we have $w_k \leq \varepsilon\mathcal{W}$. Thus, we have:

$$\begin{aligned} v_1 + \cdots + v_k > v_k/\varepsilon \quad &\Leftrightarrow \\ \varepsilon(v_1 + \cdots + v_k) > v_k \quad &\Leftrightarrow \\ \varepsilon(v_1 + \cdots + v_{k-1}) > (1-\varepsilon)v_k \quad &\Leftrightarrow \\ v_k < \tfrac{\varepsilon}{1-\varepsilon}(v_1 + \cdots + v_{k-1}) \quad &\Rightarrow \\ v_k < \tfrac{\varepsilon}{1-\varepsilon} V_{OPT} \end{aligned}$$

Hence, the solution of the greedy algorithm reaches a total value

$$\begin{aligned} v_1 + \cdots + v_{k-1} = (v_1 + \cdots + v_k) - v_k &> (1 - \tfrac{\varepsilon}{1-\varepsilon})V_{OPT} \\ &= (\tfrac{1-2\varepsilon}{1-\varepsilon})V_{OPT} \\ &> (1 - 2\varepsilon)V_{OPT} \end{aligned}$$

We now prove the approximation ratio of Algorithm 1 for the MakespanWithBoundedMemory problem. We denote by $T_{max} = \sum_i T_i$ *the makespan achieved when all blocks are compressed, and similarly* $T_{min} = \sum_i t_i$ *the makespan when no blocks are compressed. Let* T_A *be the makespan achieved by the greedy algorithm and* T_{OPT} *the optimal makespan. We have*

$$T_A = T_{max} - V_A \text{ and } T_{OPT} = T_{max} - V_{OPT}$$

where V_A *(resp.* V_{OPT}*) is the value obtained by the greedy algorithm (resp. the optimal algorithm) on the Knapsack instance detailed in the previous proof. We can thus rewrite* T_A *as:*

$$T_A = T_{OPT} + V_{OPT} - V_A$$

We notice that in this instance, for each task i,

$$w_i = S_i - s_i \leq S_i \leq \varepsilon \left(\mathcal{M} - \sum_j s_j \right) = \varepsilon \mathcal{W}$$

by assumption on the S_i*'s. Hence, thanks to Observation 2, we know that the greedy algorithm is a* $(1 - 2\varepsilon)$*-approximation algorithm on this instance, and thus* $V_A \geq (1 - 2\varepsilon)V_{OPT}$*, which gives:*

$$T_A \leq T_{OPT} + 2\varepsilon V_{OPT}$$

We also have:

$$V_{OPT} \leq \sum_i T_i - t_i \leq \sum_i T_i = T_{max} = \rho T_{min}$$

Together with $T_{OPT} \geq T_{min}$*, this gives* $T_A \leq (1 + 2\epsilon\rho)T_{OPT}$*.*

4. Predictive Models to Estimate Time and Memory

Now that we have formalized the problem and presented an approximation algorithm to compute a good solution, some information is still missing before using the proposed approach. Indeed, for each block, we need to know a priori its time and memory usage under both *early* and *lazy* modes. In Section 4.1, we list the values required to use the *memory-aware* strategy and what is available before the factorization. In Section 4.2 we present the rank model and in Section 4.3 the models used to predict the time for both strategies. In Section 4.4, we discuss the practical details of building these models, before presenting the results in Section 4.5.

4.1. Values Required for the Memory-Aware Strategy

In order to establish the *memory-aware* strategy, Algorithm 1 sorts values accordingly to $\frac{T_i - t_i}{S_i - s_i}$ for each block C_i of size $m_i \times n_i$. S_i corresponds to the memory required by the *lazy* mode, it is equal to the full-rank memory: $S_i = m_i \times n_i$. s_i is the memory for the *early* mode and depends on the rank of the block, denoted by r_i. Remember that we suppose that the rank of a block is constant throughout the factorization and thus equal to its final value. We will discuss in Section 5.5 why this assumption is sufficient for the design of our memory-aware strategy. We have $s_i = (m_i + n_i)r_i$. t_i is the total update time of C_i in *lazy* mode, *i.e.*, the sum of all elementary updates applied to the block C_i in *lazy* mode (cf. Figure 2(e)), while T_i is the same value for the *early* mode (cf. Figure 2(d)).

Unfortunately, we do not know the rank of blocks before the factorization. Indeed, it depends on numerical properties of the matrix and cannot be deduced only from static properties. As the cost of operations depends on the rank, we do not have access to t_i and T_i either. Thus, we have introduced models to estimate the values of s_i, t_i, and T_i.

4.2. Rank Model

The first model concerns the rank, which is required to estimate both the memory storage s_i for the *early* mode and the processing times in both modes. To build this model, we rely on a linear regression. First, the simplicity of this approach makes the model easily usable in another code. Then, as the time models presented in Section 4.3 are built as linear combinations of the theoretical complexity of each operation, using a linear model for the rank simplifies the global approach.

Again remember that our objective is to estimate the final rank of a block C_i and not its evolution throughout the factorization. We know that this value depends on the size of the matrix but also on numerical and geometric properties. Here, we suppose that the final rank of a block depends linearly on:

1. The initial rank of the block C_i:
2. Its height m_i;
3. Its width n_i;
4. The area of the block $m_i n_i$;
5. The number of updates the block receives.

Except 1), all those data can be easily extracted from the block-symbolic factorization. There exist methods to roughly estimate the rank of a block without fully compressing it [21], but those methods are dedicated to large matrices. In this work, as the blocks are small and mostly made of zeroes before starting the factorization, methods like Rank-Revealing QR will stop the computations quickly without performing all operations as it would happen for dense matrices. Thus, obtaining the initial rank of each block is cheap with respect to the cost of the factorization. In practice, we will use Rank-Revealing QR to obtain this information.

In order to build the model, the idea consists of running the factorization for a given training matrix to obtain the actual ranks after the factorization. Then, from the five parameters described above together with the actual rank, a linear regression can be performed. It provides a model where the rank can be predicted as a linear combination of the five parameters given above. The key point is that, as those parameters can be easily obtained before starting the factorization, one can access the predicted ranks to estimate the memory footprint of a block for both the *early* and *lazy* modes.

4.3. Time Model

In order to predict t_i and T_i, the approach consists of predicting the execution time of each elementary update, *i.e.*, *Low-rank update* and *Dense update* kernels in the DAGs on Figure 2(a). As already presented on Figure 2(d) and Figure 2(e), an update is an operation between three blocks A, B, and C that performs the operation $C = C - AB^t$.

For the full-rank case, this operation is simply a matrix-matrix product. However, in the low-rank case, both A and B can be low-rank matrices and C is either full-rank for the *lazy* mode or low-rank for the *early* mode. Thus, this update is broken down into smaller operations as presented in [3].

Depending on the strategy used, the update process can be seen as p operations $op_1, \ldots, op_p$. Each elemental operation op_i has a theoretical complexity which depends on the characteristics of the input matrices A, B, and C [3]. Thus, we know that T_i and t_i are a linear combination of $op_1, \ldots, op_p$ and can be obtained using a linear model. This type of approach was for instance used to obtain a better scheduling in the QR-MUMPS solver [22].

The issue is that most operations here depend on the ranks of matrices A, B, or C. A first idea would be to directly use the ranks obtained by the model presented in Section 4.2. However, one could fear that injecting the result of one model in another one would lead to errors from both models adding up. Instead, we replace each occurrence of a rank (r_A, r_B, or r_C) by a linear combination of the five parameters presented in Section 4.2. This increases the number of parameters for the linear regression but avoids combining errors.

For instance, let us consider that the operation op_i depends linearly on the product $r_A r_B$. Then the linear regression will involve 25 parameters, as five parameters are required for both r_A and r_B.

4.4. Practical Details

In practice, we do not use a single model for the *early* mode. Indeed, when performing the product AB^t before updating C, there is some internal recompression process and the internal rank depends on properties of matrices A and B.

For a sparse supernodal solver like PaStiX, there are many off-diagonal blocks of different sizes. As presented in Section 2.1, large column blocks are split to exhibit more parallelism. In practice, the large column blocks are split to ensure a width between 128 and 256. From the block-symbolic factorization obtained afterwards, off-diagonal blocks with a width larger than or equal to 128 and a height larger than or equal to 20 are marked as compressible. It means that there are a lot of small blocks that will never be compressed.

Depending on the properties of A and B, the efficiency of computing the contribution AB^t may vary. Thus, we split the original dataset into three subsets, following which blocks are marked as compressible. A first one contains updates where both A and B are compressible, a second one where exactly one matrix among A and B is compressible, and a last one where both A and B are not compressible. Then, three regressions are performed independently, one on each set to estimate the cost of elementary updates, before merging the results block by block to obtain the value of T_i.

This approach has not been proven efficient for the *lazy* mode, for which we keep a single regression.

4.5. Results of the Models

In this section we present some evaluation of our models. These models predict the rank of blocks and their execution times both under the *early* and *lazy* modes. More than the predicted execution times, what matters to our approach is the ordering of blocks according to the $\frac{T_i - t_i}{S_i - s_i}$ ratio. We are therefore going to compare this ordering according to the actual values and to the predicted ones.

However, before all, we should notice that there are blocks for which one mode is the best whatever the context. Let us consider a block C_i of size $m_i \times n_i$. If C_i uses more memory when compressed than when uncompressed ($s_i \geq S_i$), there is no memory advantage to compress it, we always run it under the *lazy* mode, and we label it Lazy. Otherwise, if C_i executes faster under the *early* mode than under the *lazy* mode ($T_i \leq t_i$), then C_i should always be executed under the *early* mode and we label it Early. In all other cases, the mode under which to process C_i is *To Be Determined* by the approximation algorithm and C_i is marked TBD.

We start by assessing the quality of our partitioning of blocks into Early, Lazy, and TBD in Table 4 using matrix `Geo1438` which includes 56727 compressible (*i.e.,* large enough) blocks. The training was performed using matrix `Serena` and the low-rank tolerance used for both matrices is 10^{-8}.

This table leads to mixed conclusions. One the one hand, the number of actual Early blocks predicted as Lazy, and the number of actual Lazy blocks predicted as Early is negligible (3 in total, corresponding to 0.00% of all blocks). On the other hand, among the 9.28% of all blocks which are actually Lazy blocks only 45.95% are predicted as such (which corresponds to 4.26% of all blocks). Also, among the 36.33% of all blocks which are actually Early blocks, only 29.61% are predicted as such (corresponding to 10.76% of all blocks). Moreover, out of the 54.39% of all blocks which are actually TBD blocks 8.58% are predicted to be either Early or Lazy (corresponding to 4.67% of all blocks). Overall, the results presented by Table 4 are therefore far from being random results, but they are also not perfect. As the models do not correctly predict all Early or Lazy blocks contrarily to a perfect oracle, the approximation algorithm may have to process more blocks than required to decide which blocks to process in *early* and *lazy* mode.

In Figure 3 we compare the ordering of all blocks when using either the actual characteristics of blocks or the predicted ones. For both sets of characteristics, Early blocks are ordered first, then the TBD blocks, and finally the Lazy blocks. The TBD blocks are obviously ordered according to the $\frac{T_i - t_i}{S_i - s_i}$ ratio. The other blocks are (arbitrarily) ordered using the block identifiers. This explains why the set of correctly predicted Early blocks forms a (light blue) line on the graph, and so do the correctly predicted Lazy blocks (dark green line). In this figure

Predicted \ Actual	EARLY	TBD	LAZY	Total predicted
EARLY	10.76%	3.54%	0.00%	14.30%
TBD	25.57%	49.72%	5.01%	80.30%
LAZY	0.00%	1.13%	4.26%	5.39%
Actual total	36.33%	54.39%	9.28%	100%

Table 4: Statistics on the classification of blocks according to their actual and predicted characteristics.

blocks are colored according to both their actual (suffix "Act") and predicted (suffix "Pred") types.

Once again, the conclusions are decent but not perfect. On the one hand, this figure does not look at all like a random figure (where sub-rectangles would be uniformly filled). For instance, the mislabelled actual EARLY blocks (dark blue dots) appear rather "early" in the overall order and the mislabelled actual LAZY blocks (pink dots) appear mostly very late in the order. Also, one can guess a trend looking like a diagonal for the actual TBD blocks that are correctly predicted (dark orange dots). On the other hand, one could argue that the clouds of points show huge variations in the ordering of some blocks. One might wonder, however, whether changing the position of a block in the ordering by 1000 (or 5000) positions in a list of more than 56000 blocks really matters.

We will see below that this imprecise ordering and classification of the blocks is sufficient to reach very good trade-offs between memory and time-to-solution, and to closely approach the performance of the optimal block ordering using a perfect oracle.

As a side note, we were able to check throughout our experiments that, in practice, the quality of the solution produced by Algorithm 1 is not distinguishable from the quality of the optimal solution.

5. Experimental Evaluation

We now present the results of our *memory-aware* strategy. In Section 5.1, we present the machine as well as the sparse matrices used for the experiments. In Section 5.2, we detail the implementation of the memory-aware strategy, before discussing the training step in Section 5.3. In Section 5.4, we demonstrate the potential of the method. In Section 5.5, we comment on the implementation of this approach into the PASTIX solver before detailing results on factorization time in Section 5.6.

5.1. Experimental Context

Experiments were performed on the `crunch` cluster from our LIP laboratory, [1], where a node is equipped with four INTEL XEON E5-4620 8-cores running at 2.20 GHz and 378 GB of memory. On this platform, the INTEL MKL 2018 library is used for sequential BLAS and LAPACK kernels. The PASTIX version used for our experiments is based on the public git repository[2] version at the tag `6.1.0`. We use Rank-Revealing QR to perform low-rank compression, because Singular Value Decomposition would be too expensive. We used the RRQR kernel from the BLR-MUMPS solver [1] which stops the factorization when the precision is reached. Large column blocks are split in order to obtain column blocks of width between 128 and 256. Blocks with a width larger or equal to 128 and a height larger or equal to 20 are marked as compressible. In all those experiments, a 10^{-8} tolerance has been used.

In order to validate the results for representative real-life problems, we have used ten 3D matrices extracted from the SuiteSparse Matrix Collection [23]:

- `Atmosmodl`: atmospheric model (1 489 752 non-zeroes)
- `CurlCurl3`: operator of 2nd order Maxwell's equations (1 219 574 non-zeroes)
- `dieFilterV2real`: high-order vector finite element method (1 157 456 non-zeroes)
- `Flan1565`: 3D mechanical problem discretizing a steel flange (1 564 794 non-zeroes)
- `Geo1438`: geomechanical model of Earth (1 437 960 non-zeroes)
- `Hook1498`: model of a steel hook (1 498 023 non-zeroes)
- `Pflow742`: 3D pressure-temperature evolution in porous media (742 793 non-zeroes)
- `Serena`: gas reservoir simulation (1 391 349 non-zeroes)
- `StocF1465`: flow in porous medium with stochastic permeabilies (1 465 137 non-zeroes)
- `3Dspectralwave`: 3D spectral-element elastic wave modelling (680 943 non-zeroes)

Note that these matrices come from different application fields and have thus different mathematical properties. Therefore, training the model on a matrix and using it for another matrix will test not only the efficiency of the model, but also the robustness of our overall approach. Our objective here is to predict memory consumption and computational cost using the models presented in Section 4.2 and Section 4.3, and to use these values to select which blocks to execute in *lazy* mode in order to use as much memory as possible without overcoming a given memory limit.

5.2. Implementation of the Memory-Aware Strategy

In this section, we present how we solve the *MakespanWithBoundedMemory* problem presented in Section 3 in practice. The choice of the mode for blocks (*early* or *lazy*) is done in a dynamic way before their allocation in the memory. We describe in detail the approach below.

[1] `http://www.ens-lyon.fr/LIP/`

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

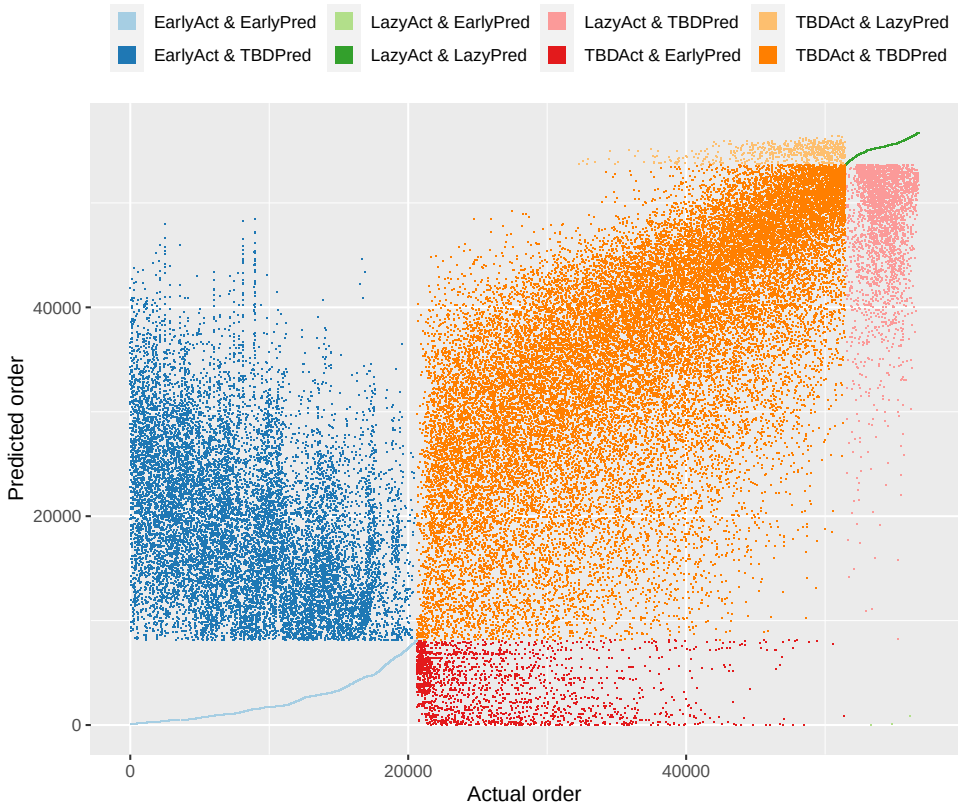


Figure 3: Comparison of the actual and predicted orders of blocks, under Algorithm 1.

1. For the training matrix, we run the real factorization using *Just-In-Time* and *Minimal Memory* strategies for all blocks. Then we extract information for each elementary update, using results of the *Minimal Memory* (respectively *Just-In-Time*) for the *early* (resp. *lazy*) mode. We perform three runs and take the median value to remove data noise before training the models. Then, we compute the rank model and both time models;

2. We apply the obtained models for the test matrix to compute the predicted values of s_i, t_i and T_i;

3. We select blocks that should always be treated in *lazy* mode ($s_i \geq S_i$) as well as blocks that should always be treated in *early* mode ($t_i \geq T_i$);

4. We sort the remaining blocks (see below in Section 5.4);

5. We consider all remaining blocks in *lazy* mode (uncompressed), which would all require S_i if allocated in memory. Then, some blocks are compressed (and thus moved to the *early* mode) following the previous order until the memory limit is achieved. The memory consumption of the current solution is computed as the sum of S_i for blocks still in the *lazy* mode plus the sum of the s_i for blocks moved to *early* mode. Note that we use the real size s_i and not the predicted one: it is known as soon as a block is compressed (and it is more accurate than the predicted one). When the memory of the current solution falls below the memory limit, blocks still in *lazy* mode may safely be loaded.

6. If the memory increases during the factorization and would exceed the memory limit, we compress the next block in the previous order (we switch this block to *early* mode). This is required for actual runs of the solver as we cannot know the exact evolution of the memory consumption of low-rank blocks before the actual factorization.

5.3. Discussion about the Training Stage

For now, we have presented an approach where a single matrix is used to perform the training. This training consists of computing a rank model as well as a time model for both *early* and *lazy* modes. It can be used for several testing matrices. In the context of an academic or an industrial utilization, the approach could consist of training the models with a matrix that is representative of the underlying application. This approach may not work well when used for several matrices issued from different applications. Indeed, if some properties (dimensions of blocks, number of updates) do not depend on the matrix properties, the rank depends on the underlying operator. Thus, the rank model learned when training with a matrix from a given domain is unlikely to be useful with a matrix from another domain.

The approach that we have used is to consider as a training set all the matrices included in our study. In practice, we ag-

gregate the results of all ten matrices and the training considers this aggregation of results as if it was a single matrix.

We acknowledge that this approach does not scale, as for instance one could not use the full SuiteSparse Matrix Collection because the training stage would be too long and would require too much memory. A possible workaround would be to randomly select a set of blocks from a large number of matrices issued from various applications. This approach would reduce the cost of training and would allow to integrate in the model the properties of each application. One could still expect that training with data issued from a specific application would be the best approach if testing matrices belong to the same application.

5.4. Potential Gain Estimation through Simulation

In order to validate the models presented in Section 4, we have implemented the previous approach using different ordering strategies for Step 4. From now on, we denote by T_i, t_i, and s_i the actual timings and memory consumption and T_i^*, t_i^*, and s_i^* the predicted ones. The four orderings are:

- Decreasing **theoretical ratio** $\frac{T_i - t_i}{S_i - s_i}$ (this is to get a bound on the best result we can obtain with our models).
- Decreasing **predicted ratio** $\frac{T_i^* - t_i^*}{S_i - s_i^*}$ (original strategy).
- Decreasing number of updates received by a block: we expect blocks receiving many updates to be more efficient in *lazy* mode, to avoid numerous expensive low-rank updates (denoted by **count** in the following figures). This strategy does not rely on predicted values, and is thus simpler to implement.
- **Random** order, for baseline comparison.

On Figure 4, we present the results of the four heuristics for the `Geo1438` matrix. The training of the models was realized with the `Serena` matrix. Both matrices were studied with a 10^{-8} tolerance. From the minimum memory consumption achievable (corresponding to the *Minimal Memory* strategy) to the maximum memory consumption possible (corresponding to the *Just-In-Time* strategy), we utilize the results of the previous algorithm to decide which mode to use for each block. To compute the simulated time, we sum the update times; the cost of common operations (remaining kernels of the considered blocks as well as all operations on small full-rank blocks such as the factorization of diagonal blocks) is omitted for better readability. The update times are extracted from the training data set, as explained above. Thus, each curve corresponds, for each heuristic, to all possible combinations memory/time.

On top-left part of the figure, we have the largest execution time and the lowest memory consumption: it corresponds to performing all blocks under the *early* mode, which is exactly the *Minimal Memory* strategy. On bottom-right, we have the opposite case, using *lazy* mode for all blocks, which is exactly the *Just-In-Time* strategy, with the largest memory consumption but an execution time much lower than the one of the *Minimal Memory* strategy. Note that for those two limit cases, the results are exactly the same for each of the four heuristics, as all blocks are executed in the same mode independently of the heuristic used.

Four points appear on the figure, two for the "theoretical ratio" (in blue) and two for "predicted ratio" (in orange). The points on the left represent the lowest execution time achievable under the condition of using as low memory as possible: all blocks are executed in *early* mode except those which do not bring memory savings ($s_i \geq S_i$ or $s_i^* \geq S_i^*$) that are processed in *lazy* mode. It is similar to the *Minimal Memory* strategy, except that blocks that do not bring any memory saving are performed in *lazy* mode. It gives the best time achievable under the condition that as low memory as possible is used. On the contrary, the two right points represent the lowest memory consumption under the condition of being as fast as possible: all blocks are executed in *lazy* mode except those which are faster in *early* mode ($t_i \geq T_i$ or $t_i^* \geq T_i^*$). It is similar to the *Just-In-Time* strategy, except that blocks that do not favor execution time gain are executed in *early* mode to reduce memory. It gives the minimal memory consumption achievable under the condition that execution time is minimized.

Between those two points, one can observe all possible combinations for the *memory-aware* strategy. For the "theoretical ratio", where perfect information is used, one can see that very nice trade-offs between execution time and memory consumption can be achieved. When considering a naive sorting such as "random" or even "count" (that sorts blocks according to the number of updates), the trade-offs are much less interesting. For instance, with 15 GB of memory, the execution time using the "count" heuristic is four times larger than the one using the "theoretical ratio" heuristic. When we use the target heuristic (*i.e.*,"predicted ratio"), using our predictive models to estimate the memory cost and the execution time of each block for both *early* and *lazy* modes, the resulting execution time is slightly larger than with the perfect strategy but much smaller than the one obtained with "random" or even "count". Using the predictive model, by increasing the memory consumption by only 20% (compared to the minimal amount of memory), execution time can be divided by 3.

On Figure 5, we present the results achieved by both "theoretical ratio" and "predicted ratio" using a 10^{-8} tolerance on a set of ten testing matrices and with four different training matrices. In Figure 5(a) and Figure 5(b), we use two training matrices that are issued from the `Janna` collection widely used in sparse direct solvers, and which give reasonable result. Note that Figure 4 corresponds to the plot in the fifth line and first column in Figure 5(a). In Figure 5(c), we use the `CurlCurl3` matrix for the `Bodendiek` as training, that we identified as a worst case for training results. Finally, in Figure 5(d), we use all our set of ten matrices for the training.

The first observation in Figure 5(a) is that the results using the predictive models are often close to the optimal solution. This proves that our approach is very robust to the choice of the training matrix. Recall that the main objective of our approach is being able to use predictive models obtained after training with data issued from a particular application, while the testing matrix can be issued from another field. One can observe that

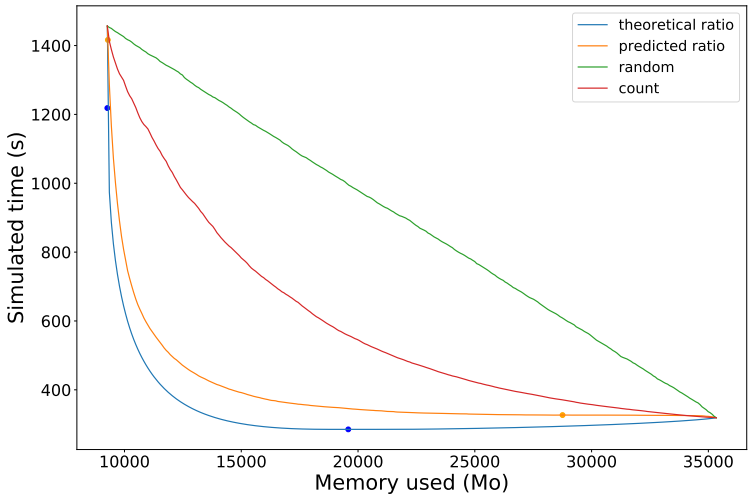


Figure 4: Simulated time of update tasks depending on the memory constraint for Geo1438 solved with a 10^{-8} tolerance for four different heuristics. The training matrix for the predictive strategy is Serena using a 10^{-8} tolerance. The top-left point represents the *Minimal Memory* strategy while the bottom-right point represents the *Just-In-Time* strategy.

the results are not very good on some matrices, for instance the `CurlCurl3`, the `PFlow742` and `3Dspectralwave` matrices.

In Figure 5(b), one can observe that surprisingly, `PFlow742` is a very good training matrix for `Serena` while the contrary is not true (even though `Pflow742` is twice smaller than `Serena`). This encourages us to use a large training set made of a large number of blocks from different matrices in order to compute better models. One can also observe that the results of testing the training matrix itself are very nice, as expected.

In Figure 5(c), we perform the training with a matrix that is known to have different low-rank properties. Here, the results are much worst (except when testing on the training matrix itself): one should be careful when choosing the training matrix, or use a large training set as proposed above.

In Figure 5, we perform the training with the set of ten matrices, including `CurlCurl3` and `3Dspectralwave` that do not behave as expected in other curves. One can observe that the training works well for almost all matrices, except for the matrix `3Dspectralwave`, although it was included in the testing set. For the `CurlCurl3` matrix, the results are slightly better than when the training uses `PFlow742`.

Overall, the curves in Figure 5 demonstrate that our new strategy allows to obtain nice trade-offs between time and memory, in most cases close to the optimal solutions. For more difficult matrices, which properties differ, the results are less impressive even if they still allow to reach interesting trade-offs.

5.5. *Integration of the Memory-Aware Strategy into the* PaStiX *Solver*

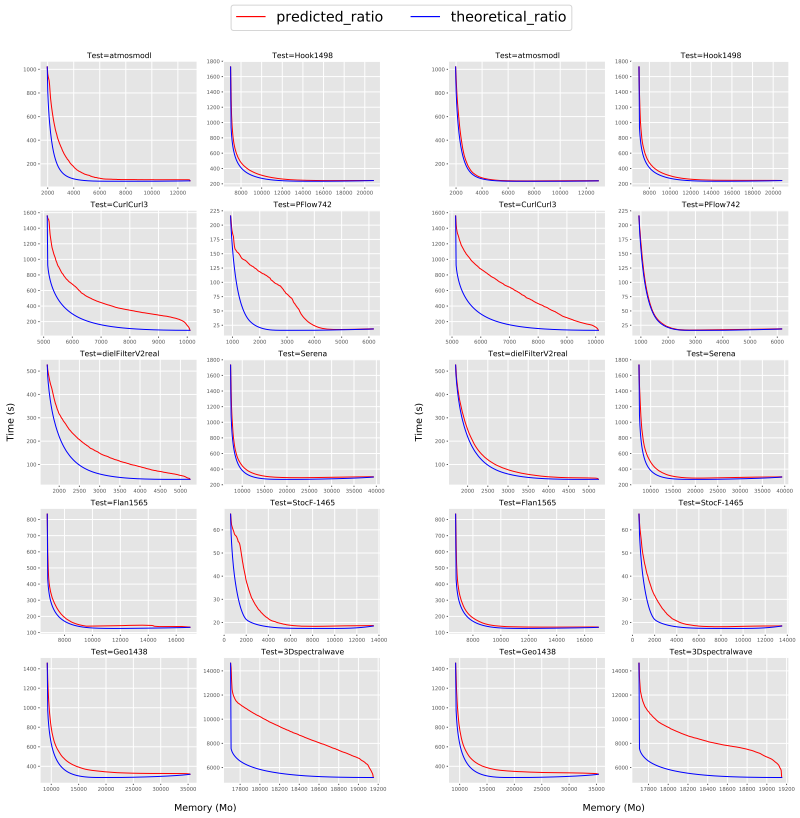
Based on these attractive preliminary results, we now present how this *memory-aware* strategy has been introduced into the PaStiX solver and the gains obtained on actual factorization time and memory usage.

We recall that in the *Minimal Memory* strategy, all blocks are in *early* mode: they are compressed before starting the factorization. During each update, the rank of a block can increase, and thus the overall memory consumption increases up to the end of the factorization. In the *Just-In-Time* strategy, all blocks are in *lazy* mode. Thus, the overall memory consumption decreases up to the end of factorization. At the end of the factorization, the *final* ranks are pretty close, independently of the mode, *early* or *lazy*.

In the modelization of our problem, we made the assumption that the rank is constant all throughout the factorization. We would like to highlight why this assumption in the modelization does not hurt the memory-aware approach. Firstly, the evolution of the rank depends on the order of the different updates. If we consider a sequential approach in this paper, we target parallel executions in future work. Secondly, we expect to have a static model before starting the factorization, while predicting the rank evolution would require the knowledge of previous contributions. As we observed in Section 4.5, the models are not perfect even if they provide a very nice trend. The memory-aware strategy using those models will inherently be dynamic.

The first step consists of selecting blocks which will be performed in the *early* mode, so that the memory consumption of the factors before the factorization starts is below the given memory limit.

As the memory associated to *early* blocks may increase (due to low-rank updates) while the memory associated to *lazy* blocks will decrease (when compressed), it is impossible to estimate the impact these evolutions will have throughout the factorization on the overall memory. Therefore, the dynamic strategy decides to compress additional blocks on-the-fly, if this is required for the memory constraint to always be respected. To



(a) Training with Serena

(b) Training with PFlow742

(c) Training with CurlCurl3

(d) Training with all ten matrices

Figure 5: Execution time of update tasks depending on the memory constraint for ten matrices using a 10^{-8} tolerance and four different training matrices. Two curves are plotted, the best results achievable knowing all information and the results obtained by our predictive models.

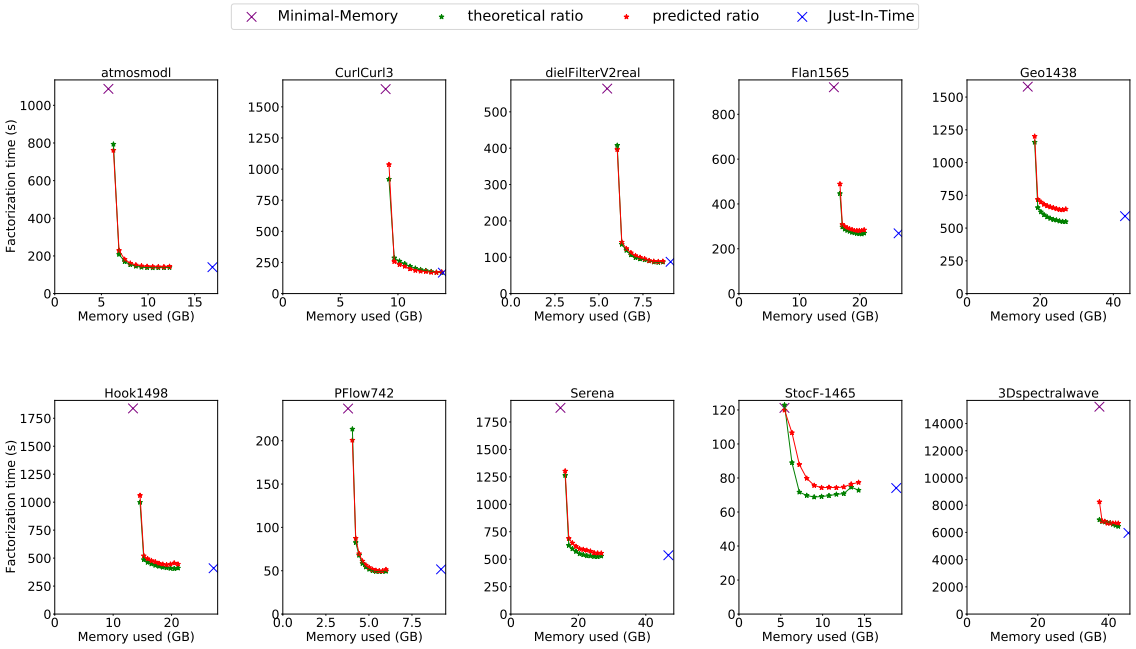


Figure 6: Memory/Performance trade-offs achievable with the proposed bi-objective strategy, compared to existing single objective strategies.

take such a decision, the strategy only needs an ordering of the blocks. Then assuming in our models that ranks are constant simplifies the model without leading to significant drawbacks.

This implementation is a proof-of-concept and has limitations. Firstly, it is only sequential for now, as compressing on-the-fly data blocks may induce deadlocks when using PaStiX in parallel. Secondly, building and using the models as well as sorting blocks is performed using an external tool. In future work, it will be fully integrated into the PaStiX solver, and a parallel memory management will be proposed.

Before presenting the results of the *memory-aware* strategy with respect to existing strategies in the PaStiX solver, we would like to discuss quickly the accuracy of the solution. The accuracy when using *early* or *lazy* updates can be different, as operations are performed differently. One could observe the numerical accuracy of both the *Minimal Memory* and the *Just-In-Time* strategies in ([3], Chap 3). The difference between both methods is very small, in the order of one digit in most cases. Our *memory-aware* strategy being in-between the existing *Minimal Memory* and *Just-In-Time* strategies, its accuracy is also in-between. Moreover, in a recent article [24], it was demonstrated that BLR factorization is numerically stable. The authors have demonstrated that using a fully-structured or a non-fully-structured approach (in our case *Minimal Memory* or *Just-In-Time*) only impacts the constant for the accuracy.

5.6. *Real Gain for the* PaStiX *Solver*

With the implementation presented before, we can now compare the behavior of both *Just-In-Time* and *Minimal Memory* strategies already presented in Table 2 with the *memory-aware* method introduced in this paper. This requires to add an extra entry parameter to the PaStiX solver, the maximum memory authorized.

Now, we run the actual factorization using the PaStiX solver, contrary to the results presented in Section 5.4, where we have estimated the potential gain from the data issued from previous runs. This is why we only performed tests on a limited subset of memory settings. In this section, we evaluate the *memory-aware* strategy with different memory limits, as well as two extreme cases: all blocks executed in *early* mode except those which do not bring memory savings (according to the model) and all blocks executed in *lazy* mode except those which do not reduce execution time (again according to the model). Those two extreme cases correspond to the points that appeared in Figure 4 and represent the first and the last values. In practice those two cases can be seen as an enhancement of existing strategies: it allows to reach almost the minimal memory consumption of the *Minimal Memory* strategy while reducing its execution time and similarly to reach almost the same (sometimes even better) performance as *Just-In-Time* while reducing the memory consumption.

In Figure 6 we present, for ten matrices solved at tolerance 10^{-8}, the memory consumption and factorization time for the *Just-In-Time* and *Minimal Memory* strategies. The models were trained using all ten matrices with a 10^{-8} tolerance. We present both the optimal time (using actual values from the execution) and the time achieved using the predictive models for various memory constraints. The results demonstrate that an interesting trade-off between time and memory can always be achieved. For instance, for the `Geo1438` matrix, increasing memory consumption by 30% divides by 2.2 the execution time of *Minimal Memory* strategy. Similarly, 60% of the memory consumption

of the *Just-In-Time* strategy could be saved by increasing by only 10% the execution time. Using the predictive values we are always within 10% of the optimal performance.

6. Conclusion

In sparse direct solvers, using low-rank compression has emerged as a solution to process larger matrices. Existing solvers use either only low-rank updates (the matrix is compressed beforehand) or only full-rank updates (the matrix is compressed during the factorization). We have presented a *memory-aware* approach, performing both types of updates, whose objective is to enable applications to run as fast as possible while keeping their memory usage under a given memory limit (*e.g.*, the size of the RAM).

We have formalized the optimization problem of choosing which blocks to compress in the *early* or *lazy* mode, and shown its equivalence to the Knapsack problem, which proves our problem NP-complete. We have successfully adapted a 2-approximation algorithm for Knapsack to obtain a 1.02 approximation algorithm for our problem (under realistic hypotheses).

To take advantage of this optimization algorithm, we need information on matrix blocks (memory consumption once compressed, processing time in both modes) which is usually unknown before the computation. To leverage this problem, we have introduced models to predict these values. We have demonstrated that using these models we can achieve a performance close to the results achievable with a perfect omniscient oracle, even in the less favorable case where the models are trained with a matrix from one application field and used with matrices from another field. Our approach sorts all blocks and selects at runtime, depending on the actual memory usage, which ones to manage with low-rank updates (*early* mode) and which ones to manage with full-rank updates (*lazy* mode). This ensures that the memory bound is respected.

Our *memory-aware* strategy has been integrated into the PaStiX solver. The obtained execution times are far lower than the existing memory-conservative strategy (for instance, twice faster), even for memory limits that are only slightly larger than the memory usage of this strategy (for instance, 30% increase in memory). Conversely, it is only marginally slower than the performance-oriented strategy (for instance, 10% increase in execution time) while using drastically less memory (for instance, half the memory). The *memory-aware* strategy thus achieves a kind of "best of both world" performance, allowing a wide range of memory-time trade-offs.

The implementation is for now sequential and the memory management has to be adapted for the parallel case, which is left for future work. The main problem here is to avoid deadlocks, as the next candidate for being compressed may be used by another thread, for instance to apply an update. The work conducted in [25] could be a starting point. Another future work consists of using geometric information of the matrix, which is known to influence ranks, to try to enhance the predictive models of processing times and memory consumption. Finally, we plan to study how to select a representative subset of blocks from different matrices to perform the training in order to obtain both a good accuracy and a fast training.

Up to our knowledge, this work is the first tentative to produce models for low-rank compression in the sparse case. It could be used to exhibit better time estimates to enhance scheduling for sparse direct solvers using low-rank compression.

Acknowledgments

This work is part of the SOLHARIS project, supported by the Agence Nationale de la Recherche, under grant ANR-19-CE46-0009.

References

[1] P. R. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L'Excellent, C. Weisbecker, Improving Multifrontal Methods by Means of Block Low-Rank Representations, SIAM Journal on Scientific Computing 37 (3) (2015) A1451–A1474.

[2] T. Mary, Block Low-Rank multifrontal solvers: complexity, performance, and scalability, Ph.D. thesis, Toulouse University, Toulouse, France (Nov. 2017).

[3] G. Pichon, On the use of low-rank arithmetic to reduce the complexity of parallel sparse linear solvers based on direct factorization techniques, Ph.D. thesis, Université de Bordeaux, Talence, France (Nov. 2018).

[4] W. Hackbusch, A Sparse Matrix Arithmetic Based on $\mathcal{H}$-Matrices. Part I: Introduction to $\mathcal{H}$-Matrices, Computing 62 (2) (1999) 89–108.

[5] W. Hackbusch, S. Börm, Data-sparse Approximation by Adaptive $\mathcal{H}^2$-Matrices, Computing 69 (1) (2002) 1–35.

[6] P. Ghysels, X. S. Li, C. Gorman, F.-H. Rouet, A robust parallel preconditioner for indefinite systems using hierarchical matrices and randomized sampling, in: 2017 IEEE IPDPS, 2017, pp. 897–906.

[7] A. Aminfar, 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 107 (6) (2016) 520–540.

[8] T. A. Davis, R. Sivasankaran, W. M. Sid-Lakhdar, A survey of direct methods for sparse linear systems, Acta Numerica 25 (2016) 383–566. doi:10.1017/s0962492916000076.

[9] A. George, Nested dissection of a regular finite element mesh, SIAM Journal on Numerical Analysis 10 (2) (1973) 345–363.

[10] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM Journal on scientific Computing 20 (1) (1998) 359–392.

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

[12] J. J. Dongarra, J. Du Croz, S. Hammarling, I. S. Duff, A set of level 3 basic linear algebra subprograms, ACM Trans. Math. Softw. 16 (1) (1990) 1–17.

[13] P. Hénon, P. Ramet, J. Roman, PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems, Parallel Computing 28 (2) (2002) 301–321.

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

[15] J. E. Savage, Models of computation - exploring the power of computing, Addison-Wesley, 1998.

[16] L. Eyraud-Dubois, L. Marchal, O. Sinnen, F. Vivien, Parallel scheduling of task trees with limited memory, ACM Trans. Parallel Comput. 2 (2) (2015) 13:1–13:37. doi:10.1145/2779052.

[17] L. Marchal, B. Simon, F. Vivien, Limiting the memory footprint when dynamically scheduling dags on shared-memory platforms, J. Parallel Distributed Comput. 128 (2019) 30–42. doi:10.1016/j.jpdc.2019.01.009.

[18] H. Kellerer, U. Pferschy, D. Pisinger, Multidimensional knapsack problems, in: Knapsack problems, Springer, 2004, pp. 235–283.

[19] A. Gupta, Dynamic programming: Knapsack, scheduling, available online at https://www.cs.cmu.edu/afs/cs/academic/class/15854-f05/www/scribe/lec10.pdf, course note of 15-854 "Approximations Algorithms" at Carnegy Mellon University.

[20] C. Chekuri, UIUC course CS 598 on approximation algorithms, lecture 4, available online at https://courses.engr.illinois.edu/cs598csc/sp2009/Lectures/lecture_4.pdf (2009).

[21] S. Ubaru, Y. Saad, Fast methods for estimating the numerical rank of large matrices, in: ICML, 2016, pp. 468–477.

[22] L. Stanisic, E. Agullo, A. Buttari, A. Guermouche, A. Legrand, F. Lopez, B. Videau, Fast and accurate simulation of multithreaded sparse linear algebra solvers, 2015 IEEE ICPADSdoi:10.1109/icpads.2015.67.

[23] T. A. Davis, Y. Hu, The University of Florida sparse matrix collection, ACM Trans. Math. Softw. 38 (1). doi:10.1145/2049662.2049663.

[24] N. Higham, T. Mary, Solving block low-rank linear systems by lu factorization is numerically stable, IMA Journal of Numerical Analysis.

[25] M. Sergent, D. Goudin, S. Thibault, O. Aumage, Controlling the Memory Subscription of Distributed Applications with a Task-Based Runtime System, in: 21st Int. Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS), Chicago, IL, 2016, pp. 318–327.