Robust memory-aware mappings for parallel multifrontal factorizations

Emmanuel Agullo, Patrick Amestoy, Alfredo Buttari, Abdou Guermouche, Jean-Yves L'Excellent, François-Henry Rouet

ROBUST MEMORY-AWARE MAPPINGS FOR PARALLEL MULTIFRONTAL FACTORIZATIONS

EMMANUEL AGULLO[*], PATRICK R. AMESTOY[†], ALFREDO BUTTARI[‡], ABDOU GUERMOUCHE[§], JEAN-YVES L'EXCELLENT[¶], AND FRANÇOIS-HENRY ROUET[‖]

Abstract. We study the memory scalability of the parallel multifrontal factorization of sparse matrices. In particular, we are interested in controlling the active memory specific to the multifrontal factorization. We illustrate why commonly used mapping strategies (e.g., the proportional mapping) cannot provide a high memory efficiency, which means that they tend to let the memory usage of the factorization grow when the number of processes increases. We propose "memory-aware" algorithms that aim at maximizing the granularity of parallelism while respecting memory constraints. These algorithms provide accurate memory estimates prior to the factorization and can significantly enhance the robustness of a multifrontal code. We illustrate our approach with experiments performed on large matrices.

AMS subject classifications. 05C50, 65F05, 65F50, 68W10

1. Introduction.

We consider the memory scalability of sparse direct methods for the solution of linear systems on distributed-memory architectures. We focus on the *multifrontal method* [6, 7]. In this method, the consumed memory space consists of the factors to be computed (e.g., the LU factors of a given matrix A) and some temporary data that we call the *active memory* and that we describe in detail in the next section. Both the size of the factors and of the active memory depend on the input matrix permutation [12] (nested dissection methods are commonly used on large size problems); in this work we focus on studying the memory consumption resulting from different mapping techniques for a fixed matrix permutation. The active memory can represent a significant fraction of the total memory, and, as we will show in Section 2, it does not naturally scale when the number of processes increases. This means that the total memory usage of the factorization can dangerously grow with the number of processes. This highlights the need for mapping and scheduling algorithms that are able to control the active memory. In Section 3, we suggest *memory-aware* mapping algorithms that aim at maximizing the granularity of parallelism under a given memory constraint (the maximum memory usage of a process). We illustrate our approach in Section 4 with experiments carried out on large matrices using up to 256 processes.

Although we focus on the multifrontal method, our study can be applied to any application with a tree-shaped workflow graph since, as we recall in the next subsection, the multifrontal method relies on the so-called *elimination tree* [24]. A similar problem has also been recently explored from a more theoretical point of view [8].

1.1. Background and context.

We briefly recall the main ingredients of the multifrontal method. We are to compute a factorization of a given matrix A, $A = LU$ if the matrix is unsymmetric, or $A = LDL^T$ if the matrix is symmetric. Without loss of generality, in the rest of the paper we assume that A is non-reducible. For a

[*]INRIA/LaBRI, Bordeaux, France (emmanuel.agullo@inria.fr).
[†]Université de Toulouse, INPT(ENSEEIHT)-IRIT, France (amestoy@enseeiht.fr).
[‡]CNRS-IRIT, Toulouse, France (alfredo.buttari@enseeiht.fr).
[§]Université Bordeaux 1/LaBRI, Bordeaux, France (abdou.guermouche@labri.fr).
[¶]Université de Lyon, Inria and ENS Lyon, France (jean-yves.l.excellent@ens-lyon.fr).
[‖]Université de Toulouse, INPT(ENSEEIHT)-IRIT, France and Lawrence Berkeley National Lab, Berkeley, USA. (fhrouet@lbl.gov).

matrix A with an unsymmetric pattern, we assume that the factorization takes place using the structure of $A + A^T$, where the summation is structural. In this case, the multifrontal method relies on a structure called the elimination tree. A few equivalent definitions are possible (we recommend the survey by Liu [17]); we use the following:

Definition 1. *Assume $A = LU$ where A is a sparse, structurally symmetric, $N \times N$ matrix. Then, the elimination tree of A is a tree of N nodes, with the ith node corresponding to the ith column of L and with the parent relations defined by:*

$$par(j) = \min\{i : i > j \text{ and } \ell_{ij} \neq 0\}, \text{ for } j = 1, \ldots, N-1$$

In practice, nodes are *amalgamated*: nodes that represent columns and rows of the factors with similar structures are grouped together in a single node. In the end, each node is a square dense matrix (referred to as a *frontal matrix*) with the following 2×2 block structure:

$$\begin{bmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{bmatrix}$$

Factoring the matrix consists in a bottom-up traversal of the tree, following a topological order (a node is processed before its parent). Processing a node consists in:

- forming (or *assembling*) the frontal matrix: this is achieved by summing the rows and columns of A corresponding to the variables in the $(1,1)$ block with temporary data that has been produced by the child nodes;
- eliminating the pivots in the $(1,1)$ block F_{11}: this is done through a partial factorization of the frontal matrix which produces the corresponding rows and columns of the factors. At this step, the so-called Schur complement or *contribution block* is computed as $F_{22} \leftarrow F_{22} - F_{21} \cdot F_{11}^{-1} \cdot F_{12}$ and stored in a temporary memory; it will be used to form the frontal matrix associated with the parent node. Therefore, when a node is activated, it "consumes" the contribution blocks of its children.

In the multifrontal factorization, the *active memory* (at a given step in the factorization) consists of the frontal matrix being processed and a set of contribution blocks that are temporarily stored and will be consumed at a later step. The multifrontal method lends itself very naturally to parallelism since multiple processes can be employed to factor a large frontal matrix together, or to process concurrently frontal matrices belonging to independent subtrees. These two sources of parallelism are commonly referred to as *node* and *tree parallelism*, respectively, and their correct exploitation is the key to achieving high performance on parallel supercomputers.

1.2. Notations, assumptions and definitions. We define some notations that will be useful in the rest of the paper: $\mathcal{T}_i$ is the subtree rooted at node i of the elimination tree; $\mathcal{P}_i$ is the set of nodes in the path that connects node i to the root of the tree; cb_i is the memory requirements for storing the contribution block F_{22} associated with node i; similarly, m_i (or m when there is no ambiguity) is the memory for the frontal matrix associated with a node i.

We rely on the following assumption:

Assumption 2. *In the case of a sequential execution, or in the case of a whole subtree processed on a single process, the corresponding tree nodes are visited following a postorder traversal, that is, a topological ordering such that nodes in any subtree are ordered consecutively.*

Among all topological orderings, postorderings have been shown to have good properties in terms of memory usage [13]. Furthermore since the memory used in a sequential context will be used in this paper as a target for global memory usage in a parallel environment, a postordering based on the work of [15] has been used for our experiments.

We also introduce the notion of *stacked set*[1]:

DEFINITION 3 (Stacked set). *The stacked set $\mathcal{S}_i$ of a node i is the set of nodes that are visited before i and still have unvisited siblings when the traversal reaches node i:*

$$\mathcal{S}_i = \{j : j \text{ is numbered before } i \text{ in the postorder and } sib(j) \cap \mathcal{P}_i \neq \varnothing\},$$

where $sib(j)$ is the set of siblings of node j.

We illustrate the notion of stacked set in Figure 1. At a given node i in the tree, the active memory consists of the frontal matrix associated with i and the contribution blocks of the nodes in $\mathcal{S}_i$. In the rest of the study, we denote by S_i the sequential peak of active memory associated with a node i, i.e., the peak of active memory yielded by a sequential traversal of the subtree rooted at i. We note that such a traversal is included in the traversal of the subtree rooted at the parent of i; therefore, by construction, the property $S_i \leq S_{par(i)}$ holds. Note that S_i is different from the peak of active memory when reaching i during the traversal of the whole tree. When reaching i, the maximum active memory usage is at least S_i plus the memory for the contribution blocks of the nodes in $\mathcal{S}_i$, the stacked set of i, that are stored when i is reached. We insist that S_i is the peak of active memory for traversing the subtree rooted at i, regardless of the rest of the tree. In order to compare sequential and parallel executions (see also Section 1.4 for notations that are specific to parallel executions), we denote by S_{seq} the peak of active memory for a sequential traversal of the entire tree; if r denotes the root of the tree, we have $S_{seq} = S_r$.

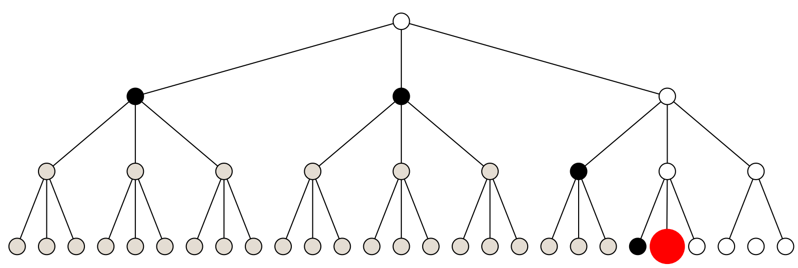


FIGURE 1. *Stacked set of a node i (solid nodes); the nodes that have been visited (but do no longer contribute to the active memory) are shaded.*

1.3. Controlling the active memory in the serial multifrontal method. Reducing the memory requirements of the serial multifrontal method (in particular the active memory) has been extensively studied [15, 16, 11, 14]. Liu [15] proposed a way to compute the postorder traversal of the tree that minimizes the active memory when considering an *in-place assembly scheme*, in which the memory for the frontal matrix of a parent node is allowed to overlap with the contribution block of its last child. Essentially, the idea is to reorder the children i of a given node in decreasing order of $\max_i\{S_i, m\} - cb_i$, where $m = m_{par(i)}$ denotes the storage of the frontal

[1] The notion is derived from [26] where it is called the "visited set" and defined for binary trees.

matrix associated with the parent node. In this paper, we consider that the memory for a frontal matrix is allocated after all its children have been processed (*terminal allocation scheme*); in this variant, the postorder that minimizes memory is obtained by processing the children of a given node in decreasing order of $S_i - cb_i$. However, other schemes exist and they have different properties with respect to the memory usage and, in the out-of-core case, disk traffic [2, 11, 14].

In terms of traversals, a postorder is generally used (Assumption 2) because it allows for using a stack mechanism for storing the contribution blocks, which significantly enhances the memory management. Liu also showed how to compute the topological ordering (which might not be a postorder) that minimizes the active memory [17], using a *tree pebble game* formulation that we briefly describe in Section 5. More recently, Jacquelin et al. [13] revisited this problem. Their experimental findings are that, on trees corresponding to large sparse matrices coming from real applications, the optimal traversal is a postorder 95% of the time. In the worst case, the memory overhead induced by using the best postorder instead of the best order is 18%. This confirms that using a postorder traversal (in the serial case) is a reasonable choice, especially since it allows for an efficient stack mechanism.

1.4. Memory efficiency and problem statement. In the parallel case, the active memory footprint depends on the node-to-process mapping followed during the factorization. We detail different strategies in the next section and highlight their influence on the active memory in further sections. Here, we define our problem. First, we emphasize that the memory usage for a parallel execution might be completely different from that of a sequential execution; therefore, the problem is not only about evenly distributing the amount of memory needed for a sequential execution among the different processes. A bad mapping and/or scheduling might yield a total memory usage significantly higher than that of a sequential execution. We give a simplistic example in Figure 2; the tree is mapped on two processes. We assume that the two leaf nodes are significantly larger than the two other nodes ($C \gg \varepsilon$) and that the size of their contribution block is negligible. The total amount of active memory for a sequential traversal is (approximately) $S_{seq} = C$. However, using the mapping proposed in the figure (which may seem reasonable since it aims at enforcing some balance), the total amount of memory is $2C$ (assuming that the two leaf nodes are active at the same time, which is very likely in a parallel execution). This demonstrates that the mapping and the scheduling strategies influence not only the balance of the memory usage but also the total consumption.

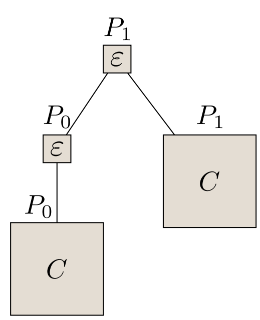


FIGURE 2. *The tree is mapped on two processes P_0 and P_1. We assume $C \gg \varepsilon$. The peak of active storage for a sequential traversal is (roughly) C. However, the total amount of active memory in a parallel execution is $2C$.*

In a parallel setting, the objective depends on the type of system we consider:

- In a shared-memory context, the objective is to minimize the total memory footprint. Unless one considers memory locality issues, balancing the memory usage on the different processes is not important.
- In a distributed-memory context, minimizing the total memory consumption is desirable but it is also crucial to maintain a balanced memory usage between the processes, to prevent a process from running out of memory (assuming that all the processes can access the same amount of memory, which is the most frequent setting).

Minimizing the overall memory consumption in a parallel setting is complicated unless a schedule with very specific properties is used. In our distributed memory context, the maximum peak of active memory (over the set of processes) is the target for our optimization problem. Note that the sum of the peaks of the different processes provides an upper bound on the total consumption (which might be loose). For a parallel execution on p processes, we denote by $S_{max}(p)$ and $S_{avg}(p)$ the maximum and average peaks of active memory among the p processes, respectively. $S_{avg}(p)$ is computed as the sum of the p peaks divided by p; with the above observation $p \cdot S_{avg}(p)$ is an upper bound on the total consumption.

The performance of a parallel algorithm executed on p processes is often assessed using a notion of *efficiency*; given the above observations, we consider two kinds of *memory efficiency* metrics that depend on p:

- $e_{avg}(p) = \dfrac{S_{seq}}{p \cdot S_{avg}(p)}$; this metric compares the total memory usage (using an upper bound, as described above) to that of a sequential execution and is relevant in a shared-memory context as well as in a distributed-memory one.
- $e_{max}(p) = \dfrac{S_{seq}}{p \cdot S_{max}(p)}$; this metric detects that one (or more) process(es) use(s) too much memory, which is only relevant in a distributed-memory context.

Assuming p_i processes are mapped on the subtree rooted at node i of the tree, a lower bound on the average storage needed by these processes to process the subtree is defined as $\bar{S}_i = S_i/p_i$. This quantity corresponds to the memory consumption per process in the ideal case where the peak S_i is uniformly distributed; as we explain below, this case is very unlikely in practice, unless very specific mapping schemes are used within the subtree.

Consider the following parallel scheme. The tree is processed following the postorder used in the sequential case (*tree serialization*) and each node of the tree is mapped on all processes; this is what we will refer to as *tree serialization mapping*. If we assume a perfect memory balance within each node and within each contribution block, this tree serialization mapping technique clearly provides a perfect memory scalability ($S_{avg} = S_{max} = \bar{S}_i$ and $e_{avg}(p) = e_{max}(p) = 1$). However, it does not exploit tree parallelism (all the branches are serialized). It also leads, by forcing a large number of processes to be used at each node, to an unnecessary increase in communication (both within nodes when performing dense partial factorizations, and between nodes when communicating contribution blocks). Finally, it does not deliver an adequate granularity of operations within nodes. These drawbacks are likely to induce a significant performance penalty, as we assess in Section 4. This is why no solver that we are aware of relies on this strategy. An improvement of this approach could consist in introducing some tree parallelism near the leaves, as long as this does not increase the overall memory peak too dangerously. In practice, minimizing the active memory is not the best formulation of the problem we want to solve. Instead of minimizing it, we would rather *control* the active memory by enforcing a given memory constraint that would be provided by the user or defined by hardware specifications. Then, for this memory constraint, we would like to maximize tree parallelism and parallelism granularity, in order to avoid as much as possible the aforementioned drawbacks related to communication volume and small task granularity; this is exactly the aim of the mapping technique we describe in Section 3.

1.5. Mapping techniques. Different mapping strategies are commonly used in multifrontal codes. The subtree to subcube mapping by Liu, George and Ng [10] and the proportional mapping by Pothen and Sun [19] are popular strategies and are the basis for more sophisticated schemes. These are special cases of the wider family of *tree partitioning* methods where the set of processes mapped on a node of the tree is partitioned into disjoint subsets and each subset is assigned to a child subtree. Tree partitioning mappings are well appreciated because they help reducing the volume of data transfers thanks to a good data locality and because they allow for a good use of both tree and node parallelism. In the proportional mapping method, this recursive splitting of processes sets is guided by a balancing criterion. This mapping technique consists in a top-down traversal of the tree where every node is assigned a set of processes. All the processes are assigned to work on the root node. This is a natural choice since the root node is the last task to be performed in the factorization. Then, for every node in the tree, the set of processes working at that node is split among its children, proportionally to the weights (determined according to a given metric) of the subtrees rooted at these children. Denoting by w_i the weight of the subtree rooted at node i, and by $par(i)$ the parent of node i, the number of processes p_i given to node i is then

$$p_i = \frac{w_i}{\sum\limits_{j;\, par(j)=par(i)} w_j} \cdot p_{par(i)}. \tag{1}$$

This procedure is applied in a recursive fashion to the whole tree, starting from the root r; the recursion stops when leaf nodes are reached or entire subtrees are mapped onto single processes, which happens because the number of nodes in the tree is commonly much larger than the number of processes. Not considering the case where fractions of processes are allowed to be mapped on different subtrees (meaning that such a process would work less than the others on a given subtree, and be assigned less memory), rounding is performed in (1) in order to ensure that:

- p_i is an integer for all nodes i; and
- the tree partitioning property holds, i.e., $p_{par(i)} = \sum\limits_{j;\, par(j)=par(i)} p_j$.

The metric used at each step of the mapping in the original method proposed by Pothen and Sun is the workload of each subtree; we refer to this case as the *workload-based proportional mapping*. Clearly, this criterion can be replaced by another one depending on the objectives. If one aims to achieve a memory balance rather than a load balance, a possible heuristic consists in using a *memory-based* variant; we report on experimental results using this strategy in the experimental section. Prasanna and Musicus proposed a scheduling strategy for tree-shaped task graphs when the time for computing a parallel task (a *malleable task*) using p processes is exactly $\frac{L}{p^\alpha}$ (with $0 < \alpha \leqslant 1$) where L is the length of the task [20]. Beaumont and Guermouche assessed this behaviour in the multifrontal method [5].

An interesting property of the tree partitioning mapping is that the traversal of every process, i.e., the set of tasks that a process executes and the order in which they are processed, is deterministic. Indeed, every process is in charge of a sequential subtree and takes part in the computation of the parallel nodes in the path between that subtree and the root of the elimination tree; this defines a single possible traversal. Denoting by i the root of the sequential subtree mapped on a given process, the traversal followed by that process consists of a postorder traversal of the subtree rooted at i, $\mathcal{T}_i$, followed by the path from i to the root node, $\mathcal{P}_i$. It is then very easy to estimate the memory usage of a process by simply simulating the traversal followed by the process together with the variations in its memory usage.

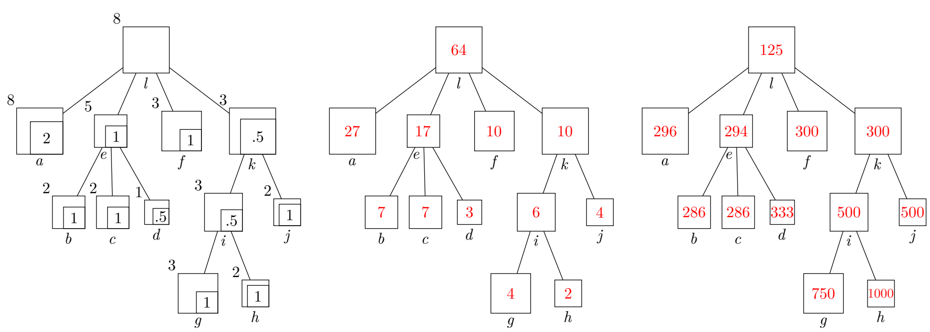
In the family of tree partitioning mappings discussed above, the set of processes associated with a node is split into disjoint sets that are distributed to its children. As we will demonstrate in Sections 2.1 and 2.2, this is not a memory-friendly strategy. One can also choose to use a *relaxed* mapping in which the sets of processes given to different siblings can overlap. Such a relaxed scheme may (slightly) reduce the memory usage in a parallel context and may allow for dynamic executions: a process being allowed to work on two parallel branches follows a traversal that interleaves the nodes of these branches depending on the progress of each branch. In that latter case, however, the traversal followed by a process can no longer be forecast and the memory usage cannot be accurately estimated. Also, note that one could devise different tree partitioning mapping techniques that provide a more balanced memory use and better memory efficiency than the proportional mapping; discussing such techniques is out of the scope of this document. In the remainder, we will thus use the proportional mapping as a representative of the whole family of tree partitioning mappings; it must be noted, though, that the novel techniques proposed in Section 3 are perfectly compatible with any tree partitioning technique.

2. Memory scalability issues. Here we show that commonly-used mapping techniques, such as the proportional mapping, do not achieve a good memory scalability. First, we provide a very simple but realistic example where we analyze step-by-step, on a given tree, the behavior of the proportional mapping with respect to the active memory. In this example, after three steps of proportional mapping, i.e., once the grand-grandchildren of the root node are mapped, the memory efficiency is bounded by 0.125, which is unacceptably low. We then provide a theoretical result showing that, on regular grids ordered with nested dissection, the memory efficiency rapidly tends to zero when the number of processes increases.

2.1. A simple example. We illustrate the behavior of the proportional mapping on a simple yet realistic elimination tree. We consider a memory-based proportional mapping strategy, but we could make the same observations about a workload-based strategy. We consider the tree in Figure 3(a) (which could correspond to the top of the elimination tree from a real problem), which is to be mapped on 64 processes. First, these 64 processes are assigned to the root node l. Then, a first step of memory-based proportional mapping is used to distribute these 64 processes among the four children of l: a, e, f, and k, as illustrated in Figure 3(b). The sequential peaks of active memory of the subtrees rooted at a, e, f, and k are 8 GB, 5 GB, 3 GB and 3 GB, respectively. Therefore, a gets $\frac{8}{8+5+3+3} \cdot 64 \approx 27$ processes; e gets $\frac{5}{8+5+3+3} \cdot 64 \approx 17$ processes and f and k get $\frac{3}{8+5+3+3} \cdot 64 \approx 10$ processes. Note that we have chosen to round the numbers of processes so that they add to 64 and are distributed without any overlap; a relaxed technique could be used but this would not significantly change the result. At this stage, we can compute a lower bound of the peak of active memory of every process. Consider the 27 processes working on the subtree rooted at a. The sequential peak of active memory of this subtree is 8 GB. Therefore, at best, i.e., assuming a perfect memory scalability can be attained for this subtree, the maximum peak of active memory among these 27 processes will be $\bar{S}_a = \frac{8\text{ GB}}{27} = 0.296$ MB. Similarly, the maximum peaks of active memory for the processes working on the subtree rooted at e (f and k respectively) are bounded from below by $\bar{S}_e = \frac{5\text{GB}}{17}$ ($\bar{S}_f = \frac{3\text{GB}}{10}$ and $\bar{S}_k = \frac{3\text{GB}}{10}$, respectively); ignoring the rounding applied to obtain integer numbers of processes, all these peaks are the same ($\bar{S}_a \approx \bar{S}_e \approx \bar{S}_f \approx \bar{S}_k \approx 0.3$ GB) since we have applied a memory-based proportional mapping. We can, thus, derive a first lower bound on the memory efficiency for this problem; since the sequential peak of active memory for the whole tree is 8 GB, the memory efficiency is bounded as follows:

$$e_{max}(p) = \frac{S_{seq}}{p \cdot S_{max}(p)} \leqslant \frac{8\text{ GB}}{64 \cdot 0.3} \leqslant 0.42$$

It is fairly easy to see why the efficiency is low. The sequential peaks of the whole tree and the subtree rooted at a are the same; however, only a small subset of the processes work on the latter subtree. Therefore, even if a perfect memory scalability is attained on the subtree rooted at a, the memory usage for the 27 processes working on that subtree is more than twice what we are targeting ($\frac{8\text{ GB}}{27}$ instead of $\frac{8\text{ GB}}{64}$).



(a) Elimination tree. (b) Proportional mapping. (c) Best-case average peak.

FIGURE 3. *An example of memory-based proportional mapping on $p = 64$ processes. In (a), we indicate next to each node the sequential peak of active memory S_i (in GB) of the corresponding subtree, and within each node, the size of its contribution block cb_i (in GB). In (b), we indicate within each node the number of processes p_i assigned by the proportional mapping. In (c), we indicate within each node the lower bound $\bar{S}_i = \frac{S_i}{p_i}$ on the average storage required per process (in MB).*

We can refine this upper bound on memory efficiency by looking at the lower levels of the tree. By considering the grandchildren of the root node, we see in Figure 3(c) that the memory usage $S_{max}(p)$ is at least $\max\left(\bar{S}_b, \bar{S}_c, \bar{S}_d, \bar{S}_i, \bar{S}_j\right) = 0.5$ GB, yielding

$$e_{max}(p) \leqslant \frac{S_{seq}}{p \cdot 0.5} = 0.25$$

We assumed that a perfect memory scalability was reached in the different subtrees rooted at the grandchildren of the root node, which means that this bound on memory efficiency is likely to be optimistic. Finally, by considering the lowermost level (see node h in Figure 3(c)), we have $e(p) \leqslant 0.125$.

This example is simplistic but illustrates real-life practical situations. In our example, the subtree rooted at a can be seen as a "memory attractor"; since the proportional mapping partitions the list of processes and distributes them among a and its siblings, the memory efficiency necessarily decreases. At this stage of the mapping process, the only way to guarantee a perfect memory scalability is to assign the 64 processes to the subtree rooted at a. Of course, this prevents us from using a strategy similar to a proportional mapping since there is thus no way to assign disjoint sets of processes to siblings. This observation is one of the motivations for the strategies presented in Section 3.

As mentioned earlier, different tree partitioning mappings that behave better than the proportional mapping can be devised. In our example, one can notice that the memory efficiency is limited by the fact that not enough processes are available on node h. Therefore, assigning more processes to the subtree rooted at node k (ancestor of node h) is likely to yield a better memory efficiency.

2.2. Theoretical analysis. We have carried out a theoretical analysis on 2D regular grids ordered with nested dissection and have shown that if a proportional mapping is used to map the elimination tree associated with the grid, the memory efficiency rapidly tends to zero when the number of processes increases. The complete proof can be found in [21]; here we report the main ideas. We used the model described by George [9] and showed that for a regular 2D grid with n^2 nodes, the sequential peak of active memory is

$$S_{seq} \simeq 6.25n^2$$

Then, in order to compute the memory efficiency of the proportional mapping, we used the same reasoning as in the previous examples: for every subtree $\mathcal{T}_i$ (rooted at node i), the memory consumption of a process working at that subtree is at least $\bar{S}_i = \frac{S_i}{p_i}$, where p_i is the number of processes assigned to $\mathcal{T}_i$. Therefore, $\max_i \bar{S}_i$ is a lower bound on S_{max} and thus allows us to derive an upper bound on $e_{max}(p)$. We found that, for p large enough, this maximum is attained at level $0.5 \log p$ (level 0 being the root node) and gives

$$S_{max} \geqslant 29n^2p^{-0.8}$$

Therefore, we deduce the following result:

Theorem 4 (Sub-optimality of the proportional mapping; [21, Theorem 8.1]). *Let T be an elimination tree corresponding to a nested dissection of a regular 2D square grid with a nine-point stencil. For a given number of processes p, the memory efficiency of a strict memory-based proportional mapping of T verifies:*

$$e_{max}(p) = \frac{S_{seq}}{p \cdot S_{max}(p)} \leqslant \frac{6.25}{29}p^{-0.2} \simeq 0.22\, p^{-0.2}$$

(for p sufficiently large).

This shows that, for example, the memory efficiency $e_{max}(p)$ is limited to 0.10 for $p = 32$, and to 0.06 for $p = 512$.

To illustrate Theorem 4, we study the memory efficiency obtained with a memory-based proportional mapping on a 2D grid of size 8000×8000 (leading to a sparse matrix of order 64 millions). For these experiments, we used a more standard 5-point scheme, but we expect the same type of behavior as for a 9-point scheme. This is confirmed by the results presented in Table 1, where the behavior of $e_{max}(p)$ is close to that of

p (MPI processes)	Proportional Mapping		
	S_{max} (MB)	$e_{max}(p)$ (observed)	$\frac{e_{max}(p)}{p^{-0.2}}$
32	966.6	0.10	0.20
64	524.9	0.09	0.21
128	336.5	0.07	0.19
256	193.8	0.06	0.19
512	112.9	0.05	0.19

TABLE 1

Memory scalability of the Proportional Mapping method on a 5-point discretization of a Laplacian on a 8000×8000 *grid* ($S_{seq} = 3112.0$ *MB). We provide the maximum peak of active memory, the memory efficiency as well as the* $\frac{e_{max}(p)}{p^{-0.2}}$ *coefficient.*

the model. These experiments confirm that the memory efficiency is low even with a small number of processes and decreases significantly when the number of processes increases. We recall that $e_{max}(p) \leq 0.10$ means that peak of active memory S_{max} is over ten times the target memory usage S_{seq}/p, which may be unacceptable in some situations.

Finally, in Table 2, we illustrate the behavior of the proportional mapping strategy for a set of matrices from real-life applications (described in Table 3). We observe that, for 64 and 256 processes, the memory efficiency e_{max} lies between 0.04 and 0.37; the average is 0.13. These experimental results, along with the theoretical result presented above, show that there is a need for a mapping technique able to achieve a much better memory scalability.

Matrix name	S_{seq} (MB)	S_{max} (MB)	e_{max}	S_{avg} (MB)	e_{avg}
cage13	21400.5	911	0.37	737	0.45
pancake2_3	11158.3	1723	0.10	619	0.27
as-Skitter	3810.3	556	0.11	190	0.31
HV15R	88857.6	23623	0.07	10126	0.15
MORANSYS1	17845.3	1732	0.16	939	0.30
meca_raff6	6638.0	2951	0.04	1740	0.06
Geoazur_192	65949.7	4416	0.06	1851	0.14

TABLE 2

Memory scalability of the Proportional Mapping method for the matrices described in Table 3 (with $p = 64$*, except for Geoazur_192 for which* $p = 256$*).*

3. Memory-aware mapping algorithms. We demonstrated that a memory-based proportional mapping leads to a low scalability of the active memory. However this mapping is interesting in terms of performance since it maximizes tree parallelism and reduces the volume of communication within parallel nodes and between nodes of the tree. We also introduced a "tree serialization mapping" which consists in a constrained traversal of the tree where all the processes work at every node, following a postorder. However, this solution generates prohibitive amounts of communications, it does not exploit tree parallelism and yields small granularity of computations on nodes at the bottom of the tree. Therefore it is not time efficient; we assess this in the experimental section. Here, we introduce a "memory-aware mapping" that hybridizes these two techniques and tries to enforce a given memory constraint (the maximum amount of active memory that a process can use). The idea was first worked on during the PhD thesis of one of the authors [1] and is described in Section 3.1. It consists in a tree serialization mapping in memory demanding parts of the tree, and a proportional mapping whenever we can be sure that it will not violate the memory constraint. We suggest in Section 3.2 some refinements that improve the granularity of parallelism while enforcing the same memory constraints.

3.1. Flat memory-aware algorithm.

3.1.1. Main idea. We assume that we are given a memory constraint M_0 that represents the maximum amount of active memory that a process is allowed to use. We will also use the notation M_i, $i > 0$, to denote the memory constraint for a subtree τ_i rooted at i ($M_0 = M_r$ in case r is the root of the entire tree). The memory-aware mapping works as follows. We assume that the tree has been reordered to reduce the sequential peak of active memory (as mentioned in Section 1.3) and that the sequential peaks S_i have been computed for every subtree $\mathcal{T}_i$. Then, a top-down traversal of the tree is performed to compute the mapping. All the processes are first assigned to the root node r. Then, recursively, once a subtree τ_r rooted at r is mapped on p_r processes, its children are mapped as follows. We first check whether a proportional mapping of the children is feasible by simulating a proportional mapping and verifying that the memory constraint is respected at every child i. Denote p_i the number of processes that a proportional mapping would assign to child i (Equation 1). For every child i, we check the condition $\bar{S}_i = \frac{S_i}{p_i} \leqslant M_r$:

- If all child subtrees $\mathcal{T}_i$ respect this condition, then the step of proportional mapping is accepted; the child subtrees will be processed in parallel on the number of processes provided by the step of proportional mapping. For the subsequent steps of the mapping procedure, the memory constraint is unchanged: $M_i = M_r$.
- If at least one of the subtrees does not respect the condition, then the step of proportional mapping is rejected. All the child subtrees τ_i inherit the processes of their parent ($p_i = p_r$) and will be processed one after another during the factorization, following the same order as the one of the sequential execution. In this case, when a child subtree $\mathcal{T}_i$ is processed, the contribution blocks of the previous siblings j ($par(j) = par(i) = r, j < i$, assuming the order of the siblings is in agreement with the postorder) are stacked and equally distributed in the memory of the $p_j = p_i = p_r$ processes. Therefore, for the next steps of the mapping procedure, the memory constraint is modified in order to take into account these contributions blocks: $M_i = M_r - \sum_{par(j)=r,j<i} \frac{cb_j}{p_j}$ (where $p_j = p_i = p_r$).

At each step of the traversal, the condition $\bar{S}_i \leqslant M_r$ means "is it possible to process the subtree $\mathcal{T}_i$ on p_i processes, using a memory at most equal to $M_i = M_r$ on each process?". Thus, when a step of proportional mapping is accepted, we ensure that every subtree will respect the memory constraint. In the end, this algorithm yields a hybrid mapping in-between a proportional mapping and a tree serialization mapping.

The memory-aware mapping algorithm is presented in a recursive fashion in the pseudo-code of Figure 4; we provide a simplified version where we only compute the number of processes given to each node, not a true mapping.

Every time a tentative proportional mapping is not acceptable (line 15) at a given subtree, the memory-aware algorithm decides that the child subtrees should be serialized (line 17). The dependency induced by this serialization is stored during the

```
subroutine ma_mapping(r, p, M, S)
  ! r     : root of the subtree
  ! p(:)  : number of processes mapped each node
  !         input  : p(r)
  !         output : p(i), i in subtree rooted at r, i < r
  ! M(:)  : memory constraint per process for each node
  !         input  : M(r)
  !         output : M(i), i in subtree rooted at r, i < r
  ! S(:)  : sequential peak memory consumption for all nodes (input)

  call prop_mapping(r, p, M)

  forall (i children of r)
    ! check if the constraint is respected (here M(i)=M(r))
    if( S(i)/p(i) > M(i) ) then
       ! reject prop. mapping and revert to tree serialization
       call tree_serialization(r, p, M)
       exit      ! from forall block
    end if
  end forall

  forall (i children of r)
    ! apply MA mapping recursively to all siblings
    call ma_mapping(i, p, M, S)
  end forall
end subroutine ma_mapping


subroutine prop_mapping(r, p, M)
  forall (i children of r)
    p(i) = share of p(r) from proportional mapping (Equation (1))
    M(i) = M(r)
  end forall
end subroutine prop_mapping


subroutine tree_serialization(r, p, M)
  stack_siblings = 0
  forall (i children of r) ! in increasing order
     p(i) = p(r)
     ! update the memory constraint for the subtree
     M(i) = M(r) - stack_siblings
     stack_siblings = stack_siblings + cb(i)/p(r)
  end forall
end subroutine tree_serialization
```

FIGURE 4. *Flat memory-aware algorithm.*

execution of the mapping algorithm and will then have to be taken into account by the parallel factorization algorithm.

We mentioned that the memory-aware mapping aims at ensuring that a given memory constraint is respected. Another favorable property is that, like a strict proportional mapping or a tree serialization mapping, it allows for computing accurate memory estimates prior to the factorization. Indeed, at a given set of siblings in the tree, the set of processes is either perfectly split (when a step of proportional mapping is accepted) or the traversal of the set of siblings is completely constrained (when a step of proportional mapping is rejected). Therefore, as in the proportional mapping and in the tree serialization mapping, it is possible to compute memory estimates simply by simulating a (partial) postorder traversal of the tree for every process.

3.1.2. Example. We illustrate a few steps of memory-aware mapping in Figure 5, using the tree of Figure 3(a) again. The tree is to be mapped on $p = 64$ processes and we choose a rather tight memory constraint $M_0 = 160$ MB (i.e., we target $e_{max} = \frac{8 \text{ GB}}{64 \cdot 160 \text{ MB}} = 0.8$). First, the 64 processes are assigned to the root node l; then the four children a, e, f and k of l are mapped. The first step consists in computing a proportional mapping of the four child nodes. a is given 27 processes, e is given 17 processes and f and k are given 10 processes each. Then, the memory constraint is checked for the subtrees. At node a, the sequential peak of active memory is 8 GB; thus $\bar{S}_a = \frac{8 \text{ GB}}{27} = 296$ MB is greater than M_0. Therefore, the subtree rooted at a cannot be processed using 27 processes without violating the memory constraint. Thus the step of proportional mapping is rejected; the four child subtrees are mapped on the 64 processes and are serialized ($\mathcal{T}_a$ will be processed first, then $\mathcal{T}_e$, and so on). Then the four subtrees are mapped using the same procedure.

Now consider the mapping of the subtree rooted at e. Since we have serialized the four child subtrees of l, we have to take into account that, when processing $\mathcal{T}_e$, the contribution block of a is stacked in memory and equally distributed among the processes. For a given process, the available memory is no longer $M_l = M_0$ but $M_e = M_l - \frac{2 \text{ GB}}{64} = 129$ MB. A proportional mapping of the three children of e assigns 25 processes to b and c and 14 processes to d. This step of proportional mapping can be accepted since the memory constraint is respected for all the subtrees of e: $\bar{S}_b = \frac{S_b}{25} = \bar{S}_c = \frac{S_c}{25} = 80 \text{ MB} < 129 \text{ MB}$ and $\bar{S}_d = \frac{S_d}{14} = 72 \text{ MB} < 129 \text{ MB}$. A step of proportional mapping is also accepted when mapping the two subtrees below k since $M_k = 160\text{MB} - \frac{2 \text{ GB}}{64} - \frac{1 \text{ GB}}{64} - \frac{1 \text{ GB}}{64} = 98$ MB (because of the contribution blocks of a, e and f) is greater than both $\bar{S}_i = \frac{S_i}{28}$ and $\bar{S}_j = \frac{S_j}{38}$. However, the children of i cannot be mapped using a proportional mapping. This would assign 23 processes to g and 15 to h, leading to $\bar{S}_g = \frac{S_g}{27} = \frac{3 \text{ GB}}{27} = 111$ MB, but the memory available for processing $\mathcal{T}_i$ is only $M_i = M_k = 98$ MB. In the end, the mapping of the tree is a combination of local proportional mappings and local tree serialization mappings; it ensures that the efficiency is at least $e_{max} = \frac{8 \text{ GB}}{64 \cdot 160 \text{ MB}} = 0.8$, which is much better than what a proportional mapping delivers (as shown in the previous section).

3.2. Aggregated memory-aware algorithm. The main idea of the memory-aware mapping is to detect memory-demanding parts of the tree. When a problematic subtree is detected among a set of siblings, the whole set of siblings it belongs to is mapped using a local tree serialization mapping; all the siblings inherit the mapping from their parent, and all the subtrees are serialized, i.e., processed one after another. Although this works well in enforcing the memory constraint, this is quite constraining as it potentially maps small subtrees on many processes and forces many serializations even in less memory-demanding parts of the tree. Indeed for a given set of siblings in the tree, subtrees rooted at these siblings might have a large sequential peak of active memory while other ones might have a small peak and do not require to be mapped on many processes. This is unlikely to happen for regular problems (e.g., PDEs with finite elements discretizations) ordered with nested dissection where the trees are quite often fairly well balanced, and it is more common on irregular problems. In this situation,

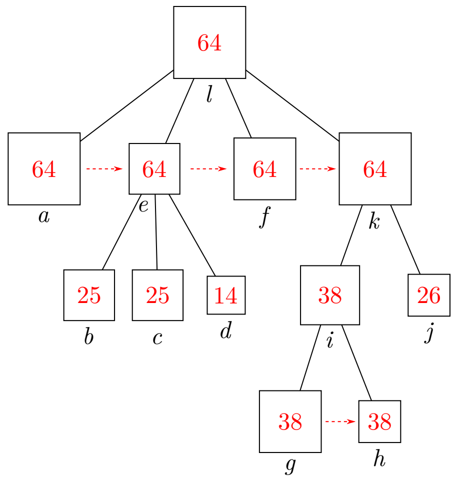


FIGURE 5. *Simple example of memory-aware mapping. The tree shown in Figure 3(a) is mapped using the flat memory-aware mapping with $M_0 = 160$ MB. The scheduling constraints resulting from tree serialization are shown with arrows. The subtree rooted at a is processed first and is followed by the subtrees rooted at e and f, and the subtree rooted at k is processed last. Similarly, the subtree rooted at i is mapped using a local tree serialization mapping.*

we would like to relax the baseline strategy and avoid serializing the whole set of siblings. In this section, we refine the main idea of the memory-aware mapping and propose a strategy where, when working on a set of siblings, we try to detect groups of subtrees within which a proportional mapping is applied and that are serialized in such a way that, like the flat strategy, memory constraints are always respected. The interest is that this variant allows us to decrease the number of serializations as well as the number of processes in the "easier parts" of the tree. In the remainder of the paper, this variant will be referred to as *aggregated memory-aware algorithm*.

3.2.1. A motivating example. We use the same example as before (see Figure 3(a)), with the same memory constraint ($M_0 = 160$ MB). As shown in the previous section, the memory-aware mapping applies a local tree serialization mapping to assign processes to the four children of the root node. We can however go one step further. Since the subtree rooted at a is the most constraining subtree, using all the processes at this subtree is a reasonable strategy, but we can try something else for the mapping of e, f, and k. A first try consists in mapping e, f, and k using a proportional mapping and in enforcing a serialization constraint that ensures that the subtrees rooted at e, f, and k start only after a is completed. This would assign 29 processes to e, 13 to f and 12 to k. This does not work with respect to the memory constraint: indeed, the remaining memory (of a single process) after a is completed is $M_e = M_0 - \frac{cb_a}{64} = 160 \text{ MB} - \frac{2 \text{ GB}}{64} = 129$ MB, but $\bar{S}_e = \frac{S_e}{29} = 172 \text{ MB} > M_e$. Once again, the idea is that one of the subtrees has to be left alone while some subtrees can be mapped using a proportional mapping. The configuration shown in Figure 6 is valid; in this setting, the subtrees rooted at e and f are mapped using a proportional mapping. A serialization constraint is set so that the subtrees rooted at e and f start only after a is completed, and another constraint is set so that the subtree rooted at k can start only when *both* the subtrees rooted at e and f are completed.

We also have to check that, taking into account what is stacked at e and f, the subtree rooted at k can be processed using 64 processes. A major difference

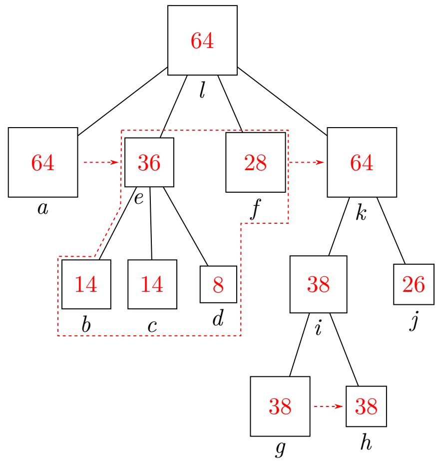


FIGURE 6. *Aggregated memory-aware mapping: the two subtrees rooted at e and f can be put in a group within which a proportional mapping is applied. The subtree rooted at k can only start when the subtree rooted at e and the subtree rooted at f, that are processed in parallel, are finished.*

compared to the flat memory-aware algorithm is that, in the flat case, whenever a tree serialization mapping is locally used, all the processes stack the same amount of data. This is not the case in the aggregated variant. In our example, the 36 processes working on the subtree rooted at e stack equal parts of the contribution block of e; similarly, the 28 processes working on the subtree rooted at f stack equal shares of the contribution block of node f. These two contribution blocks have the same size, but since they are distributed on different numbers of processes, processes stack different amounts of memory. This implies that the algorithm has to control the active memory of each process, while it is possible to rely on a global stack in the flat case. The algorithm is still able to ensure the memory constraint on every process, but it might yield more imbalance in memory because of the distribution of contribution blocks. This imbalance could prevent from processing a subtree even using all the processes used at its parent node. In this example, things work well; for the 36 processes working on the subtree rooted at e, the memory constraint (remaining memory) when mapping k is $M_0 - \frac{2\text{ GB}}{64} - \frac{1\text{ GB}}{36} = 101$ MB. For the 28 processes working on the subtree rooted at f, the memory constraint at k is $M_0 - \frac{2\text{ GB}}{64} - \frac{1\text{ GB}}{28} = 93$ MB. On every process, enough memory is available for processing the subtree rooted at k if the 64 processes are used on this subtree; indeed $\bar{S}_k = \frac{S_k}{64} = 47$ MB, which is smaller than the remaining memory on any process (whether it works on the subtree rooted at e or on the subtree rooted at f).

Note that, in this example, one can show that putting f and k together (instead of e and f) is also a valid strategy with respect to the memory constraint. In general, multiple partitions may be valid and we need a strategy for selecting a partition.

3.2.2. Heuristic. This section gives a formal presentation of how our aggregated variant of the memory-aware mapping is built. As for the flat memory-aware mapping, it consists in a top-down traversal of the tree; the only difference is the way a set of siblings is processed. In the flat memory-aware mapping, the whole set is mapped using a proportional mapping; then the memory constraint is checked for every subtree, and if the constraint cannot be ensured on at least one subtree, the whole set of siblings is reset to a tree serialization mapping. Our algorithm tries to detect groups of siblings within which a proportional mapping can be applied without violating the memory constraint, and taking into account the fact that groups are serialized which, as mentioned above, requires to track the active memory of every process. This problem could be formalized in terms of partitioning: how to partition the set of siblings so that there is a minimum number of parts and the memory constraint is ensured in each part? This partitioning problem is complex since the constraint to be checked on a part depends on the other parts, and its solution is out of the scope of this work. We propose, instead, the following scheme:

- We are given an order for traversing the list of siblings.
- Following this order, we form groups of nodes as large as possible.

By forming groups as large as possible, our aim is to decrease the number of serializations, and to obtain a mapping that is closer to a proportional mapping than with a flat memory-aware algorithm, while ensuring the memory constraint. In our experiments, the order we choose to guide our heuristic is the one corresponding to the memory-minimizing postorder for sequential executions (see Section 1.3), although any order could be used.

Following this strategy, we form a group using the following procedure. We denote by i the i^{th} node in a set of sorted siblings, and assume that the previous nodes have been processed, i.e., have been given a number of processes. First, we need to check that i can at least be a singleton, i.e., it can be alone in a group. Indeed, it could happen that, because unbalanced shares of the contribution blocks of the previous siblings have been stacked, the memory constraint cannot be ensured on at least one process, even if i forms a group by itself. In that case, we reset all the previous siblings to a tree serialization mapping; this will enforce a good balance of the contribution blocks of this set of siblings among the processes. Then, starting from i, we add nodes to the group until the memory constraint can no longer be satisfied. We repeat this process until the whole set of siblings has been partitioned. Finally, the mapping is computed (a proportional mapping is applied within each group) and the scheduling constraints are set (each group has to wait for the previous one to finish before it starts).

In the following we formalize the constraints to be ensured within each group. We consider the root r of a given subtree τ_r, mapped on p_r processes, and its ordered children. Let us assume that we are building the g-th group of children of r, denoted by $\mathcal{V}_g$, and that $g-1$ groups $\mathcal{V}_1, \dots, \mathcal{V}_{g-1}$ have been built already. We denote by j a node of the $g-1$ first groups and by i a node of the group being built. For a given child node i (or j), p_i is the number of processes i is mapped on, m_i is the size (memory) of the frontal matrix associated with i, and cb_i is the size (memory) of the contribution block of i. Finally, we keep track of the active memory of every process in an array $PSTACK$. $PSTACK(i,p)$ is the size of the contribution blocks that are stacked on process p before the factorization enters in the subtree rooted at i; this corresponds to the nodes in the stacked set of i that are mapped on p. The constraints we ensure when forming groups are the following:

- The nodes within a group can be mapped using a proportional mapping without violating the memory constraint:

  (Cstk): $\forall i \in \mathcal{V}_g, \forall p$ among the p_i processes working on i,

$$\bar{S}_i \leqslant M_0 - PSTACK(r,p) - \sum_{k=1}^{g-1} \left(\frac{cb_j}{p_j} \ , \ \ j \in \mathcal{V}_k \ ; \ j \text{ mapped on } p \right)$$

- The assembly of the parent node can be done without violating the memory constraint[2]:

  (Casm): $\forall i \in \mathcal{V}_g, \forall p$ among the p_i processes working on i,

$$\frac{cb_i}{p_i} + \frac{m_r}{p_r} \leqslant M_0 - PSTACK(r,p) - \sum_{k=1}^{g-1} \left(\frac{cb_j}{p_j} \ , \ \ j \in \mathcal{V}_k \ ; \ j \text{ mapped on } p \right)$$

The last term in both formulas, $\frac{cb_j}{p_j}$, is justified by the fact that within each group a strict tree partitioning mapping is applied. As a result, process p works only on one branch of each group $\mathcal{V}_k$. Note that the right-hand side of both formulas can also be written as $M_0 - PSTACK(i,p)$.

Algorithm 1 summarizes our grouping strategy.

Algorithm 1 Computation of groups of siblings.

// **Input:** a set of children of a node r
// a given order used to traverse the list of nodes (*order*)
while not all the children have been collected (following *order*) **do**
  i= current child node
  if i cannot be processed alone without violating (Cstk) or (Casm) **then**
    Reset previous siblings to a tree serialization mapping
  end if
  Starting from i: collect as many nodes as possible as long as both (Cstk) and (Casm) can be ensured
  Use a proportional mapping on the group, serialize with the previous ones
end while

4. Experiments. We implemented the proposed algorithms within the MUMPS (MUltifrontal Massively Parallel Solver) software package [3, 4], a general-purpose distributed-memory sparse direct solver that implements the multifrontal method.

In this section, we assess different mapping strategies on the set of matrices described in Table 3. The largest problem (Geoazur_192) is processed using 256 MPI processes while the other matrices are processed using 64 MPI processes. The Geoazur_192 matrix [18] is unsymmetric and complex and corresponds to a 27-point stencil discretization of a 3D visco-acoustic wave propagation model on a grid of size $192 \times 192 \times 192$. All the matrices are ordered using MeTiS. The experiments were carried out using a Bullx DLC B710 system at the Centre Interuniversitaire de Calcul de Toulouse (CICT). Each node of the machine has two ten-core 2.8 GHz Intel Xeon E5-2680v2 processors and 64 GB of main memory. For the Geoazur_192 problem, we used 16 nodes of the system, and we used 4 nodes for the other matrices (i.e., we used 16 MPI processes per node). In Table 3, we present the characteristics of the matrices and also statistics for the amount of active memory used by the MUMPS solver, version 5.0.0, for the factorization. The mapping strategy used in MUMPS is described in Amestoy *et al.* [4]: at the top of the tree, a relaxed proportional mapping is used whereas on lower layers of the tree, a strategy that aims at balancing memory requirements and workloads is used. The results reported in the table show that the

[2]This condition is not new and is in practice also integrated in the flat memory-aware mapping, but we had not mentioned it before for simplicity.

memory efficiency of MUMPS, although slightly better than with a strict proportional mapping, is quite low: e_{max} lies between 0.07 and 0.32 (the average is 0.19).

Matrix name	Order N	Entries ($\times 10^6$)	Factors (GB)	S_{seq} (GB)	S_{max} (MB)	e_{max}	S_{avg} (MB)	e_{avg}	Time (s)	Description; origin
cage13	445,315	7.5	30.7	21.4	1050	0.32	709	0.47	385.1	Directed weighted graph; Utrecht Univ.
pancake2_3	1,004,060	49.1	39.8	10.5	832	0.21	481	0.36	278.0	3D electromagnetism; Padova Univ.
as-Skitter	1,696,415	23.9	17.7	3.8	566	0.11	197	0.30	171.1	Internet topology graph; SNAP
HV15R	2,017,169	283.1	366.4	88.6	10638	0.13	4883	0.28	N/A	CFD, 3D engine fan; FLUOREM
MORANSYS1	2,734,008	81.3	63.5	17.9	998	0.28	715	0.39	390.3	Model Order Reduction; CADFEM
meca_raff6	3,269,763	130.2	63.5	6.6	1393	0.07	859	0.12	335.3	Thermo-mechanical coupling; EDF
Geoazur_192	7,077,888	189.1	251.9	65.9	1151	0.22	827	0.31	1308.2	3D Geophysics; Seiscope consortium

Table 3

Set of matrices used for the experiments; active memory (S_{max} and S_{avg}) and run time for the factorization using MUMPS (with $p = 64$ except for Geoazur_192 for which $p = 256$). For matrices as-Skitter and meca_raff6, symmetric, an LDL^t decomposition is computed; for the other matrices (unsymmetric) an LU decomposition is computed. Matrices pancake_2 and Geoazur_192 use single precision, complex, arithmetic; the other matrices use double precision, real, arithmetic.

First, we assess the behavior of the following strategies for matrix Geoazur_192 in Table 4:

- A plain, memory-based, proportional mapping;
- A flat memory-aware mapping, with different constraints M_0;
- An aggregated memory-aware mapping, with different constraints M_0;
- A tree serialization mapping on the whole tree, where all processes work at every node (except in case of frontal matrices with less rows than processes).

Mapping	M_0	S_{max} (MB)	e_{max}	S_{avg} (MB)	e_{avg}	Time (s)
Proportional		4417	0.06	1852	0.14	1465
Memory-aware	1288	940	0.27	672	0.38	1369
Aggregated memory-aware	1288	875	0.29	676	0.38	1381
Memory-aware	644	478	0.54	364	0.71	2061
Aggregated memory-aware	644	399	0.65	390	0.66	1961
Memory-aware	429	453	0.57	294	0.87	2695
Aggregated memory-aware	429	392	0.66	289	0.89	2301
Memory-aware	322	292	0.88	258	0.99	3141
Aggregated memory-aware	322	279	0.92	258	0.99	2799
Memory-aware	258	259	0.99	258	1.00	3765
Aggregated memory-aware	258	259	0.99	258	1.00	3370
Tree serialization		260	0.99	258	1.00	9070

Table 4

Experiments with the Geoazur_192 matrix. For the memory-aware mapping, the imposed memory bounds of 1288, 644, 429, 322, *and* 258 *MB correspond, respectively, to a memory efficiency of* 0.2, 0.4, 0.6, 0.8 *and* 1.0.

As expected, the tree serialization approach delivers a near-perfect memory scalability ($e_{max} = 0.99$ and $e_{avg} = 1.00$). However, the run time is over six times higher than the one obtained with the plain proportional mapping strategy. This is due to the prohibitive amount of communications generated by this mapping. For the Geoazur_192 problem, and using 256 MPI processes, the volume of communication with the tree serialization mapping is roughly nine times the volume generated by the proportional mapping strategy, and the number of messages is over 30 times larger than the number of messages with the default mapping. We also experimented with different numbers of processes and observed that the performance of the tree serialization strategy degrades significantly when the number of processes increases, which is due to the large number of messages and the small granularity of tasks. These observations show that the tree serialization mapping is in general prohibitive in practice, especially for large number of processes.

For matrix Geoazur_192, we also report results with four values of M_0 to illustrate how the memory-aware algorithm can be used. The four values that we use correspond to constraining the mapping such that $e_{max} \geqslant 0.2$, $e_{max} \geqslant 0.4$, $e_{max} \geqslant 0.6$, $e_{max} \geqslant 0.8$, and $e_{max} \approx 1.00$ respectively. Note that value 0.2 is similar to the value of e_{max} obtained using the default mapping in MUMPS (see Table 3), while the others are significantly higher. Thanks to the memory-aware mapping, the memory constraint is respected and performance is interesting since the run time remains comparable to our references (proportional mapping and default mapping in MUMPS 5.0.0) which need significantly more memory. On this matrix, we also observe that the larger the size of the memory is, the lower the run time is. This makes sense although it is not guaranteed by our memory-based mapping heuristics. It is also interesting to see that, with $M_0 = 258$ MB, we indeed reach a near-perfect memory scalability, but with a much better performance than the one obtained with the tree serialization strategy.

When groups are added to our memory-aware algorithm, we generally observe an improvement in the run time. This strategy exploits more tree parallelism but enforces the same memory constraints as the baseline approach. Aggregated memory-aware mapping is thus the most robust approach and will be used in the rest of this experimental section.

We visually assess the behavior of the different strategies for $e_{max} \geqslant 0.4$ and $e_{max} \geqslant 0.6$ in Figure 7. In this figure, we show how a given strategy behaves compared to a strict proportional mapping, as a function of the depth in the tree. The convention is that the root node is at depth 0, its children at depth 1, etc. There is a marker for every strategy, and the ordinate of a point is the ratio between:

1. the average number of processes given to the nodes lying at a given depth in the tree and
2. the average number of processes given by a strict proportional mapping to the nodes lying at the same depth in the tree.

At the bottom of the tree (here depth 12 and below), we notice that the ratios are close to 1, which means that all strategies behave like a proportional mapping. At the top of the tree, the ratios are greater than 1, which means that, on average, nodes are mapped onto more processes than if a proportional mapping were used. For example, for nodes at depth 2, a memory-aware mapping with $M_0 = 322$ MB assigns 4 times more processes on average than a proportional mapping. This means that, at that level, there is less tree parallelism (serialization was used), and the average task granularity decreases. We notice that when M_0 increases, the memory-aware mapping tends to behave more like a proportional mapping. Finally, for a given memory constraint, the aggregated variant ("w/ groups" in Figure 7) allows the memory-aware mapping to be closer to a proportional mapping while maintaining the same memory constraint.

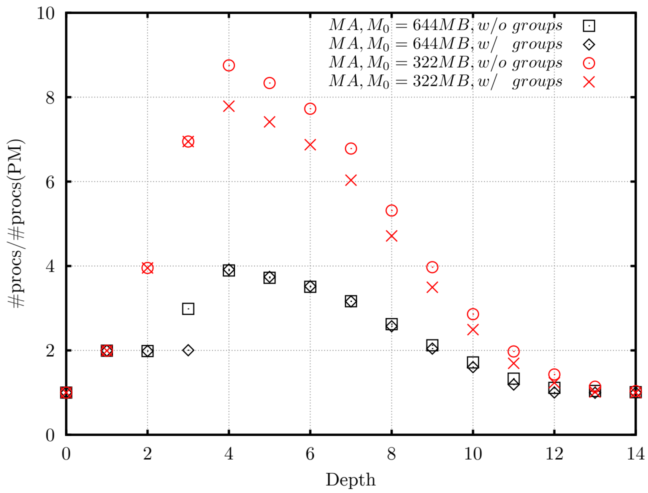


FIGURE 7. *Ratio of the average number of processes with a given mapping strategy and with a strict memory-based proportional mapping, as a function of the depth in the tree (matrix Geoazur_192).*

Matrix	Mapping	S_{max}	e_{max}	S_{avg}	e_{avg}	Time (s)
cage13	PM	912	0.37	737	0.45	406
	MA $e = 0.4$	639	0.52	516	0.66	381
	MA $e = 0.8$	360	0.93	338	0.99	372
pancake2_3	PM	1723	0.10	619	0.27	425
	MA $e = 0.4$	324	0.51	257	0.63	538
	MA $e = 0.8$	177	0.93	176	0.93	560
as-Skitter	PM	556	0.11	190	0.31	144
	MA $e = 0.4$	142	0.42	76	0.78	168
	MA $e = 0.8$	72	0.83	61	0.98	232
HV15R	PM	23624	0.07	10126	0.15	N/A
	MA $e = 0.4$	2778	0.50	1855	0.75	4718
	MA $e = 0.8$	1407	0.98	1390	0.99	4511
MORANSYS1	PM	1733	0.16	939	0.30	320
	MA $e = 0.4$	695	0.40	477	0.59	392
	MA $e = 0.8$	322	0.87	285	0.98	475
meca_raff6	PM	2951	0.04	1741	0.06	305
	MA $e = 0.4$	226	0.46	128	0.81	433
	MA $e = 0.8$	124	0.84	103	1.00	514

TABLE 5

Comparison of the memory-based proportional mapping with the aggregated memory-aware algorithm for two target efficiencies.

In Table 5 we report results on a large class of matrices; we compare a proportional mapping strategy and the aggregated memory-aware algorithm. We observe that using a memory-aware strategy that targets $e_{max} = 0.8$, we are able to decrease the memory peak by factors between 2.5 (matrix cage13) and 23.8 (matrix meca_raff6). This comes at the price of a moderate increase in run time for most problems. The worst case is meca_raff6 for which the increase in run time is about 70%. For matrix cage13, the memory-aware mapping actually delivers slightly better performance with $e_{max} = 0.8$ than with both $e_{max} = 0.4$ and proportional mapping. Although counter-intuitive, it may happen that smaller task granularities yield better time performance, especially since the proportional mapping heuristic is designed to balance memory rather than optimize time. Matrix HV15R is particularly interesting because the factorization cannot complete when proportional mapping (or the default mapping in MUMPS 5.0.0) is used. Indeed, we use 16 MPI processes per node and the average memory peak (estimated during the analysis phase) is 23.6 GB per MPI process with the default strategy, while each node of our system only has 64 GB of memory.

Overall, the memory-aware mapping exhibits very interesting results compared to the default strategy in MUMPS, since we are able to significantly decrease the memory footprint without dramatically decreasing performance. For all matrices, we have decreased the maximum memory peak by an important factor that is increasing with our target for memory efficiency e_{max}. The penalty in run time also depends on the target for memory efficiency and is typically between 40% and 60% on 64 processes with respect to the performance of the proportional mapping strategy. When comparing with the default mapping used in MUMPS 5.0.0 (Table 3), we observe similar results. Although the mapping strategy of MUMPS typically yields lower memory consumption than a proportional mapping, using a memory-aware algorithm can significantly reduce memory usage, at the price of a moderate penalty in factorization time.

5. Related work. In this section, we mention some related problems and possible directions for future work.

5.1. The tree pebble game. We mention an interesting formulation of our problem. A pebble game is a game played on an acyclic graph (here we consider only trees). Every node in the graph has a weight τ_i. Nodes can carry pebbles; if a node i carries τ_i pebbles, it is said to be *satisfied*. The rules of the game are the following:

- Initially no vertices carry pebbles.
- A pebble may be placed on a input vertex (for us, a leaf of the tree) at any time.
- If all the predecessors (for us, the children) of an internal vertex are satisfied, a new pebble may be placed on that vertex,
- A pebble can be removed from the graph at any time.

The goal is to place pebbles on output vertices (for us, the root node); this yields a topological traversal of the tree. The objective can be to minimize the number of pebbles that are used (the *pebble cost*) or the number of time steps (turns). This game has applications in the VLSI community and in semantics [23]. Liu shows how to modify the elimination tree to obtain a tree pebble game that represents the memory usage of a serial sparse factorization [16]. He provides an algorithm that minimizes the number of pebbles, i.e. that finds a topological traversal of the tree that minimizes memory requirements of the multifrontal method. It finds the same traversal as the more general (and more recent) algorithm by Jacquelin et al. [13].

In our parallel setting, the tree pebble game formulation might be generalized by using pebbles with different colors, with a color for each process. A pebble would thus represent a unit of memory *for a given process*. We are interested in finding a traversal of the tree that maximizes tree parallelism under a given memory constraint. Our memory constraints can be modeled with a limit on the number of pebbles of each color, whereas the number of turns can serve as a representation of the time for traversing the tree. Rules would need to be added in order to have a realistic model of a parallel execution. For example, several pebbles of the same color cannot be placed on different nodes at the same time. This represents the fact that a process cannot work on different tasks at the same time. Rules also need to be added to penalize some undesirable configurations; for example, intra-node communication can be penalized by setting a penalty in the number of turns whenever too many colors are used at a given node. We have not pushed this formulation further.

5.2. Scheduling algorithms. We presented a task mapping algorithm along with a constrained scheduling strategy. This strategy is not dynamic, which implies that it can not adapt on the fly to workload or memory imbalances that might occur during the factorization. We recall that in a multifrontal factorization with *delayed pivoting*, the tree can be dynamically modified since a pivot can be delayed and transferred from a node to its parent, perhaps several times. This implies variations in workload and memory usage that cannot be forecast before the factorization. This highlights the need for memory-aware dynamic scheduling strategies. The dynamic scheduling in MUMPS is already able to balance memory loads on the fly, by using global information messages [4]; using this scheduling strategy could be considered but would not provide a formal guarantee that memory constraints are ensured. Another option is the use of *deadlock avoidance* algorithms [22, 25]. In these algorithms, a critical resource (such as memory) is granted to a requesting process only if there is a guarantee that all the processes still have a way to complete their tasks. This is a direction for future work.

6. Conclusion. We demonstrated that there is a need for mapping and scheduling algorithms that are able to limit the active memory of the multifrontal factorization in a parallel setting. Indeed, we showed both theoretically and experimentally that commonly-used mapping strategies, such as the proportional mapping, can dangerously let the memory usage grow with the number of processes.

We proposed a *memory-aware* mapping method that aims at maximizing parallelism granularity (and tree parallelism) under a given memory constraint; it consists in using a proportional mapping whenever it is possible (with respect to the constraint) and in serializing a set of sibling subtrees whenever a proportional mapping cannot be used without violating the memory constraint. We also suggested a variant that improves parallelism by enforcing serializations only between groups of siblings, each group being mapped with a proportional mapping. Although we focused on a memory-based proportional mapping in our experiments, any other tree partitioning mapping (e.g., a memory-minimizing tree partitioning mapping, or a workload-based proportional mapping ensuring a good balance of the work on the processes), can be used in combination with memory-aware subtree serializations; this could help reducing the number of serializations and/or improving performance.

Our experimental results demonstrate that the memory-aware mapping is able to ensure that a given memory constraint is respected. In general, the looser the memory constraint is, the better performance is, since the memory-aware mapping tends to behave more like a proportional mapping. We also showed that the aggregated variant is closer to a proportional mapping while still ensuring memory constraints. Overall, the results are interesting since, compared to both proportional mapping and the strategy described in [4], we are able to decrease the maximum memory peak by a large amount at the cost of a very reasonable increase in the run time.

A very attractive feature of the proposed algorithms is that they allow users to adapt the memory usage of the solver to the memory available for their application; indeed, setting M_0 to the available memory, the solver will limit the amount of parallelism of the tree that is exploited to reach this objective. When the memory given gets close to the minimum (S_{seq}/p memory available on each process) then performance might strongly degrade. In cases where more memory is available, M_0 can be loosened and the performance of the solver will in general improve. We believe that this feature of the proposed memory-aware algorithms significantly improves the robustness of multifrontal solvers in a parallel environment.

7. Acknowledgments. This work was granted access to the HPC resources of CALMIP under the allocation 2014-P0989. We are grateful to all the reviewers and, especially, to Cleve Ashcraft who gave us many insightful comments and hints to help us improve the quality of this document.

REFERENCES

[1] E. Agullo, *On the out-of-core factorization of large sparse matrices*, PhD thesis, École Normale Supérieure de Lyon, Nov. 2008.

[2] E. Agullo, A. Guermouche, and J.-Y. L'Excellent, *Reducing the I/O volume in sparse out-of-core multifrontal methods*, SIAM Journal on Scientific Computing, 31 (2010), pp. 4774–4794.

[3] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent, *A fully asynchronous multifrontal solver using distributed dynamic scheduling*, SIAM Journal on Matrix Analysis and Applications, 23 (2001), pp. 15–41.

[4] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet, *Hybrid scheduling for the parallel solution of linear systems*, Parallel computing, 32 (2006), pp. 136–156.

[5] O. Beaumont and A. Guermouche, *Task scheduling for parallel multifrontal methods*, Euro-Par 2007 Parallel Processing, (2007), pp. 758–766.

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

[7] I. S. Duff and J. K. Reid, *The multifrontal solution of unsymmetric sets of linear systems*, SIAM Journal on Scientific and Statistical Computing, 5 (1984), pp. 633–641.

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

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

[10] A. George, J. W. H. Liu, and E. Ng, *Communication results for parallel sparse Cholesky factorization on a hypercube*, Parallel Computing, 10 (1989), pp. 287–298.

[11] A. Guermouche and J. Y. L'excellent, *Constructing memory-minimizing schedules for multifrontal methods*, ACM Transactions on Mathematical Software (TOMS), 32 (2006), pp. 17–32.

[12] A. Guermouche, J.-Y. LExcellent, and G. Utard, *Impact of reordering on the memory of a multifrontal solver*, Parallel Computing, 29 (2003), pp. 1191 – 1218, http://dx.doi.org/http://dx.doi.org/10.1016/S0167-8191(03)00099-1, http://www.sciencedirect.com/science/article/pii/S0167819103000991. Parallel Matrix Algorithms and Applications.

[13] M. Jacquelin, L. Marchal, Y. Robert, and B. Uçar, *On optimal tree traversals for sparse matrix factorization*, in Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, IEEE Computer Society, 2011, pp. 556–567.

[14] J.-Y. L'Excellent, *Multifrontal methods for large sparse systems of linear equations: parallelism, memory usage, performance optimization and numerical issues*, habilitation à diriger des recherches, École Normale Supérieure de Lyon, 2012. http://tel.archives-ouvertes.fr/tel-00737751.

[15] J. W. H. Liu, *On the storage requirement in the out-of-core multifrontal method for sparse factorization*, ACM Transactions on Mathematical Software (TOMS), 12 (1986), pp. 249–264.

[16] J. W. H. Liu, *An application of generalized tree pebbling to sparse matrix factorization*, SIAM Journal on Algebraic and Discrete Methods, 8 (1987), pp. 375–395.

[17] J. W. H. Liu, *The role of elimination trees in sparse factorization*, SIAM journal on matrix analysis and applications, 11 (1990), pp. 134–172.

[18] S. Operto, J. Virieux, P. R. Amestoy, J.-Y. L'Excellent, L. Giraud, and H. Ben Hadj Ali, *3D finite-difference frequency-domain modeling of visco-acoustic wave propagation using a massively parallel direct solver: A feasibility study*, Geophysics, 72 (2007), pp. 195–211, http://dx.doi.org/10.1190/1.2759835, http://link.aip.org/link/?GPY/72/SM195/1.

[19] A. Pothen and C. Sun, *A mapping algorithm for parallel sparse cholesky factorization*, SIAM Journal on Scientific Computing, 14 (1993), pp. 1253–1253.

[20] G. N. Prasanna and B. R. Musicus, *Generalized multiprocessor scheduling and applications to matrix computations*, IEEE Transactions on Parallel and Distributed Systems, 7 (1996), pp. 650–664.

[21] F.-H. Rouet, *Memory and performance issues in parallel multifrontal factorizations and triangular solutions with sparse right-hand sides*, PhD thesis, Institut National Polytechnique de Toulouse, 2012.

[22] C. Sánchez, H. B. Sipma, Z. Manna, and C. D. Gill, *Efficient distributed deadlock avoidance with liveness guarantees*, in Proceedings of the 6th ACM & IEEE International conference on Embedded software, ACM, 2006, p. 20.

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

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

[25] W. M. Sid-Lakhdar, *Scaling multifrontal methods for the solution of large sparse linear systems on hybrid shared-distributed memory architectures*, Ph.D. dissertation, ENS Lyon, Dec. 2014.

[26] S. Wang, X. S. Li, J. Xia, Y. Situ, and M. V. De Hoop, *Efficient scalable algorithms for solving dense linear systems with hierarchically semiseparable structures*, SIAM Journal on Scientific Computing, 35 (2013), pp. C519–C544.