Optimal memory minimization algorithms for the multifrontal method

Abdou Guermouche, Jean-Yves l'Excellent

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Optimal memory minimization algorithms for the multifrontal method

Abdou Guermouche (ENS Lyon) — Jean-Yves L'Excellent (INRIA)

N° 5179

May 2004

THÈME Num

ISSN 0249-6399 ISRN INRIA/RR--5179--FR+ENG

Optimal memory minimization algorithms for the multifrontal method

Abdou Guermouche (ENS Lyon), Jean-Yves L'Excellent (INRIA)

Thème Num — Grilles et calcul haute-performance
Projet GRAAL

Rapport de recherche n° 5179 — May 2004 — 31 pages

Abstract: We are interested in the active and total memory usage of the multifrontal method. Starting from the algorithms proposed by Liu, we suggest a new scheme together with a tree traversal that give an optimal peak of active memory. Significant gains are obtained compared to Liu's approach. We also study the problem of minimizing the total memory and compare various new schemes. A number of experiments shows the interest of these approaches.

Key-words: Liu's algorithm, multifrontal method, sparse matrices, memory, tree traversal.

This text is also available as a research report of the Laboratoire de l'Informatique du Parallélisme `http://www.ens-lyon.fr/LIP`.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Algorithmes optimaux pour la minimisation de la mémoire dans la méthode multifrontale

Résumé : Nous nous intéressons à l'optimisation mémoire pour l'approche multifrontale. Repartant des algorithmes proposés par Liu, nous proposons de nouveaux algorithmes et parcours d'arbre visant à minimiser la mémoire. Dans le cas out-of-core nous proposons un algorithme optimal pour la taille de la mémoire active, alors que pour le cadre in-core nous nous intéressons à la minimisation de la mémoire totale et proposons plusieurs nouvelles approches. Cette étude théorique est complétée par des expérimentations sur un grand nombre de problèmes tests qui montrent les améliorations obtenues.

Mots-clés : Algorithme de Liu, méthode multifrontale, matrices creuses, mémoire, parcours d'arbre

1 Introduction

The multifrontal method is an efficient direct method to solve sparse systems of linear equations, in which the dependencies between computations are given by a tree. In this approach two zones of memory can be distinguished. The first one corresponds to the computed factors, and the second one to the active memory, or stack memory. For an out-of-core code, the factors can be stored on disk after they are computed and it is then very important to minimize the peak of active memory (or stack memory). For an in-core approach, the minimization of the peak of total memory (factors and active memory) is more crucial.
In [12], Liu presents an algorithm to reduce the active memory usage of the multifrontal method. He raises some examples where modifying the multifrontal scheme can give better results than his algorithm. Starting from this observation, this paper presents some algorithms to optimize the memory usage of the multifrontal method both in the cases of active memory and total memory. In particular, a new scheme is proposed, where an anticipated activation of certain tasks together with a specific traversal of the dependency tree can significantly reduce the memory usage.
This paper is organized as follows. We first discuss in Section 2 the memory aspects of the multifrontal method and present algorithms related to tree traversals aiming at reducing the peak of memory. Those algorithms are either already existing or simple variants of existing algorithms. In Section 3, we present a modification of the multifrontal method when we are interested in the active memory usage. We give an algorithm that produces an optimal traversal of the tree in that case, together with an anticipated activation of some tasks. We apply the same type of ideas to the problem of minimizing the total memory in Section 4, and also obtain an optimal algorithm. In Section 5, we report on the gains obtained for the active memory and the total memory, when applying the new schemes. Section 6 studies a special case where some assembly operations are supposed to be done in-place. Finally, we discuss some more implementation-dependent issues related to the memory management in the multifrontal approach and conclude.

2 Memory aspects in the multifrontal method

2.1 The multifrontal method

In this section, our intention is to introduce the terms that will be used later in this paper rather than giving a complete description of the multifrontal method. The reader should (for example) refer to [7, 8] for further details of this technique.

In the multifrontal method, we deal with the solution of sparse systems of equations of the form $Ax = b$, where A is a large sparse matrix. In our case, an analysis phase based on the structure of $A + A^T$ is first applied that determines an appropriate ordering to preserve sparsity.
Like other direct solvers, the multifrontal method is based on an elimination tree [13], which is a transitive reduction of the matrix graph and is the smallest data structure representing dependencies between the operations. In practice, we use a structure called *assembly tree*, obtained by merging nodes of the elimination tree whose corresponding columns belong to the same supernode [5]. A supernode is a contiguous range of columns (in the factor matrix) having the same lower diagonal nonzero structure.

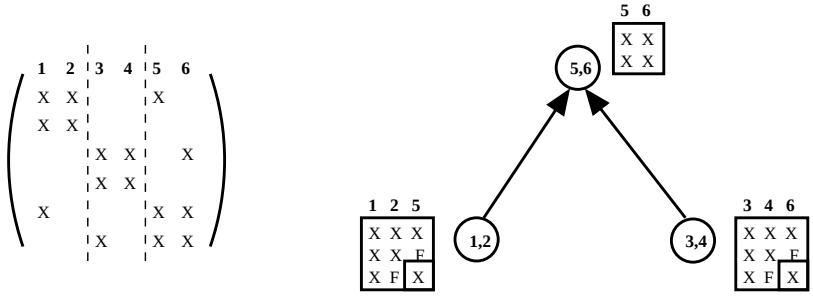


Figure 1: A matrix and the associated assembly tree.

The factorization of the matrix is then done by performing a succession of partial factorizations of small dense matrices called *frontal matrices*, which are associated to the nodes of the tree (as illustrated in Figure 1). The order of a frontal matrix is given by the number of non-zeros below the diagonal in the first column of the supernode associated with the tree node. Each frontal matrix is divided into two parts: the *factor* block, also called *fully summed* block, which corresponds to the part factorized during the elimination process; and the *contribution block* which corresponds to the square block updated when processing the frontal matrix. Once the partial factorization is done, the contribution block is passed to the parent node. When contributions from all children are available on the parent, they can be assembled (*i.e.* summed with the values contained in the frontal matrix of the parent). The elimination algorithm is a postorder traversal (we do not process parent nodes before their children) of the assembly tree [14]. It uses three areas of storage in a contiguous memory space, one for the factors, one to stack the contribution blocks,

and another one for the current frontal matrix [2]. During the tree traversal, the memory space required by the factors always grows while the stack memory (containing the contribution blocks) varies depending on the operations made: when the partial factorization of a frontal matrix is processed, a contribution block is stacked which increases the size of the stack; on the other hand, when the frontal matrix is formed and assembled, the contribution blocks of the children nodes are popped out of the stack and its size decreases. The stack memory is thus very dependent on the assembly tree topology.

2.2 Some notations

Given i a node in the tree, we define $factor_i$ to be the memory requirement for the factors of i, and cb_i to be the storage for the contribution block of i. $store_i = factor_i + cb_i$ represents the size of the complete frontal matrix of node i.
We also use $SubT_i$ to denote the subtree rooted at i, including i, and f_i to denote the amount of factors corresponding to $SubT_i$, that is:

$$f_i = \sum_{j \in SubT_i} factor_j$$

Let A_i (respectively T_i) be the storage for active memory (respectively active memory and factors) required to process the subtree $SubT_i$ rooted at i and n_i be the number of children of i. Those children are denoted by $(c_{i,j})_{j=1,\ldots,n_i}$.

2.3 Stack memory usage

Since the factors are not accessed again after they are computed, we can suppose that a basic out-of-core scheme would store them on disk, or that the system paging mechanism will put them to disk. In such cases, it is interesting to consider the stack memory peak.
In the multifrontal method, the parent node is usually activated after all its child subtrees have been processed and the storage required to assemble the contributions of the children into the parent node i is

$$store_i + \sum_{j=1}^{n_i} cb_i.$$

Furthermore, when processing a subtree rooted at a child $c_{i,j}$, the contribution blocks of all previously processed children have to be stored, leading to a storage equal to

$$A_{c_{i,j}} + \sum_{k=1}^{j-1} cb_{c_{i,k}}.$$

Therefore, the maximum storage required to process the complete subtree rooted at node i is given by:

$$A_i = \max(\max_{j=1,n_i}(A_{c_{i,j}} + \sum_{k=1}^{j-1} cb_{c_{i,k}}), store_i + \sum_{j=1}^{n_i} cb_{c_{i,j}}), \quad (1)$$

as seen in [10]. (Note that for leaf nodes with no children, $A_i = store_i$.)
Note that some implementations of the multifrontal algorithm allow the contribution of the last child to be assembled in-place into the parent node, in which case the contribution block from the last child is not counted in the last assembly step. In that case, Formula (1) becomes:

$$A'_i = \max(\max_{j=1,n_i}(A_{c_{i,j}} + \sum_{k=1}^{j-1} cb_{c_{i,k}}), store_i + \sum_{j=1}^{n_i-1} cb_{c_{i,j}}), \quad (2)$$

2.4 Total memory usage

For the in-core case, we are interested in the peak of total memory. Compared to the memory usage for the active (stack) memory, we take the memory used for the storage of the factors into account since they must be kept in memory after they are computed. Using the same notations as above, the amount of total memory needed to process a node i (and its subtree) is then given by:

$$T_i = \max(\max_{j=1,n_i}(T_{c_{i,j}} + \sum_{k=1}^{j-1}(cb_{c_{i,k}} + f_{c_{i,k}})), store_i + \sum_{j=1}^{n_i}(cb_{c_{i,j}} + f_{c_{i,j}})) \quad (3)$$

2.5 Algorithms to decrease the memory usage

Active memory usage

The tree traversal has a strong impact on the peak of memory. For example, a deep unbalanced tree will lead to a better memory usage if processed using a depth-first postorder traversal and the number of simultaneous contribution blocks will be smaller if the traversal starts from the deepest leaves. This is illustrated in Figure 2.

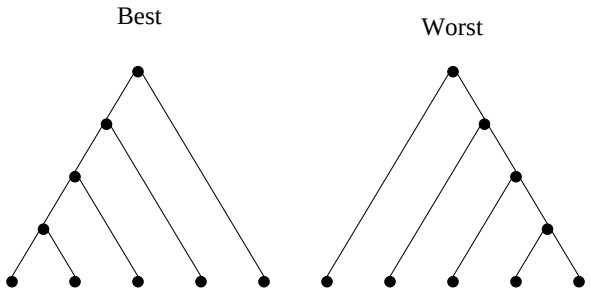


Figure 2: Importance of the tree traversal. The tree is processed using a depth-first postorder traversal starting from the left-hand-side.

In [12], Liu suggests an optimal tree traversal for elimination trees when the assembly of the last child can be done in-place. This algorithm can easily be generalized to assembly trees and consists, at each level of the tree, in sorting the children nodes in decreasing order of $\max(A_{c_{i,j}}, store_i) - cb_{c_{i,j}}$ and then use a depth-first postorder traversal of the reordered tree. This result is based on the theorem below, proved in [12], and that we will refer to as Liu's theorem in the other parts of this paper:
Liu's theorem. *Given a set of values* $(x_i, y_i), i = 1..n$, *the minimal value of* $max_{i=1,n} x_i + \sum_{j=1}^{i-1} y_i$ *is obtained by sorting the sequence* (x_i, y_i) *in decreasing order of* $x_i - y_i$, *that is,* $x_1 - y_1 \geq x_2 - y_2 \geq \ldots \geq x_n - y_n$.

Now if the assembly of the last child cannot be done in-place, [10] show that the children should then be sorted in decreasing order of $A_{c_{i,j}} - cb_{c_{i,j}}$. By doing so, the value of A_{root} as defined by (1) for the complete tree rooted at its root node is minimized. In the rest of this paper we will use the terms *variant of Liu's algorithm* to refer to this order.
However, especially for very wide trees (where $\sum_{j=1}^{j=n_i} cb_{c_{i,j}}$ may be large compared to $store_i$), A_{root} can still be large. For that reason, Liu also experiments in [12] a strategy consisting in pre-allocating the structure of the parent before children are formed. Each contribution block from each new child is then assembled directly into the structure of the parent, thus avoiding a potentially large collection of contribution blocks in stack memory. The storage obtained for this approach is then recursively defined by:

$$A_i = store_i + \max_{j=1,n_i} (A_{c_{i,j}}) \tag{4}$$

Again, it appeared that this strategy was also somewhat disappointing, because a chain of parent nodes (at each level of the tree) had to be stored, possibly also leading to a significant memory usage.
We should notice here that rather than pre-allocating the parent, the scheme from Liu can be slightly improved by allocating the parent just after the first child has been processed. Using the notations above, the memory usage is then recursively defined by:

$$A_i = \max(A_{c_{i,1}}, store_i + cb_{c_{i,1}}, store_i + \max_{j=2,n_i}(A_{c_{i,j}})) \quad (5)$$

if the assembly into the parent is not performed in-place, and

$$A_i = \max(A_{c_{i,1}}, store_i + \max_{j=2,n_i}(A_{c_{i,j}})) \quad (6)$$

if the assembly into the parent can be done in-place.
Note that both formulas (5) and (6) necessarily lead to a value of A_i smaller or equal to that of Formulas (4); furthermore they both provide an optimal value of the active memory A_i when the child with the largest peak is treated first (the order in the term $\max_{j=2,n_i}(A_{c_{i,j}} + store_i)$ has no influence and moving the first child with largest peak to that set can only increase the peak). In the following, these two strategies will be referred to as "*anticipated parent activation*" and "*anticipated parent activation, in-place*", respectively.

Total memory usage

Similarly to the case of active memory peak, sorting the children nodes in decreasing order of $T_{c_{i,j}} - (cb_{c_{i,j}} + f_{c_{i,j}})$ gives an optimal tree traversal in terms of minimization of the total memory peak defined by (3). This is explained in more details in [9], where experimental results show that this (optimal) order brings slight gains over the one from the previous paragraph ($A_{c_{i,j}} - cb_{c_{i,j}}$) when the total memory is considered. (If the graph of the matrix is not connected (reducible matrix), one should also order the root nodes corresponding to the different trees in decreasing order of their $T_i - f_i$, as this will decrease the total memory $\max_{i \in roots} T_i + \sum_{j=1}^{i-1} f_j$; this is a direct consequence of Liu's theorem cited earlier.) In the following, we call this algorithm *classical total memory minimization.*
Finally, in the case of the total memory it is also possible to activate the parent node after the first child, leading to the *anticipated parent activation* and *anticipated parent activation, in-place* variants, and a memory usage equal to

$$T_i = \max(T_{c_{i,1}}, f_{c_{i,1}} + cb_{c_{i,1}} + store_i, store_i + \max_{j=2,n_i}(T_{c_{i,j}} + \sum_{k=1}^{j-1} f_{c_{i,k}}))$$

if the assembly of the contribution from the first child into the parent is not done in-place, and

$$T_i = \max(T_{c_{i,1}}, f_{c_{i,1}} + store_i, store_i + \max_{j=2,n_i}(T_{c_{i,j}} + \sum_{k=1}^{j-1} f_{c_{i,k}}))$$

if the assembly of the contribution from the first child into the parent can be done in-place.
For this anticipated parent activation, both for the in-place and non in-place case, it makes sense to try to order the children nodes in decreasing order of their $T_{c_{i,j}} - f_{c_{i,j}}$ as this will at least minimize the term $\max_j(T_{c_{i,j}} + \sum_{k=1}^{j-1} f_{c_{i,k}})$.

3 Case of the active memory: optimal tree management

As explained in the previous section, neither the systematic pre-allocation of the parent node, nor its allocation after the last child (classical multifrontal method) provide an optimal memory usage. However it is possible to activate the parent node at an arbitrary position, after a first set of child nodes has been treated.

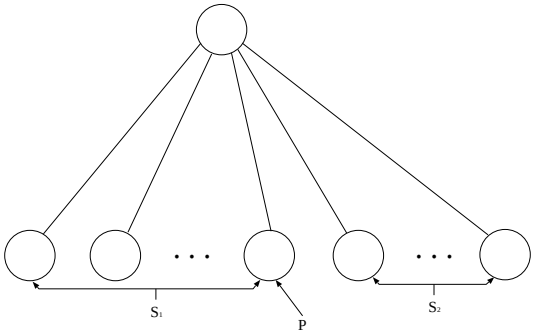


Figure 3: Example of a parent node and its children.

Given a parent node and its set of children nodes in the assembly tree, we use the following notations: supposing that the parent node is activated (allocated in

memory) just after child j has been processed, we define $p = j$. We also define $\mathcal{S}_1$ as the set of children nodes treated before the activation of the parent node and $\mathcal{S}_2$ as the set of children nodes treated after the activation of the parent node. So we consider a parent node i and its children $(c_{i,j})_{j=1,\ldots,p} = \mathcal{S}_1$ and $(c_{i,j})_{j=p+1,\ldots,n_i} = \mathcal{S}_2$ as shown in Figure 3. The storage needed to process a child node $c_{i,j}$ is $A_{c_{i,j}}$. Before the allocation of the parent node, the peak of storage is obtained by applying Formula (1) (Section 2.3) to the children nodes of $\mathcal{S}_1$. Furthermore, the amount of memory needed to process the children nodes of $\mathcal{S}_2$ is obtained by applying Formula (5) to those children (assuming that the assemblies into the parent are not done in-place). Thus, the amount of memory needed to process the subtree rooted at i is:

$$A_i = \max(\max_{j=1,p}(A_{c_{i,j}} + \sum_{k=1}^{j-1} cb_{c_{i,k}}), \\ store_i + \sum_{j=1}^{p} cb_{c_{i,j}}, \\ store_i + \max_{j=p+1,n_i}(A_{c_{i,j}})) \tag{7}$$

Lemma 1. *The order of the children nodes in $\mathcal{S}_2$ does not change the peak of active memory A_i.*

Proof. Obvious since $store_i + \max_{j=p+1,n_i}(A_{c_{i,j}}))$ does not depend on the order of the children in $\mathcal{S}_2$. □

Lemma 2. *Let j be a child node belonging to $\mathcal{S}_1$. If A_j is smaller than $\max_{i \in \mathcal{S}_2}(A_i)$, then j can be moved to $\mathcal{S}_2$ without increasing the peak.*

Proof. Removing a node from $\mathcal{S}_1$ does not increase the peak in this set. Furthermore since A_j is smaller than $\max_{i \in \mathcal{S}_2}(A_i)$, the peak in $\mathcal{S}_2$ will not increase. □

Theorem 1. *Let i be a node and let $c_{i,1} \ldots c_{i,n_i}$ be its children nodes. An optimal active memory peak is obtained by applying the following algorithm:*

for all $j = 1, \ldots, n_i$ **do**
Start with the children ordered in decreasing order of $A_{c_{i,k}}$;
Define $\mathcal{S}_1 = (c_{i,k})_{k=1,\ldots,j}$ and apply the variant of Liu's algorithm to reorder $\mathcal{S}_1$ (decreasing $A_{c_{i,k}} - cb_{c_{i,k}}$;)

Define $\mathcal{S}_2 = (c_{i,k})_{k=j+1,\ldots,n_i}$;
Compute the value of A_i (formula (7));
if A_i is smaller than any A_i previously obtained **then**
Set $p = j$ and keep the corresponding order of children nodes;
end if
end for

Proof. Let σ be an optimal order of the children of i where i is activated after the p^{th} child has been processed (in the order given by σ). The permutation σ' obtained by sorting all the child nodes j belonging to $\mathcal{S}_2$ in descending order of their respective A_j is still optimal (Lemma 1). If $\exists k \in \mathcal{S}_1$ so that $A_k \leq \max_{j \in \mathcal{S}_2}(A_j)$, the permutation σ'' obtained by moving k to $\mathcal{S}_2$ is still optimal (Lemma 2). Thus, there exists an optimal permutation σ^n such that $\min_{k \in \mathcal{S}_1}(A_k) > \max_{k \in \mathcal{S}_2}(A_k)$ obtained by repeating the previous operation.
As a consequence, an optimal permutation of the children nodes can be computed by sorting them in descending order of their respective A_j and determine the best position to activate the parent p. This can be done by trying all the possible positions for the activation of the parent and selecting the minimal one. Note that for each iteration we recompute the best order on $\mathcal{S}_1$. The determination of the best position for the activation of the parent is then done in a maximum of n_i steps. □

We should remark here that from an implementation point of view, it is not necessary to sort the nodes in $\mathcal{S}_1$ at each iteration. In fact, the following algorithm can be applied, where a pre-computed permutation allows to select efficiently the nodes with smallest $A_{c_{i,k}}$ in $\mathcal{S}_1$:

Set $\mathcal{S}_1 = (c_{i,k})_{k=1,\ldots,n_i}$, $\mathcal{S}_2 = \emptyset$ and $p = n_i$;
Sort $\mathcal{S}_1$ in decreasing order of $A_{c_{i,k}} - cb_{c_{i,k}}$ and compute A_i using Formula (7);
repeat
Find $c_{i,j}$ such that $A_{c_{i,j}} = \min_{c_{i,k} \in \mathcal{S}_1} A_{c_{i,k}}$;
Set $\mathcal{S}_1 = \mathcal{S}_1 \setminus c_{i,j}$, $\mathcal{S}_2 = \mathcal{S}_2 \cup c_{i,j}$, and $p = p - 1$;
Compute A'_i;
if $A'_i \leq A_i$ **then**
Keep the value of p, $\mathcal{S}_1$ and $\mathcal{S}_2$ and set $A_i = A'_i$;
end if
until $p == 1$ or $A'_i > A_i$

The termination condition $A'_i > A_i$ is explained by the fact that when the global peak increases, this is necessarily because of a term corresponding to $\mathcal{S}_2$. Since elements

will only be added to $\mathcal{S}_2$, the peak can only increase and it is not worth trying to add more elements to $\mathcal{S}_2$.
Finally, applying the previous theorem on the complete tree leads to Algorithm 1.

Algorithm 1 Optimal tree reordering to minimize the peak of stack memory.

Tree_Reorder (T)**:**
  Begin
    for all i in the set of root nodes **do**
      Process_Child(i);
    end for
  End
Process_Child(i)**:**
  Begin
    if i is a leaf **then**
      A_i=$store_i$;
    else
      for $j = 1$ to n_i **do**
        Process_Child($c_{i,j}$);
      end for
      Determine the position p where the parent should be activated and the order of children using Theorem 1;
      Compute A_i using Formula (7);
    end if
  End

4 Total memory optimization

In this section we are interested in the minimization of the total memory, where both the stack and the factors are taken into account. In the classical multifrontal method, the total memory is given by (3), applied to the root node of the tree and it is possible to determine an optimal tree traversal in that context. However, similarly to what has been done in Section 3 for the stack (active) memory, we can decide to activate the parent node at an arbitrary position, before all children have been processed.
Let i be a node in the assembly tree. We use the same definition as before for the sets $\mathcal{S}_1$, $\mathcal{S}_2$ and p. The peak of storage for $\mathcal{S}_1$, including the allocation of the parent

node, is obtained by applying Formula (3) to the children nodes belonging to this set:

$$\begin{aligned}\mathcal{P}_1 = \max(\max_{j=1,p}(T_{c_{i,j}} + \sum_{k=1}^{j-1}(cb_{c_{i,k}} + f_{c_{i,k}})),\\ store_i + \sum_{j=1}^{p}(cb_{c_{i,j}} + f_{c_{i,j}}))\end{aligned} \tag{8}$$

Furthermore, the amount of memory needed to process the children nodes of $\mathcal{S}_2$ is:

$$\mathcal{P}_2 = store_i + \sum_{j=1}^{p} f_{c_{i,j}} + \max_{j=p+1,n_i}(T_{c_{i,j}} + \sum_{k=p+1}^{j-1} f_{c_{i,k}}) \tag{9}$$

Indeed, when treating a node, the memory will contain the factors corresponding to all already processed brothers. In the formulas above, note also that $T_{c_{i,j}}$ includes $f_{c_{i,j}}$ so that the factors for the last child are effectively taken into account.
Finally, the amount of memory needed to process the subtree rooted at the parent node i is:

$$T_i = \max(\mathcal{P}_1, \mathcal{P}_2) \tag{10}$$

Lemma 3. *Suppose that the position p to activate the parent, the set of children nodes in $\mathcal{S}_1$ and the set of children nodes in $\mathcal{S}_2$ are given. Then, sorting the children nodes in $\mathcal{S}_1$ in decreasing order of $T_{c_{i,j}} - (cb_{c_{i,j}} + f_{c_{i,j}})$ and the children nodes in $\mathcal{S}_2$ in decreasing order of $T_{c_{i,j}} - f_{c_{i,j}}$ provides an optimal peak of memory on $\mathcal{S}_1$ and an optimal peak of memory on $\mathcal{S}_2$.*

Proof. For $\mathcal{S}_1$, see the end of Section 2.5. For $\mathcal{S}_2$, this results from the theorem from Liu which states that $x_k + \sum_{j=1}^{k-1} y_j$ is minimal when the (x_k,y_k) are sorted in decreasing order of $x_k - y_k$. The theorem is applied on Formula (9) with $x_k = T_{c_{i,k}}$ and $y_k = f_{c_{i,k}}$. □

Lemma 4. *Suppose that the max in $\mathcal{P}_2$ is obtained for child j_0, $p+1 \leq j_0 \leq n_i$. In other words, we suppose that $\mathcal{P}_2 = \mathcal{P}_2(j_0)$, where we define*

$$\begin{aligned}\mathcal{P}_2(j_0) &= store_i + \sum_{j=1}^{p} f_{c_{i,j}} + T_{c_{i,j_0}} + \sum_{k=p+1}^{j_0-1} f_{c_{i,k}} \\ &= store_i + \sum_{k=1}^{j_0-1} f_{c_{i,k}} + T_{c_{i,j_0}}\end{aligned} \tag{11}$$

Then, any configuration $(\mathcal{S}_1, \mathcal{S}_2)$ such that c_{i,j_0} belongs to $\mathcal{S}_2$ has a peak for the set $\mathcal{S}_2$ that is larger or equal to the value $\mathcal{P}_2(j_0)$ above.

Proof. (i) If we move nodes c_{i,j_1} from $\mathcal{S}_2$ to $\mathcal{S}_1$ that are before c_{i,j_0} (that is, $j_1 < j_0$), the second line in (11) will not change. (ii) If we move nodes c_{i,j_1} from $\mathcal{S}_2$ to $\mathcal{S}_1$ that are after c_{i,j_0} (that is, $j_1 > j_0$), the second line in (11) will increase (since $f_{c_{i,j_1}}$ adds up to the sum), and thus the peak on $\mathcal{S}_2$. □

Lemma 5. *Given a set $\mathcal{S}_1$, inserting a new element c_{i,j_0} to $\mathcal{S}_1$ cannot decrease the peak on $\mathcal{S}_1$: if we define $\mathcal{P}'_1$ to be the peak of total memory including the allocation of the parent using $\mathcal{S}'_1 = \mathcal{S}_1 \cup c_{i,j_0}$, we have $\mathcal{P}'_1 \geq \mathcal{P}_1$.*

Proof. See Formula (8). □

Theorem 2. *Given a set of nodes $(c_{i,k})_{k=1,n_i}$ with characteristics $T_{c_{i,k}}$, $f_{c_{i,k}}$, and $cb_{c_{i,k}}$ (as defined earlier), the following algorithm provides an optimal total memory peak T_i for the parent node i.*

Set $\mathcal{S}_1 = \emptyset$, $\mathcal{S}_2 = (c_{i,k})_{k=1,\ldots,n_i}$ and $p = 0$;
Sort $\mathcal{S}_2$ according to Lemma 3;
Compute $T_i = \mathcal{P}_2$ according to Formula (9);
repeat
  Find $c_{i,j_0} \in \mathcal{S}_2$ such that $\mathcal{P}_2 = store_i + \sum_{k=1}^{j_0} f_{c_{i,k}} + T_{c_{i,j_0}}$ (Formula (11));
  Set $\mathcal{S}_1 = \mathcal{S}_1 \cup c_{i,j_0}$, $\mathcal{S}_2 = \mathcal{S}_2 \setminus c_{i,j_0}$, and $p = p + 1$;
  Sort the nodes in $\mathcal{S}_1$ and $\mathcal{S}_2$ according to Lemma 3;
  Compute $\mathcal{P}_1$, $\mathcal{P}_2$, and $T'_i = \max(\mathcal{P}_1, \mathcal{P}_2)$;
  if $T'_i \leq T_i$ **then**
    Keep the values of p, $\mathcal{S}_1$ and $\mathcal{S}_2$ and set $T_i = T'_i$;
  end if
until $p = n_i$ or $\mathcal{P}_1 \geq \mathcal{P}_2$

Proof. We first remark that the order in $\mathcal{S}_1$ and $\mathcal{S}_2$ is imposed by Lemma 3. Starting from a configuration where $\mathcal{P}_2 > \mathcal{P}_1$, it results from Lemma 4 that the only way to decrease the peak is by moving c_{i,j_0} from $\mathcal{S}_2$ to $\mathcal{S}_1$. Thus, at each iteration either we have obtained the optimal peak T_i, or the solution with the optimal peak is such that c_{i,j_0} (which was responsible of the peak in $\mathcal{S}_2$) belongs to $\mathcal{S}_1$. Since we start with $\mathcal{S}_1 = \emptyset$, we are sure to reach the optimal configuration after a maximum of n_i iterations. (At each iteration, Lemma 3 is applied.)
For the termination criterion, we know that the optimal peak has been obtained when $\mathcal{P}_1$ becomes larger or equal than $\mathcal{P}_2$, since in that case the memory peak $T_i = \mathcal{P}_1$ will only increase if the algorithm is pursued further (Lemma 5). □

Finally, similarly to Algorithm 1, applying the previous theorem to the complete tree leads to Algorithm 2.
Remark. We used the stopping criterion $\mathcal{P}_1 \geq \mathcal{P}_2$. Note that the condition $T_i' > T_i$ is not correct to ensure that the optimal peak has been obtained. In practice, it may happen that the global peak will increase by moving the element c_{i,j_0} from $\mathcal{S}_2$ to $\mathcal{S}_1$, and decrease again at a further iteration to reach the optimal. An example producing such a situation is the one defined below:

$$\begin{cases} store_i = 100 & n_i = 3 \\ T_{c_{i,1}} = 160, & f_{c_{i,1}} = 100, \\ T_{c_{i,2}} = 140, & f_{c_{i,2}} = 120, \\ T_{c_{i,3}} = 10, & f_{c_{i,3}} = 5, \\ cb_{c_{i,1}} = cb_{c_{i,2}} = cb_{c_{i,3}} = 5 \end{cases}$$

Initially all three children are in $\mathcal{S}_2$, sorted according to Lemma 3 and $T_i = 340$ is reached for $c_{i,2}$, that the algorithm tries to move to $\mathcal{S}_1$: $\mathcal{S}_1 = \{c_{i,2}\}$ and $\mathcal{S}_2 = \{c_{i,1}, c_{i,3}\}$, leading to $T_i' = 380 > 340$. Then, moving child $c_{i,3}$ from $\mathcal{S}_2$ to $\mathcal{S}_1$ leads to a peak of total memory equal to $T_i = 330$, which is the optimal since $\mathcal{P}_1 = \mathcal{P}_2$.

5 Experimental results

In this section we present experimental results for both Algorithm 1 and, in the case of total memory minimization, the heuristics presented in the previous section.
For our experiments, we used the software package `MUMPS` (MUltifrontal Massively Parallel Solver) [4, 3], which implements parallel multifrontal solvers with threshold partial pivoting for both **LU** and $\mathbf{LDL}^T$ factorizations. We implemented the

Algorithm 2 Optimal tree reordering to minimize the peak of stack memory.

Tree_Reorder (T)**:**
  Begin
    for all i in the set of root nodes **do**
      Process_Child(i);
    end for
  End

Process_Child(i)**:**
  Begin
    if i is a leaf **then**
      T_i=$store_i$
    else
      for $j = 1$ to n_i **do**
        Process_Child$(c_{i,j})$;
      end for
      Determine the position p where the parent should be activated and the order of children using Theorem 2;
      Compute T_i using Formula (10);
    end if
  End

algorithm and the heuristics in the analysis phase of MUMPS and we compute statically the memory occupation that will be obtained during the factorization (if no pivoting occurs). In this simulator the tree is traversed using a depth-first search scheme (like the one used during the factorization phase of a multifrontal method), and the information generated by each algorithm/heuristic are used to measure the memory peak. This allows us to analyze the benefits from the algorithms without a complete implementation of the new schemes in the numerical factorization, which would be very much package/code-dependent and is outside the scope of this paper. Note that, for our experiments, we considered an unsymmetric storage of the frontal matrices even for the symmetric matrices.
We used trees resulting from various reordering techniques: AMD (Approximate Minimum Degree) [1], AMF (approximate Minimum Fill) as implemented in MUMPS, PORD [15] and METIS [11]. The motivation was to study our strategies on various types of assembly trees, knowing that the tree topology is very much influenced by the reordering technique applied [10]. The test problems are extracted from either the Rutherford-Boeing collection [6], the collection from University of Florida[1] or the PARASOL collection[2], and they are listed in Table 5, numbered from 1 to 45. They are from various application fields, with an order varying between 238 to 943695 and a number of stored nonzeros between 1128 and 39.3 millions. The test problem ULTRASOUND3 was provided by M. Sosonkina and was generated by X. Cai (Simula Research Laboratory, Norway) using diffpack. Finally, the MCHLNF problem was also provided by M. Sosonkina and represents a 3D tire design from John Melson (Michelin Research and Development Corporation). Note that our experiments were performed on one node of the IBM SP system from IDRIS[3].

Stack memory usage

Figure 4 gives the active (stack) memory usage for all matrices and the four reordering techniques considered. As expected, we observe that the optimal approach given by Algorithm 1 (normalized to 1) is always the best. The gains are good compared to both Liu's variant, and the systematic activation of the parent after the first child (where the first child is the one with largest memory peak – see Section 2.5). We observed that the largest gains with respect to Liu's algorithm are obtained for very wide trees (GUPTA matrices) where storing all contribution blocks before the parent allocation is prohibitive. We can also observe that the gains depend on the

[1] http://www.cise.ufl.edu/~davis/sparse/
[2] http://www.parallab.uib.no/parasol
[3] Institut du Développement et des Ressources en Informatique Scientifique

Symmetric problems		
1. 3DTUBE	9. GUPTA2	17. S3DKQ4M2
2. AUDIKW_1	10. GUPTA3	18. S3DKT3M2
3. BCSSTK34	11. MSDOOR	19. SHIP_003
4. BCSSTK38	12. M_T1	20. STRUCT4
5. BMWCRA_1	13. NASA1824	21. THREAD
6. CFD2	14. NASA2910	22. VIBROBOX
7. CRANKSG2	15. NASA4704	
8. GUPTA1	16. OILPAN	
Unsymmetric problems		
23. AF23560	31. LI	39. TWOTONE
24. BIG	32. MCHLNF	40. ULTRASOUND3
25. CIRCUIT_4	33. MIXING_TANK	41. VENKAT50
26. EPB3	34. ONETONE1	42. WANG1
27. GARON02	35. PRE2	43. WANG3
28. GRAHAM1	36. RMA10	44. XENON2
39. GRID48	37. SAYLR1	
30. INVEXTR1	38. THERMAL	

Table 1: Test matrices from various collections.

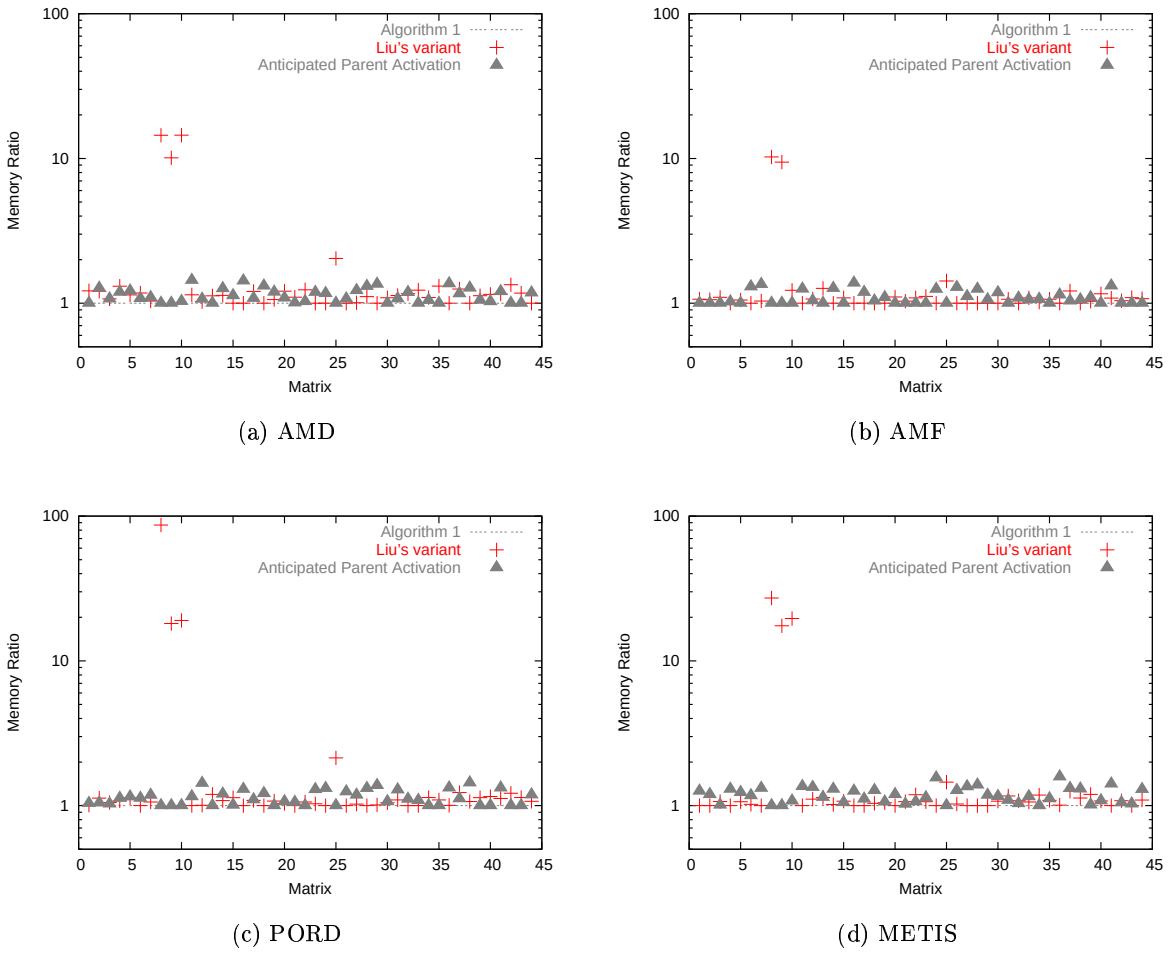
reordering technique used. For example, Algorithm 1 seems to be more effective with AMD. This can be explained by the fact that AMD generates deep trees with very large nodes in the upper part of the tree, with generally large contribution blocks [10]. Furthermore, the peak is often reached in an assembly operation in that part. Thus, anticipating the parent allocation has more potential to reduce the peak of active memory. On the other hand, with trees generated by METIS for example, Algorithm 1 gives the same results as Liu's algorithm for a lot of cases. This is mainly due to the regularity and good balance of the trees generated by METIS [10]. In addition, the peak is often reached in the upper parts of the tree where the sum of the contribution blocks of the children is generally smaller than the memory storage of the parent.
Finally, concerning AMF and PORD, There are also several cases where Liu's algorithm is comparable to Algorithm 1. The reason is that for those reordering techniques, the corresponding trees are very unbalanced with small frontal matrices and small contribution blocks, where anticipating the parent allocation does not help.
Despite these remarks it is worth applying Algorithm 1 systematically since for all orderings there are some matrices where significant gains are observed.

Total memory usage

With respect to the total memory, the results are presented in Figure 5. We compare the peak of total memory measured using the following variants:

- Algorithm 2;
- an anticipated parent activation after the first child; this is referred to as *Anticipated parent activation* and the children are sorted in decreasing order of $T_{c_{i,j}} - f_{c_{i,j}}$; and
- the classical multifrontal scheme, where the parent is only activated after all children have been processed. In that case, the children are sorted in decreasing order of $T_{c_{i,j}} - (cb_{c_{i,j}} + f_{c_{i,j}})$, as explained in Section 2.5. This is referred to as *classical total memory minimization* in Figure 5.

The results are normalized with respect to the optimal, obtained with Algorithm 2. Note that if the graph of the matrix is not connected (i.e. there are several trees in the dependency graph), we focus only on the most costly tree. Indeed, the minimization of the memory requirement of the matrix is done by applying the heuristic for each tree and then by sorting the subtrees in decreasing order of their $T - f$ (difference



(a) AMD

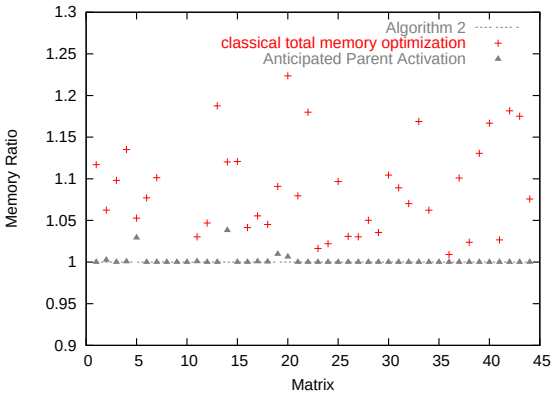
(b) AMF

(c) PORD

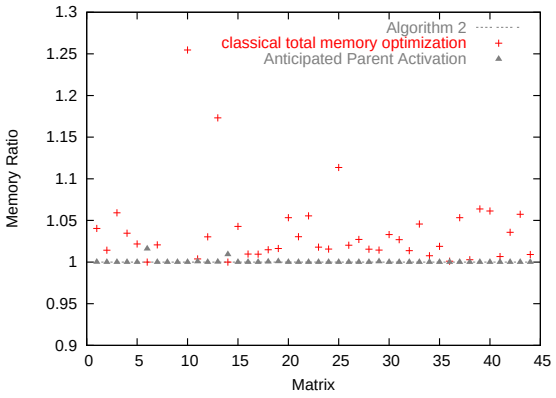
(d) METIS

Figure 4: Comparison of the active memory usage with Algorithm 1 (normalized to 1), Liu's variant, and the systematic allocation of the parent after the first child.

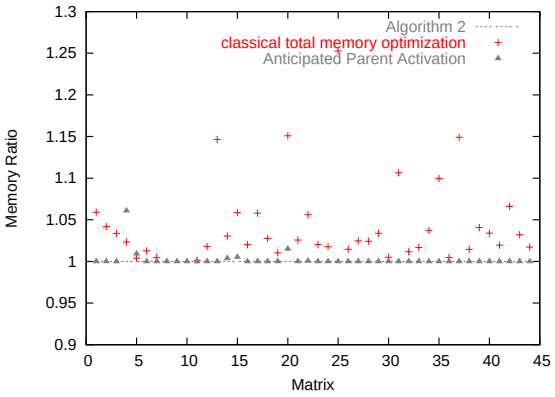
between the amount of memory needed to process the subtree and the amount of factors corresponding to it).



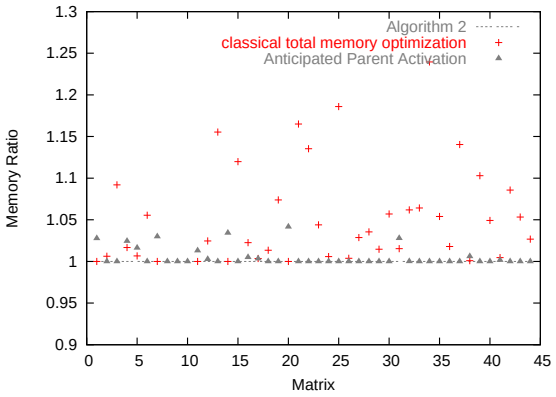
(a) AMD. Gains relative to *classical total memory optimization* are equal to 6.57, 6.78 and 4.12 for matrices 8, 9, 10, respectively.



(b) AMF. Gains relative to *classical total memory optimization* are equal to 4.26, 4.30 and 1.25 for matrices 8, 9, 10, respectively.



(c) PORD. Gains relative to *classical total memory optimization* are equal to 26.08, 8.95 and 8.09 for matrices 8, 9, 10, respectively.



(d) METIS. Gains relative to *classical total memory optimization* are equal to 9.99, 7.30 and 3.61 for matrices 8, 9, 10, respectively.

Figure 5: Comparison of the total memory usage with different algorithms and tree traversals. Memory is normalized with respect to Algorithm 2.

The gains concerning the total memory are not as large as they were for the stack. This is because the memory for the factors, which is constant, is now taken into account. When reaching the top of the tree, the amount of factors already computed is large and reduces the freedom to move nodes from $\mathcal{S}_2$ to $\mathcal{S}_1$. Compared the classical multifrontal method, gains due to Algorithm 2 are up to 25%, except for matrices 8, 9, and 10 (corresponding to matrices GUPTA1, GUPTA2, and GUPTA3) where the gain is larger because the stack is predominant. The comparison between Algorithm 2 and the anticipated parent activation shows that it is still worth allocating the parent at the optimal position for a few cases (for example, 1.06 for matrix 4 with PORD).

6 Special case: in-place assembly

In this section we describe variants of the algorithms given in the previous parts of the paper. We assume that the assembly of the contribution block corresponding to the last child treated before the allocation of the parent, the p^{th} child using the notations introduced before, is done in-place into the frontal matrix of the parent: the memory of the contribution block of the last child overlaps with the memory of the parent. In practice, this is implementation-dependent; some codes allow the assembly of the last child into the parent to be done in-place, others don't.

6.1 Active memory usage

With the assumption above, the active memory usage for a node i becomes:

$$\begin{aligned} A_i = \max\Big(& \max_{j=1,p}(A_{c_{i,j}} + \sum_{k=1}^{j-1} cb_{c_{i,k}}), \\ & store_i + \sum_{j=1}^{p-1} cb_{c_{i,j}}, \\ & store_i + \max_{j=p+1,n_i}(A_{c_{i,j}})\Big) \end{aligned} \tag{12}$$

The difference with Formula (7) comes from the term $store_i + \sum_{j=1}^{p-1} cb_{c_{i,j}}$. This is due to our assumption about the in-place assembly of the p^{th} child.
Finding an algorithm that minimizes the active memory occupation with this new scheme is equivalent to the problem presented in Section 3. The only difference comes from the computation/processing of the optimal order inside the set $\mathcal{S}_1$. Indeed,

inside this set, we have to use the algorithm proposed by Liu [12] that ensures an optimal memory occupation with the assumption that the last son is assembled in-place into the parent. This is done by sorting the children nodes in descending order of $\max(A_{c_{i,j}}, store_i) - cb_{c_{i,j}}$. Thus we can modify Algorithm 1 to sort the children nodes inside $\mathcal{S}_1$ in this order.

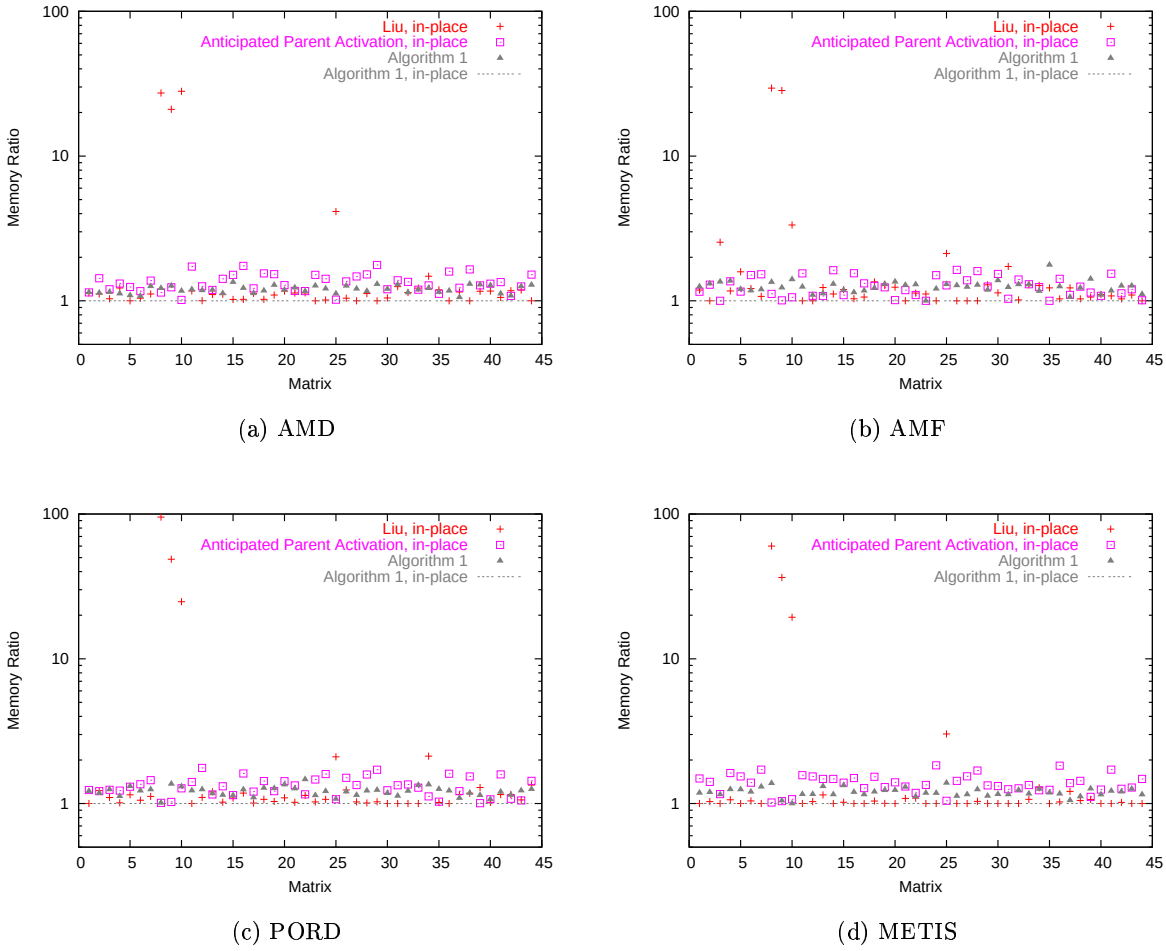
We present in Figure 6 the active memory usage obtained with this new algorithm ("Algorithm 1, in-place") compared to:

- the in-place assembly of the last child ("Liu, in-place"); children are sorted in decreasing order of $max(A_{c_{i,j}}, store_i) - cb_{c_{i,j}}$ (see Section 2.5) and this corresponds to the algorithm by Liu,
- the in-place assembly of the first child into the parent ("Anticipated activation of the parent, in-place"); the child with the largest peak of memory $A_{c_{i,j}}$ is processed first in order to minimize the memory usage,
- the situation of Section 3 ("Algorithm 1"), where the assembly into the parent is not in-place.

We observe that significant gains can be obtained compared to Algorithm 1. Indeed, with the new algorithm, the gains obtained at each level of the tree modify the global traversal and often allow a parent node to be activated earlier. This explains that we gain more than just the memory of the largest contribution block of the complete assembly tree. Comparing in-place approaches, we also remark that memory gains of up to 2 may be obtained over the case where the parent is activated after the first child, and that huge gains can still be obtained over the classical multifrontal method (*Liu, in-place*) for very wide trees (GUPTA matrices).

6.2 Total memory usage

Similarly to the case of the active memory, if the assembly of the contribution block corresponding to the p^{th} child is done in-place, the total memory becomes:



(a) AMD

(b) AMF

(c) PORD

(d) METIS

Figure 6: Comparison of the active memory usage with in-place and non in-place algorithms. Memory is normalized with respect to the in-place version of Algorithm 1.

$$T_i = \max\Big(\max_{j=1,p}(T_{c_{i,j}} + \sum_{k=1}^{j-1}(cb_{c_{i,k}} + f_{c_{i,k}})), \\ store_i + \sum_{j=1}^{p-1} cb_{c_{i,j}} + \sum_{j=1}^{p} f_{c_{i,j}}, \\ store_i + \max_{j=p+1,n_i}(T_{c_{i,j}} + \sum_{k=p+1}^{j-1} f_{c_{i,k}})\Big) \tag{13}$$

The only difference with the non in-place case comes from the peak in $\mathcal{S}_1$, more precisely the second term of the max, where the contribution block for child p is not taken into account in the assembly into the parent. Looking at Theorem 2, the same principle can be applied and the only modification comes from the peak in $\mathcal{S}_1$: instead of sorting the children in $\mathcal{S}_1$ according to Lemma 3, the order should be such that the new peak on $\mathcal{S}_1$,

$$\mathcal{P}_1 = \max(\max_{j=1,p}(T_{c_{i,j}} + \sum_{k=1}^{j-1}(cb_{c_{i,k}} + f_{c_{i,k}})), \\ store_i + \sum_{k=1}^{p-1} cb_{c_{i,k}} + \sum_{k=1}^{p} f_{c_{i,k}})$$

is minimal. We now try to find an optimal order of the children nodes $c_{i,j}, j = 1...p$ such that$\mathcal{P}_1$ is minimal. Note that we can rewrite $\mathcal{P}_1$ as

$$\mathcal{P}_1 = \max\Big(\max_{j=1,p}(T_{c_{i,j}} + \sum_{k=1}^{j-1}(cb_{c_{i,k}} + f_{c_{i,k}})), \\ store_i + \max_{j=1,p}(\sum_{k=1}^{j-1} cb_{c_{i,k}} + \sum_{k=1}^{j} f_{c_{i,k}})\Big) \\ = \max_{j=1,p}\Big(\sum_{k=1}^{j-1} cb_{c_{i,k}} + \max(T_{c_{i,j}}, store_i + f_{c_{i,j}})\Big)$$

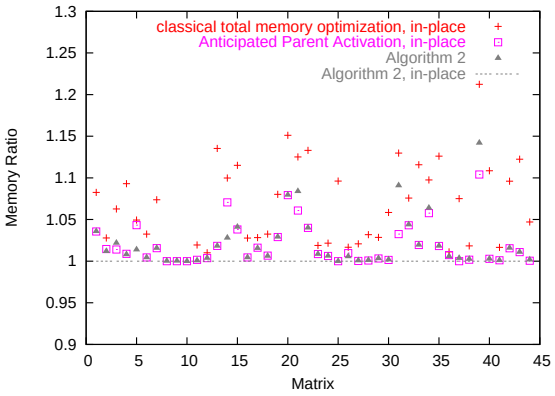
By applying Liu's theorem again, we obtain that the smallest peak in $\mathcal{S}_1$ is obtained when the children nodes are sorted in decreasing order of

$$\max(T_{c_{i,j}}, store_i + f_{c_{i,j}}) - (cb_{c_{i,j}} + f_{c_{i,j}}).$$

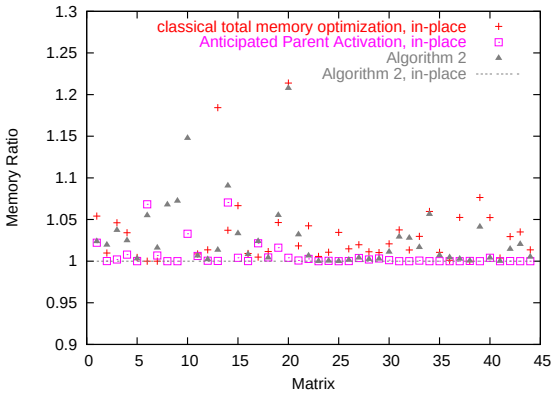
And an optimal algorithm for the total memory in the in-place case consists in applying Algorithm 2 with children nodes in $\mathcal{S}_1$ in that order rather than the one from Lemma 3. In the results that follow, we call this approach *Algorithm 2, in-place*. In Figure 7, we compare this approach (referred to as *Algorithm 2, in-place*) to:

- the anticipated parent activation (after the first child); children nodes are sorted in decreasing order of $T_{c_{i,j}} - (cb_{c_{i,j}} + f_{c_{i,j}})$,
- the classical multifrontal scheme, where the parent node is only activated after all children are processed and the last child is assembled in-place. Based on the result above, it makes sense to sort the children nodes in decreasing order of $\max(store_i, T_{c_{i,j}} + f_{c_{i,j}}) - (cb_{c_{i,j}} + f_{c_{i,j}})$, as stated in Section 2.5. This is referred to as *classical total memory minimization, in-place* in Figure 7,
- the memory obtained with the original Algorithm 2, where the assembly is not done in-place; the objective here is to appreciate how much can be gained by doing the assembly in-place, when the optimal position for the parent activation and optimal tree traversal are applied.

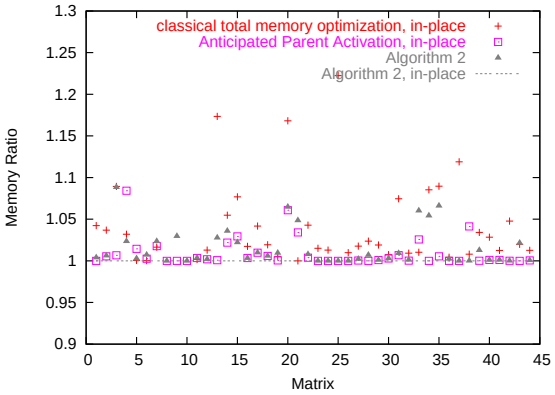
The results are normalized with respect to the best memory usage obtained for Algorithm 2, in-place. We observe in Figure 7 that by using this optimal algorithm, significant gains are obtained over the other approaches; also there seems to be more potential for gains than when the assembly was not in-place (Figure 5), thanks to the increased freedom we have with the in-place mechanism. Finally we observe that the memory ratio between the optimal in-place mechanism and the optimal non in-place mechanism can also be significant, and that it is worth doing the assembly in-place if the implementation allows it. Indeed (Table 6.2), we observe that the percentage of total memory used by the factors has significantly increased compared to the case of the non in-place memory, and that most of the memory is now used by the factors; this was often not the case with the classical multifrontal scheme.



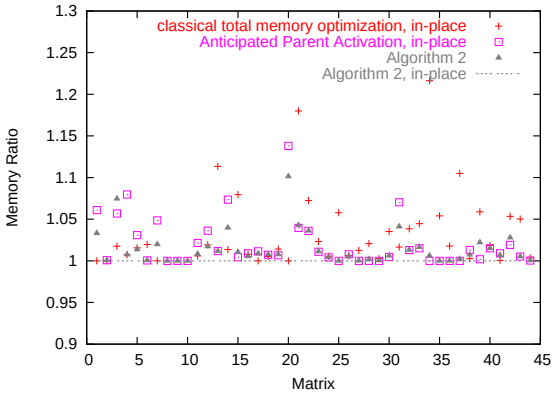
(a) AMD. Gains relative to *classical total memory optimization* are equal to 6.55, 7.64 and 4.09 for matrices 8, 9, 10, respectively.



(b) AMF. Gains relative to *classical total memory optimization* are equal to 4.49, 4.74 and 1.44 for matrices 8, 9, 10, respectively.



(c) PORD. Gains relative to *classical total memory optimization* are equal to 28.42, 9.21 and 8.34 for matrices 8, 9, 10, respectively.



(d) METIS. Gains relative to *classical total memory optimization* are equal to 10.79, 9.31 and 3.57 for matrices 8, 9, 10, respectively.

Figure 7: Comparison of the total memory usage with different algorithms and tree traversals. Memory is normalized with respect to Algorithm 2, in-place variant.

Percentage of total memory for factors	Number of test cases	
	Algorithm 2	Algorithm 2, in-place
< 50 %	2	0
50%-60%	2	3
60%-70%	9	6
70%-80%	22	22
80%-100%	141	145
Total	176	176

Table 2: Number of combinations of test cases (matrix/ordering) for different ranges of the percentage of total memory used by the final factors.

7 Implementation issues

7.1 Assembling children when the parent is already allocated

As we have seen, the best memory usage is obtained when the parent node is allocated before all children are processed. This complicates slightly the assembly mechanism compared to the classical multifrontal approach, where the assembly operation of a set of children nodes into the parent requires indirections and is generally done in the following way: the lists of indices of the contribution blocks from the children are merged using an indirection array of size N, where N is the order of the complete matrix. In fact, for each new variable v encountered of each new child, a counter k is incremented and the entry $v \in 1..N$ of the indirection array is set to k. k corresponds to the position of global variable v in the parent and is used to assemble elements of the children at the appropriate position of the parent.

In our case, when processing children nodes that are posterior to the allocation of the parent node, we cannot assume that such an array of size N associated to node i is still available in memory, since several such work arrays would then have to be stored simultaneously for all active parent nodes. It also seems to us that pre-computing such information during the symbolic factorization phase would be costly. Thus, this work array has to be recomputed for each new child in $\mathcal{S}_2$. This can be done in one pass over the variables already present in the parent node, which appears to be reasonable considering that this cost remains small compared to that of assembling the effective numerical values.

7.2 Pivoting issues

When pivoting occurs, the exact size of the parent node cannot be known a priori, i.e., before all children have been processed. It may happen in some cases that the size of the parent dynamically increases (delayed pivots) because of stability problems occurring at the children level. What can be done in that case is to allocate a parent node slightly larger (say, 10 %) to overcome most pivoting problems. If this is not enough it is possible to reallocate a larger parent node, and copy the values of the previous parent node into it. Either the copy can be done in-place, or a temporary extra-storage has to be used, increasing the total storage of at most the size (with delayed pivots) of the largest node in the tree.

7.3 Parallel case

The results described in this paper are valid in the sequential case. They can also be applied in a parallel implementation of a multifrontal approach on clustered subtrees treated sequentially on each processor. For the upper part of the tree, where nodes may themselves be parallelized, the issue is very much dependent on the effective management of memory and parallelism of the implementation considered.
Still, an approach to reduce the memory usage that can be pretty general (and that could be applied in the parallel case) consists in modifying the structure of the top of the assembly tree by inserting artificial nodes between a parent and its children. For example, a parent with 100 children initially could have 10 children (possibly smaller than the parent) of 10 children each after such a modification. For the intermediate nodes introduced, no factorization process would occur, only assembly operations would be performed.

8 Conclusion

We have presented algorithms adapted to the management of the memory in the multifrontal method. We showed how to obtain the best possible memory usage on a given assembly tree, and reported on a large range of matrices/reordering techniques the gains obtained over the classical multifrontal method and other heuristics. This represents an improvement of the algorithms proposed by [12] and [10], and the total memory is now dominated by the factors.
Different approaches have been suggested for both the active memory (out-of-core scheme where computed factors are stored on disk, either explicitly or by the system paging mechanisms) and the total memory (in-core scheme). Furthermore both the

in-place and classical (with a copy) assembly of the last child into the parent have been considered.
We have discussed implementation issues for the new scheme, including the case of delayed pivots possibly occurring during the factorization and the parallel case, which is more code-dependent. We hope that these results will be useful to developers of sparse direct solvers.

Acknowledgement

We are grateful to Arnaud Legrand for his help regarding the central theorems and proofs of this paper.

References

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

[2] P. R. Amestoy and I. S. Duff. Memory management issues in sparse multifrontal methods on multiprocessors. *Int. J. of Supercomputer Applics.*, 7:64–82, 1993.

[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(1):15–41, 2001.

[4] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods Appl. Mech. Eng.*, 184:501–520, 2000.

[5] C. Ashcraft, R. G. Grimes, J. G. Lewis, B. W. Peyton, and H. D. Simon. Progress in sparse matrix methods for large linear systems on vector computers. *Int. Journal of Supercomputer Applications*, 1(4):10–30, 1987.

[6] I. S. Duff, R. G. Grimes, and J. G. Lewis. The Rutherford-Boeing Sparse Matrix Collection. Technical Report TR/PA/97/36, CERFACS, Toulouse, France, 1997. Also Technical Report RAL-TR-97-031 from Rutherford Appleton Laboratory and Technical Report ISSTECH-97-017 from Boeing Information & Support Services.

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

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

[9] A. Guermouche, J.-Y. L'Excellent, and G. Utard. Analysis and improvements of the memory usage of a multifrontal solver. Technical Report RR-2003-08, LIP, 2003. Also INRIA report RR-4829.

[10] A. Guermouche, J.-Y. L'Excellent, and G. Utard. Impact of reordering on the memory of a multifrontal solver. *Parallel Computing*, 29(9):1191–1218, 2003.

[11] G. Karypis and V. Kumar. MeTiS – *A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices – Version 4.0.* University of Minnesota, September 1998.

[12] J. W. H. Liu. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Transactions on Mathematical Software*, 12:127–148, 1986.

[13] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11:134–172, 1990.

[14] E. Rothberg and R. Schreiber. Efficient methods for out-of-core sparse cholesky factorization. *SIAM Journal on Scientific Computing*, 21(1):129–144, 1999.

[15] J. Schulze. Towards a tighter coupling of bottom-up and top-down sparse matrix ordering methods. *BIT*, 41(4):800–841, 2001.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399