Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

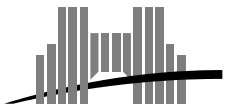
Reducing the I/O Volume in an Out-of-core Sparse Multifrontal Solver

Emmanuel Agullo ,
Abdou Guermouche ,
Jean-Yves L'Excellent

May 2007

Research Report N° 2007-22

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`



[0]This work is partially supported by ANR project SOLSTICE, ANR-06-CIS6-010.
This text is also available as a research report of INRIA.

Reducing the I/O Volume in an Out-of-core Sparse Multifrontal Solver

Emmanuel Agullo , Abdou Guermouche , Jean-Yves L'Excellent

May 2007

Abstract

High performance sparse direct solvers are often a method of choice in various simulation problems. However, they require a large amount of memory compared to iterative methods. In this context, out-of-core solvers must be employed, where disks are used when the storage requirements are too large with respect to the physical memory available. In this paper, we study how to minimize the I/O requirements in the multifrontal method, a particular direct method to solve large-scale problems efficiently. From a theoretical point of view, we show that minimizing the storage requirement can lead to a huge volume of I/O compared to directly minimizing the I/O volume. Then experiments on large real-life problems also show that the volume of I/O obtained when minimizing the storage requirement can be significantly reduced by applying algorithms designed to reduce the I/O volume. We finally propose efficient memory management algorithms that can be applied to all the variants proposed.

Keywords: Sparse direct solver; out-of-core; large matrices; multifrontal method; IO volume minimization; memory management schemes

Résumé

Les solveurs creux directs sont souvent une méthode de choix pour traiter divers problèmes de simulation. Néamoins, ils requièrent une capacité mémoire importante par rapport aux méthodes itératives. Dans ce contexte, des solveurs *out-of-core* doivent être employés, où les disques sont utilisés lorsqu'il devient nécessaire d'utiliser plus de mémoire que n'en dispose la machine. Dans ce papier, nous étudions comment minimiser le volume d'entrées/sorties dans la méthode multifrontale (une méthode directe particulière), afin de résoudre efficicacement des problèmes de grande taille. Nous montrons d'un point de vue théorique que minimiser le volume d'entrées/sorties ne se ramène justement pas à minimiser les besoins mémoires. Des expérimentations sur des problèmes académiques et industriels de grande taille montrent ensuite que le volume d'entrées/sorties obtenu en minimisant les besoins mémoire peut être réduit de manière significative en appliquant des algorithmes spécifiquement conçus pour réduire le volume d'entrées/sorties. Nous proposons finalement des algorithmes de gestion mémoire efficaces pour toutes les variantes étudiées.

Mots-clés: Solveur creux direct ; out-of-core ; hors-mémoire ; matrices de grande taille ; méthode multifrontale ; minimisation du volume d'entrées/sorties ; schémas de gestion mémoire

1 Introduction

We are interested in solving a sparse system of linear equations of the form $Ax = b$ by a so-called direct method. Such methods work in three phases: (i) an analysis phase, that orders the variables of the problem to limit the computations and prepares the work for the factorization; (ii) a numerical factorization phase, where A is factored under the form LU, LL^t or LDL^t; and (iii) a solve phase, where triangular factors are used to obtain the solution of the problem. Because of their large memory requirements, several authors have worked on out-of-core sparse direct solvers [1, 4, 8, 12, 15, 16, 17]. Left-looking and multifrontal methods are two main classes of sparse direct methods that can be extended to an out-of-core context. In that case, a left-looking approach allows to reduce significantly the minimal memory requirements, while the multifrontal method may lead to large frontal matrices that prevent processing arbitrarily large problems [16] if frontal matrices are not assembled and factored with out-of-core algorithms. On the other hand, for problems in which the largest frontal matrix fits in memory or can be treated reasonably using an out-of-core algorithm, the multifrontal method remains interesting [14, 7] and motivates the design of robust software solutions [2, 15].

In the multifrontal method, the factorization of a sparse matrix A is done by a succession of partial factorizations of small dense matrices called *frontal matrices*. Since the frontal matrices are dense, this method allows an efficient use of memory hierarchy and caches, where optimized dense kernels (BLAS) can be applied. For matrices with a symmetric structure (or in approaches like [6] when the structure of matrix **A** is unsymmetric), each frontal matrix is associated with a node of a so-called *a*ssembly tree which represents the dependencies of the tasks in the factorization algorithm.

Before a partial factorization of a parent node can be performed, temporary data (so-called *contribution blocks*) extracted from the frontal matrices of children are assembled into the frontal matrix of the parent. After the parent is factored, the contribution block of the parent is kept in memory for later use at the upper layer of the tree.

In this paper, we are concerned with out-of-core multifrontal methods. Since the factors are terminal data for the factorization phase, it appears natural to write them to disk as soon as they are produced. Focusing on memory handling issues, the multifrontal algorithm may be presented as follows:

For each `node` k **in** `the tree (postorder traversal)`
  $al_k(x)$: `Allocate memory (of size` x`) for the frontal matrix associated to` k
  If k `is not a leaf:`
    $as_k(x)$: `Assemble contribution blocks from children (of total size` x`)`
  End If
  $f_k(x)$: `Perform a partial factorization of the frontal matrix of` k`,`
    `writing factors (of size` x`) to disk on the fly`
End For

Note that, because we rely on a post-order traversal, the multifrontal algorithm can use

a stack mechanism to store the contribution blocks: the contribution blocks produced last are the first ones assembled. Still, there is a lot of freedom to order the siblings at each level of the tree so that the tree traversal can have a significant impact on both the number of contribution blocks stored simultaneously and the memory usage. Liu [13] (and, more recently, [11, 10]) have shown the impact of the tree traversal on the memory behaviour and proposed tree traversals that minimize the storage requirements of the multifrontal method when factors are systematically written to disk. With this assumption, Liu suggested in the conclusion of [13] that minimizing the storage requirements was well adapted to an out-of-core execution.

In this paper we focus on the volume of *I/O* related to the stack of contribution blocks and we aim at designing optimal tree traversals with respect to minimizing the volume of *I/O*. By expressing this volume in a formal way, we show that minimizing the storage requirements is different from minimizing the volume of *I/O*.

Note that we consider several minor variants of the multifrontal algorithm. We call *last-cb in-place* a variant of the assembly scheme (available, for example, in a code like `MA27` [9]) where the memory of the frontal matrix at the parent node is allowed to overlap with the contribution block of the last child. In that case, we save memory by not summing the memory of the frontal matrix of the child with the memory of the frontal matrix of the parent (a maximum between these two values is enough). We also propose a new variant, where we overlap the memory for the frontal matrix of the parent with the memory of the child having the largest contribution block (even if that child is not processed last). For each variant, we present the tree traversal that minimizes memory (algorithms so called `MinMEM`); then, we show by how much the volume of *I/O* can be reduced (depending on the physical memory available) with new algorithms (called `MinIO`) that minimize the *I/O* volume. We also discuss possible memory management algorithms corresponding to each variant, and show that these variants can be implemented reasonably, without complicated garbage collection mechanisms.

The paper is organized as follows. In Sections 2 and 3, we explain how to model and minimize the volume of *I/O* induced by the *classical* and *last-cb in-place* schemes, respectively. In Section 4, we discuss the new variant of the *in-place* algorithm. We then show in Section 5 that the volume of *I/O* induced by `MinMEM` may be arbitrarily larger than the volume induced by `MinIO`. Section 6 illustrates the difference between `MinMEM` and `MinIO` on matrices arising from real-life problems, and shows the interest of the new *in-place* variant proposed.

2 Limiting the amount of I/O

2.1 Some notations

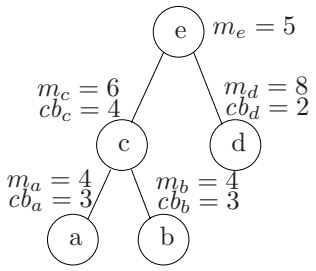
In a limited memory environment, we define M_0 as the volume of core memory available for the multifrontal factorization. As described in the introduction, the multifrontal method is based on a tree in which a parent node is allocated in memory after all its child subtrees

have been processed. When considering a generic parent node and its n children numbered $j = 1, \ldots, n$, we note:

- cb / cb_j, the storage for the contribution block of the parent node / of child j (note that $cb = 0$ for the root of the tree);
- m / m_j, the storage of the frontal matrix associated to the parent node / to child j (note that $m_j > cb_j$ and $m_j - cb_j$ is the size of the factors produced by child j);
- S / S_j, the storage required to process the subtree rooted at the parent / at child j (note that if $S_j < M_0$, no *I/O* is necessary to process the whole subtree rooted at j);
- $V^{I/O}$ / $V_i^{I/O}$ the volume of *I/O* required to process the subtree rooted at node j given an available memory of size M_0.

2.2 Illustrative example

To illustrate the memory behaviour, let us first take the toy example described in Figure 1(left): we consider a root node **(e)** with two children **(c)** and **(d)**. The frontal matrix of **(e)** requires a storage $m_e = 5$. The contribution blocks of **(c)** and **(d)** require a storage $cb_c = 4$ and $cb_d = 2$, while the storage requirements for their frontal matrices are $m_c = 6$ and $m_d = 8$ respectively. **(c)** has itself two children with characteristics $cb_a = cb_b = 3$ and $m_a = m_b = 4$. We assume that the core memory available is $M_0 = 8$.



Sequence a-b-c-d-e

Storage $S = 12$

I/O $V^{I/O} = 8$

$\Rightarrow$ Memory minimized

Sequence d-a-b-c-e

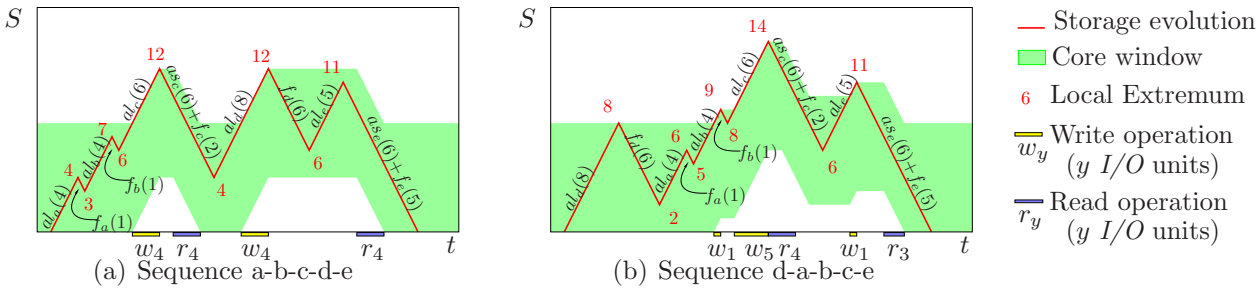
Storage $S = 14$

I/O $V^{I/O} = 7$

$\Rightarrow$ *I/O* minimized

Figure 1: Influence of the tree traversal on the storage requirement and on the volume of *I/O* (with $M_0 = 8$).

To respect a postorder traversal, there are two possible ways to process this tree: (a-b-c-d-e) and (d-a-b-c-e). (Note that **(a)** and **(b)** are identical and can be swapped.) For each sequence we now describe the memory behaviour and *I/O* operations. We first consider sequence (a-b-c-d-e), see Figure 2(a). **(a)** is first allocated ($m_a = 4$) and factored (we write its factors of size $m_a - cb_a = 1$ to disk), and $cb_a = 3$ remains in memory. After **(b)** is processed, the memory contains $cb_a + cb_b = 6$. Then a peak of storage $S_c = 12$ is reached when the frontal matrix of **(c)** is allocated. Since only 8 (MegaBytes, say) can be kept in core memory, this leads to write to disk a volume of data equal to 4. During the assembly



(a) Sequence a-b-c-d-e (b) Sequence d-a-b-c-e

Figure 2: Evolution of the storage requirement when processing the sample tree of Figure 1 with the two possible postorders, and subsequent I/O operations. Notations of the algorithm introduced in Section 1 are employed.

process we first assemble contributions that are in memory, and then read (r_4) data from disk to assemble them in turn in the frontal matrix of **(c)**. Note that (here but also more generally), in order to fit the memory requirements, the assembly of data initially on disk may have to be performed by panels (interleaving the read and assemble operations). After the factors of **(c)** of size $m_c - cb_c = 2$ are written to disk, its contribution block $cb_c = 4$ remains in memory. When leaf node **(d)** is processed, the peak of storage reaches $cb_c + m_d = 12$. This leads to a new volume of *I/O* equal to 4 (and corresponding to cb_c). After **(d)** is factored, the storage requirement is equal to $cb_c + cb_d = 6$ among which only $cb_d = 2$ is in core (cb_c is already on disk). Finally, the frontal matrix of the parent (of size $m_e = 5$) is allocated, leading to a storage $cb_c + cb_d + m_e = 11$: after cb_d is assembled in core (into the frontal matrix of the parent), cb_c is read back from disk and assembled in turn. Overall the volume of data written to (and read from) disk[1] is $V_e^{I/O}$(a-b-c-d-e)= 8 and the peak of storage was S_e(a-b-c-d-e)= 12.

When the tree is processed in order (d-a-b-c-e) (see Figure 2(b)), the storage requirement successively takes the values $m_d = 8$, $cb_d = 2$, $cb_d + m_a = 6$, $cb_d + cb_a = 5$, $cb_d + cb_a + m_b = 9$, $cb_d + cb_a + cb_b = 8$, $cb_d + cb_a + cb_b + m_c = 14$, $cb_d + cb_c = 6$, $cb_d + cb_c + m_e = 11$, with a peak S_e(d-a-b-c-e)= 14. Nodes **(d)** and **(a)** can be processed without inducing *I/O*, then 1 unit of *I/O* is done when allocating **(b)**, 5 units when allocating **(c)**, and finally 1 unit when the frontal matrix of the root node is allocated. We obtain $V_e^{I/O}$(d-a-b-c-e)= 7.

We observe that the postorder (a-b-c-d-e) minimizes the peak of storage, while (d-a-b-c-e) minimizes the volume of *I/O*. This shows that minimizing the peak of storage is different from minimizing the volume of *I/O*.

[1]Remember that we do not count *I/O* for factors since factors are written to disk systematically in all variants considered.

2.3 Expressing the volume of I/O

Since contribution blocks are stored thanks to a stack mechanism, some contribution blocks (or parts of contribution blocks) may be kept in memory and consumed without being written to disk [4, 3]. Assuming that the contribution blocks are written only when needed (possibly only partially), that factors are written to disk as soon as they are computed, and that a frontal matrix must fit in core memory, we focus on the computation of the volume of *I/O* on this stack of contribution blocks.

When processing a child j, the contribution blocks of all previously processed children have to be stored. Their memory size sums up with the storage requirements S_j of the considered child, leading to a global storage equal to $S_j + \sum_{k=1}^{j-1} cb_k$. After all the children have been processed, the frontal matrix (of size m) of the parent is allocated, requiring a storage equal to $m + \sum_{k=1}^{n} cb_k$. Therefore, the storage required to process the complete subtree rooted at the parent node is given by the maximum of all theses values, that is:

$$S = \max\left(\max_{j=1,n}(S_j + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1}^{n} cb_k\right) \quad (1)$$

Knowing that the storage requirement S for a leaf node is equal to the size of its frontal matrix m, applying this formula recursively (as done in [13]), allows to determine the storage requirement for the complete tree.

In our out-of-core context, we now assume that we are given a core memory of size M_0. If $S > M_0$, some *I/O* will be necessary. Since the contribution blocks are accessed with a stack mechanism, writing the bottom of the stack first results in an optimal volume of *I/O*.

To simplify the discussion we first consider a set of leaf nodes with their parent. In that case, S_j is simply equal to m_j. The volume of contribution blocks that will be written to disk corresponds to the difference between the memory requirement at the moment when the peak S is obtained and the size M_0 of the memory allowed (or available). Indeed, each time an *I/O* is done, an amount of temporary data located at the bottom of the stack is written to disk. Furthermore, data will only be reused (read from disk) when assembling the parent node. More formally, the expression of the volume of *I/O*, $V^{I/O}$, using Formula (1) for the storage requirement, is:

$$V^{I/O} = \max\left(0, \max(\max_{j=1,n}(S_j + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1}^{n} cb_k) - M_0\right) \quad (2)$$

As each contribution written is read once, $V^{I/O}$ may be considered either as (and is defined as) the volume of data written or that read. Let us consider now a more general context where each child may root a subtree. In this new context, for a child j, S_j denotes the peak of storage observed while processing its subtree. If we suppose that $\forall j : S_j \leq M_0$, Formula (2) continues to be applicable to compute the volume of I/O needed to process the tree rooted at the parent node.

Suppose now that $\exists j : S_j > M_0$. We know that child j will have an intrinsic volume of I/O $V_j^{I/O}$ (recursive definition based on a bottom-up traversal of the tree). In addition, we know that it cannot occupy more than M_0 in memory. Thus, we can consider it as a child using exactly M_0 memory ($A_j \stackrel{def}{=} \min(S_j, M_0)$), and inducing an intrinsic volume of I/O equal to $V_j^{I/O}$. With this definition of A_j as the *active memory*, *i.e.* the amount of core memory effectively used to process the subtree rooted at child j, we can now generalize the expression given in Formula (2) which becomes:

$$V^{I/O} = \max\left(0, \max(\max_{j=1,n}(A_j + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1}^{n} cb_k) - M_0\right) + \sum_{j=1}^{n} V_j^{I/O} \qquad (3)$$

To compute the volume of I/O on the whole tree, we can simply apply recursively Formula (3) at each level of the tree (knowing that $V^{I/O} = 0$ for leaf nodes). The volume of I/O of the tree is then given by the $V^{I/O}$ value of its root node.

2.4 Tree traversals

It results from Formula (3) that minimizing the volume of I/O is equivalent to minimizing the expression $\max_{j=1,n}(A_j + \sum_{k=1}^{j-1} cb_k)$, since it is the only term sensitive to the order of the children.

Theorem 2.1. (Liu, 86) *Given a set of values $(x_i, y_i)_{i=1,\ldots,n}$, the minimal value of $\max_{i=1,\ldots,n}(x_i + \sum_{j=1}^{i-1} y_j)$ is obtained by sorting the sequence (x_i, y_i) in decreasing order of $x_i - y_i$, that is, $x_1 - y_1 \geq x_2 - y_2 \geq \ldots \geq x_n - y_n$.*

Thanks to Theorem 2.1 (proved in [13]), we deduce that we should process the children nodes in decreasing order of $A_j - cb_j = \min(S_j, M_0) - cb_j$. (This implies that if all subtrees require a storage $S_j > M_0$ then `MinIO` will simply order them in increasing order of cb_j.) An optimal postorder traversal of the tree is then obtained by applying this sorting at each level of the tree, constructing Formulas (1) and (3) from bottom to top. We will name `MinIO` this algorithm.

Note that, in order to minimize the peak of storage (defined in Formula (1)), children had to be sorted (at each level of the tree) in decreasing order of $S_j - cb_j$ rather than $A_j - cb_j$. Therefore, on the example from Section 2.2, the subtree rooted at **(d)** ($A_d - cb_d = 8 - 2 = 6$) must be processed before the subtree rooted at **(c)** ($A_c - cb_c = M_0 - 4 = 4$). The corresponding algorithm (that we name `MinMEM` and that leads to sequence (a-b-c-d-e)) is thus different from `MinIO` (that leads to (d-a-b-c-e)): minimizing the storage requirement is *not* minimizing the I/O volume; it may induce a volume of I/O larger than needed.

3 In-place assembly of the last contribution block

This is a variant of the *classical* multifrontal algorithm (used in `MA27` [9] and its successors, for example) in which the memory of the frontal matrix of the parent is allowed to overlap

with (in fact to include) that of the contribution block from the last child. The contribution block from the last child is then expanded (or assembled *in-place*) in the memory of the parent. Since the memory of a contribution block can be large, this scheme can have a strong impact on both storage and *I/O* requirements. In this new context, the storage requirements needed to process a given node (Formula (1)) becomes:

$$S = \max \left(\max_{j=1,n}(S_j + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1}^{\boxed{n-1}} cb_k \right) \tag{4}$$

The main difference with Formula (1) comes from the *in-place* assembly of the last child (see the boxed superscript in the sum in Formula (4)). In the rest of the paper we will use the term *last-cb in-place* to denote the memory management scheme where an *in-place* assembly scheme is used for the contribution block coming from the last child. Liu has shown[13] that Formula (4) could be minimized by ordering children in decreasing order of $\max(S_j, m) - cb_j$.

In an out-of-core context, the use of this *in-place* scheme induces a modification of the amount of data that has to be written to/read from disk. As previously for the memory requirement, the volume of I/O to process a given node with n children (Formula (3)) becomes:

$$V^{I/O} = \max \left(0, \max(\max_{j=1,n}(A_j + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1}^{\boxed{n-1}} cb_k) - M_0 \right) + \sum_{j=1}^{n} V_j^{I/O}$$

Once again, the difference comes from the *in-place* assembly of the contribution block coming from the last child. Because $m + \sum_{k=1}^{n-1} cb_k = \max_{j=1,n}(m + \sum_{k=1}^{j-1} cb_k)$, this formula can be rewritten as:

$$V^{I/O} = \max \left(0, \max_{j=1,n}(\max(A_j, m) + \sum_{k=1}^{j-1} cb_k) - M_0 \right) + \sum_{j=1}^{n} V_j^{I/O} \tag{5}$$

Thanks to Theorem 2.1, minimizing the above quantity can be done by sorting the children nodes in decreasing order of $\max(A_j, m) - cb_j$. Again, the I/O volume minimization algorithm in the *last-cb in-place* context is obtained by applying this order for each family in the tree, in a bottom-up process.

4 In-place assembly of the largest contribution block

In order to do better than equation (4), one is tempted to try to overlap the memory of the parent not with the contribution from the last child, but with the largest child contribution block. In that case, the largest contribution block is known in advance and we can apply

the memory-minimization order from the *classical* scheme (decreasing order of $S_j - cb_j$). The main difference is in the computation of the peak of storage, where cb_{max} must be subtracted from the term $m + \sum_j cb_j$ in equation (1). From an implementation point of view, note that the *in-place* assembly of the largest contribution block requires storing it in a particular area, rather than in the main stack. While processing the tree using a postorder, we thus need to use two stack mechanisms: one for the normal contribution blocks (for example on the left of a workarray), and one for the largest contribution blocks of each family (for example in the right part of the same workarray). The second one is used to extend the adequate contribution block into the frontal matrix of the parent. We call this scheme *max-cb in-place*. In that context, `MinMEM` is optimal by sorting a set of children in decreasing order of $S_j - cb_j$.

In an out-of-core context, it is not immediate or easy to generalize `MinIO`. Indeed, there is no guarantee that we will be able to keep the largest contribution block of a family in core memory to enable its *in-place* assembly (suppose, for example, that a subtree ordered after that which induces the largest contribution block forces us to write this contribution to disk). Therefore, we propose the following heuristic. We first try to apply `MinMEM` + *max-cb in-place* to a given family (in a bottom-up process). If this leads to a storage smaller than M_0, we keep this approach to process this family. Otherwise, we *switch* to `MinIO` + *last-cb in-place* to process this family and any parent family. In the following we name this heuristic `MinIO` + *max-cb in-place*.

5 Theoretical comparison of `MinMEM` and `MinIO`

Theorem 5.1. *The volume of* I/O *induced by* `MinMEM` *may be arbitrarily larger than the volume induced by* `MinIO`.

This result is valid both for the *classical* and *last-cb in-place* assembly schemes. In the following, we provide a formal proof for the *classical* (non in-place) context.

Proof. Let M_0 be the core memory available and $r(>2)$ an arbitrarily large real number. We aim at building an assembly tree (to which we may associate a matrix) for which the *I/O* volume induced by `MinMEM`, $V^{I/O}$ (`MinMEM`), is at least r times larger than the one induced by `MinIO`, $V^{I/O}$ (`MinIO`), *i.e.* for which $V^{I/O}(\texttt{MinMEM})/V^{I/O}(\texttt{MinIO}) \geq r$.

We first consider a sample tree T_0 as described in Figure 3(a) composed of a root node (r) and two leaves (a) and (b). The frontal matrices of (a), (b) and (r) require respectively a storage $m_a = M_0$, $m_b = M_0$ and $m_j = M_0/2$. Their respective contribution blocks are of size $cb_a = 3M_0/4$, $cb_b = 3M_0/4$ and $cb_r = M_0/3$. Finally the storage required to process T_0 is $S_0(\texttt{MinMEM}) \stackrel{def}{=} S_r(\texttt{MinMEM}) = 2M_0$ leading to a volume of *I/O* of $V_0^{I/O} \stackrel{def}{=} V_r^{I/O} = M_0$.

We say that a subtree T_k verifies $\mathcal{P}_k$ when it is of height $k+1$, has a peak of storage equal to $S_k(\texttt{MinMEM}) = 2M_0$, a frontal matrix at its root of size $m_r = M_0/2$ with a contribution block of size $cb_r = M_0/3$. We have just shown that T_0 verifies $\mathcal{P}_0$. Given a subtree T_k which verifies $\mathcal{P}_k$, we now build recursively a tree T_{k+1} which verifies $\mathcal{P}_{k+1}$. To proceed we root

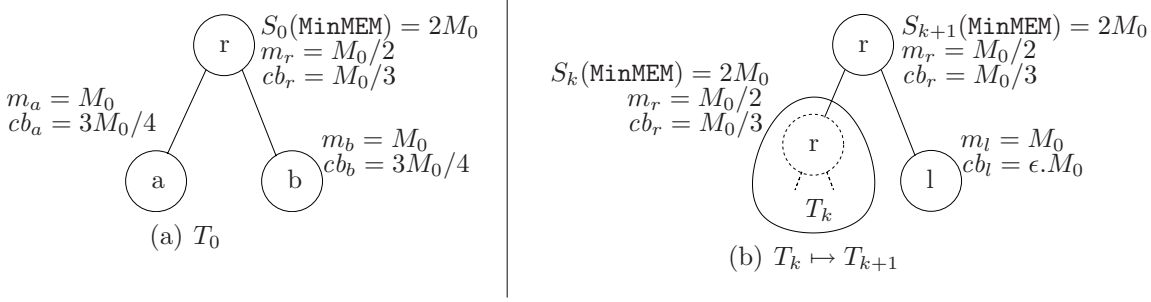


Figure 3: Recursive construction of an assembly tree illustrating Theorem 5.1.

T_k and a leaf node (l) to a new parent node (r), as illustrated in Figure 3(b). The frontal matrix of the root node has characteristics $m_r = M_0/2$ and $cb_r = M_0/3$, and the leaf node (l) is such that $m_l = S_l = M_0$ and $cb_l = \epsilon M_0$. The value of ϵ is not fixed yet but we suppose $\epsilon < 1/10$. The active memory usage for T_k and (l) are thus $A_k = \min(S_k, M_0) = M_0$ and $A_l = \min(S_l, M_0) = M_0$. `MinMEM` would process such a family in the order (T_k-l-r) because $S_k - cb_k > S_l - cb_l$. This leads to a peak of storage equal to $S_{k+1}(\texttt{MinMEM}) = 2M_0$ (obtained when processing T_k). Thus T_{k+1} verifies $\mathcal{P}_{k+1}$. We note that `MinMEM` would lead to a volume of *I/O* (see Formula (3)) equal to $V^{I/O}_{k+1}(\texttt{MinMEM}) = M_0/3 + V^{I/O}_k(\texttt{MinMEM})$.

`MinIO` would process it in the order (l-T_k-r) because $A_l - cb_l > A_k - cb_k$. In that case, we obtain a peak of storage $S_{k+1}(\texttt{MinIO}) = (2+\epsilon)M_0$ and a volume of *I/O* $V^{I/O}_{k+1}(\texttt{MinIO}) = \epsilon M_0 + V^{I/O}_k(\texttt{MinIO})$.

Recursively, we may build a tree T_n by applying n times this recursive procedure. As we had $V^{I/O}_0(\texttt{MinMEM}) = V^{I/O}_0(\texttt{MinIO}) = M_0$, we conclude that $V^{I/O}_{\texttt{MinMEM}}(n) = nM_0/3 + M_0$ while $V^{I/O}_{\texttt{MinIO}}(n) = n\epsilon M_0 + M_0$. We have thus:

$$\frac{V^{I/O}_n(\texttt{MinMEM})}{V^{I/O}_n(\texttt{MinIO})} = \frac{(1+n/3)}{(1+n\epsilon)}$$

Fixing $n = \lceil 6r \rceil$ and $\epsilon = 1/\lceil 6r \rceil$ we finally get as intended:

$$\frac{V^{I/O}_n(\texttt{MinMEM})}{V^{I/O}_n(\texttt{MinIO})} \geq r$$

□

The proof in the *last-cb in-place* case is exactly the same except that T_0 has a third leaf, say (c), with the same characteristics as (a) and (b).

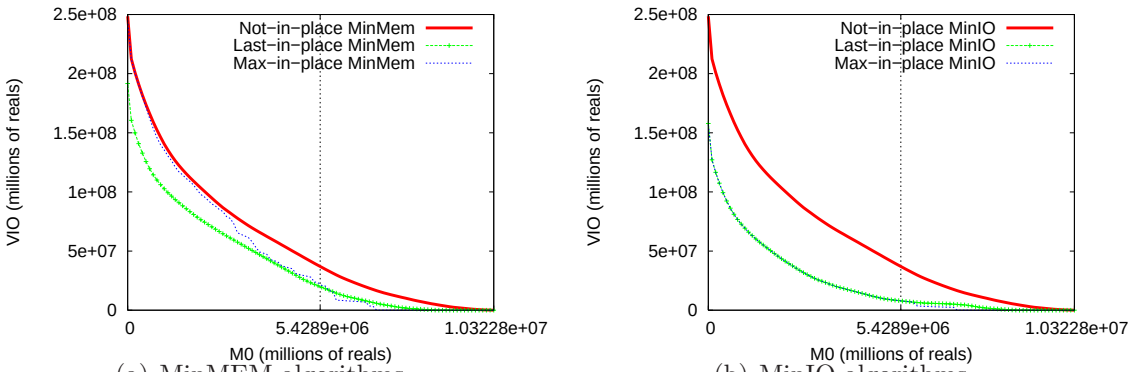
6 Experimental results

In this section we experiment the behaviour of strategies presented in Sections 2, 3, and 4 on different matrices issued from the Parasol, Rutherford-Boeing or university of Florida

collections. The matrices used, numbered from 1 to 30, are: `AUDIKW_1`, `BCSSTK`, `BMWCRA_1`, `BRGM`, `CONESHL_MOD`, `CONV3D_64`, `GEO3D-20-20-20`, `GEO3D-50-50-50`, `GEO3D-80-80-80`, `GEO3D-20-50-80`, `GEO3D-25-25-100`, `GEO3D-120-80-30`, `GEO3D-200-200-200`, `GUPTA1`, `GUPTA2`, `GUPTA3`, `MHD1`, `MSDOOR`, `NASA1824`, `NASA2910`, `NASA4704`, `SAYLR1`, `SHIP_003`, `SPARSINE`, `THERMAL`, `TWOTONE`, `ULTRASOUND3`, `ULTRASOUND80`, `WANG3` and `XENON2`. Matrices `GEO3D*`, `BRGM` and `CONV3D_64` come from `Geosciences Azur`, `BRGM`, and `CEA-CESTA (code AQUILON)`, respectively.

We used several ordering heuristics, that, for a given matrix, define the task dependency graph (or assembly tree) and impact the computational complexity. The volumes of *I/O* were computed by instrumenting the analysis phase of `MUMPS` [5] (for MUltifrontal Massively Parallel Solver) which allowed us to experiment four ordering heuristics: `AMD`, `AMF`, `METIS` and `PORD`. The matrices have a size from very small up to very large (a few million equations) and can lead to huge factors (and storage requirements). For example, the factors of matrix `CONV3D_64` with `AMD` ordering represent 53 *GB* of data.

As previously mentioned, the *I/O* volume depends on the amount of core memory available. Figure 4 illustrates this general behaviour on a sample matrix, `TWOTONE` ordered with `PORD`, for the 3 assembly schemes presented above, for both `MinMEM` and `MinIO` algorithms. For all assembly schemes and algorithms used, we first notice that exploiting all

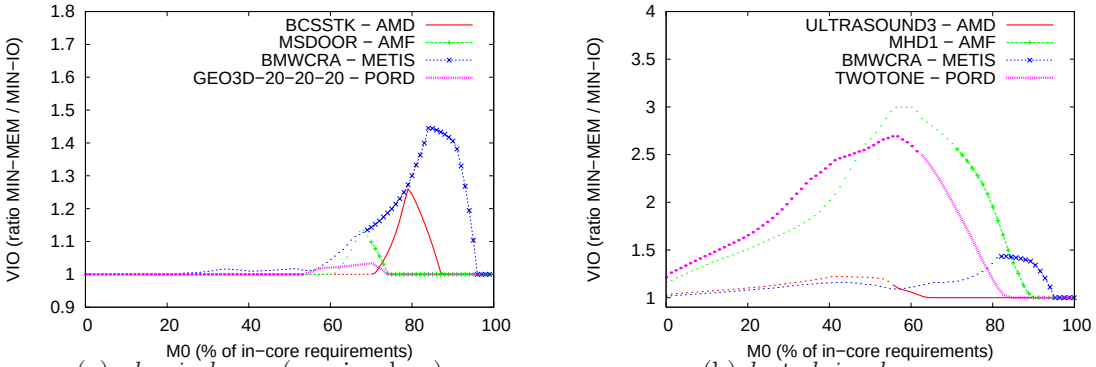


(a) MinMEM algorithms (b) MinIO algorithms

Figure 4: I/O volume on matrix `TWOTONE` with `PORD` ordering as a function of the core memory available, for the 3 assembly schemes presented above, for both `MinMEM` and `MinIO` algorithm. The vertical bar represents the size of the largest frontal matrix.

the available memory is essential to limit the *I/O* volume. Before discussing the results we remind the reader that the *I/O* volumes presented are valid under the hypothesis that the largest frontal matrix may hold in-core. With a core memory lower than this value (*i.e.* the area on the left of the vertical bar in Figure 4), the *I/O* volumes presented are actually lower bounds on the effective *I/O* volume. They are computed as if we could process the out-of-core frontal matrices with a read-once write-once scheme. However, it remains meaningful because the extra-cost due to the specific treatment of frontal matrices will be independent of the assembly scheme used. We first notice that the *last-cb in-place* assembly schemes strongly decrease the amount of *I/O* compared to the *classical* assembly

schemes. In fact, using an *in-place* assembly scheme is very useful in an out-of-core context: it divides the *I/O* volume by more than 2 on most of our test matrices. With the *classical* assembly scheme (presented in Section 2) we observe (on this particular matrix) that the `MinIO` and `MinMEM` algorithms produce the same *I/O* volume (their graphs are identical). Let us come back to Formula (3) to explain this behaviour. We have minimized $\max\left(\max_{j=1,n}(A_j + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1}^{n} cb_k\right)$ by minimizing the first member, because the second one is constant. But if the second member surpasses the first, it becomes useless to decrease the value of the first one. In other words, the largest the frontal matrices (m in the formula) compared to the other metrics (contribution blocks cb_k and active memory requirements for the subtrees A_j), the lowest the probability that reordering the children will impact the *I/O* volume is. From the list of matrices presented above, we have extracted four cases (one for each ordering strategy) for which the gains are significant and we report them in Figure 5(a). To better illustrate the gains resulting from the `MinIO` algorithm, we



(a) *classical* case (non in-place) (b) *last-cb in-place* case

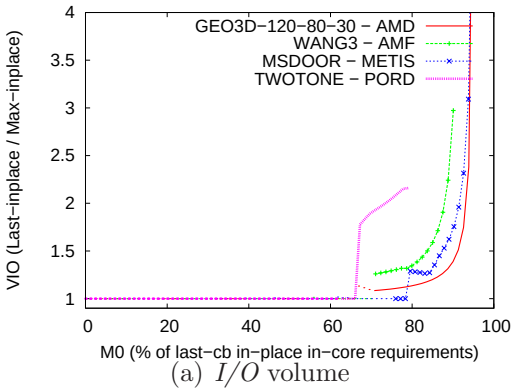
Figure 5: *I/O* volume obtained with `MinMEM` algorithm divided by the one obtained with `MinIO` algorithm. For each matrix/ordering, the filled (right) part of the curve matches the area where the amount of core memory is larger than the size of the largest frontal matrix, while the dotted (left) part matches the area where this amount is lower. For each matrix, we normalized the memory (x-axis) to the in-core minimum requirement (for the given assembly scheme).

analyze the *I/O* volume ratios as a function of the amount of core memory available (in percentage of the core memory requirements). For instance, a value of $(x = 80\%, y = 1.3)$ means that `MinMEM` leads to 30% more volume of *I/O* when 80% of the core memory is provided. Values lower than 1 are not possible because `MinIO` is optimal.

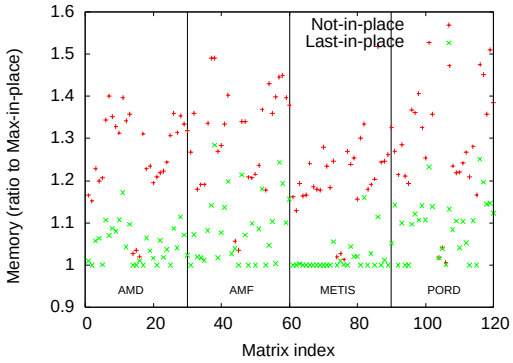
We now focus on the *in-place* assembly scheme (described in Section 3). As we may not show the graphs obtained for our whole collection of matrices, we decided to present in Figure 5(b) again four cases (one for each ordering strategy) for which `MinIO` was much more efficient than `MinMEM` (*I/O* volume was divided for instance by more than 2 for a large range of core memory amounts on `MHD1-AMF` matrix). In general `MinIO` decreases the *I/O* volume, especially when the matrices are pre-processed with orderings which tend to build

irregular assembly trees (like `AMF` and `PORD` and to a lesser extent `AMD` - see [11] for more details). Indeed, first `MinIO` has a stronger impact in the case of an *in-place* assembly scheme because the term $\sum_{k=1}^{n} cb_k$ in Formula (3) is decreased (see Formula (5)). This leaves more freedom to order the children and makes the storage requirement and volume of *I/O* much more sensitive to the order of the children. Second, with large differences in the values of the metrics (size of contribution blocks, storage requirements for subtrees), there is a higher probability to be sensitive to the order of children. That is why our algorithms can provide larger gains in the case of ordering heuristics that produce irregular trees.

We show in Figure 6(a) by how much the `MinIO` algorithm with a *max-cb in-place*



(a) *I/O* volume



(b) Storage requirements (`MinMEM` algorithms). Thirty test matrices are ordered with four ordering heuristics (`AMD`, `AMF`, `METIS` and `PORD`), separated by vertical bars.

Figure 6: Impact of *max-cb in-place* assembly scheme.

assembly scheme improved the `MinIO` *last-cb in-place* one, again on four matrices of the collection (one for each ordering heuristic) for which we observed large gains. An extensive study has shown that, again, the highest profits are usually obtained on irregular assembly trees. To extend a contribution block different from the last one, this block must be kept in memory. But when the core memory available decreases, keeping that data in-core may become a handicap. In this case the `MinIO` heuristic for the *max-cb in-place* assembly scheme switches (as explained in Section 4) to a *last-cb in-place* scheme. Thus, with a small amount of core memory, the *last-cb in-place* and *max-cb in-place* `MinIO` heuristics have a similar behaviour (the left part of their curves are identical in Figure 4(b); the ratio is equal to 1 in Figure 6(a)).

Finally, Figure 6(b) shows that the peak of storage (critical for the in-core case) can also be decreased significantly. This allows us to interpret the extreme right parts of the curves in Figure 6(a) which tend to (or are equal to) infinity: the *max-cb in-place* assembly scheme does *not* induce *I/O* while the *last-cb in-place* scheme *does*.

7 Memory management algorithms

The different `MinMEM` and `MinIO` algorithms presented in this paper provide a particular postorder of the assembly tree. They are executed during the analysis phase of a sparse direct solver. Then the numerical factorization phase relies on this traversal to respect the forecasted optimal metrics (memory usage, *I/O* volume).

In this section we suppose that a postorder has been given (thanks to one of the algorithms presented earlier) and we present some memory management algorithms for the numerical factorization phase that match the different assembly schemes we have considered. The objective is to show that our models are meaningful and that they can lead to a reasonable implementation during the numerical factorization phase. A particular attention is paid to avoid complicated garbage collection mechanisms and extra memory copies.

Remember that the factors are written to disk on the fly (this can be done via a small intermediate buffer, panel by panel). Thus we only have to store frontal matrices and contribution blocks. We propose to use a preallocated workarray W of size M_0 (the amount of core memory available). In this workarray, we manage one or two stacks depending on the assembly scheme used. One stack (StackL) is on the left of the workarray from index botL $= 1$ to index topL; botL $=$ topL means an empty stack. The other one (StackR) is on the right, reversed, from botR $= M_0$ to topR, as illustrated in Figure 7.

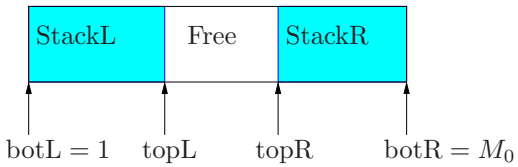


Figure 7: Subdivision of the main workarray, W, into two stacks.

7.1 In-core stack memory

In the three following subsections, we describe possible memory management algorithms for the in-core case, *i.e.* when the storage requirement is smaller than the memory available M_0.

7.1.1 Classical assembly scheme

The memory management for the in-core *classical* assembly scheme is described in Algorithm 1. One stack suffices; we arbitrarily use StackR.

7.1.2 In-place assembly of the last contribution block

Algorithm 2 describes a memory management mechanism for the in-core *last-cb in-place* assembly scheme. In that case, the memory for the parent node can overlap the top of

topR $\leftarrow M_0$ (Initialization);
foreach *node* $\mathcal{N}$ *(in the given postorder)* **do**
  % $\mathcal{N}$ has a frontal matrix of size m, produces a contribution block of size cb, and is parent of a family involving contributions $(cb_j)_{j=1,\ldots,n}$
  Allocate (reserve) m locations in $W(1 : topR)$, for instance $W(1 : m)$, for the frontal matrix of node $\mathcal{N}$;
  for $j = n$ **downto** 1 **do**
    Assemble contribution block of child j, available in $W(\text{topR} + 1 : \text{topR} + cb_j)$, into the frontal matrix of $\mathcal{N}$;
    topR $\leftarrow$ topR $+ cb_j$;
  Factor frontal matrix of $\mathcal{N}$ and write factors to disk on the fly;
  if $cb \neq 0$ **then**
    Move the contribution block produced, of size cb, to $W(\text{topR} - cb + 1 : \text{topR})$ (*);
    topR $\leftarrow$ topR $- cb$;

Algorithm 1: Memory management for the in-core *classical* assembly scheme. Note that the initial and final positions of cb may overlap when performing (*). In that case the copy must start from the right-most part of the contribution block.

StackR. One stack is still enough.

7.1.3 In-place assembly of the largest contribution block

Our new *max-cb in-place* assembly scheme consists in overlapping the memory of the parent with the memory of the largest child contribution block. As this contribution block is not necessarily the one of the last child processed, we need two stacks. The first stack (StackR, say) stores the largest contribution block while the other one (StackL) stores the normal contribution blocks. Storing the largest contribution block on the right allows to keep the property that only StackR is concerned with overlapping the top of the stack with the current frontal matrix. We propose a corresponding memory management mechanism in Algorithm 3.

7.2 Out-of-core extension

We now assume that the stack(s) of contribution blocks may be written to disk when needed (while the frontal matrices are still processed in-core).

7.2.1 Cyclic memory management: dynamic bottom-up approach

In the *classical* and *last-cb in-place* cases, a natural extension consists in substituting StackR by a cyclic stack memory. From a conceptual point of view, the cyclic memory

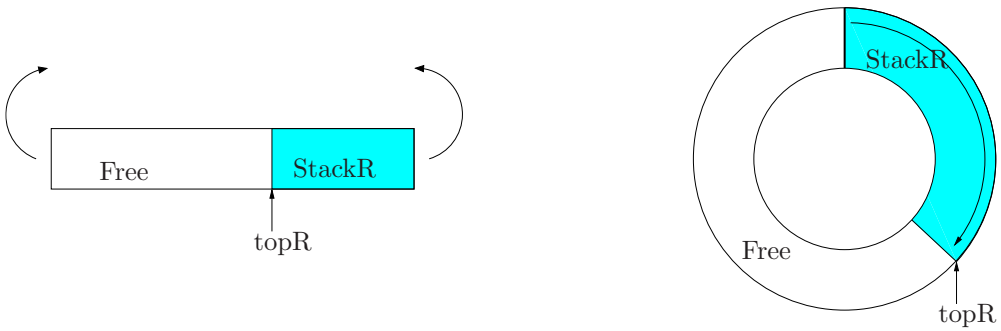
```
topR ← M_0 (Initialization) ;
foreach node N (in the given postorder) do
    % N has a frontal matrix of size m, produces a contribution block
      of size cb, and is parent of a family involving contributions
      (cb_j)_{j=1,...,n}
    if N is a leaf (n = 0) then
        Allocate (reserve) m locations in W(1 : topR) for the frontal matrix of node
        N;
    else
        % W(topR + 1 : topR + cb_n) contains the contribution block of the
          last child
        Allocate (reserve) m locations in W(1 : topR + cb_n) for the frontal matrix of
        node N;
        % Note that m and cb_n can overlap
        Expand (scatter) W(topR + 1 : topR + cb_n) in the frontal matrix of N ;
        topR ← topR + cb_n (but current frontal matrix is still in memory);
        for j = n − 1 downto 1 do
            Assemble contribution block W(topR + 1 : topR + cb_j) into the frontal
            matrix of N ; topR ← topR + cb_j;
    Factor N and write factors to disk on the fly;
    if cb ≠ 0 then
        Move the contribution block produced, of size cb, to W(topR − cb + 1 : topR)
        ;
        topR ← topR − cb;
```

Algorithm 2: Memory management for the in-core *last-cb in-place* assembly scheme.

```
topL ← 1; topR ← M_0 (Initialization);
foreach node N (in the given postorder) do
    % N has a frontal matrix of size m, produces a contribution block
      of size cb, and is parent of a family involving contributions
      (cb_j)_{j=1,...,n}
    if N is a leaf then
        Allocate (reserve) m locations in W(topL : topR) for the frontal matrix of
        node N ;
    else
        % W(topR + 1 : topR + cb_maxi) contains the largest child
          contribution block
        Allocate (reserve) m locations in W(topL : topR + cb_maxi) for the frontal
        matrix of node N;
        % Note that m and cb_maxi may overlap
        Expand (scatter) W(topR + 1 : topR + cb_maxi) in the frontal matrix of N ;
        topR ← topR + cb_maxi (but current frontal matrix is still in memory) ;
        for j = n downto 1, j ≠ maxi do
            Assemble contribution block W(topL − cb_j : topL − 1) into the frontal
            matrix of N ;
            topL ← topL − cb_j ;
    Factor frontal matrix of N and write factors to disk on the fly;
    if cb ≠ 0 then
        if N has the largest cb among its siblings or is the only child then
            Move the contribution block produced, of size cb to
            W(topR − cb + 1 : topR);
            topR ← topR − cb;
        else
            Move the contribution block produced to W(topL : topL + cb − 1) ;
            topL ← topL + cb;
```

Algorithm 3: Memory management for the in-core *max-cb in-place* assembly scheme.

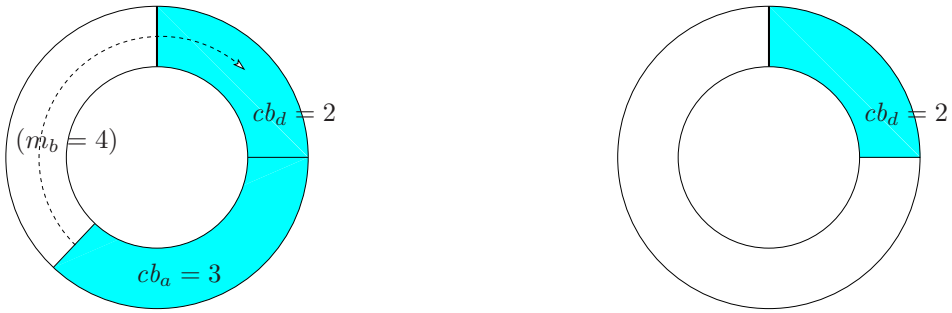
management is obtained by joining the end of the memory zone to its beginning as illustrated in Figure 8. When the free space vanishes, a part of the bottom of the stack is written to disk and this space is reused to store the new data produced. The decision to free a part of the bottom of the stack is taken dynamically, when the memory is almost full. We illustrate this on the sample tree of Figure 1 processed in the postorder (d-a-b-c-e) with a *classical* assembly scheme. After processing nodes (d) and (a), one only discovers that *I/O* has to be performed on the first contribution block produced (cb_d) at the moment of allocating the frontal matrix of (b), of size $m_b = 8$ (see Figure 9(a)).



(a) Workarray (b) Cyclic workarray

Figure 8: Folding a linear workarray (left) into a cyclic workarray (right).

Note that one drawback of this approach is that a specific management has to be applied to the border when a contribution block or a frontal matrix is split on both sides of the memory area (as occurs for frontal matrix m_b in Figure 9(a)). Moreover, in the



(a) Bottom-up approach (b) Top-down approach

Figure 9: Memory state while dealing with the subtree rooted at (c) when processing the sample tree of Figure 1 in the postorder (d-a-b-c-e). With a bottom-up approach (left), one knows that *I/O* will be performed on cb_d only before to deal with node (b). With a top-down approach (right), we know it *a priori* (thanks to metrics computed during the analysis phase).

max-cb in-place case, such an extension is not as natural. That is why we propose in the next subsection another approach, which allows to handle efficiently the *max-cb in-place* assembly scheme and avoids a specific management of the borders for the *classical* and *last-cb in-place* cases.

7.2.2 Using pre-computed metrics: static top-down formulation

In order to minimize the *I/O* volume in the previous approach, a contribution is only written to disk when the memory appears to be full: the decision of writing a contribution block (or a part of it) is taken dynamically. However, one should notice that the amount of contributions $V_{family}^{I/O}$ that will have to be written to disk among the direct children of a given family can be known much earlier (in fact, during the analysis phase). This value is given by the first member (the recursive amount of *I/O* on the subtrees is not counted) of Formulas (3) and (5) respectively for the *classical* and *in-place* cases. $V_{family}^{I/O}$ is thus defined by Formula (6) when a *classical* assembly scheme is used

$$V_{family}^{I/O} = \max\left(0, \max(\max_{j=1,n}(A_j + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1}^{n} cb_k) - M_0\right) \tag{6}$$

and by Formula (7) in the *in-place* cases:

$$V_{family}^{I/O} = \max\left(0, \max_{j=1,n}(\max(A_j, m) + \sum_{k=1}^{j-1} cb_k) - M_0\right) \tag{7}$$

Thanks to this information, one may thus actually know *a priori* that the contribution block of node (d) (cb_d) will have to be written to disk completely (see Figure 9(b)). Indeed, we already know that the subtree rooted at (e) (the parent of node (d)), requires an amount of 3 units of *I/O* among the contributions of its direct children. As the oldest contributions are written first, cb_d will have to be (fully) written.

Using a top-down formulation, more natural in this context, we now present a recursive out-of-core algorithm, `AlgoOOC_rec`(tree T, workarray W of size M_0), which anticipates the *I/O* operations. Formally defined in Algorithm 4, it proceeds as follows. The algorithm starts from the value of the *I/O* volume $V_{family}^{I/O}$ for the family composed of the root node and its direct children. This value was pre-computed during the analysis phase and corresponds to the amount of *I/O* that will be performed on the first contribution blocks produced to respect the rule consisting in writing the oldest contributions first. The subtrees are then processed in the order forecasted by the analysis phase $(1, \ldots, n)$. If a subtree T_j fits individually in the workarray ($S_j \leq M_0$), it is then processed in-core. By construction, we have written enough contributions to disk, and the ones that are left in memory are compactly stored in the right-most part of the workarray. Therefore, we have: (`property` 1) topR $\geq S_j$. Subsequently, a contiguous free memory space of size (at least) S_j is free on the left part of the workarray in which T_j is processed (call to `AlgoIC`(T_j, $W(1 : \text{topR})$)). Note that we use the generic notation `AlgoIC`(subtree T, contiguous memory zone V)) for

any of the algorithms 1, 2 or 3, applied to a given subtree T_j, using a contiguous memory zone V. Compared to the algorithms described in Section 7.1, the only differences are that (i) V can be of size smaller than M_0, and (ii) a contribution block is produced at the root of the subtree, which is available in the right-most part of the workarray V. If T_j does not hold in-core ($S_j > M_0$), the whole memory (`property` 2) is used for recursively processing T_j out-of-core (call to `AlgoOOC_rec`(T_j, W)). `Properties` 1 and 2 (ensuring respectively that (`property` 1) a subtree which fits individually in-core may be processed in a large enough contiguous memory zone and that (`property` 2) the whole memory is available for a subtree which does not fit in-core) are valid because the contributions are written as soon as possible. Indeed, each time a contribution block (or part of it, of size v_j) from a direct child is written to disk, the pending amount of contributions to be written (v in Algorithm 4) is decreased ($v \leftarrow v - v_j$). Thus we maintain the information that a newly produced contribution has to ($v > 0$), or does not have to ($v = 0$), be written to disk. Once all the subtrees have been processed, the contributions from the direct children are assembled into the frontal matrix of the root node. Data already in-core are assembled first. Then, data on disk are loaded into memory (possibly by panels) and assembled in turn. The size p of a panel is at most $p = M_0 - m$ (the remaining space in the workarray, once the frontal matrix has been allocated). This recursive algorithm is initially called on the whole tree using the whole workarray (see Algorithm 5).

This top-down approach allows in particular to handle the out-of-core heuristic presented in Section 4 for the *max-cb in-place* assembly scheme. The switch mechanism is indeed naturally handled: when `AlgoOOC_rec`() is called, the last contribution block is extended *in-place* whereas the largest contribution block is extended when `AlgoIC`() is called.

The key point of this algorithm is that `properties` 1 and 2 ensure that a cyclic memory management is not required anymore. A simple linear workarray is used. Thus data are not split on both sides of the memory area. Moreover this mechanism does not imply any extra memory copy and does not perform more *I/O* than forecasted at the analysis phase.

This algorithm relies on the use of metrics computed during the analysis phase (the values of $V^{I/O}_{family}$ and S for each non leaf node of the tree). In static codes which *do* respect the forecasted metrics, they can thus be implemented as they are presented here. In more dynamic codes (allowing for dynamic pivoting) which do *not* respect exactly the forecasted metrics, a specific treatment (emergency *I/O*, ...) will be required when the storage effectively used by a subtree is larger than forecasted. Another possibility consists in relaxing the forecasted metrics, but this implies extra, possibly unnecessary, *I/O*.

8 Conclusion and on-going work

Table 1 summarizes the contributions of this paper. We have reminded the existing memory-minimization algorithms for the *classical* and *last-cb in-place* assembly schemes. We have then shown that these algorithms are not optimal to minimize the *I/O* volume and that they can be arbitrarily bad. We have proposed optimal algorithms for the *I/O* volume

```
% Initialization: V^{I/O}_{family} is the part of sum_j cb_j (contribution blocks
  from the direct children of T) that will be written to disk; it is
  initially equal to v, the amount of these contributions not yet
  written;
v ← V^{I/O}_{family} (see Formulas (6) and (7));
topR ← M_0;
for j = 1 to n do
    if S_j ≤ M_0 (T_j can be processed in-core in W) then
        % Property 1: topR ≥ S_j
        AlgoIC(T_j, W(1 : topR)) ;
    else
        % Property 2: topR = M_0
        AlgoOOC_rec(T_j, W) ;
    % On exit, cb_j is stored in W(topR − cb_j + 1 : topR), update topR
    topR ← topR − cb_j ;
    v_j =def min(v, cb_j);
    Write to disk W(topR + 1 : topR + v_j) ;
    v ← v − v_j ;
    topR ← topR + v_j ;
Allocate (reserve) m locations in memory, for instance in W(1 : m), for the frontal
matrix of the root of T* ;
for j = n downto 1 do
    Assemble cb_j in the frontal matrix of the root of T (reading from disk v_j units of
    data, possibly by panels) ;
    topR ← topR + cb_j − v_j ;
if cb ≠ 0 then
    Store the contribution block (of size cb) of subtree T in W(M_0 − cb + 1 : M_0);
```

Algorithm 4: AlgoOOC_rec(subtree T, workarray W of size M_0).
* The algorithm is presented for the *classical* assembly scheme. In the *in-place* cases, $W(1:m)$ can overlap with the last contribution block $W(\text{topR}+1 : \text{topR}+cb_n)$. Note that this is true even with the *max-cb in-place* scheme, which switches (see Section 4) to *last-cb in-place* as T is out-of-core.

```
if S_T ≤ M_0 (T can be processed in-core in W) then
    AlgoIC(T, W) ;
else
    AlgoOOC_rec(T, W) ;
```

Algorithm 5: AlgoOOC(tree T, workarray W of size M_0).

Assembly scheme	Algorithm	Objective function		Memory management	
		Memory minimization	**I/O minimization**	In-core	**Out-of-core**
classical	`MinMEM`	• Optimum ([11], adapting[13])	• **Arbitrarily bad in theory** • **Reasonable in most cases**	One stack	**Cyclic or top-down**
	MinIO	• Not suited	• **Optimum**		
last-cb in-place	`MinMEM`	• Optimum[13]	• **Arbitrarily bad in theory** • **Bad in practice on some irregular assembly trees**	One stack	**Cyclic or top-down**
	MinIO	• Not suited	• **Optimum**		
max-cb in-place	**MinMEM**	• **Optimum**	• Not suited	**Two stacks**	**Top-down**
	MinIO	• **Optimum**	• **Efficient heuristic**		

Table 1: Summary. Contributions of this paper are in bold.

minimization and have shown that significant gains could be obtained on real problems (especially with the *in-place* assembly scheme). We have then proposed a new assembly scheme (which consists in extending the child with the largest contribution block) and a corresponding tree traversal which is optimal to minimize memory and leads to an efficient heuristic when the objective is to minimize the *I/O* volume. From a practical point of view, we have shown that efficient memory management schemes (not inducing extra core memory traffic) could be obtained for all variants, and have proposed algorithms that appear to be reasonable to implement.

This work is particularly important when applied to large-scale problems (millions of equations) in limited-memory environments (which is actually always the case, even on high-end platforms). It is applicable for shared-memory solvers relying on threaded `BLAS` libraries. In a parallel distributed context, it will help to limit memory requirements and to decrease the *I/O* volume in the sequential (often critical) parts of the computations.

We are currently working on adapting this work to a flexible task allocation scheme, where the parent node is allowed to be allocated before all children have been processed [10]. Again, instead of limiting the storage requirement of the methods, the goal consists in minimizing the volume of *I/O* involved. The work presented in this paper is a basis to this new and more difficult flexible context.

Acknowledgement

We are grateful to Patrick Amestoy for his comments on a preliminary version of this report.

References

[1] The BCSLIB Mathematical/Statistical Library. http://www.boeing.com/phantom/bcslib/.

[2] E. Agullo, A. Guermouche, and J.-Y. L'Excellent. A preliminary out-of-core extension of a parallel multifrontal solver. In *EuroPar'06 Parallel Processing*, pages 1053–1063, 2006.

[3] E. Agullo, A. Guermouche, and J.-Y. L'Excellent. Parallel out-of-core factorization of large sparse matrices with a multifrontal approach. *Parallel Computing, Special Issue on Parallel Matrix Algorithms*, Submitted, 2007.

[4] E. Agullo, A. Guermouche, and J.-Y. L'Excellent. Towards a parallel out-of-core multifrontal solver: Preliminary study. Research report 6120, INRIA, 02 2007. Also appeared as LIP report RR2007-06.

[5] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, 32(2):136–156, 2006.

[6] P. R. Amestoy and C. Puglisi. An unsymmetrized multifrontal LU factorization. *SIAM Journal on Matrix Analysis and Applications*, 24:553–569, 2002.

[7] F. Dobrian. *External Memory Algorithms for Factoring Sparse Matrices*. PhD thesis, Old Dominion University, 2001.

[8] F. Dobrian and A. Pothen. Oblio: a sparse direct solver library for serial and parallel computations. Technical report, Old Dominion University, 2000.

[9] I. S. Duff and J. K. Reid. MA27—a set of Fortran subroutines for solving sparse symmetric sets of linear equations. Technical Report R.10533, AERE, Harwell, England, 1982.

[10] A. Guermouche and J.-Y. L'Excellent. Constructing memory-minimizing schedules for multifrontal methods. *ACM Transactions on Mathematical Software*, 32(1):17–32, 2006.

[11] 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.

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

[13] 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.

[14] J. W. H. Liu. The multifrontal method and paging in sparse Cholesky factorization. *ACM Transactions on Mathematical Software*, 15:310–325, 1989.

[15] J. K. Reid and J. A. Scott. An out-of-core sparse Cholesky solver. Technical report, Rutherford Appleton Laboratory, 2006.

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

[17] Vladimir Rotkin and Sivan Toledo. The design and implementation of a new out-of-core sparse Cholesky factorization method. *ACM Trans. Math. Softw.*, 30(1):19–46, 2004.