On the I/O Volume in Out-of-Core Multifrontal Methods with a Flexible Allocation Scheme*

Emmanuel Agullo[1,6,3,7], Abdou Guermouche[5,4,8], and Jean-Yves L'Excellent[2,6,3,9]

[1] École Normale Supérieure de Lyon
[2] Institut National de Recherche en Informatique et en Automatique (INRIA)
[3] Laboratoire de l'Informatique du Parallélisme (UMR CNRS-ENS Lyon-INRIA-UCBL), ÉNS Lyon, 46 allée d'Italie, 69364 Lyon cedex 07, France
[4] Laboratoire Bordelais de Recheche en Informatique (UMR 5800) - 351, cours de la Liberation F-33405 Talence cedex, France
[5] Université de Bordeaux 1
[6] Université de Lyon
[7] Emmanuel.Agullo@ens-lyon.fr
[8] Abdou.Guermouche@labri.fr
[9] Jean-Yves.L.Excellent@ens-lyon.fr

Abstract. Sparse direct solvers, and in particular multifrontal methods, are widely used in various simulation problems. Because of their large memory requirements, the use of *out-of-core* solvers is compulsory for large-scale problems, where disks are used to extend the core memory. This study is at the junction of two previous independent works: it extends the problem of the minimization of the volume of *I/O* [3] in the multifrontal method to the more general *flexible parent allocation* algorithm [7]. We explain how to compute the *I/O* volume with respect to this scheme and propose an efficient greedy heuristic which significantly reduces the *I/O* volume on real-life problems in this new context.

1 Introduction

We are interested in the direct solution of systems of equations of the form $Ax = b$, where A is a large sparse matrix. In direct methods, because the storage requirements are large compared to the initial matrix A, out-of-core approaches may become necessary. In such cases, left-looking [12, 13] and multifrontal [1, 11] methods are the two most widely used approaches. One drawback of multifrontal methods comes from large dense matrices that give a lower bound on the minimum core memory requirements. However, those dense matrices may fit in memory, or they can be treated with an out-of-core process. Apart from these dense matrices, the out-of-core multifrontal method follows a write-once/read-once scheme, which makes it interesting when one is interested in limiting the volume of *I/O*. For matrices A with a symmetric structure (or in approaches like [5] when the structure of A is unsymmetric), the dependency graph of the multifrontal approach is given by a tree, processed from leaves to root. The tree structure results from the sparsity of the matrix and from the order in which the variables

* Partially supported by ANR project SOLSTICE, ANR-06-CIS6-010.

of the sparse matrix are eliminated; different branches of the tree may be processed independently. To each node of the tree is associated a so called *frontal matrix*, which is a dense matrix divided into two parts (see Figure 1, left): (i) a *fully summed block*, that will be factorized, and a non-fully summed block that cannot be factorized yet but will be updated and used later at the parent node, after it has been replaced by a *Schur complement*, or *contribution block*. To be more precise, the following tasks are performed at each node of the tree:

(i) allocation of the *frontal matrix* in memory;
(ii) assembly of data (*contribution blocks*) coming from the child nodes into that frontal matrix;
(iii) partial factorization of the fully summed part of the frontal matrix, and update of the rest.

After step (iii), the fully summed part of the frontal matrix has been modified and contains *factors*, that will only be reused at the solution stage, whereas the non fully summed part contains the Schur complement, that will contribute to the frontal matrix from the parent node (see Figure 1, right). Because factors are not re-accessed during the multifrontal factorization, they can be written to disk directly, freeing some storage. Then remains the temporary storage associated to the contribution blocks waiting to be assembled and to the current frontal matrix. In the classical multifrontal scheme, the frontal matrix of a parent is allocated (and then factored) only after all children have been processed. We call this approach *terminal allocation*. Assuming that a postorder of the tree is used, contribution blocks can then be managed thanks to a stack mechanism. Furthermore, the order in which the children are processed (or tree traversal) has a strong impact on both the storage requirement for the stack and the volume of *I/O*, should this stack be processed *out-of-core*.

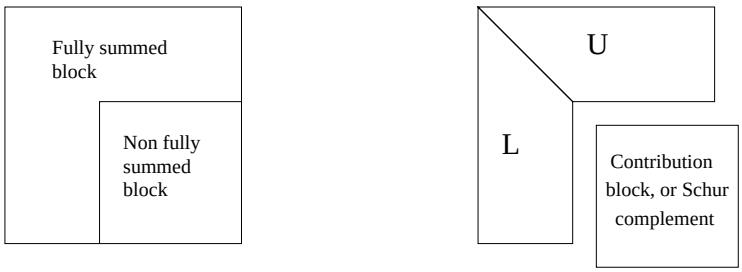


Fig. 1. Frontal matrix at a node of the tree before (left) and after (right) the partial factorization of step (iii) in the unsymmetric case (LU factorization).

A more extensive description of the multifrontal approach can be found in, for example, [6, 10]. In general a large workarray is pre-allocated, in order to store the current frontal matrix and the contribution blocks. Allowing the frontal matrix of the parent to overlap with the contribution block of the last child is possible, and significantly reduces the overall storage requirement. Considering a so-called *family* composed of a parent

node, with a frontal matrix of size m, and its set of n children that produce contribution blocks of size $cb_i,\ i = 1, \ldots, n$, [9] shows that the storage requirement to process the tree rooted at the parent is

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

(where $S_j^{terminal}$ is recursively the storage for the subtree rooted at child j) and can be minimized by sorting the children in decreasing order of $\max(S_j^{terminal}, m) - cb_j$. By applying this formula and this ordering at each level of the tree, we obtain the volume of *I/O* for the complete tree, together with the tree traversal. Starting from (1), we have shown in [3] that for a given amount of available memory, M_0, the volume of *I/O* (=volume written=volume read) associated to the temporary storage of the multifrontal method is

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

which is minimized by sorting the children in decreasing order of

$$\max(\min(S_j^{terminal}, M_0), m) - cb_j$$

at each level of the tree. This gives an optimal tree traversal which is different from the one from [9]: minimizing the *I/O* volume is different from minimizing the overall storage requirement.

2 Flexible parent allocation

With the *terminal allocation* scheme, steps (i), (ii) and (iii) for a parent node are only performed when all children have been processed. However, the main constraint is that the partial factorization (step (iii) above) at the parent level must be performed after the assembly step (ii) has been performed for all child contributions. Thus, the allocation of the parent node (step (i)), and the assembly of the contributions of some children can be performed (and the corresponding contribution block freed) without waiting that all children have been processed. This flexibility has been exploited by [7] to further reduce the storage requirement for temporary data. Assume that the parent node is allocated after p children have been processed, and that the memory for the p^{th} child overlaps with the memory for the parent. The storage required for a parent in this *flexible* scheme is then given by:

$$S^{flex} = \max\left(\max_{j=1,\boxed{p}}(S_j^{flex} + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1}^{\boxed{p}-1} cb_k, m + \max_{j=p+1,n} S_j^{flex}\right) \quad (3)$$

When the parent is allocated, all the contributions from its factored children are assembled and discarded. From that point on, each child that is factored sees its contribution block immediately assembled and its memory is released. [7] shows how to choose the point (*split point*) where the parent should be allocated and how to order the children so that the storage requirement S^{flex} is minimized.

Now if the value of S^{flex} is larger than the available memory, then disk storage must be used. In that case, rather than minimizing S^{flex}, it becomes more interesting to minimize the volume of *I/O*: this is the objective of the current paper. To limit the volume of *I/O*, minimizing S^{flex} can appear like a good heuristic. In [11], the authors have done so, adapting [7] with respect to some additional constraints imposed by their code. However, by computing the volume of *I/O* formally, we can show the limits of a memory-minimizing approach when aiming at decreasing the *I/O* volume: similarly to the terminal allocation case, minimizing the volume of *I/O* in the flexible allocation scheme is different from minimizing the storage requirement.

3 Volume of *I/O* in a flexible multifrontal method

The main difference compared to (2) is that with a *flexible* allocation scheme, a child j processed after the parent allocation ($j > p$) may also generate *I/O*. Indeed, if this child cannot be processed *in-core* together with the frontal matrix of the parent, then part (or the whole) of the frontal matrix has to be written to disk in order to make room and process the child with a maximum of available memory. This possible extra-*I/O* corresponds to underbrace **(a)** of Formula (4). After that, the factor block of the frontal matrix of child j is written to disk and its contribution block is ready to be assembled into the frontal matrix of the parent; to do so, and because we cannot easily rely on a simple property to find which rows of the contribution block, if any, can be assembled into the part of the frontal matrix available in memory, we assume that this latter frontal matrix is fully re-loaded into memory (reading back from disk the part previously written). This operation may again generate *I/O*: if the contribution block of child j and the frontal matrix of its parent cannot hold together in memory, a part of cb_j has to be written to disk, then read back (panel by panel) and finally assembled. This second possible extra-*I/O* is counted in underbrace **(b)** of Formula (4). All in all, and using the storage definition from Formula (3), the volume of *I/O* required to process the subtree rooted at the parent node is given by:

$$V^{flex} = \max\left(0, \max\left(\max_{j=1,\boxed{p}}\left(\min(S_j^{flex}, M_0) + \sum_{k=1}^{j-1} cb_k\right), m + \sum_{k=1}^{\boxed{p}-1} cb_k\right) - M_0\right) + \sum_{j=1}^{n} V_j^{flex} + \underbrace{\sum_{j=p+1}^{n}\Big(\max(0, m + \min(S_j^{flex}, M_0) - M_0)\Big)}_{\textbf{(a)}} + \underbrace{\sum_{j=p+1}^{n}(\max(0, m + cb_j - M_0))}_{\textbf{(b)}} \tag{4}$$

Again, a recursion gives the *I/O* volume for the whole tree.

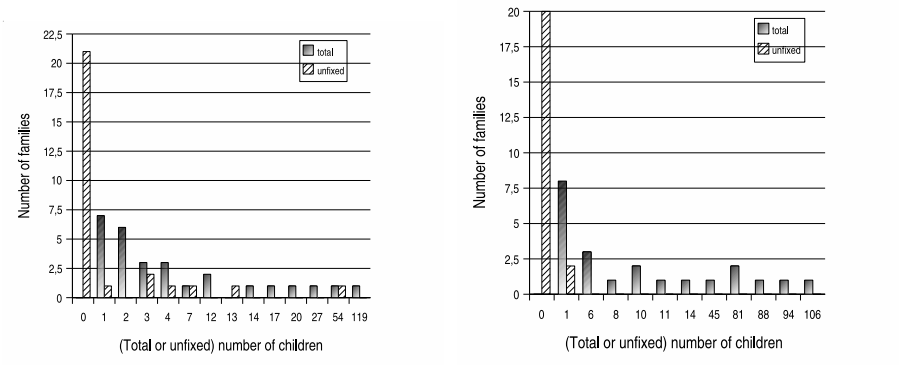
4 Minimizing the *I/O* volume in the flexible multifrontal method

With the terminal allocation scheme, the *I/O* volume (on a parent node and its n children) is minimized by sorting the children in an appropriate order. With the flexible scheme, one should *moreover* determine the appropriate *split point*, *i.e.* the best value for p. In other words, the flexible *I/O* volume is minimized when together **(i)** the children processed *before* the parent allocation are correctly separated from the ones processed *after* and **(ii)** each one of this set is processed in an appropriate order. Exploring these $n.n!$ combinations is not always conceivable since some families may have a very large number n of children (more than one hundred for instance for the `GUPTA3` matrix). However, relying on [3] and Formula 2, we know that an optimal order among children processed before the parent allocation is obtained when they are sorted in decreasing order of $\max(\min(S_j^{flex}, M_0), m) - cb_j$. Moreover, the *I/O* volume on the children processed after the allocation is independent of their relative processing order. Said differently, these two remarks mean that **(ii)** is actually obvious when **(i)** is determined: we only have to determine to which set (before or after the parent allocation) each child belongs to. But this still makes an exponential (2^n) number of possibilities to explore.

Actually, the decision problem associated to this minimization problem is NP-complete. In other words, given an arbitrary target amount of *I/O* V, there is no deterministic polynomial algorithm that can consistently decide whether there exists a partition of the children inducing a volume of *I/O* lower than or equal to V (except if *P = NP*). The proof of the NP-completeness (reduction from `2-PARTITION`) is technical and out-of-scope for this paper.

To further reduce the complexity, remark that if a child is such that $m+S_j^{flex} \leq M_0$, ordering this child *after* the parent allocation does not introduce any additional *I/O* (**(a)** and **(b)** are both 0 in (4)), whereas this may not be the case if it is processed before the parent allocation. Therefore, we conclude that we can place all children verifying $m + S_j^{flex} \leq M_0$ after the parent allocation. Furthermore, consider the case where $S_j^{flex} \geq M_0 - m + cb_j$ and $m + cb_j \leq M_0$. Processing this child *after* the parent allocation (see Formula (4)) leads to a volume of *I/O* either equal to m (if $S_j^{flex} >= M_0$) – which is greater than cb_j, or to $S_j^{flex} - M_0 + m$ (if $S_j^{flex} \leq M_0$) – which is also greater than cb_j. On the other hand, treating that child *first* (this may not be optimal) will lead to a maximum additional volume of *I/O* equal to cb_j. Therefore, we can conclude that we should process it *before* the parent allocation. For the same type of reasons, children verifying $S_j^{flex} \leq 2(M_0 - m)$ and $m + cb_j > M_0$ should also be processed *before* the parent allocation.

We will say that a child is *fixed* if it verifies one of the above properties: a straightforward analysis - independent of the metrics of its siblings - determines if it should be processed before or after the allocation of the parent node. Even though the number of *fixed* children can be large in practice, some matrices may have a few families with a large number of *unfixed* children, as shown in Figure 2 for two sparse problems. For instance, among the 28 families inducing *I/O* for the `GUPTA3` matrix ordered with `METIS` when a memory of $M_0 = 684686$ reals is available, 21 families have no unfixed children (thus for them the optimum process is directly known), but one family keeps having 54 unfixed children. In such cases, heuristics are necessary and we designed one



(a) `GUPTA3` matrix - `METIS` ordering M_0=684686 (b) `TWOTONE` matrix - `PORD` ordering M_0=7572632

Fig. 2. Distribution of the families in function of their total and unfixed number of children. After a straightforward analysis, most families have few (or no) unfixed children.

consisting in moving after the allocation the child which is responsible for the peak of storage until one move does not decrease the volume of *I/O* anymore.

5 Experimental results

In order to evaluate the impact of this flexible allocation scheme on the volume of *I/O*, we compare the results of our heuristic (`Flex-MinIO`) both to the terminal allocation scheme with the IO-minimizing algorithm of [3] (`Term-MinIO`) and to the flexible allocation scheme with the memory-minimizing algorithm of [7] (`Flex-MinMEM`).

The volumes of *I/O* were computed by instrumenting the analysis phase of `MUMPS` [4] which allowed us to experiment several ordering heuristics. We retained results with both `METIS` [8] and `PORD` [14]. For a given matrix, an ordering heuristic defines the order in which the variables of the matrix are eliminated and an associated task dependency graph (or tree, see Section 1). It impacts the computational complexity as well as different metrics such as the volume of *I/O*.

We have selected four test problems that we present in Table 1 and for which we have observed significant gains. Figure 3 shows the evolution of the corresponding volume of *I/O* with the available memory on the target machine. When a large amount of memory is available (right part of the graphs), the flexible allocation schemes (both `Flex-MinMEM` and `Flex-MinIO`) induce a small amount of *I/O* compared to the terminal allocation scheme (`Term-MinIO`). Indeed, with such an amount of memory, many children can be processed after the allocation of their parent without inducing any *I/O* (or inducing a small amount of *I/O*): the possible extra-*I/O*s corresponding to underbraces **(a)** and **(b)** of Formula (4) are actually equal (or almost equal) to zero for those children.

When the amount of available memory is small (left part of the graphs), the memory-minimizing algorithm (`Flex-MinMEM`) induces a very large amount of *I/O* compared

Matrix	Order	nnz	Type	$nnz(L\|U)$ ($\times 10^6$)	Flops ($\times 10^9$)	Description
CONV3D_64	836550	12548250	UNS	4690.6	48520	Provided by CEA-CESTA; generated using AQUILON (http://www.enscpb.fr/master/aquilon).
GUPTA3	16783	4670105	SYM	10.1	6.3	Linear programming matrix (AA'), Anshul Gupta (Univ. Florida collection).
MHD1	485597	24233141	UNS	1169.7	8382	Unsymmetric magneto-hydrodynamic 3D problem, provided by Pierre Ramet.
TWOTONE	120750	1224224	UNS	30.7	39.7	AT&T,harmonic balance method, two-tone. More off-diag nz than onetone (Univ. Florida collection).

Table 1. Test problems. Size of factors ($nnz(L|U)$) and number of floating-point operations (Flops) were computed with PORD ordering, except the ones of GUPTA3 for which METIS ordering was used.

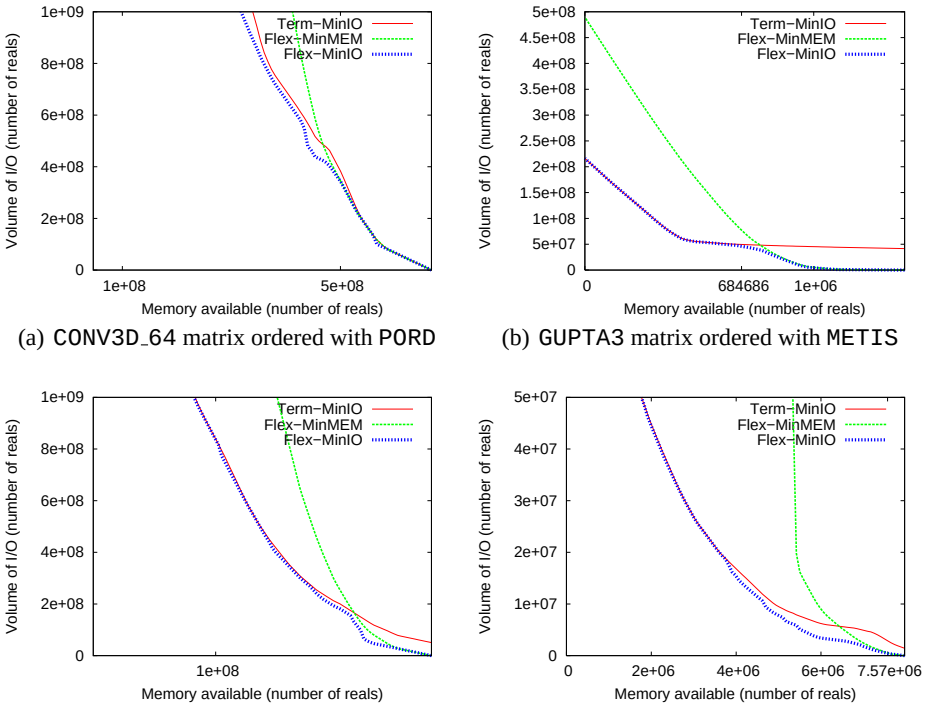
to the *I/O*-minimization algorithms (both Flex-MinIO and Term-MinIO). Indeed, processing a child after the parent allocation may then induce a very large amount of *I/O* (M_0 is small in underbraces **(a)** and **(b)** of Formula (4)) but memory-minimization algorithms do not take into account the amount of available memory to choose the *split point*.

Finally, when the amount of available memory is intermediate, the heuristic we have proposed (Flex-MinIO) induces less *I/O* than both other approaches. Indeed, according to the memory, not only does the heuristic use a flexible allocation scheme on the families for which it is profitable, but it can also adapt the number of children to be processed after the parent allocation.

The algorithms presented in this paper should allow to improve the new generation of serial [11] *out-of-core* multifrontal codes, based on the flexible allocation scheme, as well as the serial parts of the parallel ones [2].

References

1. The BCSLIB Mathematical/Statistical Library. http://www.boeing.com/phantom/bcslib/.
2. E. Agullo, A. Guermouche, and J.-Y. L'Excellent. Towards a parallel out-of-core multifrontal solver: Preliminary study. Research report 6120, INRIA, February 2007. Also appeared as LIP report RR2007-06.
3. Emmanuel Agullo, Abdou Guermouche, and Jean-Yves L'Excellent. On reducing the I/O volume in a sparse out-of-core solver. In *HiPC'07 14th International Conference On High Performance Computing*, number 4873 in Lecture Notes in Computer Science, pages 47–58, Goa, India, December 17-20 2007.
4. P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, 32(2):136–156, 2006.
5. P. R. Amestoy and C. Puglisi. An unsymmetrized multifrontal LU factorization. *SIAM Journal on Matrix Analysis and Applications*, 24:553–569, 2002.
6. 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.



(a) CONV3D_64 matrix ordered with PORD

(b) GUPTA3 matrix ordered with METIS

(c) MHD1 matrix ordered with PORD

(d) TWOTONE matrix ordered with PORD

Fig. 3. I/O volume on the stack of contribution blocks as a function of the core memory available for the three heuristics with four different matrices.

7. A. Guermouche and J.-Y. L'Excellent. Constructing memory-minimizing schedules for multifrontal methods. *ACM Transactions on Mathematical Software*, 32(1):17–32, 2006.
8. 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.
9. 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.
10. J. W. H. Liu. The multifrontal method for sparse matrix solution: Theory and Practice. *SIAM Review*, 34:82–109, 1992.
11. J. K. Reid and J. A. Scott. An out-of-core sparse Cholesky solver. Technical Report RAL-TR-2006-013, Rutherford Appleton Laboratory, 2006. Revised March 2007.
12. E. Rothberg and R. Schreiber. Efficient methods for out-of-core sparse Cholesky factorization. *SIAM Journal on Scientific Computing*, 21(1):129–144, 1999.
13. V. Rotkin and S. Toledo. The design and implementation of a new out-of-core sparse Cholesky factorization method. *ACM Trans. Math. Softw.*, 30(1):19–46, 2004.
14. J. Schulze. Towards a tighter coupling of bottom-up and top-down sparse matrix ordering methods. *BIT*, 41(4):800–841, 2001.