Towards a Parallel Out-of-core Multifrontal Solver: Preliminary Study.

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

Laboratoire de l'Informatique du Parallélisme

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

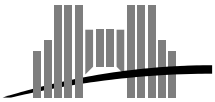
Towards a Parallel Out-of-core Multifrontal Solver: Preliminary Study

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

February 2007

Research Report N° 2007-06

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



Towards a Parallel Out-of-core Multifrontal Solver: Preliminary Study

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

February 2007

Abstract

The memory usage of sparse direct solvers can be the bottleneck to solve large-scale problems involving sparse systems of linear equations of the form $Ax = b$. This report describes a prototype implementation of an *out-of-core* extension to a parallel multifrontal solver (MUMPS), where disk is used to store data that cannot fit in memory. We show that, by storing the factors to disk, larger problems can be solved on limited-memory machines with reasonable performance. We illustrate the impact of low-level IO mechanisms on the behaviour of our parallel *out-of-core* factorization. Then we use simulations to analyze the gains that can be expected when also storing the so called active memory on disk. We discuss both the minimum memory requirements and the minimum volume of *I/O* in a limited memory environment. Finally we summarize the main points that we identified to be critical when designing parallel sparse direct solvers in an *out-of-core* environment.

Keywords: Sparse direct solver; large matrices; multifrontal method; out-of-core; parallel factorization; IO volume; direct IO; performance study

Résumé

Lors de la résolution de systèmes linéaires creux de la forme $Ax = b$, le volume mémoire nécessaire aux méthodes dites directes peut rapidement devenir le goulet d'étranglement pour les problèmes de grande taille. Dans ce rapport, nous décrivons un prototype d'une extension hors-mémoire (*out-of-core*) d'un solveur parallèle multifrontal, MUMPS, où les disques durs sont utilisés pour stocker les données qui ne peuvent pas tenir en mémoire centrale. Nous montrons qu'en stockant les facteurs sur disque, des problèmes de plus grande taille peuvent être traités sur des machines à mémoire limitée tout en conservant une efficacité raisonnable. Nous illustrons l'impact des mécanismes bas-niveau d'E/S sur le comportement de la factorisation parallèle *out-of-core*. Nous utilisons ensuite des simulations pour analyser les gains envisageables en stockant de surcroît sur disque les données numériques temporaires (*mémoire active*). Nous discutons à la fois des besoins minimaux mémoires et du volume minimal d'E/S que nous pourrions ainsi obtenir sur une machine à mémoire limitée. Finalement, nous résumons les principaux points critiques que nous avons identifiés lorsqu'il s'agit de concevoir des méthodes directes de résolution de systèmes linéaires creux dans un environnement *out-of-core*.

Mots-clés: Matrices creuses ; méthode directe ; méthode multifrontale ; hors-mémoire (out-of-core) ; factorisation parallèle ; volume d'E/S ; E/S directes ; étude de performance

1 Introduction

The solution of sparse systems of linear equations is a central kernel in many simulation applications. Because of their robustness and performance, direct methods can be preferred to iterative methods. In direct methods, the solution of a system of equations $Ax = b$ is generally decomposed into three steps: (i) an analysis step, that considers only the pattern of the matrix, and builds the necessary data structures for numerical computations; (ii) a numerical factorization step, building the sparse factors (e.g., L and U if we consider an unsymmetric LU factorization); and (iii) a solution step, consisting of a forward elimination (solve $Ly = b$ for y) and a backward substitution (solve $Ux = y$ for x). For large sparse problems, direct approaches often require a large amount of memory, that can be larger than the memory available on the target platform (cluster, high performance computer, ...). In order to solve increasingly large problems, *out-of-core* (OOC) approaches are then necessary, where disk is used to store data that cannot fit in physical main memory.

Although several authors have worked on sequential or shared-memory *out-of-core* solvers [1, 13, 26], sparse *out-of-core* direct solvers for distributed-memory machines are less common. In this work, we will use an existing parallel sparse direct solver, `MUMPS` [4, 5] (for MUltifrontal Massively Parallel Solver), to identify the main difficulties and key points when designing an *out-of-core* version of such a solver. `MUMPS` is based on a parallel multifrontal approach which is a particular direct method for solving sparse systems of linear equations. It has to be noted that recent contributions by [21] and [22] for uniprocessor approaches pointed out that multifrontal methods may not fit well an *out-of-core* context because large dense matrices have to be processed, that can represent a bottleneck for memory; therefore, they prefer left-looking approaches (or switching to left-looking approaches at some point during the factorization). However, in a parallel context, increasing the number of processors can help keeping such large blocks in-core.

After presenting other direct out-of-core approaches for sparse linear systems (Section 3) and the available tools to manage *I/O*, the objective of Section 4 is to justify and describe an out-of-core approach where only the factors produced at each node of the multifrontal tree are stored to disk. We will observe that this approach allows us to treat larger problems with a given memory, or the same problem with less memory. Both a synchronous approach (writing factors to disk as soon as they are computed) and an asynchronous approach (where factors are copied to a buffer and written to disk only when half of the buffer is full) are analyzed, and compared to the in-core approach on a platform with a large amount of memory. Before designing a future full *out-of-core* multifrontal method, these prototypes allow us to identify the difficulties in obtaining an efficient *I/O*-intensive application. In particular, we will see that even the validity and reproducibility of the *I/O* performance benches strongly depends on the low-level *I/O* tools. In order to process significantly larger problems, we present in Section 5 simulation results where we suppose that the active memory of the solver is also stored on the disk and study how the overall memory can further be reduced. This study is the basis to identify the bottlenecks of our approach when confronted to arbitrarily large problems. Finally, we analyze in Section 6 the volume of *I/O* involved by our strategies, and compare it to the volume of *I/O* for the factors. In Section 7, we draw up an assessment of the lessons learned.

2 Memory management in a parallel multifrontal method

Like many other direct methods, the multifrontal method [14, 15] is based on the elimination tree [19], which is a transitive reduction of the matrix graph and is the smallest data structure representing dependencies between 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 [8]. We recall that a supernode is a contiguous range of columns (in the factor matrix) having the same lower diagonal nonzero structure.

In the multifrontal approach, the factorization of a matrix is done by performing a succession of partial factorizations of small dense matrices called *frontal matrices*, which are associated to the nodes of the tree. Each frontal matrix is divided into two parts: the *factor* block, also called *fully summed* block, which corresponds to the variables factored when the elimination algorithm processes the frontal matrix; and the *contribution block* which corresponds to the variables updated when processing the frontal matrix. Once the partial factorization is complete, 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 topological (we do not process parent nodes before their children) of the postorder traversal assembly tree. It uses three areas of storage, one for the factors, one to stack the contribution blocks, and another one for the current frontal matrix [3]. During the tree traversal, the memory space required by the factors always grows while the stack memory (containing the contribution blocks)[1] 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 discarded and the size of the stack decreases. We will refer by "active memory" to the memory area containing both the active frontal matrices[2] and the stack of contribution blocks.

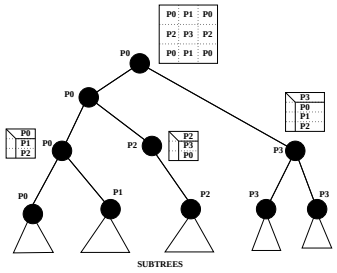


Figure 1: Example of the distribution of an assembly tree over four processors.

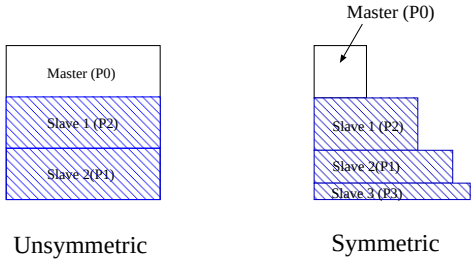


Figure 2: Distribution of the processors on a parallel node in the unsymmetric and symmetric cases.

From the parallel point of view, the parallel multifrontal method as implemented in MUMPS uses a combination of static and dynamic scheduling approaches. Indeed, a first partial mapping is done statically (see [6]) to map some of the tasks to the processors. Then, for parallel tasks corresponding to large frontal matrices of the assembly tree, a *master task* is in charge of the elimination of the so-called fully summed rows, while dynamic scheduling decisions are used to select the "slave" processors in charge of updating the rest of the frontal matrix (see Figures 1 and 2). The rest of the frontal matrix and the associated memory is thus scattered among several processors, each processor receiving a *slave task* to process, whereas the fully summed part can only be treated by one processor (the one responsible for the master task). Figure 2 illustrates the impact of the load distribution on the processors responsible for the tasks associated to such a node. Note that those decisions are taken to balance workload, possibly under memory constraints (see [7]). Finally, in order to limit the amount of communication, the nodes at the bottom of the tree are statically merged into *subtrees* (see

[1] In parallel, the contribution blocks management may differ from a pure stack mechanism.
[2] In the parallel case, it may happen that more than one matrix is active

Figure 1), each of them being processed sequentially on a given processor.

3 State of the art

3.1 Out-of-core direct methods

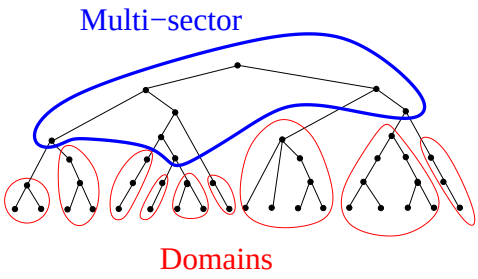
Approaches based on virtual memory

Paging consists in delegating the *out-of-core* management to the system: it handles an amount of memory greater than the physical memory available and is composed of memory pages either in physical memory or on disk. Some authors have developed their own paging mechanism in Fortran [20]. When relying on the system, paging mechanisms do not usually exhibit high performance [11, 21] because it has no particular knowledge of the memory access pattern of the application. However, through *paging monitoring* [10] the application can adapt the paging activity to the particularities of its memory access scheme at the level of the operating system kernel. The application can then define the priority of a page to be kept in memory and specify which pages are obsolete so that they can be freed. This improvement can reach a performance 50 % higher than the LRU (Least Recently Used) policy. However, this approach is too closely related to the operating system and not adapted when designing portable codes.

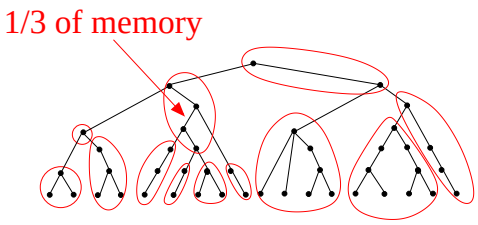
Another approach consists in mapping parts of memory to files using C primitives such as `mmap`. Again, it is difficult to obtain portable code and attain good performance (even if some mechanisms like the `madvise` system call can help).

Sparse direct methods based on explicit *I/O*

In [21] Rothberg and Schreiber compare in the sequential case the impact of the sparse direct method chosen (*left-looking*, *right-looking* or *multifrontal*) on the volume of *I/O* produced and on the performance of the Cholesky factorization. As the multifrontal method fits well to the *in-core* case, they first propose an *out-of-core* multifrontal factorization. Since factors are not re-used during the factorization, they are written to disk systematically. Then the authors identify the largest subtrees (called *domains*) in the elimination tree that can fit *in-core*. All such subtrees are processed with their active memory *in-core*, while their last contribution block (*i.e.* the information necessary to the factorization of the top of the tree) is written to disk. Once all the domains are factored and the contribution blocks of their root are written to disk, the top of the tree (called *multi-sector*) is then factored. When a node is processed, its frontal matrix is divided into panels, *i.e.* an aggregation of columns so that each panel might hold in half of the available memory. To assemble a panel, first the contribution blocks which have to update it are loaded in the other half of the memory (and are themselves cut if they are too large) and the updates are performed; next, the panels of the same frontal matrix are loaded into memory (one by one) and the subsequent updates are performed. Then the panel is factored and written to disk. This method, named `MF`, is robust as it never runs out-of-memory: at any time, at most two panels are together in memory. However, a very large volume of *I/O* is required when the frontal matrices become large. The authors propose an hybrid variant, named `MF-LL`, which consists in (i) an update phase for which the factors of all the descendants are read back from the disk then assembled, and (ii) a factorization phase. The size of the panels is adapted to fit the memory requirements of the left-looking method for which the elementary data to process is not anymore a frontal matrix but a factor (smaller). An experimental validation allowed them to show that this second method usually requires less *I/O* on very large problems. As illustrated in Figure 3(a), this is an hybrid method be-



(a) `MF` et `MF-LL`



(b) `LL-RR`

Figure 3: direct *out-of-core* approaches

cause it merges a multifrontal approach at the bottom of the elimination tree (on subtrees that can be processed *in-core*) and a left-looking approach above.

In [22], Rotkin and Toledo propose a modification of this hybrid approach and then focus on a *left-looking-right-looking* variant (named `LL-RR`): they do not use multifrontal methods because of the presence of big frontal matrices that can be the bottleneck for memory in the sequential case. They do not rely on *domains* but on a formal concept that they call *cold subtrees assumption*: they suppose (i) to partition the set of supernodes into disjoint subsets that are factored one after the other and (ii) that the factorization of each subset starts with an empty (or *cold*) main memory. They show that each subset may be a connected subset[3] of the elimination tree without increasing the volume of *I/O*. Then they limit the size of both an individual supernode and of the maximal single root-to-leaf path (sum of the sizes of the supernodes along this path) of a subset to one third of the available memory (see Figure 3(b)). A subset s is factored in a depth-first traversal as follows:

1. when a supernode j is first visited, its nonzero structure is loaded from disk;
2. for each child i of j that is *not in* s, individual supernode-supernode updates (for which is reserved the last third of memory in order to ensure efficiency with dense computation combined with scattering-gathering operations) are applied from every supernode of the subtree rooted at i to j;
3. j is factored;
4. the ancestors of j *in* s are updated with a partial right-looking approach;
5. j is written to disk and released from main memory.

In [12] Dobrian compares the efficiency (in terms of *I/O* volume and internal memory traffic) of the left-looking, right-looking and multifrontal methods (in the sequential case too). Thanks to analytical results (for model problems) and experimental ones from simulations (for irregular problems), he shows that the multifrontal method is a good choice even when the core size available is small. However, this method is not well suited for matrices whose peak of active memory is larger than the volume of factors. He concludes that to achieve good performance on a large range of problems, the solver should provide several algorithmic options including left-looking, right-looking and multifrontal as well as hybrid approaches.

[3] They call *subtree* such a subset (which differs from the classical definition); hence the name of their assumption.

Although both [21] and [22] insist on the problem of large frontal matrices arising in multifrontal methods, note that those large dense frontal matrices can be processed out-of-core (as done in [1]) and that in a parallel context, this may be less critical since a frontal matrix can be distributed over several processors.

3.2 I/O mechanisms

I/O mechanisms are essential for *out-of-core* applications. Their performance impacts directly the whole application performance. We give below an overview of some existing *I/O* tools.

Different file access modes. By default, system caches (pagecache) may be used by the operating system to speed-up *I/O* requests. The management of the pagecache is system-dependent and not under user control (it is usually managed with a `LRU` policy and its size may vary dynamically). Thus depending on whether the data is copied to the pagecache or written to disk (for example when the pagecache is flushed), the performance of *I/O* operations may vary. In order to enforce a synchronization between the pagecache and the disk, the *O_SYNC* flag can be specified when opening a file. However, the user still has no control of the size of the pagecache in this context. Thus depending on the pagecache management policy, the behaviour of the user-space applications may be perturbed by the virtual memory mechanisms.

One way to avoid the caching at the system level consists in using direct *I/O*. This is a specific feature existing in various operating systems. In our experimental environment, it can be activated by specifying the *O_DIRECT* flag when opening the file. Furthermore, data must be aligned in memory when using direct *I/O* mechanisms: the address and the size of the buffer must be a multiple of the page size and/or of the cylinder size. The use of this kind of *I/O* operations ensures that a requested *I/O* operation is effectively performed and that no caching is done by the operating system.

C and Fortran libraries. The C standard *I/O* routines *fread/fwrite* (to read from or write to a binary stream), *read/write* (to read from or write to a file descriptor), or *pread/pwrite* when available (to read from or write to a file descriptor at a given offset) are known to be efficient low-level kernels. Nevertheless building a complete efficient asynchronous *I/O* mechanism based on them remains a challenge as we must design a robust communication scheme between an *I/O* thread which manages disk accesses and the main thread (`MUMPS`). The new Fortran 2003 includes an asynchronous *I/O* API as a standard. But this version is too recent to be portable.

AIO. AIO is a POSIX asynchronous *I/O* mechanism. It should be optimized at the kernel level of the target platform and then permits high performance.

MPI-IO. The Message Passing Interface MPI has an efficient *I/O* extension MPI-IO [25] that handles *I/O* in a parallel environment. However, this extension aims at managing parallel *I/O* applications which is not the case in our application: disks are there to extend the memory of each process and we are not planning to share *out-of-core* files between several processes.

FG. The FG [9] framework for high-performance computing applications aims at making *I/O* designing more efficient. It allows the developer to use an efficient asynchronous buffered *I/O* mechanism at a high level. However we will not use it at the moment because it manages concurrent *I/O* threads whereas, in our case, *I/O* threads do not interfere with each other.

We decided to implement two *I/O* mechanisms: the standard C *I/O* library and AIO. However, since AIO was not available on our main target platform (described in Section 4.1), we focus on the C *I/O* library in the rest of this paper.

4 Out-of-core multifrontal factorization

4.1 Experimental environment

For our study, we have chosen four large test problems (see top of Table 1). Occasionally, we illustrate some particular properties using auxiliary matrices presented in the second part of the table. Some of these problems are extracted from the PARASOL collection[4] while others are coming from other sources.

Our main target platform is the IBM SP system from IDRIS[5], which is composed of several nodes of either 4 processors at 1.7 GHz or 32 processors at 1.3 GHz. On this machine, we have used from 1 to 128 processors with the following memory constraints: we can access 1.3 GB per processor when asking for 65 processors or more, 3.5 GB per processor for 17-64 processors, 4 GB for 2-16 processors, and 16 GB on 1 processor. The *I/O* system used is the `IBM GPFS` [23] filesystem. With this filesystem we observed a maximal *I/O* bandwidth of 108 MBytes per second (using direct *I/O* to ensure that the *I/O* is effectively performed, without intermediate copy). However, since it was not possible to write files on disks local to the processors, some performance degradation occurs when several processors write/read an amount of data simultaneously to/from the filesystem: we observed a speed-down of about 3 on 8 processors (and 12 on 64 processors) when each processor writes one block of 500 MBytes. Note that we chose to run on this platform because it allows us to run large problems *in-core* and thus compare *out-of-core* and *in-core* approaches (even if the behaviour of the filesystem is not optimal for performance studies).

In our performance analysis, we sometimes also use another platform with local disks in order to avoid specific effects linked to GPFS: a cluster of linux bi-processors from PSMN/FLCHP[6], with 4 GB of memory and one disk for each node of 2 processors. On each node, the observed bandwidth is 50 MB / second per node, independently of the number of nodes, and the filesystem is ext3. However, when not otherwise specified, results correspond to the main IBM platform described above.

By default, we used the METIS package [17] to reorder the matrices and thus limit the number of operations and fill-in arising in the subsequent sparse factorization. The results presented in the following sections have been obtained using the dynamic scheduling strategy proposed in [7].

4.2 Preliminary study

In the multifrontal method, the factors produced during the factorization step are not re-used before the solution step. It then seems natural to first focus on writing *them* to disk. Thus, we present a preliminary study which aims at evaluating by how much the in-core memory can be reduced by writing the factors to disk during the factorization. To do so, we simulated an *out-of-core* treatment of the factors: we free the corresponding memory as soon as each factor is computed. Of course the solution step cannot be performed as factors are definitively lost, but freeing them allowed to analyze real-life problems on a wider range of processors (in this initial study).

[4] `http://www.parallab.uib.no/parasol`
[5] Institut du Développement et des Ressources en Informatique Scientifi que
[6] Pôle Scientifi que de Modélisation Numérique/Fédération Lyonnaise de Calcul Haute Performance

Main test problems						
Matrix	Order	nnz	Type	$nnz(L\|U)$ ($\times 10^6$)	Flops ($\times 10^9$)	Description
AUDIKW_1	943695	39297771	SYM	1368.6	5682	Automotive crankshaft model (PARASOL)
CONESHL_mod	1262212	43007782	SYM	790.8	1640	provided by SAMTECH; cone with shell and solid element connected by linear constraints with Lagrange multiplier technique
CONV3D64	836550	12548250	UNS	2693.9	23880	provided by CEA-CESTA; generated using AQUILON (http://www.enscpb.fr/master/aquilon)
ULTRASOUND80	531441	330761161	UNS	981.4	3915	Propagation of 3D ultrasound waves, provided by M. Sosonkina
Auxiliary test problems						
Matrix	Order	nnz	Type	$nnz(L\|U)$ ($\times 10^6$)	Flops ($\times 10^9$)	Description
BRGM	3699643	155640019	SYM	4483.4	26520	large finite element model for ground mechanics (provided by BRGM)
CONESHL2	837967	22328697	SYM	239.1	211.2	Provided by SAMTECH
GUPTA3	16783	4670105	SYM	10.1	6.3	linear programming matrix (AA'), Anshul Gupta
SHIP_003	121728	4103881	SYM	61.8	80.8	PARASOL collection
SPARSINE	50000	799494	SYM	207.2	1414	Structurual optimization (CUTEr)
THREAD	29736	2249892	SYM	24.5	35.1	Threaded connector/contact problem (PARASOL collection)
QIMONDA07	8613291	66900289	UNS	556.4	45.7	Circuit simulation. Provided by Reinhart Schultz (Infineon Technologies).
WANG3	26064	177168	UNS	7.9	4.3	Discretized electron continuity, 3d diode, uniform 30-30-30 mesh
XENON2	157464	3866688	UNS	97.5	103.1	Complex zeolite,sodalite crystals. D Ronis

Table 1: Test problems. Size of factors and number of floating point operations (Flops) computed with METIS.

We measure the size of the new peak of memory (which actually corresponds to the *active memory* peak) and compare it to the one we would have obtained with an *in-core* factorization (*i.e.* the *total memory peak*). In a distributed memory environment, we are interested in the maximum peak obtained over all the processors as this value represents the memory bottleneck.

In Figure 4, we present the typical memory behaviour of the parallel multifrontal method as implemented in MUMPS for a large sparse matrix of 943695 equations, called AUDIKW_1 (see Table 1). First, we observe that the peak of factors zone is often close to the peak of total memory. This means that treating only the active memory *out-of-core* would not lead to large memory gains.

For a small number of processors, we observe that the active memory is much smaller than the total memory. In other words, if factors are written to disk as soon as they are computed, only the active memory remains *in-core* and the memory requirements decrease significantly (up to 80 % in the sequential case).

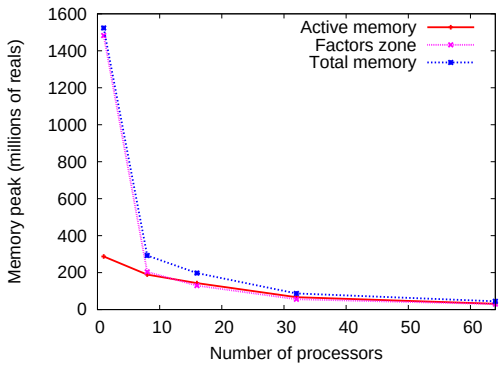


Figure 4: Typical memory behaviour (AUDIKW_1 matrix) with `METIS` on different numbers of processors.

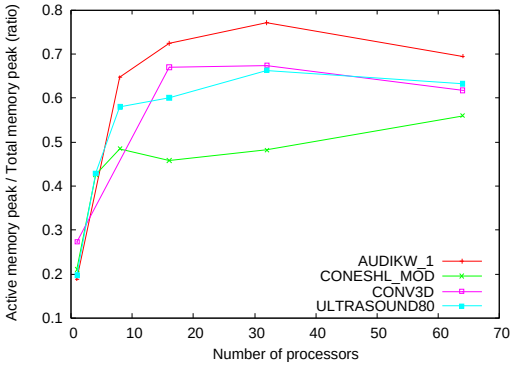


Figure 5: Ratio of active and total memory peak on different number of processors for several large problems (`METIS` is used as the reordering technique).

On the other hand, when the number of processors increases, the peak of active memory decreases more slowly than the total memory as shown in Figure 5 for our four main test problems. For example, on 64 processors, the active memory peak reaches between 50 and 70 percent of the peak of total memory. In conclusion, on platforms with small numbers of processors, an *out-of-core* treatment of the factors will allow us to process significantly bigger problems; the implementation of such a mechanism is the object of Section 4.3. Nevertheless, either in order to further reduce memory requirements on platforms with only a few processors or to have significant memory savings on many processors, we may have to treat both the factors and the active memory with an *out-of-core* scheme. This will be studied in Section 5.

Note that we have been focussing in this discussion on the number of real entries in the factors, in the active memory, and in the total memory. The ratios presented in Figure 5 only consider the number of reals used for the numerical factorization. To be more precise, we should also take care of the amount of memory due to the integer workspace (indices of the frontal matrices, tree structure, mapping information,...) and the communication buffers. Table 2 provides the size in MegaBytes of the different memory areas in the multifrontal code MUMPS, for 1 processor and 32 processors: integers for active memory and factors, integer arrays to store the tree, the mapping and various data structures, communication buffers at the application level to manage asynchronous communications. We observe that communication buffers, that depend on the largest estimated message sent from one

Matrix		Factors and active memory integers	reals	Other data structures (tree, ...)	Comm. buffers	Initial matrix (1)	Total
AUDIKW_1	1P	98	11839	26	0	479	12443
AUDIKW_1	32P	8	758	33	264	33	1097
CONESHL_MOD	1P	107	7160	34	0	526	7828
CONESHL_MOD	32P	9	314	44	66	24	458
CONV3D64	1P	83	(2)	(2)	0	157	(2)
CONV3D64	32P	7	927	32	286	9	1260
ULTRASOUND80	1P	51	8858	16	0	401	9326
ULTRASOUND80	32P	4	348	19	75	19	464

Table 2: Average memory (MegaBytes) per processor for the different memory areas. Those numbers are the ones estimated during the analysis step of the solver, and they are used to allocate the memory at the factorization step. (1) This corresponds to a copy of the initial matrix that is distributed (with some redundancy) over the processors. (2) For these values, an integer overflow occurred in the statistics computed by MUMPS.

processor to another, also use a significant amount of memory in parallel executions. If this becomes critical in the future, we will have to study how subdividing large messages into series of smaller ones can be done. Although the memory for the integer indices corresponding to active memory and factors is small compared to the memory for real entries, the study of this paper can also be applied to the integers, and processing them out-of-core is also a possibility. A copy of the initial matrix is distributed over the processors in a special format for the assemblies occurring at each node of the tree. Some parts of the initial matrix are replicated on several processors to allow some tasks to be mapped dynamically. However, once a node is assembled, the corresponding part of the initial matrix could be discarded; this is not done in the current version of the code.

In any case, for the four large matrices of our study, we observe that the storage corresponding to real entries for factors and active memory is predominant, and that reducing it is a priority. This is the objective of this paper.

4.3 Out-of-core management of the factors: prototype implementation

We present in this section a prototype of an *out-of-core* parallel multifrontal factorization scheme. In order to reduce the memory requirements of the factorization phase, factors are written from memory to disk as soon as they are computed. Before designing a full *out-of-core* method, the goal of this prototype is to better understand the difficulties arising, to check if the problems we can process are as large as forecasted in Section 4.2, and most of all, to test the behaviour of the low-level *I/O* mechanisms in the context of our method.

We designed several *I/O* schemes allowing us to write the factors during the factorization. Our aim is to study both a synchronous *I/O* scheme, and an asynchronous buffered *I/O* scheme.

Synchronous *I/O* scheme. In this scheme, the factors are directly written with a synchronous scheme using the standard *I/O* subroutines (either *fread/fwrite*, *read/write*, or *pread/pwrite* when available).

Asynchronous *I/O* scheme. In this scheme, we associate with each MPI process of our application an *I/O* thread in charge of all the *I/O* operations. This allows us to overlap the time needed by *I/O* operations with computations. The *I/O* thread is designed over the standard `POSIX`

thread library (pthread library). The computation thread produces (computes) factors that the *I/O* thread consumes (writes to disk) according to the producer-consumer paradigm. Each time an amount of factors is produced, the computation thread posts an *I/O* request: it takes a lock, inserts the request into the *queue of pending requests*, and releases the lock. As in a classical implementation of the paradigm, two semaphores are used additionally to avoid busy waiting: one (initialized to 0) counts the number of pending requests while another one (initialized to the maximum authorized number of pending requests) limits it to the capacity of the queue. The *I/O* thread loops endlessly: at each iteration it waits for requests that it handles using a `FIFO` strategy. Symmetrically, the *I/O* thread informs the computation thread of its advancement with a second producer-consumer paradigm in which this time the *I/O* thread produces the finished requests (inserts them into the *queue of finished requests*) that the computation thread consumes (removes them from the queue when checking for their completion). This second mechanism is independent from the first one: it does not share the same synchronization variables. The whole synchronization scheme is illustrated in Figure 6. Note that we limited our description to the case where only one *I/O* thread is attached to each computational thread. It could be interesting to use multiple *I/O* threads to improve overlapping on machines with specific hardware configurations (multiple disks per node, high performance parallel file systems, ...).

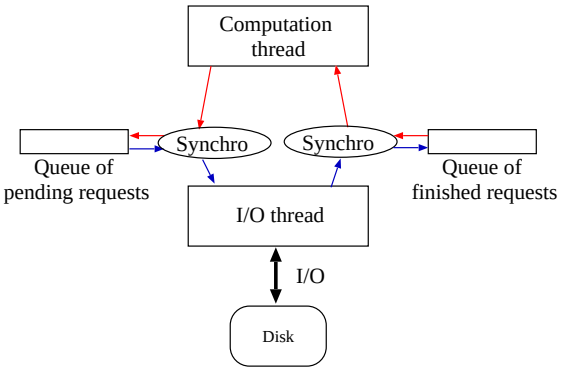


Figure 6: Asynchronous synchronization scheme.

Together with the two *I/O* mechanisms described above, we designed a buffered *I/O* scheme. This approach relies on the fact that we want to free the memory occupied by the factors as soon as possible without necessarily waiting for the completion of the corresponding *I/O*. Thus, and in order to avoid a complex memory management in a first approach, we added a buffer where factors are copied before they are written to disk. The buffer is divided into two parts so that while an asynchronous *I/O* operation is occurring on one part, factors that are being computed can be stored in the other part (double buffer mechanism allowing the overlap of *I/O* operations with computation). Although we have implemented the possibility to use a buffer with the synchronous scheme (when studying the behaviour of low-level *I/O* mechanisms), by default only the asynchronous scheme uses an intermediate buffer. In this prototype, the size of half a buffer is as large as the largest factor: this may be too costly from the memory point of view but allows us in a first implementation to assess the efficiency we can expect thanks to asynchronism.

4.4 Solving larger problems

In order to validate the interest of this prototype implementation, we have experimented it on our test problems (see Table 1). First, we used the synchronous approach. We have been able to observe that for a small number of processors we use significantly less memory with the *out-of-core* approach: the total memory peak is replaced by the active memory peak, with the improvement ratios of Table 5. Thus the factorization can be achieved on limited-memory machines: same problems are solved with either less memory (see Table 3(a)) or less processors (see Table 3(b)). Moreover the `CONV3D64` and the `BRGM`[7] problems can be processed *out-of-core* sequentially (with 16 GB of memory) whereas the *in-core* version runs out of memory with one processor. In other words, we can solve larger problems (although not arbitrarily larger).

	in-core	*out-of-core*
1 proc (16GB)	1101	218
4 procs	360	154

(a) Maximum memory (in millions of reals) needs per processor in the *in-core* and *out-of-core* cases for the `ULTRASOUND80` problem.

Matrix	Strategy	Min procs
`ULTRASOUND80`	*in-core*	8
	out-of-core	2

(b) Minimal number of processors needed to process the `ULTRASOUND80` and `CONV3D64` problems in the *in-core* and *out-of-core* cases.

Table 3: Processing the same problems with either less memory (left) or less processors (right).

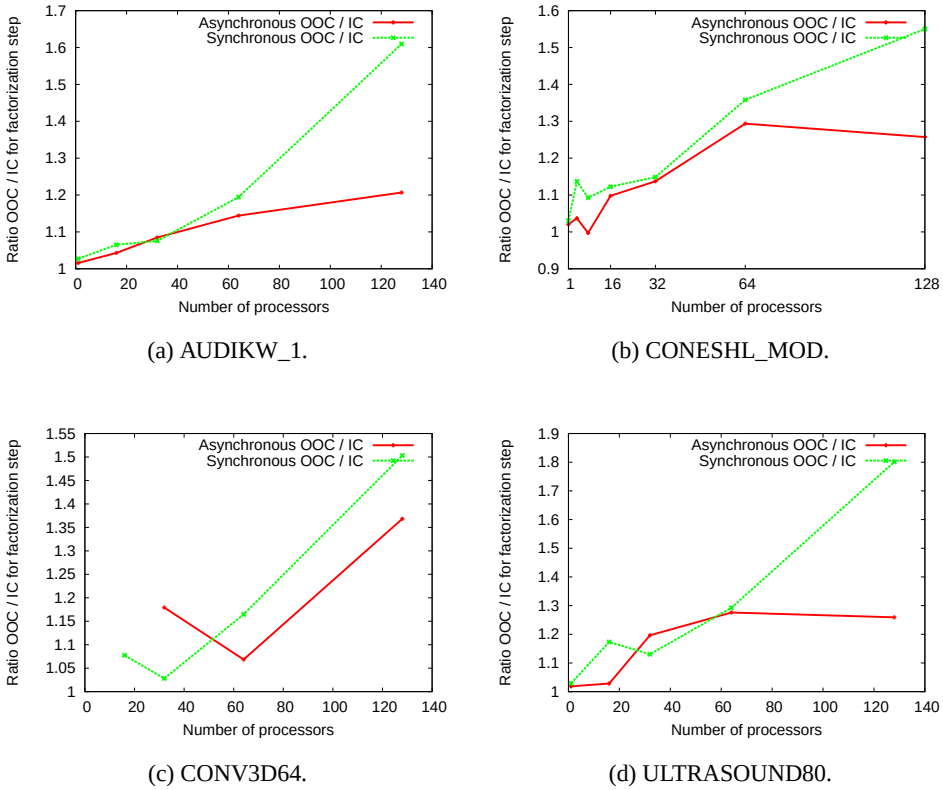
4.5 Performance of a system-based approach

We now focus on rough performance results and report in Figure 7 a comparative study of the *in-core* case, the synchronous *out-of-core* scheme and the asynchronous buffered scheme, when varying the number of processors. All jobs were submitted to the batch system simultaneously, and some of the executions may have interfered with each other when accessing the *I/O* nodes.

Note that for the buffered case, the size of the *I/O* buffer is set to twice the size of the largest factor block (to have a double buffer mechanism). As we can see, the performance of the *out-of-core* schemes is indeed close to the *in-core* performance for the sequential case. We use here our four main problems, AUDIKW_1, CONESHL_MOD, CONV3D64, ULTRASOUND80 to illustrate the discussions. Recall that we were not able to run the BRGM matrix in parallel because the memory per processor is too small for the analysis phase. We were not successful in running the CONV3D64 matrix on 1 processor with the *in-core* scheme because the total memory requires more than 16 GB. On 16 processors, the double buffer is so large on this matrix that the asynchronous approach could not be executed while both the synchronous version and the *in-core* version succeeded !

In sequential, the *out-of-core* schemes are at most 2% slower than the *in-core* case while they need an amount of memory that can be 80 percent smaller as shown in Figure 5 (for one processor). Concerning the parallel case, we observe that with the increase of the number of processors, the gap between the *in-core* and the *out-of-core* cases increases. The main reason is the performance degradation of the *I/O* with the number of processors that we mentioned at the end of Section 4.1. To

[7] The analysis step for the BRGM matrix requires more than 3.5 GB of memory with the version of MUMPS we used; thus this matrix could not be processed on more than 16 processors. Between 2 and 16 processors, the *in-core* numerical factorization step ran out-of-memory for this matrix.



(a) AUDIKW_1.

(b) CONESHL_MOD.

(c) CONV3D64.

(d) ULTRASOUND80.

Figure 7: Execution times (normalized with respect to the *in-core* case) of the synchronous and asynchronous *I/O* schemes.

fix the ideas, the average bandwidth per processor decreased from 115 MB/s to 11 MB/s between 1 and 128 processors for the `ULTRASOUND80` problem in the asynchronous scheme. Consequently, the ratio *I/O*s / computation strongly increases and overlapping the *I/O*s by computation becomes harder. Moreover, in the parallel case, the delay induced by a non overlapped *I/O* on a given processor may have repercussions and generate a similar delay on other processors when those are dependant on data coming from the processor performing a large *I/O*. We underline here an important aspect of our future work : taking into account the *I/O*s at the dynamic scheduling level [7] to better balance the load.

Concerning the comparison of the *out-of-core* schemes, we can see that the asynchronous buffered approach usually performs better than the synchronous one. However, it has to be noted that even in the synchronous scheme, the system allocates data in memory that also allows to perform *I/O* asynchronously, in a way that is hidden to the application. Otherwise, the performance of the synchronous approach might be worse (see section 4.6). The asynchronous buffered approach does actually remain generally better because *I/O* system mechanisms (copies of data to the pagecache and possibly copies from the pagecache to disks) are overlapped by computation in this case whereas, in the synchronous scheme, the system may have to write data from pagecache to disks when an *I/O* request is performed without overlapping this operation. We measured the average time a processor spends in *I/O* mode which we name the *time spent in* I/O *mode*. In the synchronous scheme, it corresponds to the time spent by the system to copy data to the pagecache and possibly copying a part of the pagecache to

disk. In the asynchronous scheme, it corresponds to the time spent by the computational thread to post its *I/O* requests: for a given request, either there is enough free space in the *I/O* buffer and then the corresponding time spent in *I/O* mode is the time used to copy the factor to the buffer, or the *I/O* thread is late and then it includes moreover the time the *I/O* thread needs to complete the writing (again by the system) of a half-buffer[8]. To sum up, in the two cases, this time represents the overhead due to the cost of the *I/O*s[9] We show in Table 4, with the example of the `ULTRASOUND80` matrix, that the synchronous approach spends much more time in *I/O* mode than the asynchronous one. In particular, we can see that in the asynchronous scheme overlapping is very efficient and thus that the *I/O* overhead is very small compared to the elapsed time for factorization (which is equal to 1376 seconds in sequential and 30 seconds on 128 processors for the `ULTRASOUND80` problem in the asynchronous scheme).

Number of processors	1	4	8	16	32	64	128
Synchronous scheme	51.2	17.0	9.7	3.5	2.5	4.2	3.0
Asynchronous scheme	6.6	2.4	1.3	0.7	0.7	0.7	3.7

Table 4: Average elapsed time (seconds) spent in *I/O* mode per processor for the synchronous and asynchronous schemes on the `AUDIKW_1` problem for a various number of processors.

When the number of processors becomes very large (128) the synchronous approach has a poor efficiency (between 50 and 80 percents slower than the *in-core* case). Indeed, on so many processors, we get a drop of bandwidth (due to `GPFS` filesystem). Moreover, the system pagecache, shared by processors that belong to the same SMP node, can be disturbed by so many simultaneous *I/O* requests. Finally, as explained above the impact of one *I/O* delay leads to other delays on other processors waiting for it. The combination of these three aspects leads to such a drop of efficiency. In the asynchronous scheme, the drop of bandwidth also impacts the global efficiency of the factorization but the good overlapping mechanism balances this overhead. We also noticed that the results are particularly difficult to reproduce with a large number of processors. Indeed, the efficiency depends on (i) the number of concurrent applications that access simultaneously the same *I/O* node through the `GPFS` file system, (ii) the state of the pagecache of the SMP node(s) and (iii) the (non deterministic) evolution of the parallel execution. For instance, we have perturbed a 64 processors factorization of the `CONESHL_MOD` matrix by several other concurrent and simultaneous *out-of-core* executions also on large numbers of processors. The asynchronous *out-of-core* factorization was then 2.3 times longer (62.7 seconds) than the *in-core* one (27.4 seconds) and the synchronous one was 3.3 times longer (90.0 seconds).

Keeping in mind that this version is a prototype for a whole (factors and active memory) *out-of-core* multifrontal method in which the *I/O* volume may be significantly larger (see Section 4.8) we realize the importance of an asynchronous approach (which allows a better overlapping) and of new load balancing criteria which would take into account the *I/O* constraints.

Concerning the solution phase, the size of the memory will generally not be large enough to hold all the factors. Thus, factors have to be read from disk, and the *I/O* involved increase significantly the

[8] We recall that we use a double buffer mechanism to allow for overlapping.

[9] In the asynchronous approach, this time is only measured on the computational thread and so takes into account the cost of the synchronization of the computational thread with the *I/O* thread (but not the cost of the management of the *I/O* thread: some `CPU` time may be used by the *I/O* thread to synchronize with the computational thread and possibly to perform the *I/O*s. Note that, on modern systems, effi cient mechanisms such as Direct Memory Access do not use many CPU cycles when performing *I/O*s).

time for solution. Note that we use a basic demand-driven scheme, relying on the synchronous low-level *I/O* mechanisms from Section 4.3. We observed that the performance of the *out-of-core* solution step is often more than 10 times slower than the *in-core* case, especially in parallel. Although disk contention might be an issue on our main target platform in the parallel case, the performance of the solution phase should not be neglected; it becomes critical in an *out-of-core* context and prefetching techniques in close relation with scheduling issues have to be studied. This is the the object of current work by the MUMPS group in the context of the PhD of Mila Slavova.

4.6 Interest of direct I/O

As said in the previous section, relying on the system for *I/O* is not satisfactory and can make things difficult to analyze. With direct *I/O*, the cost of *I/O* is more stable and the system is not allowed to allocate intermediate buffers. This should enable the *I/O* to be more efficient, and avoids consuming extra memory due to system buffers (hidden to the user).

We have implemented a variant of our low-level mechanism that allows the use of direct *I/O*. On our platforms, we had to use the O_DIRECT flag for that. Because the data to write have to be aligned in memory we had to rely on an intermediate buffer, written to disk when full. The size of the buffer has been experimentally tuned to maximize bandwidth: we use a buffer of size 10 MB, leading to an average bandwidth of 90 MB/s. In the case of a buffered asynchronous scheme (see Section 4.3), we use this aligned buffer in addition to the *I/O* buffer.

Sequential case.

We first experimented this approach in the sequential case. Table 5 shows the performance obtained. First we see that the use of direct *I/O* coupled with an asynchronous approach is usually at

Matrix	Direct *I/O* Synch.	Direct *I/O* Asynch.	P.C. Synch.	P.C. Asynch.	IC
AUDIKW_1	2243.9	2127.0	2245.2	2111.1	2149.4
CONESHL_MOD	983.7	951.4	960.2	948.6	922.9
CONV3D64	8538.4	8351.0	[[8557.2]]	[[8478.0]]	(*)
ULTRASOUND80	1398.5	1360.5	1367.3	1376.3	1340.1
BRGM	9444.0	9214.8	[[10732.6]]	[[9305.1]]	(*)
QIMONDA07	147.3	94.1	133.3	91.6	90.7

Table 5: Elapsed time (seconds) for the factorization step in the sequential case depending on the use of direct *I/O*s or pagecache (P.C.) for both the synchronous (Synch.) and asynchronous (Asynch.) approaches, and compared to the *in-core* case (IC), for several matrices.
(*) The factorization step ran out-of-memory. [[8857.2]] Side effects (swapping,...) of the pagecache management policy.

least as efficient as any of the approaches coupled with the use of the pagecache. When coupled with the synchronous scheme, the use of direct *I/O* leads to an increase of the elapsed time. Indeed, in this case, there is no overlap at all and the costs of *I/O* requests are entirely paid[10]. For the matrices AUDIKW_1, CONESHL_MOD, CONV3D64, ULTRASOUND80, BRGM and QIMONDA07, the costs of the *I/O* requests involved are 134, 55, 383, 83, 401 and 36 seconds, respectively. Those costs are

[10] We nevertheless recall, that even in the synchronous scheme, a 10 MB buffer aligned in memory is used which ensures an effi cient bandwidth (about 90 MB/s).

summed to the computation costs with no overlapping, and we can expect the total execution time for this version to be equal to the cost of the *in-core* version with the *I/O* costs. The reason why the effective execution time is sometimes smaller than that (for example, $2243.9 < 2149.4 + 134$ for `AUDIKW_1`) is that we may have a gain of CPU time thanks to good cache effects arising from a better memory locality of the *out-of-core* approach. Furthermore, when the *I/O* volume becomes critical (20 GB for the `CONV3D64` matrix), we notice a worse behaviour of the pagecache-based schemes. Indeed, the pagecache policy is adapted to a general purpose and is not well-suited for very *I/O*-intensive applications. Notice that Table 5 provides a representative set of results among several runs, each matrix corresponding to one submission at the batch-scheduler level. However, because performance results vary from execution to execution, we were able to observe up to 500 seconds gain thanks to the use of direct *I/O*s (asynchronous version) compared to the use of the pagecache (best of asynchronous and synchronous versions) on `CONV3D64`.

The `QIMONDA07` matrix is the one for which the *I/O* time for treating the factors *out-of-core* is the largest relatively to the time required for the *in-core* factorization (see Table 11). We observe here that relying only on the pagecache is not enough to correctly overlap *I/O* with computation. However, both the approach relying on the pagecache and the approach relying on direct *I/O* are very good when using an asynchronous buffered scheme at the application level.

Parallel case.

As explained previously, the *I/O* overhead is more critical in the parallel case as the delay from one processor has repercussions on other processors waiting for it. Nevertheless, we show in Table 6 the good behaviour of our approaches with the use of direct *I/O* (in particular when coupled with an asynchronous scheme) with the example of the `ULTRASOUND80` matrix. This time, the job submissions are done one after the other, limiting the possible interferences occurring on the nodes dedicated to *I/O* (and supposing that jobs from other users are not too much *I/O* intensive). This explains why results are generally better than the ones from Figure 7.

I/O mode	Scheme	1	2	4	8	16	32	64	128
Direct *I/O*	Synch.	1398.5	1247.5	567.1	350.9	121.2	76.9	44.6	36.5
Direct *I/O*	Asynch.	1360.5	(*)	557.4	341.2	118.1	74.8	45.0	33.0
P.C.	Synch.	1367.3	1219.5	571.8	348.8	118.5	69.6	44.8	90.0
P.C.	Asynch.	1376.3	(*)	550.3	339.2	109.4	73.8	45.2	30.0
IC		1340.1	(*)	(*)	336.8	111.0	64.1	40.3	29.0

Table 6: Elapsed time (seconds) for the factorization step of the `ULTRASOUND80` matrix with the use of direct *I/O*s or pagecache (P.C.) for both the synchronous (Synch.) and asynchronous (Asynch.) approaches, and compared to the *in-core* case (IC), for various numbers of processors.
(*) The factorization step ran out-of-memory.

Let us compare the asynchronous and synchronous schemes when using direct *I/O*s. When the number of processors becomes very large (64 or 128) the average volume of *I/O* per processor is very small for this test problem (15.3 MB on 64 processors, 7.7 MB on 128) and the average time spent in *I/O* mode is very low (less than 2.4 seconds) even in the synchronous scheme. Thus, the advantage of an asynchronous version is balanced by the cost of the management of the *I/O* thread. Except sometimes in these extreme cases (here 44.6 seconds versus 45.0 seconds on 64 processors), the asynchronous approach is more efficient than the synchronous one, as expected.

Concerning the comparison of the use of direct *I/O*s with the use of the system pagecache, we cannot decide which one is the best. Nevertheless, when we get a critical situation (here on a large number

of processors) the use of the system pagecache may penalize the factorization time, as observed on 128 processors in the synchronous case. One hypothesis is that the general purpose pagecache policy is not necessarily well-suited for so many simultaneous *I/O* requests.

In Table 7, we report the results obtained on one large symmetric matrix. In that case, we do not observe strong problems when using the pagecache. However, we can see that the asynchronous approach based on direct *I/O* has a good behaviour.

I/O mode	Scheme	1	2	4	8	16	32	64	128
Direct *I/O*	Synch.	983.7	595.3	361.3	158.2	69.8	41.6	26.9	21.5
Direct *I/O*	Asynch.	951.4	549.5	340.5	156.9	65.7	41.5	24.7	16.3
P.C.	Synch.	960.2	565.6	358.8	159.0	68.2	41.8	28.1	18.9
P.C.	Asynch.	948.6	549.6	336.6	153.7	65.8	40.4	26.8	16.1
IC		922.9	(*)	341.4	162.7	64.3	39.8	20.7	14.7

Table 7: Elapsed time (seconds) for the factorization step of the `CONESHL_MOD` matrix with the use of direct *I/O*s or pagecache (P.C.) for both the synchronous (Synch.) and asynchronous (Asynch.) approaches, and compared to the *in-core* case (IC), for various numbers of processors.
(*) The factorization step ran out-of-memory.

In conclusion, if the system pagecache fits well our needs in most cases, it seems that it may lead to singularly bad values when it is very stressed (either by a large amount of *I/O* - matrix `CONV3D64` - or by a high ratio *I/O*/Computation - matrix `QIMONDA07`). However, on this platform with GPFS and distant disks, leading to performance variations from one run to the other, we cannot draw definitive conclusions on the interest of direct *I/O*. This is why we now focus on the use of a machine with local disks (in the next section).

4.7 Experiments on machines with local disks

We have presented the behaviour of our algorithms on a platform with distant disks and seen that they were sometimes difficult to interpret. We now study their behaviour on a machine with disks local to the processors in order to validate our approaches on such an architecture and furthermore to show that we have a better scaling thanks to local disks when the number of processors increases. For these experiments, we use the cluster of bi-processors from PSMN/FLCHP presented in Section 4.1. As this machine has less memory, some of the main test problems have swapped or run out-of-memory, even when the factors are stored on disk. We thus first focus on results concerning some smaller problems among our auxiliary ones in order to have an *in-core* reference and then discuss some results about larger ones. Table 8 sums up the results.

Sequential case.

For the problems small enough so that the *in-core* factorization succeeds (see top of Table 8), we notice that the asynchronous *out-of-core* schemes are at most 10% slower than the *in-core* one. Moreover, our approaches behave globally better when based on the pagecache than when relying on direct *I/O*. Indeed the amount of data involved is low ; thus, when an *I/O* is requested, only a simple memory copy from the application to the pagecache is performed. Indeed, the corresponding average bandwidths observed are around 300 MB/s whereas the disk bandwidth cannot exceed 60 MB/s (maximum bandwidth) as observed when performing direct *I/O*s.

When comparing the two asynchronous approaches, we notice a slight advantage to rely on the pagecache. Actually it arises mainly from the cost of the last *I/O*. After the last factor is computed,

Matrix	Direct *I/O* Synch.	Direct *I/O* Asynch.	P.C. Synch.	P.C. Asynch.	IC
SHIP_003	43.6	36.4	37.7	35.0	33.2
THREAD	18.2	15.1	15.3	14.6	13.8
XENON2	45.4	33.8	42.1	33.0	31.9
WANG3	3.0	2.1	2.0	1.8	1.8
AUDIKW_1	2129.1	[2631.0]	2008.5	[3227.5]	(*)
CONESHL2	158.7	123.7	144.1	125.1	(*)
QIMONDA07	152.5	80.6	[[238.4]]	[[144.7]]	(*)

Table 8: Elapsed time (seconds) for the factorization step in the sequential case depending on the use of direct *I/O*s or pagecache (P.C.) for both the synchronous (Synch.) and asynchronous (Asynch.) approaches, and compared to the *in-core* case (IC), for several matrices, on a machine with local disks (PSMN/FLCHP).
(*) The factorization step ran out-of-memory. [2631.0] Swapping occurred. [[238.4]] Side effects of the pagecache management policy.

the *I/O* buffer is written to disk and the factorization step ends without any computation to overlap it. When relying on direct *I/O*, this last *I/O* is performed synchronously and then represents an explicit overhead for the elapsed time of the factorization. On the contrary, when based on the pagecache, only a memory copy to the pagecache is performed, and the system may perform the effective *I/O* later, after the factorization ends. Indeed, if we stop the timer just after the last factor is computed (*i.e.* at the end of the factorization, before this last *I/O* is performed), then we observe that the difference between the two (pagecache-based and direct *I/O*-based) asynchronous schemes is reduced from 1.4, 0.5, 0.8 and 0.21 seconds respectively for the SHIP_03, THREAD, XENON2 and WANG3 matrices to only 0.4, 0.2, 0.2 and 0.06 seconds.

As the main test problems (AUDIKW_1, CONESHL_mod, CONV3D64, ULTRASOUND80) are very large for this machine, the *in-core* case ran out-of-memory and the asynchronous *out-of-core* approaches swapped or could not be processed. This is illustrated by the AUDIKW_1 matrix in Table 8. Indeed, the use of such a huge *I/O* buffer requires a memory overhead of twice the size of the largest factor. The factorization step of this matrix requires 3657 MB with the synchronous approach and 4974 MB with the asynchronous one while the physical memory of a node is only 4 GB. Since swapping occurred, the asynchronous approaches were slower (resp. 2631 and 3228 seconds when based on direct *I/O* or on the pagecache) than the synchronous pagecache-based one (2009 seconds). This shows the importance of limiting the extra memory used for buffers. Using twice the size of the largest factor, as done in this prototype is clearly too much and we plan to reduce this size in the future, either by writing asynchronously to disk much smaller blocks or by enabling the memory management to write directly from the user area to disk in an asynchronous manner. Thus, the asynchronous approaches should be able to use approximatively the same amount of memory as the synchronous approaches.

When comparing the two asynchronous approaches to each other, we notice a higher overhead of the pagecache-based one, which consumes some extra memory hidden to the application. To illustrate this phenomenon, we used another ordering, PORD [24], which sometimes reduces memory requirements for the factorization step in comparison to METIS. The results with PORD on matrix AUDIKW_1 are reported in Table 9. In this case the new memory requirements for the asynchronous approach is of 3783 MB. We observed that the asynchronous scheme allows a factorization step in

Direct *I/O* Asynch.	P.C. Asynch.
1674	[[2115]]

Table 9: Elapsed time (seconds) for the factorization of matrix AUDIKW_1 when the ordering strategy PORD is used. Platform is PSMN/FLCHP. [2115] Swapping occurred. [[2115]] Side effects of the pagecache management policy.

1674 seconds when based on direct *I/O* without apparent swapping. However, when relying on the pagecache, the factorization step requires 2115 seconds: the allocation of the pagecache makes the application swap and produces an overhead of 441 seconds. This shows that it can be really dangerous to rely on the system pagecache, since memory is allocated that is not at all controlled by the user and it can induce swap in the application.

In Table 8, we observe that the CONESHL2 matrix is large enough so that the *in-core* execution ran out-of-memory and small enough so that no swapping occured during any *out-of-core* run. The results for this matrix confirm the good behaviour of the asynchronous approach based on direct *I/O*, as said just above for the AUDIKW_1 matrix in conjunction with the PORD ordering.

Nevertheless, again, the last *I/O*, which cannot be overlapped by any computation, appears more critical when relying on direct *I/O*. Indeed, for the AUDIKW_1 matrix coupled with the PORD ordering, in the asynchronous scheme, the size of the last *I/O* is 194.7 MB. As a direct *I/O* it costed 3.8 seconds (with an observed bandwidth of 51.5 MB/s corresponding to the characteristics of the disk) whereas, when based on the pagecache, it only costed 1.2 seconds (with an apparent bandwidth of 163.7 MB/s, higher than the bandwidth of the disk). For the CONESHL2 matrix reordered with METIS, it occurs alike and the penalty of the last *I/O* increases the factorization of 2.5 seconds when performing direct *I/O*s and only of 0.4 seconds when relying on the pagecache.

Let us now discuss the case of the matrix of our collection that induces the most *I/O*-intensive factorization, QIMONDA07. Assuming a bandwidth of 50 MB/s, the time for writing factors (85 seconds) is greater than the time for the in-core factorization (estimated to about 60 seconds). When relying on direct *I/O*, the asynchronous scheme is very efficient. Indeed, computation is well overlapped by *I/O*s: the factorization step only takes 80.6 seconds during which 60 seconds (estimated) of computation and 78.8 seconds (measured) of disk accesses are performed (with a 53.8 MB/s measured average bandwidth). Of course, on the contrary, for the synchronous approach based on direct *I/O*, these times are cumulated and the factorization step takes 152.5 seconds (among which 91.0 seconds spent in *I/O* mode). But the most original result (that confirms results obtained on our main target platform) is that the time for the pagecache-based schemes (both synchronous and asynchronous) are more than twice longer than the time for the asynchronous one with direct *I/O*. This shows that the pagecache policy targets general purpose applications and is not well adapted to very intensive *I/O* requirements.

Parallel case.

Table 10 gives the results obtained in the parallel case: we can draw similar conclusions as in the sequential case. First, for large matrices (see results for CONESHL_MOD and ULTRASOUND80), the use of the asynchronous approach relying on direct *I/O* has a good behaviour. Moreover, when the execution swaps (CONESHL_MOD on 1 processor or ULTRASOUND80 on 4 processors), the use of additional space either at the kernel level (pagecache) or at the application level (*I/O* buffer) increases the potential number of page faults which leads to a slow down so that the benefits of asynchronism (at the kernel level as well as at the application level) are lost. In the *I/O* dominant case (QIMONDA07 matrix) the pagecache still has difficulties to ensure efficiency. At last, to underline the importance of

benching *I/O* on large test problems, we report results obtained on a smaller problem (THREAD): the pagecache-based approaches seem to have a better behaviour; but this actually only means that most of the factors are kept in the pagecache and never written to disk.

Provided that enough data are involved, the *out-of-core* approaches appear to have a good scalability, as illustrated, for example, by the results on matrix CONESHL_MOD. The use of local disks allows to keep efficiency for parallel *out-of-core* executions.

Matrix	#P	Direct *I/O* Synch.	Direct *I/O* Asynch	P.C. Synch	P.C. Asynch	IC
CONESHL_MOD	1	4955.7	[5106.5]	4944.9	[5644.1]	(*)
	2	2706.6	2524.0	2675.5	2678.8	(*)
	4	1310.7	1291.2	1367.1	1284.9	(*)
	8	738.8	719.6	725.6	724.7	712.3
ULTRASOUND80	4	[373.2]	[399.6]	[349.5]	[529.1]	(*)
	8	310.7	260.1	275.6	256.7	(*)
QIMONDA07	1	152.5	80.6	[[238.4]]	[[144.7]]	(*)
	2	79.3	43.4	[[88.5]]	[[57.1]]	
	4	43.5	23.1	[[42.2]]	[[31.1]]	[750.2]
	8	35.0	21.1	[[34.0]]	[[24.0]]	14.6
THREAD	1	18.2	15.1	15.3	14.6	13.8
	2	11.3	9.9	9.3	8.9	8.6
	4	6.1	5.7	5.0	4.9	4.7
	8	6.3	5.2	4.4	4.3	3.3

Table 10: Elapsed time (seconds) for the factorization step on 1, 2, 4, and 8 processors, depending on the use of direct *I/O*s or pagecache (P.C.) for both the synchronous (Synch.) and asynchronous (Asynch.) approaches, and compared to the *in-core* case (IC), for several matrices, on a machine with local disks (PSMN/FLCHP).
(*) The factorization step ran out-of-memory. [85.7] Swapping occurred. [[238.4]] Side effects of the pagecache management policy.

A third platform.

In order to confirm the good general efficiency of our *out-of-core* approach, we have experimented with the smallest of our main large test problems on a another machine which has also local disks (CRAY XD1 system at CERFACS). Figure 8 shows that the *out-of-core* schemes perform as well as or even better than the *in-core* one on this third machine. On 4 and 8 processors, the superiority of *out-of-core* methods comes from cache effects (which are machine-dependent) resulting from freeing the factors from main memory and using always the same memory area for active frontal matrices. However, note that on two processors, the reason why the *in-core* run is slower is that swapping occurred.

4.8 Discussion

To conclude this section, let us write down the lessons learned and how this can help in the future developments. We were able to process large matrices on the IBM machine but have observed that, with GPFS and system *I/O* relying on a system cache, it was difficult to obtain perfectly reproducible

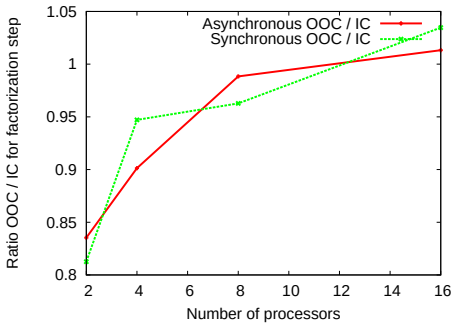


Figure 8: Elapsed time for the *out-of-core* factorization (normalized to the the *in-core* case) on a CRAY machine with local disks and the use of the pagecache for the CONESHL_MOD matrix on different number of processors. Note that the *in-core* run on two processors swapped.

results. Using direct *I/O* helped and does not imply uncontrolled extra-memory allocated by the kernel of the operating system.

In order to confirm these effects without the possible perturbations of GPFS and its accesses to distant disks (parallel accesses by different users, possibly several disks for a single file), we confirmed that the behaviour of direct *I/O* is efficient and stable on both machines. Furthermore, the use of local disks avoided the degradation of the bandwidths in parallel that were sometimes observed with GPFS.

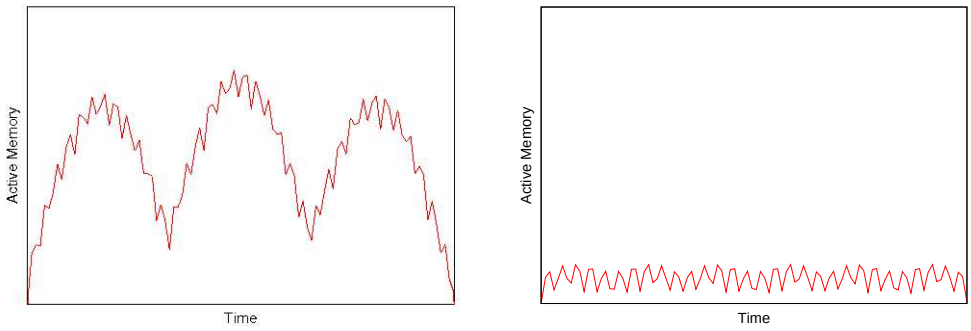
Globally, we could obtain better results with the use of direct *I/O*, but the system approach was not far behind, because all in all, there is some regularity in the disk accesses: we only write the factors sequentially to the disk. So, in most cases, efficiency is ensured by the system (asynchronism relies on the pagecache) as well as by explicit asynchronism at the application level. But in both cases the cost in terms of additional space (either at the kernel level with the pagecache or at the application level with the *I/O* buffer) is not negligible. As observed, it may induce swapping or even prevent from completing. We plan to develop an asynchronous algorithm which does not use any large (neither system or application) large buffer and would still allow efficiency. A careful implementation with an adequate memory management will be required.

Towards an out-of-core stack.

Keeping in mind that we plan to design an out-of-core version that also performs *I/O* on the contribution blocks, we can then expect to reach critical cases more often. Indeed the ratio *I/O*/computation will increase. In such cases, we have noticed that approaches relying on direct *I/O* are more robust and efficient. Moreover, files will be written and read in a more complicated manner. Similarly to [2], we can expect that the system will have more difficulty for read operations and that controlling the buffers at the application level (again with the use of direct *I/O*) will improve efficiency. For read operations this could imply to cleverly prefetch data from disk, which will be harder in the parallel case than in the sequential case. This is the object of future work.

In Table 11, we present the volume of factors, the volume of stack, and the estimated time for *I/O* operations assuming that the factors are all written to disk and that each contribution block is written to disk once and read from disk once. We will discuss in more details the volumes of *I/O* on the stack in Section 6, but note that algorithms aware of the physical memory available will not perform all the *I/O*s (by keeping as much data as possible *in-core*). So these estimations on the time for *I/O* operations with an *out-of-core* stack are an upper bound of what could be performed. We observe that in most cases this upper bound is of the same order as the time for the *in-core* factorization (obtained on our main target platform) while only the QIMONDA07 matrix reaches such a ratio if the stack remains

in-core. It confirms that we switch to a context in which the *I/O* layer may become critical for most matrices, and thus reinforces the interest of relying on direct *I/O* which was proved to be robust in this case.



(a) (b)

Figure 9: Active memory behaviour of two extreme matrices

We may already try to get some insight on how much *I/O* we could save with an algorithm aware of the physical memory available. We consider two extreme cases. In the first one, say (a) case, suppose that the volume of accesses to the active memory is only equal to a few times the peak of active memory. The typical memory behaviour of such matrices is represented in Figure 9(a). Then a large part of this volume is likely to be treated *out-of-core* as soon as the memory available decreases a little. On the contrary, say (b) case, suppose that this volume is very large compared to the peak of active memory as in Figure 9(b). Then, the whole active memory may be kept *in-core* even if the physical memory available is very low and so no *I/O* is needed. In practice, some particular matrices arising of linear programming have properties similar to the (a) case. For instance, the `GUPTA3` matrix accesses a volume of stack memory (1.06 GB) not very large compared to the peak of active memory (293 MB). So a large part of the volume of stack memory might be written. Moreover, as its volume of factors (80 MB) is relatively low, the challenging part of the *out-of-core* processing for this matrix is clearly the management of the stack memory. On the opposite, the `QIMONDA07` matrix, arising from a circuit simulation problem, has a very little peak of active memory (29 MB) compared to the volume of stack (7.2 GB), as in (b) case: it can be kept *in-core* and thus none of this huge volume has to be written to disk ! With such matrices, processing the factors *out-of-core* is enough.

These extreme cases have been added to the table for their special memory characteristics and they fix the limits of the scope of our work. Most matrices have an intermediate memory behaviour and a more accurate study of the volume of *I/O* on the stack that is to be performed is needed. This is the issue of Section 6.

Note that, on the `GUPTA3` matrix, modifications of the task allocation scheme of the multifrontal method could help significantly [16]. More generally, the *I/O* volume needed to process the whole factorization can be decreased by processing the tasks in adequate order on each processor, and by designing appropriate scheduling strategies for the parallel case.

Assuming that we will be able to design a parallel out-of-core approach that overlaps *I/O* with computations and minimizes the volume of *I/O*, we focus in the next section on the intrinsic limits of parallel multifrontal methods when both the factors and the contribution blocks are stored to disk: what are the minimum *in-core* memory requirements of the approach ?

Matrix	Volume (GB) of factors	Volume (GB) of stack	Time (seconds) for in-core factorization	*I/O* time (seconds) for factors	*I/O* time (seconds) for factors + 2 x stack
AUDIKW_1	11.4	53.5	2149.4	142.5	1480.0
CONESHL_MOD	6.3	28.5	922.9	78.8	791.3
CONV3D_64	39.4	64.9	8351.0 (*)	492.5	2115
ULTRASOUND80	7.9	19.4	1340.1	98.8	583.8
BRGM	37.8	182.5	9214.8 (*)	472.5	5035.0
GUPTA3	0.09	1.14	6.6	1.1	29.6
QIMONDA07	4.5	7.2	90.7	56.3	236.3
SPARSINE	2.2	14.0	711.7	27.7	37.8

Table 11: Time for factorization (obtained on our main target platform) and time for *I/O* when assuming a disk bandwidth (both for writing and reading) of 80 MB / second, in the sequential case, for several test problems.
(*) For these matrices, we actually report the best *out-of-core* time obtained as the *in-core* case ran out-of-memory.

5 Simulation of an *out-of-core* stack memory management

In Section 4, we presented a first *out-of-core* approach for the parallel multifrontal factorization, consisting in writing factors to disk as soon as possible. The results obtained have shown the potential of the approach and how larger problems can be treated. However this approach also has certain memory limitations and the active memory now becomes the limiting factor. Also, for cases where the stack memory was already predominant (typically, certain matrices on a large number of processors), this approach may not be sufficient. Therefore, the next step is to manage the stack of contribution blocks with an *out-of-core* scheme, where a contribution block may be written to disk as soon as it is produced, and read from disk when needed (either with a prefetching mechanism or with a demand-driven scheme). We focus on memory aspects: thus, combined with the lessons learned in the previous section thanks to the performance study on our prototype, we hope the results of this section will allow us to develop a whole efficient *out-of-core* parallel multifrontal method.

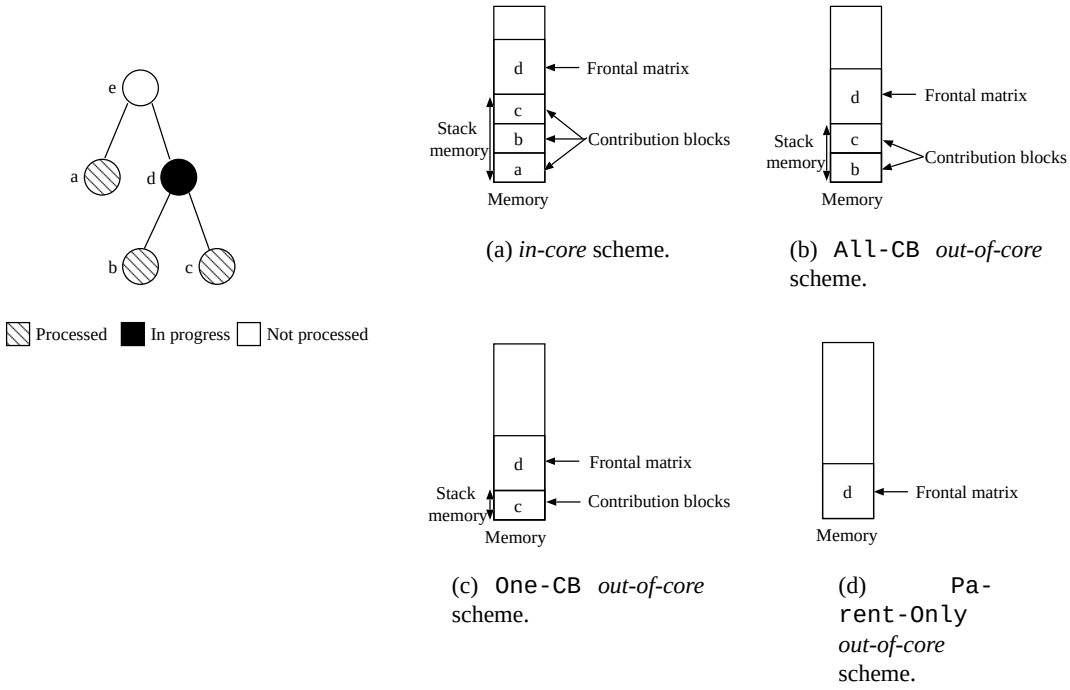
In this section, we present simulation results with various stack management strategies, with the objective to better understand the memory behaviour of our parallel multifrontal code, and identify the possible bottlenecks to treat arbitrarily large problems.

The various management schemes for the stack memory are given in Figure 10: we compare the *in-core* stack memory management strategy that corresponds to what has been done in Section 4 with the three following *out-of-core* schemes:

- `All-CB` ***out-of-core*** **stack memory.** In this scheme, we suppose that during the assembly step of an active frontal matrix, all the contribution blocks corresponding to its children have been prefetched in memory. Thus, the assembly step is processed as in the *in-core* case.

- `One-CB` ***out-of-core*** **stack memory.** In this scheme, we suppose that during the assembly step of an active frontal matrix, only one contribution block corresponding to one of its children is loaded in memory, while the others stay on disk. Thus we interleave the assembly steps with *I/O* operations.

- `Parent-Only` ***out-of-core*** **stack memory.** In this scheme, we suppose that during the assembly step of an active frontal matrix, no contribution block is loaded in memory. Thus, the



(a) *in-core* scheme.

(b) `All-CB` *out-of-core* scheme.

(c) `One-CB` *out-of-core* scheme.

(d) `Parent-Only` *out-of-core* scheme.

Figure 10: Stack memory management schemes. (Left) Front of d is being assembled. (Right) State of the memory.

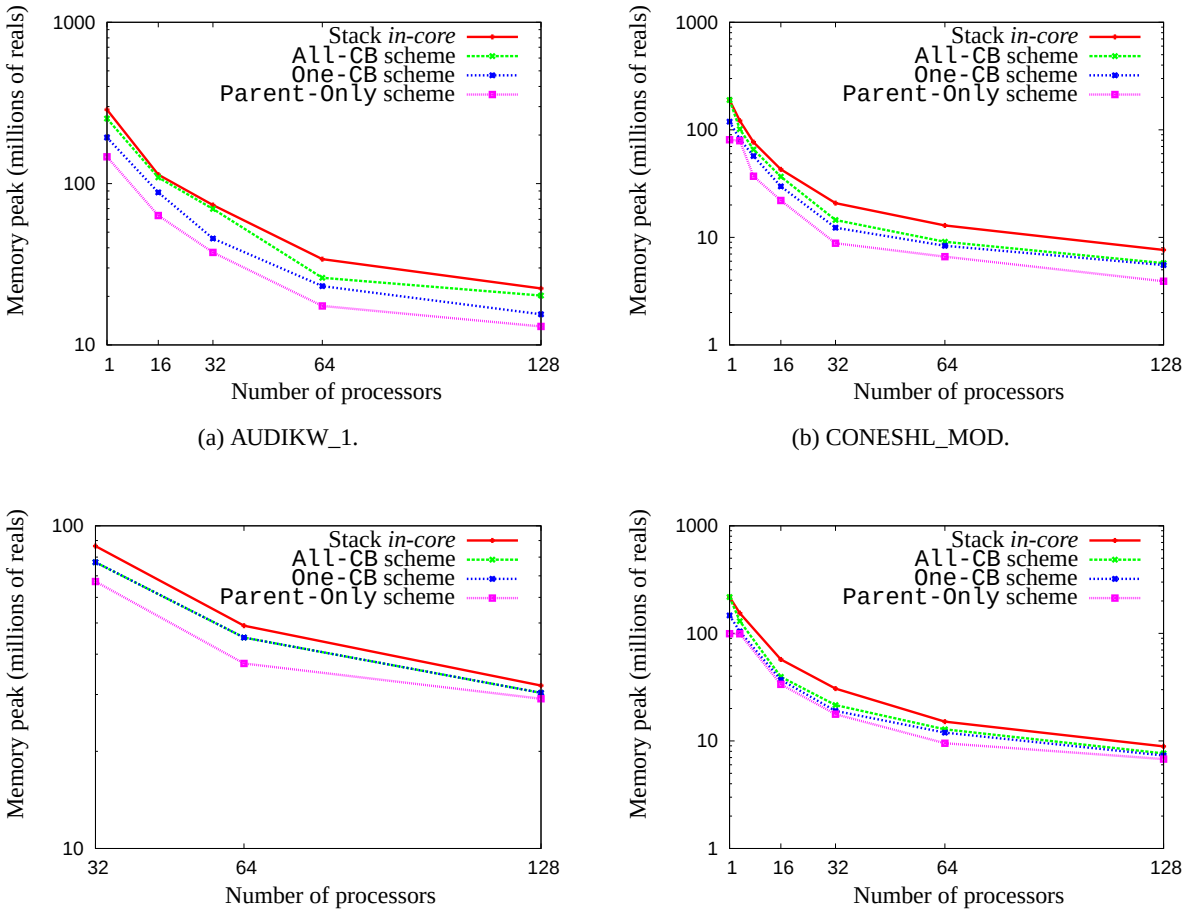
assembly step is done in an *out-of-core* manner. Note that the implementation of such a strategy will not be efficient at all since the assembly steps are not very costly and there is no way to overlap *I/O* operations with computations. But this strategy corresponds to an ideal scenario concerning the minimum core memory requirement.

Note that for the three scenarios, we suppose that a contribution block is written to disk as soon as it is computed. In addition, we assume that all the active frontal matrices remain in memory until the end of their factorization. For our experiments we have performed executions of an instrumented version of `MUMPS`, that simulates the different scenarios during the numerical factorization step and traces the memory usage. On each processor, the peak of memory is stored, and we are then interested in the maximal memory peak over the processors.

5.1 Results and discussion

We show in Figure 11 the peaks of memory obtained when the active memory is fully kept *in-core* and when it is managed out-of-core using the different *out-of-core* memory management strategies, for our main test problems on different numbers of processors.

As expected, we see that the strategies for managing the stack *out-of-core* provide a reduced memory requirement with a scalability as good as the one of the *in-core* stack. We also observe that the `Parent-Only` *out-of-core* stack memory management is the one that best decreases the memory needed by the factorization. Although this strategy might not be good for performance, it is here to provide some insight on the best we can do with our assumptions and with the current version of the code. One interesting phenomenon we observed is that the *out-of-core* stack memory



(a) AUDIKW_1.

(b) CONESHL_MOD.

(c) CONV3D64.

(d) ULTRASOUND80.

Figure 11: Memory behaviour with different memory management strategies on different numbers of processors for our main test problems (`METIS` is used as reordering technique).

management strategies give better results with symmetric matrices (see Figures 11(a) and 11(b)) than with unsymmetric ones (see Figures 11(c) and 11(d)). We will analyse more accurately the reasons in the next subsection.

Concerning the comparison between the three *out-of-core* stack memory management strategies, the `All-CB` management reduces the amount of active memory by a factor of around 15% for both the symmetric and the unsymmetric matrices compared to the case where the stack is in core. For the `One-CB` approach, we can see that the reduction of the size of the active memory is around 30% (resp. 20%) for the symmetric (resp. unsymmetric) matrices. Finally, the `Parent-Only` approach reduces the size of the active memory by an average factor of 50% (resp. 25%) for the symmetric (resp. unsymmetric) matrices. Of course keeping a good efficiency represents a challenge of increasing difficulty with the management targeted from the `All-CB` scheme to the `Parent-Only` one. Nevertheless, these figures show that each step may lead to important memory improvements and thus motivates a thorough study (see Section 6) to decide which one to develop.

We saw that the strategies for managing the stack *out-of-core* provide a reduced memory requirement with a scalability as good as the one of the *in-core* stack. We also observed (as expected) that

the `Parent-Only` *out-of-core* stack memory management is the one that best decreases the memory needed by the factorization. We give in Section 6 an estimation on the overhead of these approaches in terms of *I/O* volume.

In our approaches, both the factors and the contribution blocks are supposed to be managed *out-of-core* and so only the frontal matrices remain *in-core*. Consequently, if we want now to decrease further the memory requirements, we have to bound the memory occupation of the frontal matrices. Several directions are possible for this. First, we could reduce as much as possible the number of simultaneous active tasks on a processor. This can be done by modifying the scheduling strategies currently existing in the parallel multifrontal method. A second possibility would consist in relaxing the assumption that an active task always fits in memory. This actually corresponds to develop an *out-of-core* frontal matrices mechanism. Nevertheless, these two directions (limiting the number of simultaneous active tasks and managing the frontal matrices *out-of-core*), imply respectively a loose of freedom for the scheduling and an *I/O* overhead. Both these drawbacks make it hard (but interesting) to follow these proposed directions while keeping a good efficiency. Moreover, limiting the number of active tasks by processor may lead to strongly unbalance the processing of the assembly tree which paradoxically may overload the memory of some processors. It is what we observed by bounding the number of simultaneous master tasks to one. Another approach consists in limiting the size of the tasks used for the treatment of the frontal matrices. It is natural to look first in this direction because it does not require heavy developments to evaluate its potential and because we guess that the overhead on the global execution time should be smaller. In the next two subsections we present a finer memory study together with some basic mechanisms that aim at further reducing the core memory needed according to this last direction.

5.2 Analysing how the memory peaks are obtained

We now analyze in more details what type of tasks cause the peaks for each strategy. Table 12 shows the state of the memory when the peak is reached on the processor responsible for the peak, in the case of an execution on 64 processors for the AUDIKW_1 problem. Note that, based on load balancing criteria, the dynamic scheduler may allocate several tasks to one processor (each type of task is defined in part 2). We notice that for the `Parent-Only` and `One-CB` *out-of-core* schemes as well as for the active memory *in-core* case, the memory peak is reached when a subtree is processed (more precisely when the root of that subtree is assembled). In the `Parent-Only` case, the processor also has a slave task activated. For the `All-CB` scheme, the peak is reached because the scheduler has allocated simultaneously too many slave tasks (3) to one processor, reaching together 42.97% of its memory. Note that it was also responsible for a master task but its size is less important (5.93%). Similarly to matrix `AUDIKW_1`, we have indeed studied the memory state for almost all the problems presented in Table 1, on various numbers of processors. Rather than presenting all the results, we preferred to only present here the main phenomena observed on a representative example. We nevertheless give another example for an unsymmetric matrix (`CONV3D64`) in Table 13.

For the symmetric problems (`AUDIKW_1`, but also `SHIP_003`, `CONESHL2`, `CONESHL_MOD`, for example), between 8 and 128 processors, the peak is reached[11] when the root of a sequential subtree is assembled; this occurs for all *out-of-core* schemes. Sometimes a slave task may still be held in memory when the peak arises (and it can then represent between 25 and 75 % of the memory of the

[11] Except that (i) for the `CONESHL_MOD` problem on 64 processors the peak for the `Parent-Only` scheme arises when the root of the overall tree is processed; and (ii) for the `AUDIKW_1` problem on 64 processors, the peak for the `All-CB` scheme is reached early in the factorization process (22% of the factorization time is then elapsed) while one single processor is simultaneously responsible for three slave tasks.

Scheme	Memory ratio of the active tasks			Memory ratio of
	master tasks	slave tasks	sequential subtrees	the contribution blocks
Stack *in-core*	0%	0%	$27,11\%^*$	$72,89\%$
`All-CB`	$5,93\%$	$42,97\%^*$	0%	$51,10\%$
`One-CB`	0%	0%	$75,10\%^*$	$24,90\%$
`Parent-Only`	0%	$48,32\%$	$51,63\%^*$	$0,04\%$

Table 12: Memory state of the processor that reaches the global memory peak when the peak is reached, for each *out-of-core* scheme and for the stack *in-core* case, on the `AUDIKW_1` problem with 64 processors. Symbol * in a column refers to the last task activated before obtaining the peak, which is thus *responsible* for it. When a sequential subtree is responsible for the peak, we observed that it is (here) at the assembly step of its root; so the numerical value reported in the corresponding column represents the amount of memory of the frontal matrix of the root of this subtree.

Scheme	Memory ratio of the active tasks			Memory ratio of
	master tasks	slave tasks	sequential subtrees	the contribution blocks
Stack *in-core*	0%	$40.19\%^*$	0%	59.81%
`All-CB`	0%	$65.71\%^*$	0%	34.29%
`One-CB`	38.89%	$46.27\%^*$	0%	14.84%
`Parent-Only`	47.82%	$52.06\%^*$	0%	0.12%

Table 13: Memory state of the processor that reaches the global memory peak when the peak is reached, for each *out-of-core* scheme and for the stack *in-core* case, on the `CONV3D64` problem with 64 processors. Symbol * in a column refers to the last task activated before obtaining the peak, which is thus *responsible* for it.

active tasks on the processor). For `CONESHL2`, a smaller symmetric problem, this behaviour remains globally true but it is less systematic (except for the `Parent-Only` scheme for which it remains systematic). Indeed, the main reason is that the memory of the active tasks is low compared to the one of the contribution blocks; the memory peak may thus arise just because we get one or several large contribution blocks.

For the unsymmetric problems (`CONV3D64`, `ULTRASOUND80`), on many processors (from 16 to 128), the peak is generally obtained because of a large master task. This is increasingly true when we tend to the `Parent-Only` scheme. With fewer processors (less than 8), the assembly of a root of a subtree is more often responsible for the peak. Nevertheless, these effects are sometimes hidden when many (2 up to 6) tasks are simultaneously active. For example, on 64 processors with the `All-CB` scheme, for the `CONV3D64` problem, the peak is obtained while a processor has four slave tasks simultaneously in memory.

Thanks to parallelism, memory needs of a particular task can be parcelled out over many processors. Ideally, platforms with an arbitrarily large number of processors should thus enable the factorization of arbitrarily large problems. However, in order to be efficient, some tasks are sequential and become the memory bottleneck when the other ones are parallelized.

First, in `MUMPS`, to bound the number of communications, the nodes at the bottom of the tree are aggregated into subtrees which are treated sequentially (see Figure 1). Such a subtree may then be critical in terms of memory, its peak (usually arising when its root is performed) being the memory bottleneck of the whole factorization step. We observed it was particularly true for symmetric problems.

Next, the processor responsible for a master task treats sequentially all the fully summed rows of

the corresponding frontal matrix (only the blocks matching the Schur complement can be distributed over several processors). This way, with a large number of processors, their treatment becomes critical. Figure 2 shows that the memory needs corresponding to master tasks are more important for unsymmetric cases than for symmetric ones. On the range of processors used, the limiting factor observed is indeed the treatment of master tasks for unsymmetric problems and the one of the subtrees in the symmetric case.

5.3 Decreasing the memory peaks

It results from the previous section that in order to decrease the memory needs, the size of the master tasks has to be limited for the unsymmetric problems whereas the size of the subtrees has to be diminished for the symmetric ones. Furthermore, applying together these two approaches could further improve scaling. On a limited number of processors, the number of simultaneous active tasks should moreover be bounded.

Concerning large master tasks, we can use the splitting algorithm of [4]. Since the factorization of the pivot rows of a frontal matrix is performed by a single (master) processor, we replace the frontal matrix in the assembly tree by a chain of frontal matrices with less pivot rows, as illustrated in Figure 12. This limits the granularity of master tasks, at the cost of increasing the cost of assemblies from children to parents.

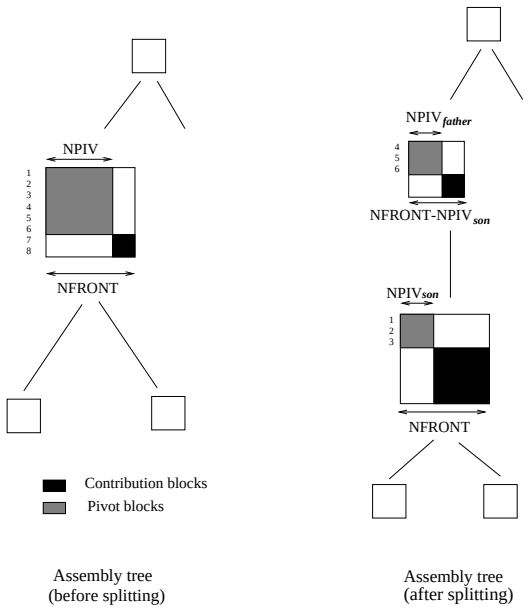
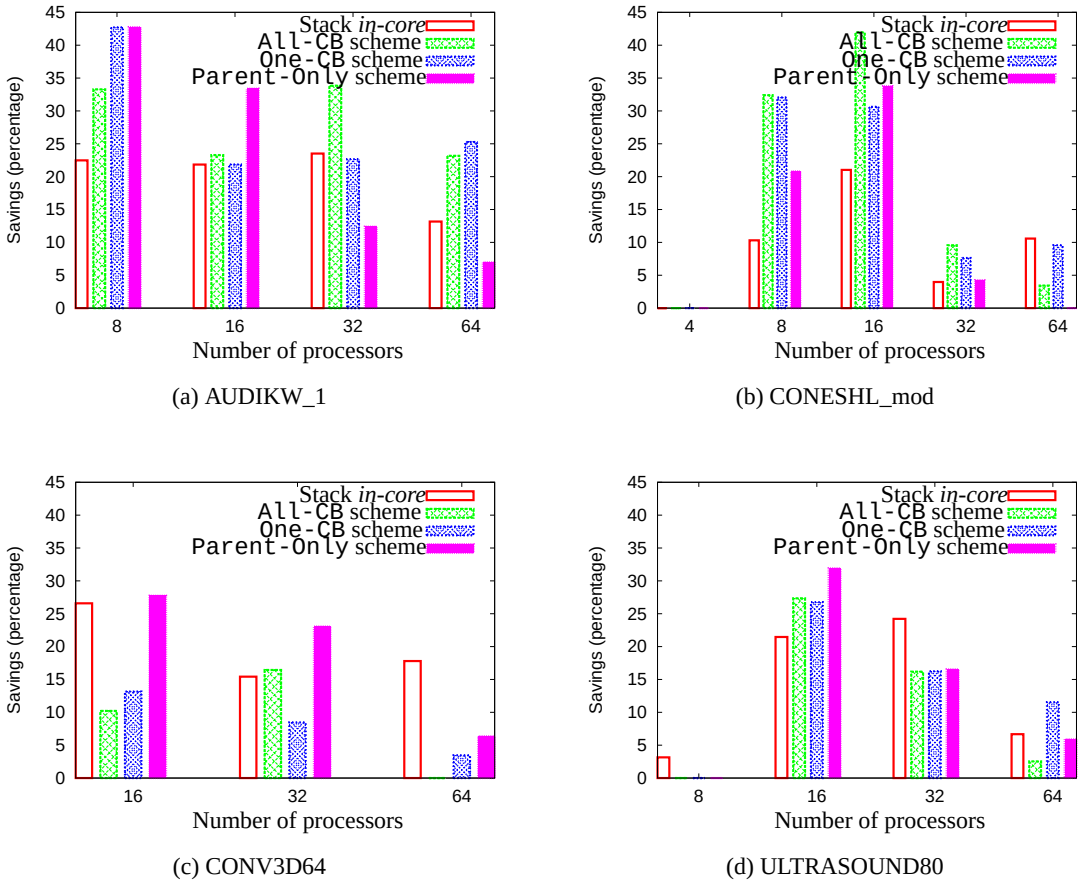


Figure 12: Tree before and after the subdivision (or splitting) of a frontal matrix with a large pivot block.

Concerning the size and the topology of the subtrees, they are currently based on load balancing criteria. For the symmetric problems, we have modified the corresponding threshold by hand to diminish the size of the subtrees. As shown in Figure 13(a) for the `AUDIKW_1` problem, we can save up to more than 40% on large symmetric problems. In particular, the `One-CB` scheme, which (as shown above) is a good balance between performance and memory, saves more than 20% at every execution on the range of processors used (8 - 64). Note that decreasing the size of the subtrees allows to decrease the global memory peak not only because (in most cases) it was obtained when processing

a root of a subtree but also (sometimes) because decreasing the granularity of these sequential tasks allows a better load balancing when processing the tasks just above these ones in the assembly tree. For the `AUDIKW_1` problem on 64 processors, this second reason (and only this one) explains the saving of 23% of memory with the `All-CB` scheme (Figure 13(a)).



(a) AUDIKW_1 (b) CONESHL_mod

(c) CONV3D64 (d) ULTRASOUND80

Figure 13: Memory savings for two symmetric problems, 13(a) and 13(b) cases (resp. for two unsymmetric problems, 13(c) and 13(d) cases), obtained by decreasing the size of the subtrees (*resp.* by splitting the master tasks), for several stack memory management schemes, on various numbers of processors. `METIS` is used to permute the matrices.

For the unsymmetric matrices, we have split the largest master tasks. The corresponding node is replaced by a chain of nodes. Figure 13(c) shows that for the `CONV3D64` problem we get important savings (from 8.5 up to 27.8%) except on 64 processors for which we did not manage to get a good tuning.

When we split master tasks of unsymmetric matrices, we have observed that the new memory peak then sometimes arises when a subtree is processed. Thus, we have tried to both split large master tasks and reduce the size of the subtrees. For the `CONV3D64` problem, with the `One-CB` strategy on 32 processors, splitting the master tasks allows us to process the problem with a memory of 69 million reals per processor, that is a 8.5% saving (see Figure 13(c)); but additionally decreasing the size of the subtrees makes it possible to treat it with 62 millions of reals which represents this time a 17.8%

saving. Reciprocally, splitting the master tasks of the symmetric problems after reducing their subtrees sizes allowed us to increase the memory savings in several cases. The `AUDIKW_1` problem illustrates this phenomenon for which on 32 processors 23.2% of memory is then saved instead of 12.4% without splitting. Nevertheless, with these memory improvements a new problem arises: the elapsed time for the factorization step increases. For example, for the `CONV3D64` problem on 32 processors with the `Parent-Only` strategy, splitting does allow us to save 23% but we then observed an overhead of 20% on the elapsed factorization time (418.8s $\rightarrow$ 496.9s). We face a key point of the future work: decreasing memory requirements while keeping good performance. Indeed, in the current scheme implemented, the mapping of the chain of nodes built when splitting nodes with a large master task implies a communication overhead that we plan to reduce in the future.

These results show the potential of the parallel *out-of-core* multifrontal method: it seems that the intrinsic limits of the sequential multifrontal method become much less critical thanks to parallelism. In the two cases (symmetric and unsymmetric), we have modified thresholds of load balancing constraints to save memory. The thresholds have been tuned specifically for each case. New criteria and algorithms based on memory constraints now have to be designed to determine the size of the subtrees and control the splitting strategies for master tasks. Furthermore, all scheduling decisions must be adapted to fit in the *out-of-core* scheme and avoid too many simultaneous active tasks.

Now that we have a reasonable idea of the core memory requirement with an *out-of-core* stack, another important issue consists in estimating the volume of *I/O* that will be involved. The next section aims at estimating the cost of such an extension in terms of *I/O* volume overhead.

6 *I/O* volume overhead

6.1 Defining the volume of *I/O*

Assuming that temporary data (*i.e.* contribution blocks) are written to disk as soon as possible, the volume of *I/O* will exactly correspond to the volume of stack memory. This represents an upperbound of the volume of *I/O*, since each contribution block is exactly written once and read once whereas this is not stricly necessary. The volume of *I/O* in this scheme was discussed earlier, in Section 4.8. It is important to note that for a given memory size, moving from an "as soon as possible" scheme to a "as late as possible" scheme can reduce the volume of *I/O* significantly. Indeed, in the latter case, given a volume of available memory, some contribution blocks (or parts of contribution blocks) can be kept in memory and consumed without being written to disk. In this section, we will thus focus on the computation of the volume of *I/O* for the stack obtained in this context. For this purpose, we assume that the contribution blocks are written only when needed (possibly only partially), that factors are still written to disk as soon as they are computed and that a frontal matrix must completely fit in memory. The objective of this study is to provide a better estimation of the volume of *I/O* associated to the stack than in Table 11, although this smaller bound will require more efforts on the implementation side regarding memory management issues, and will be more difficult to obtain, especially in a parallel context.

We first introduce some notations that we will use to compute the volume of *I/O* in a formal way. Note that we focus here on the sequential case, as this is a natural starting point, easier to analyze, and the implications of this section will generalize (to some extent) to the parallel case.

Let M_0 be the memory available for the multifrontal factorization. As described in Section 2, 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 m the storage requirement for the frontal matrix of the parent node, m_j the

storage for the frontal matrix of child j and cb_j the storage requirement for the contribution block of child j. We also denote by A the active memory required to process a complete subtree rooted at a given node. Note that for a leaf node, we have: $A = m$. Considering only a parent and its children, we denote by A and A_j the active memory requirement to process the parent and the child j, respectively.

When processing a child j, the contribution blocks of all previously processed children have to be stored. Their memory size sums up with the active memory need A_j of the considered child, leading to a storage equal to $A_j + \sum_{k=1}^{j-1} cb_k$. Furthermore, since the parent is allocated after all its children have been processed, the active memory contains the contribution blocks of all children. Thus, when the parent (of size m) is allocated, this leads to 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:

$$A = \max \left(\max_{j=1,n} (A_j + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1}^{n} cb_k \right) \tag{1}$$

Now assume that for our *out-of-core* factorization we are given a memory of size M_0. If $A > M_0$, some *I/O* will be necessary. Since the contribution blocks are accessed with a stack mechanism, writing the bottom of the stack when there is a lack of memory 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, A_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 A 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 memory peak is:

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

Note that $V^{I/O}$ is both the volume of data written and then read. Let us consider now a more general context where each child may root a subtree. In this new context, for a child j, A_j denotes the active memory peak observed while processing its subtree. If we suppose that $\forall j : A_j \leq M_0$, 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 $\exists k : A_k > M_0$. We know that the child k will have an intrinsic volume of *I/O* $V_k^{I/O}$ (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 having exactly M_0 as memory needs, and having $V_k^{I/O}$ as intrinsic volume of *I/O*. We can now generalize the expression given in Formula (2), which becomes:

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

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.

6.2 Experiments

Figure 14 presents the *I/O* volume depending on the memory available with the "as late as possible" scheme introduced in the previous subsection and computed with Formula (3) for most of the matrices presented in Table 1. Note that this volume of *I/O* does not depend on the *out-of-core* scheme management (`Parent-Only`,`One-CB`, or `All-CB`). Nevertheless, the domain of validity of the schemes differ from each other. The one corresponding to the `Parent-Only` scheme is the largest one: it includes all the other ones and is limited by the size of the largest frontal matrix. When the memory available grows, we can also use the `One-CB` scheme (the frontier corresponds to the vertical plain line of Figure 14). When the memory grows further, the `All-CB` scheme can also be applied (vertical dashed line). Finally with enough memory (more than the peak of active memory), the stack can be processed *in-core*.

When the memory available is close to the peak of active memory, we observe that most matrices only need to process *out-of-core* a volume equal to the difference between the peak of active memory (which corresponds to the notation A of the previous section for the root node of the tree) and the memory available M_0. Geometrically we observe that for this interval, the *I/O* curve matches the line of equation $y(M_0) = peak - M_0$: when the memory available decreases by 1 MB (say), the volume of *I/O* increases by 1 MB.

For a smaller amount of memory, the *I/O* volume does not follow this ideal case anymore and grows faster than it: when the memory available decreases by 1 MB, the volume of *I/O* increases by more than 1 MB.

When the memory available is close to the minimum value for which we can process the matrix (with a `Parent-Only` scheme), the volume generally increases strongly. However, it remains far from the upper bound given by the volume of stack in the third column of Table 11 (except for the `GUPTA3` matrix), which corresponds to the volume of *I/O* in a scheme for which the whole stack would be processed systematically *out-of-core*.

Let us now discuss again the two extreme cases `QIMONDA07` and `GUPTA3` matrices identified in Section 4.8. On the `QIMONDA07` matrix (arising from circuit simulations), the *I/O* volume for the stack never exceeds the straight line of equation $y(M_0) = peak - M_0$ and represents less than 1% of the volume of factors. Thus treating the stack *out-of-core* is cheap. However, the peak of active memory (29 MB) is also extremely small compared to the volume of factors (7.2 GB) and treating the stack *out-of-core* is thus not necessary: once factors are on disk, the stack can be kept *in-core*.

On the contrary, the `GUPTA3` matrix (arising from a Linear Programming problem), requires a huge active memory. However, this matrix does not have any large frontal matrices; subsequently with an *out-of-core* stack management, we can process it with a very little amount of core memory (compared to the memory required to process it with an *in-core* stack) . . . at the cost of performing a huge amount of *I/O* (up to 11.2 times the volume of factors).

6.3 On the shape of the graphs: Reinterpretation

In the previous subsection, we have noticed that the straight line $y(M_0) = peak - M_0$ is a lower bound for the volume of *I/O* on the stack memory as a function of M_0 ($V^{I/O} = f(M_0)$). We now aim at extending and reinforcing this remark. The motivation is the following: we have observed geometrically that the volume of *I/O* matches this lower bound when the memory available is close to the peak of active memory but that the gap to this bound increases when the memory available decreases; we wonder whether we can quantify more accurately how much the gap increases when the memory available decreases.

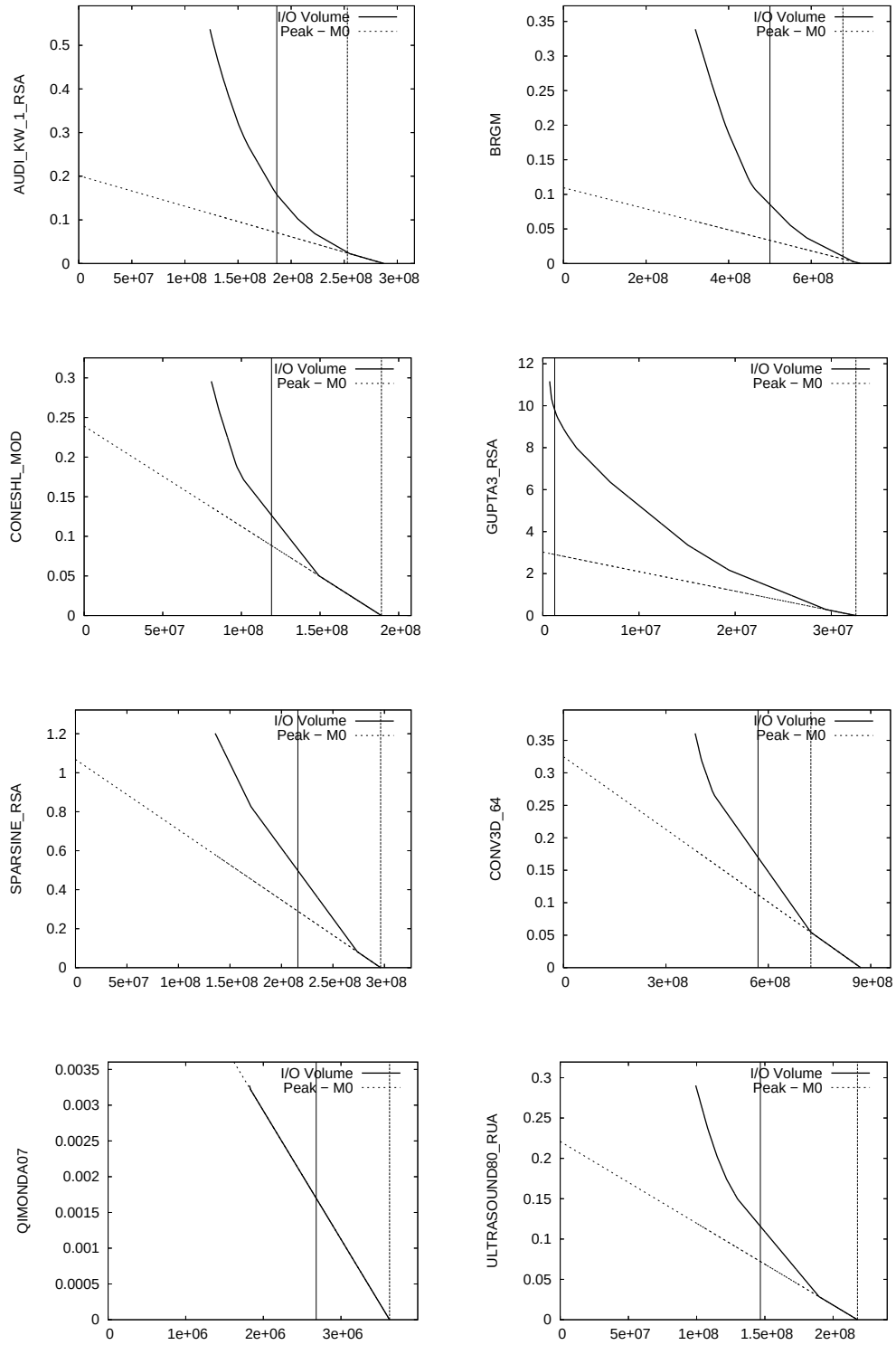


Figure 14: Volume of *I/O* for the stack divided by the volume of factors with an "as late as possible" write scheme for several problems depending on the memory available M_0 (x axis, expressed in number of real entries). The vertical plain (*resp.* dashed) line represents the minimum amount of memory necessary for processing the matrix with a `One-CB` (*resp.* `All-CB`) scheme. The line of equation $y(M_0) = peak - M_0$ given as reference represents the difference between the peak of active memory and the memory available M_0 and is a lower bound of the *I/O* volume

This quantification is formally given by Corollary A.2 in Annex A and allows one to reinterpret the previous results as follows. When the memory available is larger than the peak of active memory, no *I/O* on the stack memory is needed. When it becomes slightly smaller than it, if the memory available decreases by 1 MB (say), the volume of *I/O* increases by 1 MB. For a still smaller amount of memory, if the memory available decreases by 1 MB (say), the volume of *I/O* increases by **2** MB. And so on, the volume of *I/O* may increase by **3**, **4**, **5** MB, ... if the memory available decreases by 1 MB (say) relatively to smaller and smaller given values of memory available. We have no guarantee that each integer value is reached, but Corollary A.2 states that we know that they are *integer values* and that they *increase* when the memory available decreases.

More formally we may express it in terms of steepness (as in the proof of Corollary A.2). Observing that $y(M_0) = peak - M_0$ is a linear function of steepness equal to -1, we have to show that the function $V^{I/O} = f(M_0)$ is a piecewise affine function and that the steepness of each piece is a negative integer multiple of -1 whose absolute value decreases when the value of M_0 increases. For example (see again Figure 14), we can exhibit 3 steepnesses of values -3, -2, -1 in the case of the `CONV3D64` matrix, one of value -1 in the case of the `QIMONDA07` matrix (this is the ideal case, the associated function is strictly affine), and a large range of such negative integer values in the case of the `GUPTA3` matrix (this is a costly case, the associated function does not event look piecewise affine at first sight). We refer the reader to the annex for a more detailed description of this property.

7 Lessons learned

We have presented in this report a first implementation of an *out-of-core* extension of the parallel multifrontal solver `MUMPS`. The selected approach was to drop factors from memory as soon as they are computed and to overlap the *I/O* operations as much as possible with computations. We illustrated the good behaviour of this approach on a small number of processors and its limitations on larger ones. From a performance point-of-view, we have shown that the low-level *I/O* mechanisms must not be neglected. They have to be designed with care as the system is not tuned for applications with *I/O*-intensive needs or large memory requirements. We have seen that an asynchronous approach coupled with direct *I/O* could provide a good solution.

One bottleneck of this asynchronous approach is the dimension of the *I/O* buffer that can be critical with respect to the overall memory. To address this problem, we have to improve the memory management algorithms of `MUMPS` during the factorization either to allow asynchronous write operations directly from the `MUMPS` space (thus avoiding copies to the *I/O* buffer), either to allow for a mechanism that writes smaller parts of frontal matrices to the buffer while the frontal matrix is being factored. In the latter case, the *I/O* buffer may also be used mainly to aggregate too small *I/O* requests (for performance reasons).

We have then studied the minimum memory requirements of a parallel multifrontal method when parts of the active memory (contribution blocks) are also stored on disk. In that case, the contribution blocks can be considered as read-once/write-once data accessed with a near-to-stack mechanism (for the parallel case the accesses are more irregular). We identified some key parameters (size of subtree tasks, size of master tasks) that impact significantly the scalability of the minimum core memory when increasing the number of processors. The main improvements came (i) from splitting master tasks that involve a large amount of memory into a chain of parallel nodes, and (ii) limiting the size of the subtrees when those lead to a critical memory behaviour.

We have also made a study of the volume of *I/O* in the sequential case, and shown that, in general, the volume of *I/O* for the stack memory can remain reasonable compared to the volume of factors,

when one avoids to write contribution blocks systematically to disk. Having observed that the *I/O* for the factors can overlap nicely with the computations, this means that an *out-of-core* factorization allowing the stack memory to be stored on disk could have a good efficiency as long as we are not blocked too often on read operations. This statement should also be true (although to a lesser extent given the non-perfect memory scalability of the stack memory) in the parallel case.

As noted in [21, 22] one drawback of the sequential multifrontal approach in an *out-of-core* context consists in the large frontal matrices that can be a bottleneck for memory: allowing the *out-of-core* storage of the contribution blocks sometimes only decreases the memory requirements by a factor of about 2. However, we have seen that parallelism can further decrease these memory requirements significantly. Going further requires to process frontal matrices *out-of-core* (assembly and factorization).

In the parallel case, the number of contribution blocks and active tasks that a processor has in memory is closely related to the scheduling decisions made. Both the static and dynamic aspects of scheduling could help limiting the *I/O* volume that each processor has to perform. For example, the dynamic scheduler could give priority to tasks that depend on/consume contribution blocks already in memory. We also envisage to work on techniques inspired from [18] and [16] to reduce the amount of data written and read from disk (finding the best tree traversal, constructing memory-minimizing schedules). Finally, avoiding too many simultaneous tasks and limiting the granularity of the tasks can help reducing both the *I/O* volume and the memory requirements but an adequate tradeoff must be found in order to maintain performance.

References

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

[2] P. Amestoy, I.S. Duff, A. Guermouche, and T. Slavova. A preliminary analysis of the out-of-core solution phase of a parallel multifrontal approach. Technical report, CERFACS-ENSEEIHT-IRIT-INRIA, 2006. In preparation.

[3] 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.

[4] 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.

[5] 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.

[6] P. R. Amestoy, I. S. Duff, and C. Vömel. Task scheduling in an asynchronous distributed memory multifrontal solver. *SIAM Journal on Matrix Analysis and Applications*, 26(2):544–565, 2005.

[7] 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.

[8] 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.

[9] T. H. Cormen, E. Riccio Davidson, and Siddhartha Chatterjee. Asynchronous buffered computation design and engineering framework generator (abcdefg). In *19th International Parallel and Distributed Processing Symposium (IPDPS'05)*, 2005.

[10] O. Cozette. *Contributions systèmes pour le traitement de grandes masses de données sur grappes*. PhD thesis, Université de Picardie Jules Vernes, 2003.

[11] Olivier Cozette, Abdou Guermouche, and Gil Utard. Adaptive paging for a multifrontal solver. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 267–276. ACM Press, 2004.

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

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

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

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

[20] J. K. Reid and J. A. Scott. HSL_OF01, a virtual memory system in Fortran. Technical report, Rutherford Appleton Laboratory, 2006.

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

[22] 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.

[23] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proc. of the First Conference on File and Storage Technologies*, 2002.

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

[25] R. Takhur, W. Gropp, and E. Lusk. On implementing mpi-io portably and with high performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 23–32. ACM Press, 1999.

[26] S. Toledo. TAUCS: A library of sparse linear solvers, version 2.2, 2003. Available online at http://www.tau.ac.il/˜stoledo/taucs/.

A On the shape of the graphs: Formalization

In this annex we prove in a formal way the result presented and discussed in Section 6.1.

Let us first consider the more general context of any out-of-core application where data is produced and consumed with a stack mechanism (last data produced is consumed first). Up to the end of this subsection, we will use the term *memory* for all data relative to the application (that data may be either in core memory or on disk); and we define a *memory access* as an access to either the core memory or the disk (which implies in this second case a given amount of *I/O*). We have the following result:

Theorem A.1. *Given an out-of-core application which accesses the memory as a stack which is empty both initially and eventually; given a sequence of memory accesses, the optimum volume of* I/O $V^{I/O}$ *as a function of the available memory* M_0 *(*$V^{I/O} = f(M_0)$*) is a piecewise affine function; the steepness of each piece is an integer multiple of* -1 *whose absolute value decreases when the value of* M_0 *increases.*

Proof. The hypothesis that the stack is empty both initially and eventually implies that all data are reused; so any data written to disk will have to be read back. Subsequently, the volume of writes is equal to the volume of reads and there is an intrinsic way of defining the volume of *I/O* as proportional to the volume of write. Let us *set* 1 MB of *I/O* as 1 MB of write.

Let us focus on the evolution of the amount of memory (M) relative to the amount of memory accesses ($M_{accessed}$). At the beginning, the amount of memory is zero (stack initially empty). When (say) 1 MB of data is pushed, both the amount of data accessed (x axis) and the amount of memory (y axis) increase by 1 MB. When (say) 1 MB of data is popped, the amount of data accessed still increases by 1 MB while the amount of memory decreases by 1 MB. Geometrically, the function $M = f(M_{accessed})$ is a piecewise affine function for which each piece has a steepness equal to 1 (pushes) or -1 (pops); its graph is composed of a succession of peaks and hollows. At the end, the amount of memory is zero (stack eventually empty).

A memory access may be defined as a pair (T, Q) where T is the type of access ($push$ or pop) and Q is the amount of data involved (in MB). From a memory point of view, if n is the number of accesses, such an application is then exactly defined by a sequence $S = ((T_i, Q_i))_{i\in\{1;...;n\}}$ that verifies the two following properties:

$$(\forall j \in \{1; ...; n\})\Big(\sum_{i\in\{1;...;j\}|T_i=push} Q_i \geq \sum_{i\in\{1;...;j\}|T_i=pop} Q_i\Big) \tag{4}$$

$$\sum_{i\in\{1;...;n\}|T_i=push} Q_i = \sum_{i\in\{1;...;n\}|T_i=pop} Q_i \tag{5}$$

Moreover, even if it means packing consecutive accesses of same type we may suppose without loss of generality that pushes and pops are alternated. Then, we can define a *local peak* P_i (*resp.* a *local hollow*) as two successive memory accesses $(push, Q_{push}), (pop, Q_{pop})$ (*resp.* $(pop, Q_{pop}), (push, Q_{push})$), in this order. We define $\mathcal{P}$ as the (ordered) set of peaks. Note that $\mathcal{P}$ also defines the sequence S.

For a given amount of available physical memory M_0, the (minimum) volume of *I/O* can be directly computed with a greedy algorithm on the sequence S as shown in Algorithm 1. Each time the memory required exceeds M_0 (after $T_i = push$), we write the bottom of the stack to disk. When a *pop* operation is performed, we only read the bottom of the stack only if needed. As earlier, note that since the volume written and read are equal, we only take write operations into account, so that $V^{I/O}$

represents the volume of data *written* to disk.

Input: $S = ((T_i, Q_i))_{i \in \{1;...;n\}}$: Sequence of memory accesses
Input: M_0: Memory available
Output: $V^{I/O}$: *I/O* volume
`% Initialization:`
$current\_mem \leftarrow 0$;
$i \leftarrow 1$;
while $i \leq n$ **do**
  if $T_i = push$ **then**
    `% Memory required is` $current\_mem + Q_i$ `but only` M_0 `is available`
    `% Write the overhead to disk`
    $V^{I/O} \leftarrow V^{I/O} + \max(current\_mem + Q_i - M_0, 0)$;
    $current\_mem \leftarrow \min(current\_mem + Q_i, M_0)$;
  else
    `%` $T_i = pop$
    `% We do not count read operations`
    $current\_mem \leftarrow \min(current\_mem - Q_i, 0)$;
  $i \leftarrow i + 1$

Algorithm 1: *I/O* volume computation of a sequence of memory accesses S with an available memory M_0.

However the continuity of $V^{I/O}$ with respect to M_0 does not appear obviously with this approach. That is why we first carry out a transformation independent from M_0 which will bring to light the true potential sources of *I/O*.

Let us illustrate Algorithm 1 on simple examples. We consider the sequence (*push*,4); (*pop*,4) (see first picture of Figure 16(a)). If $M_0 > 4$ (for example $M_0 = 4.5$), no *I/O* will be necessary. If $M_0 = 2$, applying Algorithm 1 will lead to a volume of *I/O* equal to 2. If now $M_0 = 0.5$, we obtain a volume of *I/O* equal to 3.5. When the physical memory available M_0 decreases, we observe that the maximum volume of *I/O* that we can obtain is 4. We say that we have a *potential of* I/O equal to 4. Indeed on such a sequence the volume of *I/O* will be equal to $\max(4 - M_0, 0)$. If we now consider sequence (b) ((*push*,4); (*pop*,4); (*push*,4); (*pop*,4)) there are two peaks which constitute two potential sources of *I/O*. In that case the volume of *I/O* is equal to $2 \times \max(4 - M_0, 0)$. The potentials of *I/O* corresponding to the two peaks of memory are both equal to 4.

As shown in the two trivial examples above, to each peak i in $\mathcal{P}$ we have associated a *potential of* I/O Pot_i, leading to an overall volume of *I/O* equal to $V^{I/O}(M_0) = \sum_{i \in \mathcal{P}} \max(Pot_i - M_0, 0)$.

Let us now take a slightly more complex example: sequence (*push*,4); (*pop*,2); (*push*,1); (*pop*,3) from Figure 16(c). In that case, we again start doing *I/O* when the physical memory available M_0 becomes smaller than 4. If $M_0 = 2$, then the first peak $M = 4$ will force us to write 2 MB from the bottom of the stack. Then the memory M decreases until $M = 2$. When M increases again until reaching the second peak $M = 3$, the bottom of the stack is still on disk and no supplementary *I/O* is necessary. Finally M decreases to 0 and the bottom of the stack (2 MB) that was written will be read from disk and consumed by the application. For this value of M_0 (2), the volume of (written) *I/O* is only equal to 2 MB. In fact if $M_0 > 1$ the second peak has no impact on the volume of *I/O*. In this example, even if there are two peaks of sizes 4 MB and 3 MB, we can indeed notice that 2 MB are shared by these two peaks. This common amount of data can only be processed *out-of-core* once.

By trying other values of M_0, we would see that the volume of *I/O* $V^{I/O}(M_0)$ is in fact equal to $\max(4-M_0,0)+\max(1-M_0,0)$. Therefore we associate a potential of *I/O* of 4 to the first peak but a potential of *I/O* of only 1 to the second. Indeed the potential of *I/O* for the second peak is obtained by subtracting 2 (data common to the two peaks, for which *I/O* is only performed once) to 3 (value of the second peak).

We now describe more precisely the process consisting in replacing peaks by potentials of *I/O*. Each potential of *I/O* is equal to the maximum volume of *I/O* due to each peak. The key point is that each data accessed is attributed to one peak and only one as follows. The first potential source of *I/O*, corresponding to the highest peak, is selected first and receives a potential of *I/O* equal to the memory of this peak. Data corresponding to this peak will be written to disk at most once. But part of these data is shared with other peaks. That is why we carry out a transformation consisting in *subtracting* data shared with other peaks from these other peaks.

Formally, this subtraction process is described by the operation $S' \leftarrow \textit{Subtract}(S, P_i)$ from Algorithm 2. For any value of M_0, it is such that $V^{I/O}(S, M_0) = \max(Pot_i - M_0, 0) + V^{I/O}(S', M_0)$, where $Pot_i = \sum_{j=1}^{i} Q_{push_j} - \sum_{j=1}^{i-1} Q_{pop_j}$ is the *potential* associated to P_i. Recall that for this relation to hold, we have to choose P_i as the one that corresponds to the largest volume of memory (or potential), $i.e.$ the one first responsible of *I/O* when M_0 decreases. For instance, in example (d) from Figure 16, applying this subtraction to the peak associated to a memory of 3 MB (instead of the one associated to a memory of $4\ MB$) would give an incorrect volume of *I/O* equal to $\max(3-M_0,0)+\max(2-M_0,0)$ (instead of $\max(4-M_0,0)+\max(1-M_0,0)$), whereas *I/O* clearly starts occurring as soon as M_0 is smaller than 4 MB. Algorithm 3 now applies recursively the transformation to the new sequence (after the suitable subtractions). At the end, we have got a series of potentials $(Pot_i)_{i\in\mathcal{P}}$ - that we keep in the same order as the peaks they are associated to for a better readability. We call the result of this recursive transformation the *potential transform*. By construction, and as we have seen on the examples, the volumes of *I/O* for each potential are cumulated, and the total volume of *I/O* is thus given by:

$$V^{I/O}(M_0) = \sum_{i\in\mathcal{P}} \max(Pot_i - M_0, 0). \tag{6}$$

To achieve the proof, let us notice that the transformation is independent from M_0 and so a potential Pot_i too. Thus the function $M_0 \mapsto \max(Pot_i - M_0, 0)$ is a piecewise affine with steepness -1 for $M_0 < Pot_i$ and 0 for larger values of M_0. Finally, $M_0 \mapsto V^{I/O}(M_0)$, as the sum of such functions is a piecewise affine function whose pieces have steepnesses of decreasing (in absolute value) negative integer values.

□

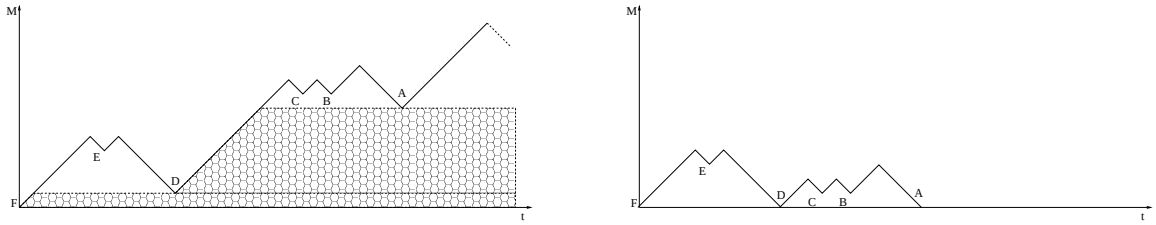
For each example from Figure 16, we unroll the algorithm and successively replace the largest peak by a potential of *I/O* equal to the memory associated with that peak. We represent each potential of *I/O* obtained by a vertical bar. At the end of the transformation, all the peaks have been replaced by their respective potentials as shown in the third picture of the figure.

Finally (for each example), the subsequent volume of *I/O* is illustrated by the fourth series of pictures of Figure 16. This result may be interpreted from a geometric point of view. The steepness of the graph of the function $V^{I/O}(M_0)$ for a given value M_0 is the number of potentials crossed by the horizontal line M_0. For instance, with sequences (c) and (d) (that have the same *potential transform*), if the amount M_0 of available memory is more than 4 (say equal to 4.5), the corresponding horizontal line (says 4.5) does not cross any potential: no *I/O* is required. If M_0 is between 1 and 4 (says 2), the horizontal line (say 2) crosses one potential: the steepness is one. In other words, locally, the volume

of *I/O* grows as fast as the memory decreases. Finally, when M_0 is less than 1 (say 0.5), the horizontal line (say 0.5) crosses two potentials: the steepness is two. The volume of *I/O* grows *twice* as fast as the physical memory available decreases.

Input: $S = (P_1, \ldots, P_n)$: The sequence of memory accesses as a list of *local peaks*
Input: $P_h = (push, Q_{push_h}), (pop, Q_{pop_h})$: A *local peak* to subtract from the sequence
Output: S': Sequence of memory accesses after subtraction of peak P_h
% Recompute potential of P_h
$Pot_h \leftarrow \sum_{i=1}^{h} Q_{push_i} - \sum_{i=1}^{h-1} Q_{pop_i}$ % Pop P_h from the sequence:
$S' \leftarrow S \setminus P_h$;
$pos\_current\_peak \leftarrow h$;
% (1) Decrease peaks prior to P_h and sharing data with it.
$current\_hollow \leftarrow Pot_h - Q_{push_h}$;
$lower\_hollow \leftarrow current\_hollow$;
% While there are data shared with other peaks
while $lower\_hollow > 0$ **do**
  % Look for the previous peak
  $pos\_current\_peak \leftarrow pos\_current\_peak - 1$;
  % Evaluate its local hollow
  $current\_hollow = current\_hollow + Q_{pop_{pos\_current\_peak}} - Q_{push_{pos\_current\_peak}}$;
  % If there is shared data with h
  if $current\_hollow < lower\_hollow$ **then**
    % Subtract shared data from current peak
    $Q_{push_{pos\_current\_peak}} \leftarrow Q_{push_{pos\_current\_peak}} + lower\_hollow - current\_hollow$;
    % Update $lower\_hollow$ value
    $lower\_hollow \leftarrow current\_hollow$;
%
% (2) Decrease peaks that are after P_h and that share data with P_h.
% Similar to 1 except that we decrease Q_{pop} values.

Algorithm 2: Subtraction of a peak from a sequence of memory accesses: $S' \leftarrow$ *Subtract(*S,P_h*)*. Only peaks before P_h are treated (point (1)); peaks after P_h receive a similar treatment (point (2)).



(a) Initial sequence

(b) Sequence after subtract

Figure 15: Illustration of Algorithm 2 on a toy sequence of memory accesses and the subtraction of its highest peak. Initially the highest peak is subtracted and $current\_hollow$ and $lower\_hollow$ are equal to A. Next $current\_hollow = B$ but $lower\_hollow$ does not change because $B > A$. When $current\_hollow = D$, an amount of data equal to $A - D$ is subtracted from the corresponding peak and the value of $lower\_hollow$ is set to D. Then $current\_hollow = E$ but $lower\_hollow$ does not change as $E > D$. Finally $current\_hollow$ is equal to F and this induce the subtraction of an amount of data equal to $D - F$ from the corresponding peak. Note that we only illustrated the process for peaks that are before the subtracted peak (point (1) of the algorithm).

Input: $S = (P_1, \ldots, P_n)$: sequence of memory accesses as a list of *local peaks*
Output: T: *Potential transform* as a list of potentials
% Initialization
$T = ;$
% Main loop
while $S \neq$ **do**
  % Find the highest *local peak* P_h, of potential Pot_h:
  $Pot_h = \max_{h=1..n} \sum_{i=1}^{h} Q_{push_i} - \sum_{i=1}^{h-1} Q_{pop_i}$;
  % Add its potential to the list of potentials:
  $T \leftarrow Pot_h :: T$;
  % Subtract P_h from S:
  $S' \leftarrow$ Subtract(S,P_h);

Algorithm 3: Computing the *potential transform* of a sequence of memory accesses: *Transform(S)*

Corollary A.2. *In the sequential case, the volume of* I/O *on the active memory for the factorization step of our application as a function of M_0 ($V^{I/O} = f(M_0)$) is a piecewise affine function; the steepness of each piece is an integer multiple of -1 whose absolute value decreases when the value of M_0 increases.*

Proof. In practice, we do not have a pure stack mechanism in our multifrontal factorization: a frontal matrix is first allocated; the contribution blocks of the children are then consumed; the frontal matrix is factored and its factors are stored to disk; and finally the contribution block of the active frontal matrix is moved to top of the stack.

However, the key point is that we may assess that the active memory is accessed as a stack without

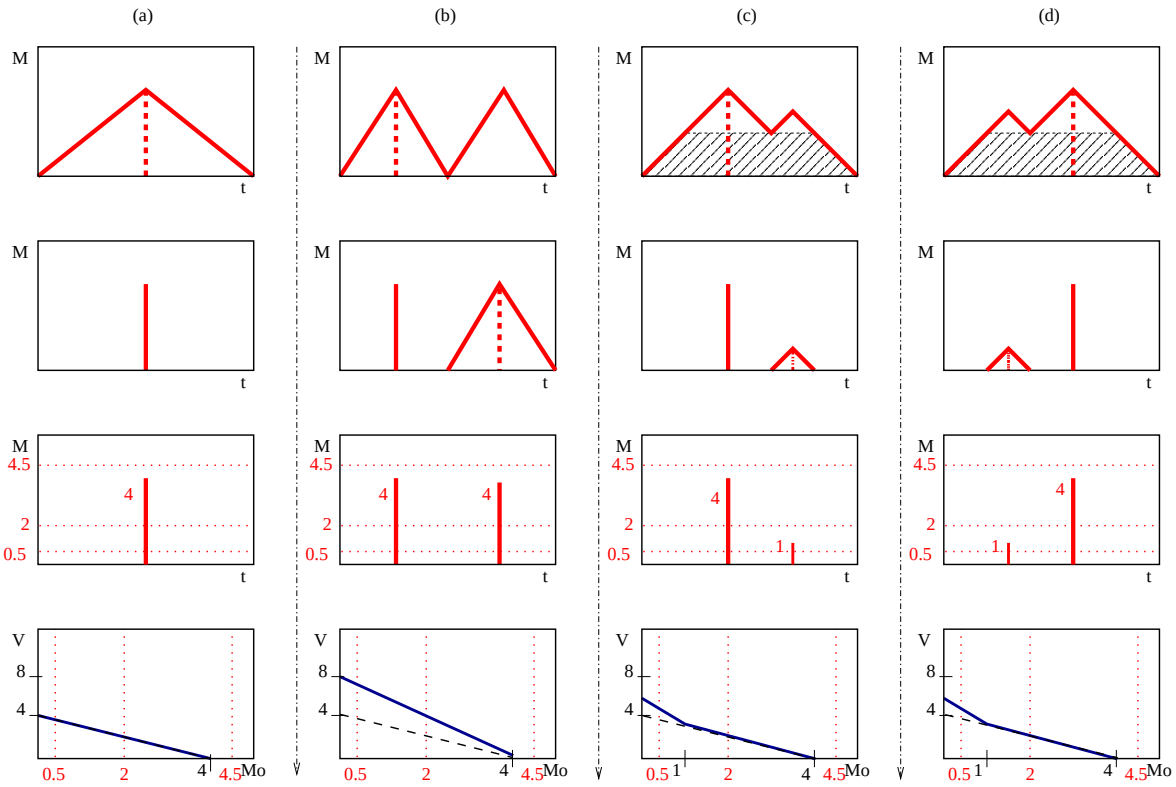


Figure 16: Computing the *potential transform* and deducing the *I/O* volume: four instructive examples. On each column, corresponding to a given sequence of memory accesses, the transformation is unrolled on the first three pictures (the potentials are the vertical bars) and the deduced *I/O* volume V (as a function of the memory available M_0) is given by the fourth one (the lower bound function "peak - M_0" is there represented with dashed line).

modifying the volume of *I/O*. Indeed, when a frontal matrix is just factorized, we may consider that we pop this complete frontal matrix as well as all the contribution blocks of its children and that we finally push its own contribution block.Because we have the assumption that a frontal matrix holds *in-core*, this involves the same amount of *I/O* as the real mechanism implemented.

Subsequently, considering that (i) we may assess that the active memory is accessed as a stack, that (ii) the active memory is empty both initially and eventually (any contribution block or frontal matrix will be reused during the factorization step and popped), that (iii) the sequence of accesses does not depend on M_0 (the tree traversal is fixed), and that (iv) the volume of *I/O* performed is minimum (use of a "as late as possible" scheme) for this sequence of accesses, Theorem A.1 can be directly applied. Note that the *memory* defined above corresponds in this context to the active memory of our application (see again Section 2).

□

Note that the *potential transform* also easily gives the volume of accesses to the memory: it is the sum of the potentials and it is also equal to the volume of *I/O* when M_0 tends to 0. However this model can only be applied to our application if M_0 remains larger than the largest frontal matrix. When a frontal matrix cannot fit *in-core* (because its size is larger than M_0) we have no more guarantee that we may respect a *read-once / write-once* scheme. For such values of M_0, the volume of *I/O* computed with this model becomes a lower bound of the actual volume of *I/O*. Subsequently, the sum of the potentials is a lower bound on the amount of data accessed.