Analysis and Improvments of the Memory Usage of a Multifrontal Solver

Abdou Guermouche, Jean-Yves l'Excellent, Gil Utard

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Analysis and Improvments of the Memory Usage of a Multifrontal Solver

Abdou Guermouche, Jean-Yves L'Excellent, Gil Utard

No 4729

February 2003

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4729--FR+ENG

Analysis and Improvments of the Memory Usage of a Multifrontal Solver

Abdou Guermouche, Jean-Yves L'Excellent, Gil Utard

Thème 1 — Réseaux et systèmes
Projet ReMaP

Rapport de recherche n ° 4729 — February 2003 — 51 pages

Abstract: We are concerned with the memory usage of sparse direct solvers. We particularly focus on the influence of state-of-the-art sparse matrix reordering techniques on the dynamic memory usage of a multifrontal solver, MUMPS, and present algorithms to modify the assembly tree traversal that aim at reducing the memory usage. A large number of experiments show the interest of the approach for sequential executions.

Key-words: Sparse matrices, multifrontal method, assembly tree, reordering techniques, memory, tree traversal.

(Résumé : tsvp)

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

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN (France)
Téléphone : 04 76 61 52 00 - International : +33 4 76 61 52 00
Télécopie : 04 76 61 52 52 - International : +33 4 76 61 52 52

Étude et optimisation de l'utilisation mémoire d'un solveur multifrontal

Résumé : Nous nous intéressons à l'occupation mémoire des méthodes directes, et nous penchons plus particulièrement sur l'influence des principaux algorithmes de renumérotation sur le comportement mémoire de la méthode multifrontale, et du solveur MUMPS. De plus, nous montrons l'importance du parcours de l'arbre d'assemblage (qui représente le graphe de dépendance des tâches) pour l'optimisation mémoire. Ainsi, nous proposons des algorithmes qui modifient le parcours de l'arbre pour une occupation mémoire optimale. L'intérêt de nos approches est illustrée par une série de résultats expérimentaux.

Mots-clé : Matrices creuses, méthode multifrontale, arbre d'assemblage, algorithmes de renumérotation, mémoire, parcours d'arbre.

1 Motivations

Sparse direct methods and in particular multifrontal methods are robust and efficient techniques to solve large sparse systems of linear equations. However, they are known for their relatively large memory requirements compared to iterative methods so that an in-core execution is not always possible.

Two types of memory can be distinguished in the process of solving sparse linear systems with a direct solver: a *static memory* needed to store the final factors of the sparse system; and an additional dynamic memory needed to store temporary values used by the computation that we call *active memory*. In the case of multifrontal methods, this active memory is handled using a stack mechanism. The size of the active memory can be large, and is sometimes larger than the static memory: for some problem the computation cannot complete because of a lack of main memory.

Reordering (ie, renumbering the unknowns of a sparse linear system) is a well known technique to reduce the fill in the final factors, and this has a big impact on the static memory size. In this paper we show that reordering techniques also have a big impact on the active memory size. In the multifrontal method, the active memory size depends on the shape of a dependency graph so called assembly tree, which depends on the reordered matrix. We present an extensive study of the assembly tree shapes resulting from the combinations of sparse matrices and reorderings.

The active memory size also depends on the way the assembly tree is traversed during the factorization process. In this paper we revisit the algorithm of Liu [13] to determine an optimal tree traversal for the multifrontal solver `MUMPS`.

In Section 2, we give some generalities concerning direct methods, and show how sparse matrix reordering techniques influence the static memory (factor size). In Section 3, we recall some general mechanisms of the multifrontal method and its active memory usage which depend of the assembly tree. In Section 4 we study the impact of reordering techniques on the shape of the assembly tree. In Section 5 we study the evolution of the active memory in `MUMPS`. Section 6 presents some variants and improvements of the algorithm by Liu [13] to modify the traversal of the multifrontal assembly tree in our context. We show how a reduction of the active memory size can be obtained.

2 Static memory consumption of sparse direct solvers

2.1 Sparse Direct Solvers

In this paper we discuss the direct solution of linear systems

$$\mathbf{A}x = b$$

where A is a large sparse matrix of order N ($A = (a_{ij})_{1 \leq i,j \leq N}$) where only NZ coefficients a_{ij} are non zero ($NZ \ll N^2$). Direct methods are based on Gaussian elimination. More precisely, the matrix A is decomposed into triangular factors $\mathbf{LU}$ or $\mathbf{LDL}^T$ depending on the numerical properties of $\mathbf{A}$. The solution is then obtained by triangular solves with these new factors. We focus here on the $\mathbf{LU}$ decomposition: it consists in decomposing the matrix $\mathbf{A} = (a_{ij})_{1 \leq i,j \leq N}$ into the product of two matrices $\mathbf{L} = (l_{ij})_{1 \leq i,j \leq N}$ and $\mathbf{U} = (u_{ij})_{1 \leq i,j \leq N}$,

$$\mathbf{A} \quad = \quad \mathbf{LU}$$

where $\mathbf{L}$ is *lower triangular* (i.e. $l_{ij} = 0$ for $1 \leq j < i \leq N$) and $\mathbf{U}$ is *upper triangular* (i.e. $u_{ij} = 0$ for $1 \leq i < j \leq N$). The $N^2 + N$ entries of $\mathbf{L}$ and $\mathbf{U}$ are solution of N^2 equations and a solution is:

$$1 \leq j \leq N \begin{cases} l_{jj} = 1 \ \text{ (in practice, } l_{jj} \text{ is not stored)} \\ u_{ij} = a_{ij} - \sum_{k=1}^{k=i-1} l_{ik} u_{kj} \text{ for } 1 \leq i \leq j \\ l_{ij} = \frac{1}{u_{jj}} (a_{ij} - \sum_{k=1}^{k=j-1} l_{ik} u_{kj}) \text{ for } j < i \leq N \end{cases}$$

To reduce the computational cost, direct solvers compute only on nonzero values. Similarly, only nonzero values are stored to reduce the storage requirements, but at the cost of complex data structures and indirect accesses to the memory.
Usually, the factors $\mathbf{L}$ and $\mathbf{U}$ are significantly denser than the original matrix $\mathbf{A}$ because of *fill-in* that appears during the factorization process: u_{ij} or l_{ij} will eventually be non zero if there is a k ($k < i$ and $k < j$) such that l_{ik} and u_{kj} are non zero, although a_{ij} is zero.
When we consider memory usage of sparse direct solvers, fill-in is the first measurement: it determines the size of the final factors. Moreover, an increase of the fill-in leads to an increase of the computational cost. Therefore, a lot of work in the field of sparse direct solvers focuses on techniques to reduce fill-in by permuting the unknowns of the original matrix. These techniques are known as *sparse matrix reordering techniques*, or *reordering techniques*.

2.2 Reordering techniques

As we said, reordering the variables of a sparse linear system, i.e. permuting columns and rows (with some respect to numerical stability), aims at reducing the amount of fill-in. For instance, if $a_{ij} = 0$ for a fixed $i < j$ and there is a $k < i$ such that both a_{ik} and a_{kj} are non zero, then the permutation of variables k and i avoids the computation and the introduction of a non zero u_{ij} value. Unfortunatly, finding a good permutation which globally reduces the fill-in is known to be an NP-hard problem and several heuristics have been developed.
Here we only consider symmetric reordering techniques. Those can also be applied to an unsymmetric matrix $\mathbf{A}$ by considering the structure of $\mathbf{A} + \mathbf{A}^T$ (after some column permutation for very unsymmetric matrices [7]).
Two popular schemes for symmetric reordering are bottom-up heuristics such as the minimum degree (AMD [1], MMD [12]) or minimum fill (MMF [15, 19]) and global or top-down heuristics based on partitioning the graph of the matrix, such as nested dissection [10]. The good properties of both schemes may be combined by hybridizing top-down nested dissection with bottom-up minimum degree.
Note that although these heuristics mainly focus on the reduction of fill-in, (and thus size of the factors and number of operations), they also have a significant impact on the parallelism (see, e.g. [3]). Here, we are essentially interested in the influence of such techniques on the memory usage and consider the following bottom-up and hybrid heuristics:

- AMD: the Approximate Minimum Degree [1];
- AMF: an implementation of the Approximate Minimum Fill, available in `MUMPS`;
- PORD: a tight coupling of bottom-up and top-down sparse reordering methods [20];
- METIS: the METIS package [11] provides an hybrid approach based on multi-level nested dissection and multiple minimum degree;
- SCOTCH: we use a modified version of `SCOTCH` [16] provided by the author that couples nested dissection and (halo) Approximate Minimum Fill (AMF), in a way very similar to [17].

In the rest of this paper, we simply use the terms AMD, AMF, METIS, SCOTCH and PORD to refer to these heuristics.

Finally, note that we had initially considered a pure nested dissection algorithm [10], but this one is competitive only in a few cases, for extremely regular problems. Since we are more interested in irregular problems, we decided to discard it.

2.3 Impact of reordering on factor size

In this section we present the impact of the reordering techniques described above on the final factor size. We measured this impact on a large range of symmetric and unsymmetric assembled test matrices from various application areas (Table 1). They are extracted from either the Rutherford-Boeing collection [6], the collection from University of Florida[1] or the PARASOL collection[2].
The software package `MUMPS` [4, 3], that implements both $\mathbf{LU}$ and $\mathbf{LDL}^T$ factorizations with partial pivoting is used for the experimentations related in this paper. This package will be presented more precisely in the next part.
Concerning the factors, Table 2 shows that PORD is the reordering technique with the smallest corresponding factors. For small matrices, hybrid reorderings like SCOTCH or METIS give factors often bigger than the ones generated by AMF or AMD but this tends to be exactly the opposite for large test problems, where the purely local heuristics become much less competitive than hybrid ones.

3 Dynamic memory usage of the multifrontal method

In the previous section, we focused on the static memory usage of sparse direct solvers which is characterized by the factor size. This size is reduced by using appropriate reordering techniques. Now, we focus on the active memory of one of the most popular techniques for sparse direct factorization: the multifrontal method. We study the evolution of the working storage memory used for the computation of the factors. In multifrontal solvers, this temporary space is known to be as important as the the space used for factors.
In this section we describe the dynamic memory usage of multifrontal solvers, in particular we outline the impact of assembly tree on this memory consumption. Finally, we present the instrumentation we made on `MUMPS`, a popular multifrontal solver, to obtain statistics on the assembly tree and memory usage for various reorderings and sparse matrices.

[1] available from http://www.cise.ufl.edu/ davis/sparse/
[2] available from http://parallab.uib.no/parasol

Matrix	Order	NZ	Type	Description
3DTUBE	45330	1629474	S	3-D pressure tube
M_T1	97578	5328	S	Tubular joint
OILPAN	73752	1835470	S	Oilpan
SHIP_003	121728	4103881	S	Ship structure from production run
THREAD	29736	2249892	S	Threaded connector/contact problem
BCSSTK34	588	11003	S	NASTRAN buckling problem stiffness matrix
BCSSTK38	8032	181746	S	Stiffness matrix, airplane engine component
CFD2	123440	1605669	S	CFD, symmetric pressure matrix, from Ed Rothberg, Silicon Graphics, Inc.
GUPTA1	31802	1098006	S	Linear programming matrix (A*A'), Anshul Gupta
GUPTA2	62064	2155175	S	Linear programming matrix (A*A'), Anshul Gupta
GUPTA3	16783	4670105	S	Linear programming matrix (A*A'), Anshul Gupta
NASA1824	1824	20516	S	Structure from NASA langley, 1824 degrees of freedom
NASA2910	2910	88603	S	Structure from NASA langley, 2910 degrees of freedom
NASA4704	4704	54730	S	Structure from NASA langley, 4704 degrees of freedom
STRUCT4	4350	121074	S	Finite element matrix, from Ed Rothberg, Silicon Graphics, Inc.
VIBROBOX	12328	177578	S	Vibroacoustic problem (flexible box, structure only) Andre Cote
AF23560	23560	484256	U	1 NACA 0012 airfoil M=0.8, eigenvalue calculation
BIG	13209	91465	U	1Structuresymmetric Matrix big K. Gaertner ETH Zurich Oct 1996.
CIRCUIT_4	80209	307604	U	Circuit DAE with BDF method & Newton. W. Bomhof, Univ. Utrecht
EPB3	84617	463625	U	Plate-fin heat exchanger (large case), (DAE) David.Averous@ensigct.fr
GARON02	13535	390607	U	2D FEM, Navier-Stokes, CFD. Square inlet and outlet on opp. sides
GRAHAM1	9035	335504	U	Galerkin FE disc. of Nav.Stokes 2phase fluid flow. D Graham, U Illinois
GRID48	47045	3705625	U	Unsymmetric assembled finite element problem. Nine node, five variable ELT48
LI	22695	1350309	U	3D Finite element matrix, magnetohydrodynamic problem with 5 variables
ONETONE1	36057	341088	U	AT&T,harmonic balance method, one-tone. Approx Jacobian for SSOR precon
PRE2	659033	5959282	U	AT&T,harmonic balance method, large example
RMA10	46835	2374001	U	3D CFD model, Charleston harbor. Steve Bova, US Army Eng., WES
SAYLR1	238	1128	U	1UNSYMMETRIC MATRIX OF PAUL SAYLOR - 14 BY 17 2D GRID MAY, 1983
THERMAL	3456	66528	U	Noel.Brunetiere@lms.univ-poitiers.fr FEM, thermal problem
TWOTONE	120750	1224224	U	AT&T,harmonic balance method, two-tone. More off-diag nz than one-tone
VENKAT50	62424	1717792	U	Unstructured 2D EULER solver, V. Venkatakrishnan NASA, time step = 50
WANG1	2903	19093	U	Discretized electron continuity, 3d diode, nonuniform 14-14-16 mesh
WANG3	26064	177168	U	Discretized electron continuity, 3d diode, uniform 30-30-30 mesh
XENON2	157464	3866688	U	Complex zeolite,sodalite crystals. D Ronis ronis@onsager.chem.mcgill.ca

Table 1: Description of the test problems. In column "Type", S stands for symmetric and U stands for unsymmetric.

	METIS	SCOTCH	PORD	AMF	AMD
3DTUBE	*22.27*	22.35	22.49	32.61	**35.14**
M_T1	**38.95**	38.80	*32.56*	33.59	38.74
OILPAN	10.86	11.11	*9.93*	10.93	**11.66**
SHIP_003	73.34	79.80	73.57	*68.52*	**91.42**
THREAD	31.44	**33.56**	24.91	*24.50*	32.03
BCSSTK34	**0.07**	0.06	0.05	*0.05*	0.06
BCSSTK38	0.97	**1.23**	0.86	*0.85*	0.93
CFD2	45.05	45.83	*41.55*	66.14	**86.29**
GUPTA1	2.72	**3.84**	2.77	*2.49*	2.68
GUPTA2	8.55	12.97	**9.77**	*7.96*	8.08
GUPTA3	10.71	**47.91**	9.41	19.76	*7.02*
NASA1824	0.11	**0.12**	*0.08*	0.09	0.10
NASA2910	0.27	**0.30**	0.23	*0.23*	0.25
NASA4704	0.38	**0.41**	*0.30*	0.32	0.35
STRUCT4	*1.59*	1.85	1.70	2.35	**3.42**
VIBROBOX	2.74	**3.01**	*2.12*	2.35	2.86
AF23560	9.00	10.98	*7.67*	**11.23**	8.34
BIG	0.71	0.69	*0.66*	0.70	**0.75**
CIRCUIT_4	0.70	0.64	*0.63*	0.70	**0.70**
EPB3	5.30	*5.08*	5.12	5.85	**6.90**
GARON02	2.15	2.11	*2.01*	2.15	**2.45**
GRAHAM1	**1.49**	1.31	*1.30*	1.40	1.31
GRID48	**15.80**	14.90	*14.24*	14.88	14.99
LI	*8.70*	**50.89**	12.84	39.28	36.27
ONETONE1	**5.26**	4.96	*4.61*	4.96	4.72
PRE2	107.26	*102.30*	127.82	117.61	**132.65**
RMA10	**9.78**	9.23	*8.36*	8.46	8.87
SAYLR1	**0.01**	0.01	0.00	*0.00*	0.00
THERMAL	0.31	*0.30*	0.31	**0.37**	0.32
TWOTONE	25.04	25.64	**28.38**	22.65	*22.12*
VENKAT50	11.39	11.56	*10.84*	11.32	**11.97**
WANG1	*0.29*	**0.43**	0.34	0.32	0.37
WANG3	*7.65*	9.74	7.99	8.90	**11.48**
XENON2	*94.93*	100.87	107.20	144.32	**159.74**

Table 2: Factor size (millions of entries). For symmetric matrices, only the entries in the triangular factor **L** are reported. For each matrix, bold corresponds to the largest factors, and italic to the smallest.

3.1 The Multifrontal method

Like other direct methods, the multifrontal method [8, 9] is based on an elimination tree [14], which is a transitive reduction of the graph of $\mathbf{L}$, where L is the Cholesky factor of the matrix $(\mathbf{A}+\mathbf{A}^T)$, and is the smallest data structure representing dependencies between operations. In practice, we use a structure called the *assembly tree*, obtained by merging nodes of the elimination tree whose corresponding columns belong to the same *supernode* [5]. We recall that a *supernode* is a contiguous range of columns having the same nonzero structure in the factor matrix. The assembly tree is, for most cases, given by the reordering algorithm.
The multifrontal algorithm is an approach to organize right-looking sparse matrix factorization in which the factorization of the matrix is done by performing a succession of partial factorizations of small dense matrices called *frontal matrices*. The factorization process is given by the assembly tree where a frontal matrix is associated to each node.
Figure 1 gives an example of a matrix and its associated assembly tree. From the initial matrix, an assembly tree with three nodes (each corresponding to one supernode) is derived. The two first independent leaf nodes contribute to the computation of the third. The order of the frontal matrix is given by the number of nonzeros

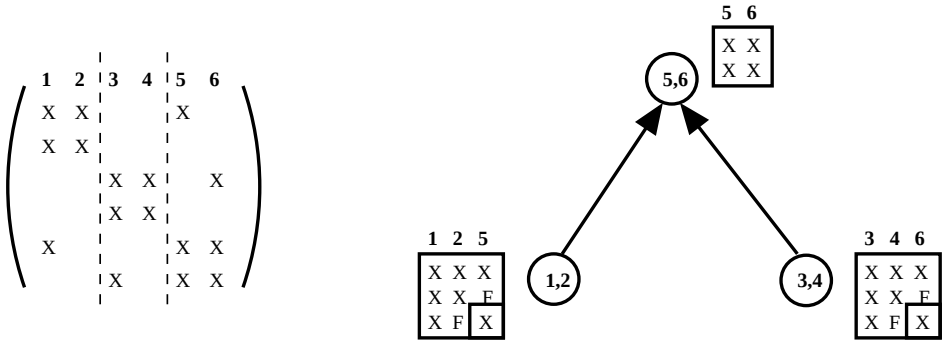


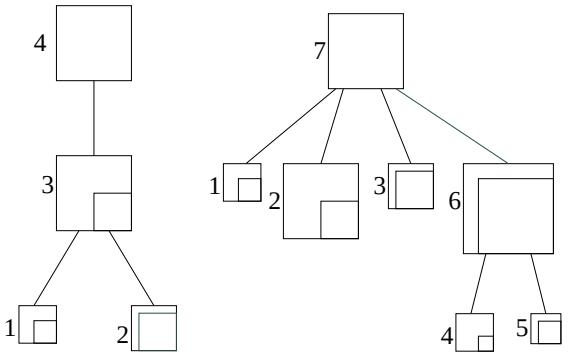
Figure 1: A matrix and the associated assembly tree. Supernodes are $(1,2)$, $(3,4)$ and $(5,6)$ (doted line), they regroup variables with same non zero structure in the final factor. To each (super)node is associated a frontal matrix where the factor part is distinguished from the contribution one ('F' represents fill-in).

below the diagonal in the first column of the supernode associated with the tree node. The frontal matrix is divided into two parts: the *factor* block, also called *fully*

summed block, which corresponds to the variables which are factorized when the elimination algorithm processes the frontal matrix; and the *contribution block* which corresponds to the variables which are updated when processing the frontal matrix. Once the partial factorization is processed, the contribution block is passed to the parent node to be assembled (i.e. summed with the values contained in the frontal matrix of the parent). The elimination algorithm is a postorder traversal (children node must be processed before their parent) [18] of the assembly tree.
The algorithm uses three areas of storage, one for the factors, one to stack the contribution blocks, and another one for the current frontal matrix [2]. During the tree traversal, the memory space required by the factors always grows while the stack memory (containing the contribution blocks) varies depending on the operations made: when the partial factorization of a frontal matrix is processed, a contribution block is stacked which increases the size of the stack; in opposition, when the frontal matrix is formed and assembled, the contribution blocks of the children nodes are popped from the stack and its size decreases.
Note that in our description all the contribution blocks for children nodes are assembled at once. Another approach could be to perform the assembly of a contribution block on the fly, each time it is computed, avoiding the use of a memory stack. This is generally not done in multifrontal solvers because this strategy implies the use of more complex memory management algorithms and the structure of the frontal matrix of the parent is unpredictable when there is pivoting. Also, except for wide trees, this is not necessarily a more efficient memory scheme because it also implies allocating a chain of frontal matrices (each containing all future contribution blocks of the subtree).
In the other parts of the paper, if not specified otherwise, we only distinguish between two areas of storage: the factors, and the stack, where the term stack includes both contributions blocks and the storage for the current frontal matrix.

3.2 Impact of the tree and its traversal on memory usage

To better explain the impact of the assembly tree on the memory usage, we consider two examples of assembly trees, as shown in Figure 2. The corresponding memory evolution for the factors, the stack and the current frontal matrix is given Figure 3. First storage for the current frontal matrix is reserved (see "Allocation of 3" in Figure 3(a)); then the frontal matrix is assembled using values from the original matrix and contribution blocks from the children nodes, and those can be freed ("Assembly step for 3" in 3(a)); the frontal matrix is factorized ("Factorization step for 3" in 3(a)). Factors are stored in the factor area on the left in our figure and the



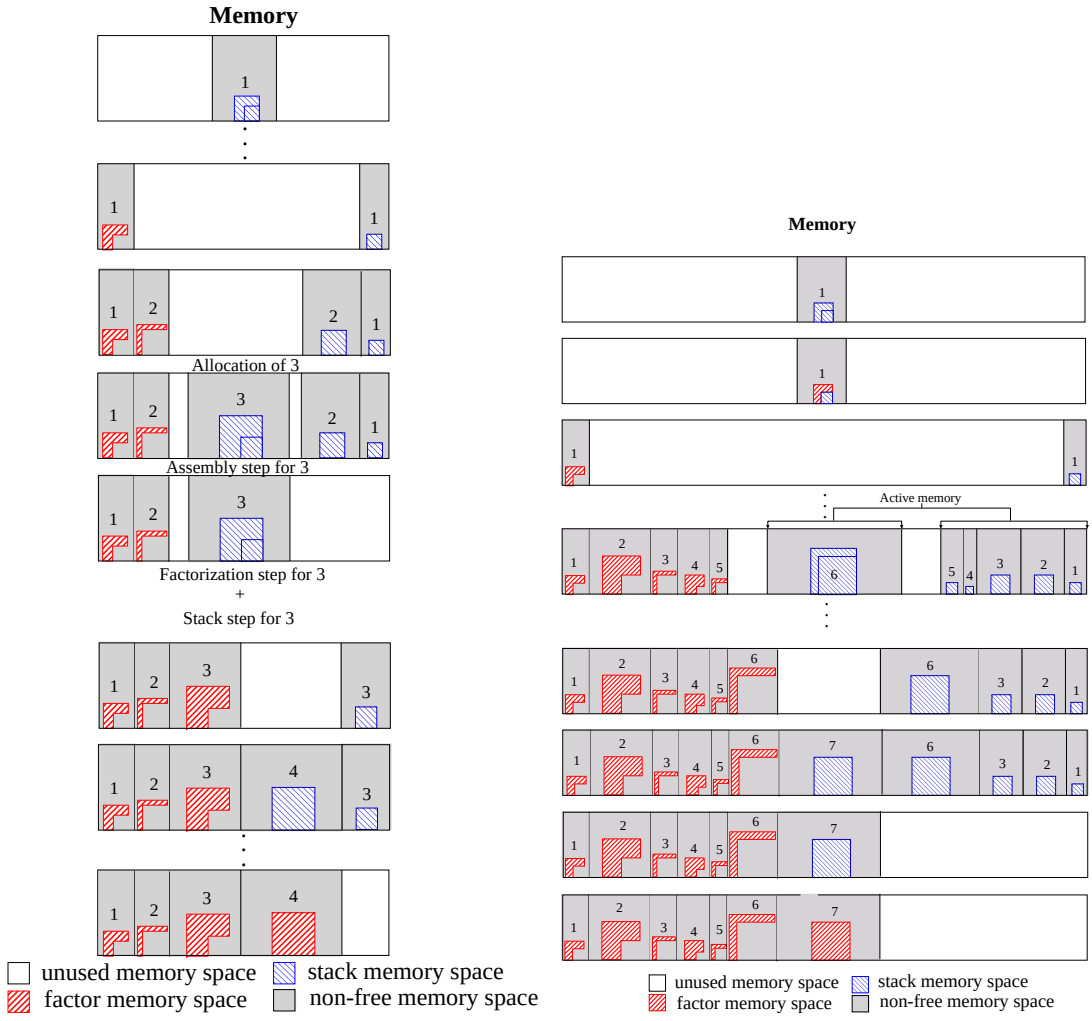
(a) tree A – deep (b) tree B – large

Figure 2: Examples of assembly trees.

contribution block is stacked ("Stack step for 3"). The process continues until the complete factorization of the root node(s).

In the previous examples, we clearly observe that the stack memory is greater for tree B than for tree A. Indeed, since node 7 in tree B has a large number of children, there are more contribution blocks to be stored. For tree A, because of its shape, there is a maximum of two contribution blocks to be stored simultaneously. This illustrates the impact of the characteristics of the assembly tree on the memory evolution.

Given an assembly tree, another very important factor impacting the memory usage is the order in which the nodes of the tree are visited. The only constraint in the traversal of the tree is that parent nodes are computed after their children and in general, for sparse multifrontal solvers the traversal is the depth-first search. In other terms, it's a traversal where we try to process the parent node as soon as it is possible to do so, as this allows to limit the amount of temporary contribution blocks. If we consider the trees of Figure 4, and assuming that the depth-first search is used where the nodes on the left are processed first, the best case in terms of memory usage is the tree on the left because we need to store contribution blocks for at most two nodes, while the tree on the right-hand-side of the picture corresponds to the worst case because contribution blocks must be stored simultaneously for all leaves.



(a) Memory usage for tree A

(b) Memory usage for tree B

Figure 3: Memory evolution for the trees given in Figure 2.

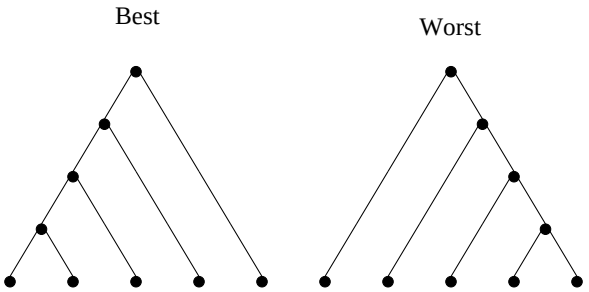


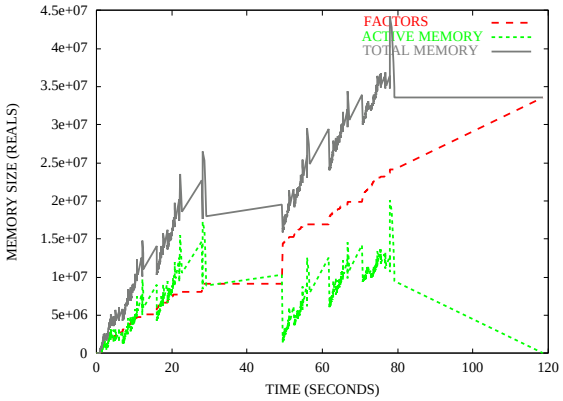
Figure 4: Importance of the tree traversal.

3.3 Instrumentation of a multifrontal solver, `MUMPS`

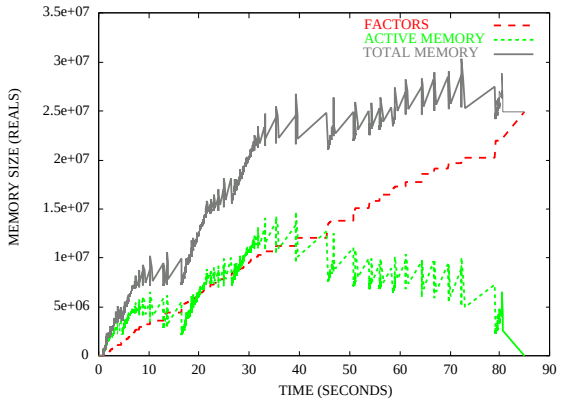
In order to make an experimental study of the dynamic memory evolution of the multifrontal approach, we instrumented the `MUMPS` software package. `MUMPS` (MUltifrontal Massively Parallel Solver) [4, 3] implements parallel multifrontal solvers with threshold partial pivoting for both $\mathbf{LU}$ and $\mathbf{LDL}^T$ factorizations, and the software allows to use the reordering packages described in Section 2.2. For our purpose, we first experiment with the sequential version, and the tree is processed using a depth first search traversal. To be able to obtain statistics on the assembly tree and on the memory usage, we have instrumented the code. Figures 5(a) and 5(b) show the memory evolution (stack, factors and total) for the matrix THREAD with SCOTCH and PORD respectively. We can see that the curves have different shapes, which illustrates the impact of the reordering on the memory behaviour. Notice also that for this matrix, the size of the stack is in the same order of magnitude as the factors. An exhaustive set of results concerning both the characteristics of the assembly tree and the memory usage are given later in this paper.

3.4 Memory accesses in the multifrontal scheme

During the factorization process, once a factor is computed, it is not accessed again until the end of the computation. The only part of the memory which is accessed is the active one, i.e. the stack plus the current front. Thus, the working space of the factorization process can be defined has the peak size of the active memory achieved during the computation. To deal with bigger problems, we can imagine two types of execution schemes, one using the paging system, another one using an out-of-core method:



(a) SCOTCH



(b) PORD

Figure 5: Memory evolution for THREAD with SCOTCH and PORD.

Paging. Assuming that there is a paging system, the working space size previously defined is the memory size threshold above which we can observe a significant degradation of performance.

Out-of-Core execution. In an out-of-core implementation, the factors are put on disk once computed. The memory size needed for the computation is then the working space previously defined.

Reducing the working space will allow us to treat bigger and bigger problems.
As previously described, the size of the working space is defined by the shape of the assembly tree and how it will be traversed. In the next sections we first study the impact of reordering techniques on the shape of the assembly tree and then their impact on the active memory size. Then we will present a new algorithm which gives the optimal tree traversal for a fixed reordering.

4 Impact of reordering techniques on the assembly tree

As outlined in the previous section, to understand the dynamic evolution of the memory, we must first study the structure of the assembly tree. The assembly tree is mainly determinated by the reordered matrix. Some other optimization such that node amalgamation to increase granularity are usually done during the reordering phase, so the assembly tree is generally build during reordering.
In this section, we study the impact of the reordering technique used on the shape of the corresponding assembly tree. We consider the test problems presented in Table 1 and the reordering techniques METIS, SCOTCH, PORD, AMF and AMD introduced in Section 2.2. The general shape of reordering tree (width and depth) was estimated by the number of nodes (Table 4), the number of leaves (Table 5) and the percentage of leaves (Table 6). Regularity of the shape was estimated by the standard deviation of the depth of the leaves (Table 7), the maximum depth of a leaf (Table 8), the average number of children (Table 9). Because the stack size is also determinated by the size of frontal matrix associated to each node, we determinated for each tree, the maximum size of a frontal matrix (10), the average front size (Table 11), and the average standard deviation of front orders between brothers (Table 12).
As previously noticed AMD, AMF, SCOTCH and PORD return directly the assembly tree. But for METIS, only the permutation is returned and `MUMPS` rebuilds an assembly tree based from that permutation.

General shape

We observe in Table 4 that for most test problems, SCOTCH generates the tree with the smallest number of nodes. Then AMD and METIS provide approximately the same number of nodes, and finally, AMF and PORD give trees with a much larger number of nodes compared to SCOTCH.

In addition, we observe from Tables 5 and 6 that usually, SCOTCH and METIS generate trees with a large number of leaves when compared to the trees generated by AMF, AMD or PORD. Effectively, the trees generated by METIS and SCOTCH are rather large (because of the global partitioning performed at the top), while the trees generated by AMD, AMF and PORD tend to be deeper.

Regularity

According to Tables 7 and 8, we can see that PORD and AMF generate unbalanced trees (where depth of leaves varies a lot depending on the branches) while SCOTCH and METIS generate much better balanced trees. Finally, according to Table 9, we can see that PORD, AMD and AMF have trees where the average number of children for a node is smaller than for the the METIS and SCOTCH cases; this also illustrates that the tree is not very large (but deep).

These remarks make sense when we know that AMF, AMD and PORD are based on local methods only aiming at minimizing either the degree or the fill.

Front size analysis

We now give some remarks on the frontal matrices and their distribution between brother nodes. According to Tables 10 and 11, we can say that in most cases, SCOTCH an METIS generate trees with frontal matrices bigger than those generated by the other reorderings. This observation will help us to explain some results in the next sections.

Table 12 gives the average standard deviation of the order of the frontal matrices of brother nodes. This illustrates if brother nodes have the same size. We can observe that in general AMD generates trees where brother nodes have approximatively the same size. For the other reorderings there does not appear to be a general rule except that SCOTCH has a tree where the order of brother nodes does not vary a lot (standard deviation is small).

Summary

Figure 6 gives the illustration of what we have observed in previous sections concerning the impact of reordering technique on assembly trees. For the shape of the tree, we have observed that hybrid heuristics like METIS and SCOTCH generate wide well-balanced trees (with a smaller number of nodes for SCOTCH). On the other hand, PORD, AMD and AMF give deep trees; it is interesting to notice that AMD provides better balanced trees than AMF and PORD (with a large number of nodes for AMF). From the frontal matrices point of view, METIS and SCOTCH give trees with bigger frontal matrices than the ones generated by other reorderings. In addition AMD, it is characterized by the fact that it gives trees with wide nodes on the tree.

5 Memory evolution in `MUMPS`

After studying the impact of reordering techniques on the structure of the assembly tree, we now focus on its consequence on the memory occupation in `MUMPS`. In this section, the unit used for memory occupation is the number of real entries, that is we do not take the integers into account.

As stated in Section 3.2, the memory occupation not only depends on the structure of the assembly tree, but also on its traversal order. Note that `MUMPS` makes the following transformation on the assembly tree during the analysis step: for each set of brother nodes, `MUMPS` orders first the node with the largest frontal matrix (the left-most position). The result is that large nodes (which normally correspond a large subtree) are processed first; thus, we can expect this order to help decreasing the peak of the stack, at least we expect it to be better than a random order.

5.1 Memory traffic

Table 13 gives the sum of the sizes for the contribution blocks for all the nodes of the tree, which represents the stack memory traffic of the factorization process. We can observe that the stack memory traffic for SCOTCH is the smallest (in most cases). This is due to the fact that SCOTCH has a smaller number of nodes compared to other reorderings. We also see that AMF leads to the biggest global stack memory traffic.

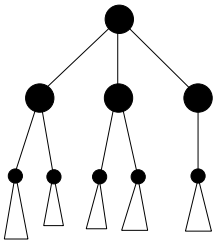
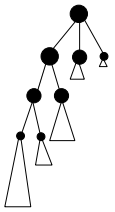
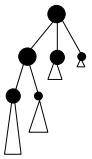
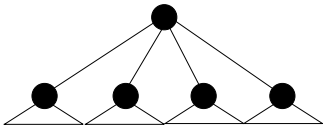
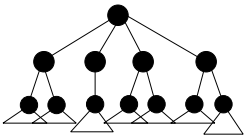
Reordering technique	Shape of the tree	observations
AMD		• Deep well-balanced tree • Large frontal matrices on top of the tree
AMF		• Very deep unbalanced tree • Small frontal matrices • Very large number of nodes
PORD		• deep unbalanced tree • Small frontal matrices • Large number of nodes
SCOTCH		• Very wide well-balanced tree • Large frontal matrices • Small number of nodes
METIS		• Wide well-balanced tree • Large number of nodes • Smaller frontal matrices (than SCOTCH)

Figure 6: Shape of the trees resulting from various reordering techniques.

5.2 Average stack size

Table 14 gives the average size of the stack during execution. The average size is defined to be the average values of the stack memory for all variations observed in the traces generated. We can observe that for AMF and PORD the average size of stack memory is smaller than for the other reorderings. This is because AMF and PORD have deep trees (as shown in the previous sections) and do not store a large number of contribution blocks at the same time. In addition the frontal matrices for AMF and PORD are smaller than those generated by the other reorderings.

5.3 Peak stack size

Finally, Table 15 gives the peak of the stack memory observed during the factorization. We can see that the reorderings giving deep trees provide better (*i.e.*, smaller) peaks of stack memory. Indeed, for our test problems, PORD and AMF have the smallest peak. This result is natural since deep trees do not need to store as many contribution blocks simultaneously as large trees as given by SCOTCH or METIS. We can also observe that the peak of stack memory for AMD (which has a deep tree) is greater than for PORD and AMF (which also have deep trees). In addition, for large problems, AMD tends to have the biggest peak. The first property of AMD's tree that can help us to explain this phenomena is that we have observed that the nodes on the top of the tree for AMD are larger than the ones for other reorderings. Indeed, once the processing begin to treat those nodes, the stack memory will contain large contribution blocks which will increase the size of the stack for the remaining subtrees. Furthermore, remember that the trees of AMF and PORD are more unbalanced than the one for AMD (Table 7, which gives the standard deviation of the depth of the leaves). Figure 6 illustrates the structural difference between AMD's and PORD-AMF's trees.
In addition, the global effect of the basic optimization used by `MUMPS` described at the beginning of this section is that it puts the biggest subtree (normally, the biggest node roots the biggest subtree) at the left-most position.
Therefore, and for all the reorderings, the factorization should start with the biggest subtree. For AMF and PORD, once the deepest subtree is treated, only smaller subtrees still need to be processed, requiring less memory. In opposition, for AMD, after treating the first node, subtrees that are not far from the first one in terms of size still need to be processed. This will cause an increase of the stack memory because of the storage due to the corresponding additional contribution blocks involved. This explains why PORD and AMF behave better in terms of stack size than AMD,

although all three have deep trees. We will discuss more the impact of the tree traversal on the stack memory size in Section 6.

Summary

To summarize this Section, we have seen that since reordering techniques have a strong impact on the shape of the assembly tree, they have also a strong impact on the memory usage in the factorization. The Figure 7 summaries the memory usage according to the ordering techniques. We have seen that PORD and AMF are the reorderings that use the smallest stack size (peak and average).From the factors point of view, for which the following has been observed we observed that for small matrices, factors with PORD and AMF are smaller than with SCOTCH and METIS, while for large matrices, METIS, SCOTCH and PORD give the smallest factors.

	Peak of the stack	**Average size of the stack**
METIS	+	+
SCOTCH	+	+
PORD	–	–
AMF	-	-
AMD	++	++

Figure 7: Characteristics of the stack memory for different reordering techniques. The symbol "+" means a bigger value, the symbol "-" means a lower value.

6 Optimal tree traversal order for memory usage

In the previous section, we measured the dynamic memory usage according to the reordering technique. But as noticed the dynamic memory usage also depends on the tree traversal. The simple strategy of `MUMPS` for tree traversal may not be optimal in terms of active memory usage. In this section we derive an algorithm which determines for a given assembly tree what is the optimal tree traversal for minimizing the dynamic memory usage.
We first define some notations which will be used for the description of the algorithm. Let i be a node in the tree and $nb\_children(i)$ the number of children of i. Children of i are noted $s_{i,j}$ where j varies between 1 and $nb\_children(i)$. We note cb_i the memory requirement to store the contribution block of the frontal matrix i. Finally, we use the notation $factor_i$ for the storage of the factors of the frontal matrix i (as shown in Figure 8).

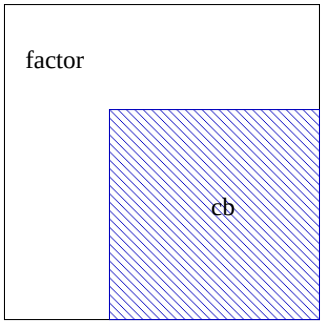


Figure 8: Structure of a frontal matrix.

In [13], Liu proposes an algorithm that finds the best traversal of the tree in terms of peak stack size for the code `MA27` (available in the Harwell Subroutine Library). Based on his work, we present here a variant which is more appropriate to the solver `MUMPS`. One specificity of Liu's algorithm is that it assumes that the space for the frontal matrix of a node reuses the space of the contribution block coming from its last child, resulting in a memory gain of the size of this last contribution block. In the case of `MUMPS` this optimization cannot be implemented simply in a distributed memory parallel context. Thus, the application of Liu's algorithm on the `MUMPS` memory management scheme does not always give the best traversal. Indeed, if we consider the tree given in Figure 9, the order given by Liu's algorithm is (**a-c-d-b**) which gives a peak of the stack of 13 (= 2+1+5+(3+2)), obtained when **b** is assembled and before the contribution blocks of **c** and **d** are released. However, using the order (**d-c-b-a**), a peak of 11 is obtained (also when **b** is assembled). This explains why we cannot apply Liu's algorithm in our case. Note also that Liu's initial algorithm was applied to elimination trees where only one pivot is eliminated at each node. In our case we work on amalgamated assembly trees. Finally, we are not only interested in decreasing the peak of the stack but we would also like to decrease the average stack size when this does not perturb the peak. Indeed, the minimization of the average is helpful in case of limited memory and in multi-user environment. This is illustrated in Figure 10 where the shaded curve, corresponding to the minimal stack memory average, is better since the stack is more often under the core limit. Note that by average, we mean the average of all successive sizes taken by the stack memory, as this corresponds to read and write accesses.

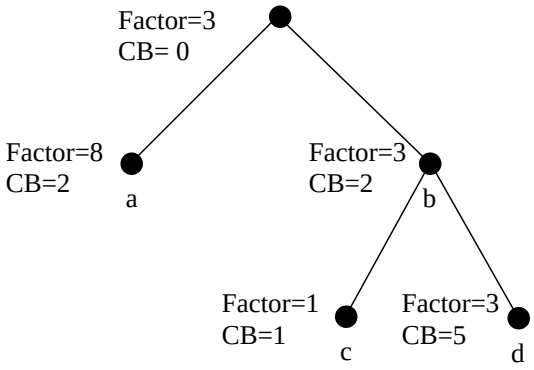


Figure 9: Example of the application of Liu's algorithm.

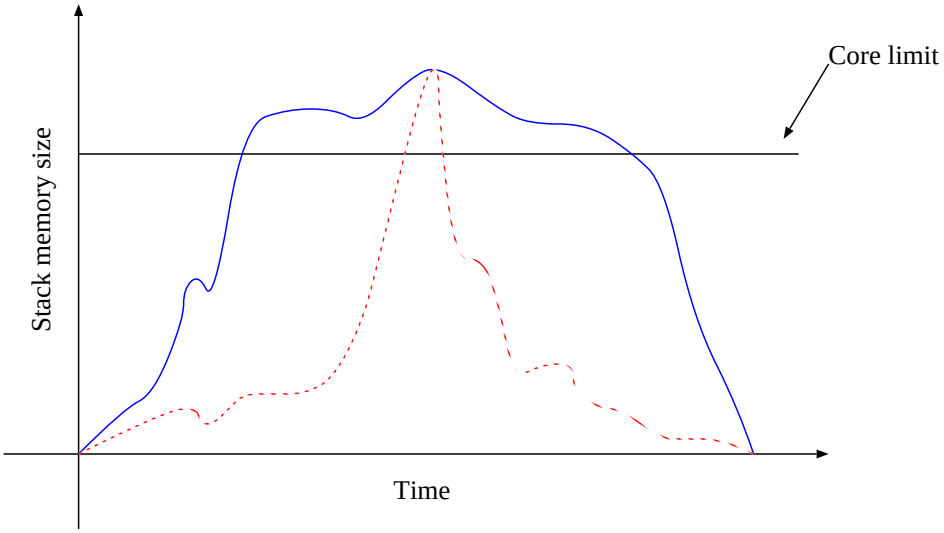


Figure 10: Stack memory evolution.

Let M_i be the amount of stack memory necessary to process node i. If i is a leaf then M_i is equal to $store_i = factor_i + cb_i$ real locations. For a parent node i, we must store in memory all contribution blocks of the children (if any) and the current frontal matrix; thus the assembly step requires a storage:

$$store_i + \sum_{j=1}^{nb\_children(i)} cb_{s_{i,j}}$$

The amount of memory needed to form the frontal matrix associated with a node $s_{i,j}$ is $M_{s_{i,j}}$. Moreover, the stack must contain the first $j-1$ contribution blocks of the brothers of $s_{i,j}$ that have already been processed. The result is that the storage requirement when factorizing the frontal matrix associated with $s_{i,j}$ is:

$$M_{s_{i,j}} + \sum_{k=1}^{j-1} cb_{s_{i,k}}$$

Finally, it results that the storage requirement to process node i is recursively defined as:

$$\begin{aligned} M_i = \max(& \max_{j=1,nb\_children(i)} (M_{s_{i,j}} + \sum_{k=1}^{j-1} cb_{s_{i,k}}), \\ & store_i + \sum_{j=1}^{nb\_children(i)} cb_{s_{i,j}}) \end{aligned} \tag{1}$$

Since we want to minimize the peak of the stack, we should reduce the value of M for the root node(s). Adapting a theorem from [13] which says that the minimum of $\max_j x_j + \sum_{i=1}^{j-1} y_i$ is obtained when the sequence (x_i, y_i) is sorted in decreasing order of $x_i - y_i$, we deduce that an optimal child sequence is obtained by rearranging the children nodes in decreasing order of $M_{s_{i,k}} - cb_{s_{i,k}}$.
Considering a tree T with n nodes, based on this result, Algorithm 1 gives an optimal traversal of the tree in terms of the peak of the stack.
The function **Sort_Sons** is the main part of the algorithm. It sorts the children nodes so that the stack memory peak is minimized and decreases the average of stack memory. Note that the function computes two possible orders for the children nodes. The first one reorders the children in descending order of $M_{s_{i,j}} - cb_{s_{i,j}}$ (in case of equality we sort the children nodes concerned in increasing order of $cb_{s_{i,j}}$).

This order gives the best traversal of the tree in terms of peak of stack memory. The second order computed is based on the sizes of the contribution blocks – increasing order of $cb_{s_{i,j}}$ – so that it only decreases the average size of the stack. Then, we compute the new value M_i of the parent node i for those two orders. If the peak is the same, that is if both orders give the optimal peak (this happens if the term $store_i + \sum_{j=1}^{nb\_children(i)} cb_{s_{i,j}}$ in (1) is large), we choose the latter order because it has good chances to better decrease the average stack size (large contribution blocks treated last). Note that there could exist a better order than this one in terms of average stack memory but finding it could become too costly. We give three remarks illustrating this point.

- Although sorting brother nodes in growing order of contribution blocks is intuitively a good way to decrease the average stack, there are counter examples. Figure 11 gives a tree that illustrates this point: if we consider the traversal (**a-b-c-d-e-f-g-h**), which represents the traversal given by our method to decrease the stack memory average, we obtain an average size of 36.47. However, with the order (**c-d-e-f-g-a-b-h**), the average size of the stack is equal to 33.23.

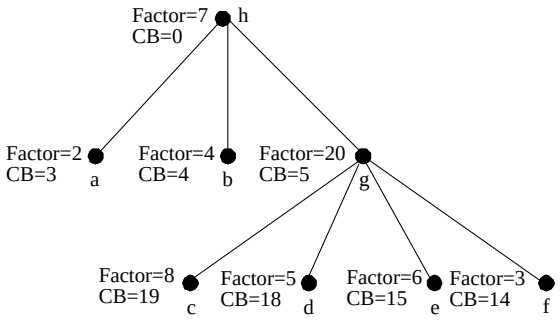


Figure 11: Example of an assembly tree.

- To be more precise about the average we give a formal definition of the problem. We consider the tree given in Figure 12 where a parent node i has three sons. Let $cb_j, j = 1, 2, 3$ be the size of the contribution block for child j. We note A_j the average size of the stack during the process of the subtree rooted by j. Let α_j be the number of variations of the stack during the factorization process of subtree j. During the treatment of the successive subtrees, the memory is equivalent to A_1 during α_1 steps, then $A_2 + cb_1$ during α_2 steps, and for the

final subtree, $A_3 + cb_1 + cb_2$ during α_3 steps. Thus the average size of the stack is

$$A_i = \frac{\alpha_1 \times A_1 + \alpha_2 \times (A_2 + cb_1) + \alpha_3 \times (A_3 + cb_1 + cb_2)}{\alpha_1 + \alpha_2 + \alpha_3}$$

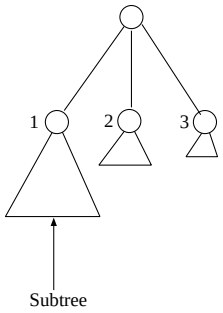


Figure 12: Node with 3 subtrees.

Since $\sum_j \alpha_j \times A_j$ does not depend on the order of the subtrees, we want to find an order of the subtrees that minimizes $\alpha_2 \times cb_1 + \alpha_3 \times (cb_1 + cb_2)$. More generally, the problem consists in finding a permutation σ for each node of the tree which is a solution of:

$$\min_{\sigma} \sum_{j=1}^{nb\_children(i)-1} (\sum_{k=1}^{j} cb_{s_{i,\sigma(k)}}) \times \alpha_{s_{i,\sigma(j+1)}}$$

Since this is not compatible with our first goal which is to minimize the stack memory peak, we only use our simple heuristic for the average described above, which will be exact only if the α_j are equal.

- We sort children nodes in growing order of contribution blocks only when it does not increase the stack memory peak for the given subtree. If we consider a case where the peak of the stack is obtained for the assembly operations for the root node, we can rearrange all the nodes in the tree so that we minimize stack memory average (as long as the global peak is not bigger). However, this is not taken into account in our algorithm.

Tree_Reorder (T)**:**
begin
  for all i in the set of root nodes **do**
    Process_Child(i);
  end for
  if the stack peak is the same as before and the average stack is worse **then**
    Use the initial tree traversal
  else
    Use the tree traversal computed
  end if
end
Process_Child(i)**:**
begin
  if i is a leaf **then**
    M_i=$store_i$
  else
    for $j = 1$ to $nb\_children(i)$ **do**
      Process_Child($s_{i,j}$);
    end for
    M_i=Sort_Sons(i);
  end if
end
Sort_Sons (i) **:**
begin
  build the permutation P_1 that sorts the children nodes j of i in descending order of $(M_{s_{i,j}} - cb_{s_{i,j}})$. In case of equality between two children sort them in increasing order of $cb_{s_{i,j}}$
  build the permutation P_2 that sorts the children nodes j of i in growing order of $cb_{s_{i,j}}$;
  compute $M_i(P_1)$ and $M_i(P_2)$ for P_1 and P_2 respectively using the formula (1);
  if $(M_i(P_1) = M_i(P_2))$ **then**
    sort the children nodes of i according to P_2;
    return$(M_i(P_2))$;
  else
    sort the children nodes of i according to P_1;
    return$(M_i(P_1))$;
  end if
end

Algorithm 1: Optimal tree reordering for minimizing stack memory peak.

We have implemented this algorithm to demonstrate its effectiveness. Figure 13 illustrates the impact of the algorithm on the size of the stack memory during execution.

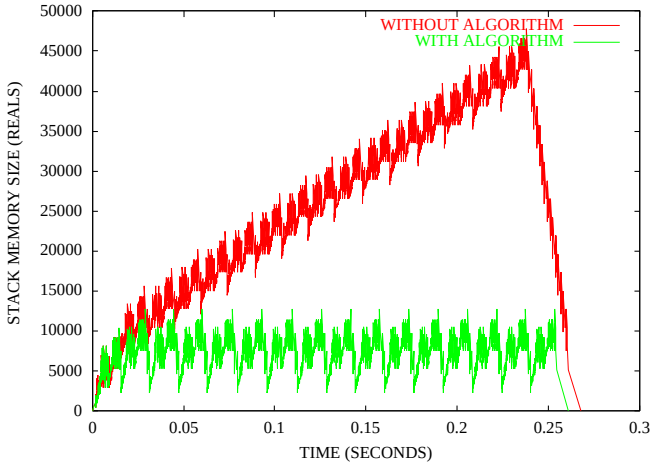


Figure 13: Impact of Algorithm 1 on the size of stack memory for the matrix THERMAL with PORD.

As we said, Algorithm 1 was designed to decrease the stack memory space and does not take the memory for the factors into account. It is well adapted for out-of-core-like algorithms where we store on disk the data (factors) that will not be reused. In the case where we have a large enough amount of memory to run in-core, it is more interesting to minimize the total memory (i.e. factor storage+stack memory storage). The Algorithm 2 follows the same idea as the Algorithm 1. Let T_i be the set of nodes in the subtree rooted at node i. The storage requirement for the factors associated to this T_i is noted $factor_{T_i}$. Note that it contains the factors for node i. Formally,

$$factor_{T_i} = \sum_{j \in T_i} factor_j$$

The memory requirement to perform the assembly operation for a node i is given by

$$store_i + \sum_{j=1}^{nb\_children(i)} (cb_{s_{i,j}} + factor_{T_{s_{i,k}}})$$

Indeed, to perform an assembly operation on node i, the memory must be large enough to contain the contribution blocks of each child, the factor already computed and the frontal matrix.
The amount of memory needed to form the frontal matrix associated with a node $s_{i,j}$ is $M_{s_{i,j}}$. Moreover, the stack must contain the first $j-1$ contribution blocks (as explained for the previous algorithm). In addition, the factor memory must contain all the factors processed until this step. This results in a storage requirement for the processing of the matrix associated with $s_{i,j}$ is given by:

$$M_{s_{i,j}} + \sum_{k=1}^{j-1} (cb_{s_{i,k}} + factor_{T_{s_{i,k}}})$$

Finally, the storage requirement for processing node i is given by:

$$\begin{aligned} M_i = \max(& \max_{j=1,nb\_children(i)} (M_{s_{i,j}} + \sum_{k=1}^{j-1} (cb_{s_{i,k}} + factor_{T_{s_{i,k}}})), \\ & store_i + \sum_{j=1}^{nb\_children(i)} (cb_{s_{i,j}} + factor_{T_{s_{i,k}}})) \end{aligned} \quad (2)$$

Following the same logic as before, we can tell that rearranging the children nodes in decreasing order of $M_{s_{i,k}} - (cb_{s_{i,k}} + factor_{T_{s_{i,k}}})$ gives an optimal child sequence in terms of minimization of M_i.
Algorithm 2 (based on the result given above) processes the optimal tree traversal in terms of memory peak for a tree T with n nodes.

6.1 Impact of the tree traversal on the stack size

Algorithm 1 was designed to obtain the optimal peak of the stack. Intuitively, the potential gains will be largest for deep unbalanced trees. We perform the experiments of Section 5 again, after Algorithm 1 has been applied to the tree.
In Table 16 (values on the left), we report on the gain in stack memory usage after applying Algorithm 1. The gain is computed between the value of peak of memory stack of standard `MUMPS` and `MUMPS` where we postprocess the tree using the Algorithm 1. We observe that the algorithm gives better gains for reorderings that generate wide well-balanced tree (SCOTCH and METIS).
This behaviour could at first appear surprising, but is indeed due to the sorting of brother nodes already implemented in `MUMPS`. Indeed, for deep unbalanced trees,

Tree_Reorder (T)**:**
begin
  for all i in the set of root nodes **do**
    Process_Child(i);
  end for
  if the memory peak is the same as before and the average is worse **then**
    Use the initial tree traversal
  else
    Use the tree traversal computed
  end if
end

Process_Child(i)**:**
begin
  if i is a leaf **then**
    M_i=$store_i$
  else
    for $j = 1$ to $nb\_children(i)$ **do**
      Process_Child($s_{i,j}$);
    end for
    M_i=Sort_Sons(i);
  end if
end

Sort_Sons (i)**:**
begin
  build the permutation P_1 that sorts the children nodes j of i in descending order of $(M_{s_{i,j}} - (cb_{s_{i,j}} + factor_{T_{s_{i,j}}}))$. In case of equality between two children sort them in increasing order of $cb_{s_{i,j}} + factor_{T_{s_{i,j}}}$
  build the permutation P_2 that sorts the children nodes j of i in growing order of $cb_{s_{i,j}} + factor_{T_{s_{i,j}}}$;
  compute $M_i(P_1)$ and $M_i(P_2)$ for P_1 and P_2 respectively using the formula (2);
  if $(M_i(P_1) = M_i(P_2))$ **then**
    sort the children nodes of i according to P_2;
    return$(M_i(P_2))$;
  else
    sort the children nodes of i according to P_1;
    return$(M_i(P_1))$;
  end if
end

Algorithm 2: Optimal tree reordering for minimizing global memory peak.

treating the largest brother node first tends to provide a good tree traversal, close to the optimal.

The gains obtained if this mechanism is switched off are reported in Table 16 (values on the right). As expected, largest gains are obtained for deep unbalanced trees (AMD-AMF-PORD).

In fact, compared to `MUMPS`, our algorithm gives the best results for well-balanced wide trees. This shows that although `MUMPS` can generally improve the stack for the worse cases, its simple strategy is not sufficient for this type of trees. Even for deep unbalanced trees, there are some cases where we still obtain a large gain compared to `MUMPS`. This is the case for matrix THERMAL with PORD where Algorithm 1 reduces the stack by 73.5%. In some exceptional cases it can even happen that `MUMPS` degrades the original traversal (gain on the left better than the one on the right). This happens for example for THERMAL with AMF where the traversal obtained by the strategy in `MUMPS` is worse than the original order (which in that case was optimal).

Finally, Table 17 shows the impact of the algorithm on the average stack size. Although the average stack is usually reduced by our simple heuristic, there are some cases where decreasing the peak implies increasing the average stack size when using our simple heuristic.

6.2 Impact of the tree traversal on the total memory

We now focus on the total memory. Table 18 gives some results about the percentage of decrease of the memory occupation after applying Algorithm 2. As could be expected, we can see that the algorithm does not decrease the total memory peak for test problems where the stack memory is too small compared to the factors. On the other hand, if the stack is the major part of the memory, we can see that the decrease can be large, as shown for example with GUPTA3 and METIS where it reaches 30.7 %.

However, note that for such problems where the stack is large, Algorithm 1 would also decrease the peak of total memory. Thus, for most cases, Algorithm 2 cannot give huge gains in comparison to Algorithm 1. Comparing the total memory peak obtained with both algorithms, we observed that the gain due to Algorithm 2 did not exceed 2 %.

However, Algorithm 2 gives the optimal total memory peak in all cases and is worth using if this is objective.

Final comments

To summarize, we have proposed an algorithm that provides the optimal peak of the stack and in most cases reduces the average stack size. Even when the stack and the factors are of the same order, the second algorithm that obtains the optimal peak of total memory, does not provide so useful gains compared to Algorithm 1. In fact, our first algorithm will be of primary interest for an out-of-core implementation of multifrontal solvers where the factors are put on disk during the factorization process.
We finally report in Table 19 the optimal peak stack, measured after the application of Algorithm 1 which minimizes the stack. We can see that the behaviour is similar to the one observed in Table 15: the reordering techniques that generate a well-balanced wide tree like METIS and SCOTCH have a larger peak of the stack than the others. In addition, the well balanced deep trees of AMD give the largest peak of the stack in a lot of cases – because AMD trees have very large frontal matrices at the top of the tree. Finally, very deep unbalanced trees like the ones given by AMF and PORD are the most effective in terms of the size of the stack.

7 Conclusion and future work

In this paper we have presented an experimental study of the memory usage of a multifrontal solver. This memory usage can be divided into two parts: a static one which corresponds to the factor size, and a dynamic one which corresponds to the active working space (the stack) needed to perform the computation. Dynamic memory usage is determined by the assembly tree which itself depends on the reordering technique applied. Reordering techniques have a significant impact on the memory usage: by the amount of fill-in for the static memory, and on the shape of the assembly tree for the dynamic memory.
Whereas there are a lot of studies on the impact of reordering on fill-in, to our knowledge this is the first work which studies the links between the reordering technique and the stack evolution of multifrontal solvers. We began by studying the impact of reordering techniques on assembly trees. We have observed that reordering techniques like METIS and SCOTCH give large well-balanced trees while reordering techniques like AMF and PORD (resp. AMD) give very deep unbalanced (resp. balanced) trees with a large number of nodes. After that we used these results to explain the memory behaviour of the solver with each reordering technique. We have seen that the deep unbalanced trees are better than the large well-balanced ones for the memory occupation. We remind that in our study, the starting point

is the amalgamated assembly tree, ie, the tree after the symbolic factorization has been performed. Clearly post-treating the tree with node splitting or amalgamation techniques would also have an impact on the memory. (For example SCOTCH which provides amalgamated trees with a small number of large nodes clearly has a smaller stack traffic.)
We have also shown that the stack memory evolution not only depends on the shape of the tree but also on the tree traversal during the factorization. We proposed an algorithm to find the best tree traversal (in terms of memory occupation) in `MUMPS`. Actually we have designed two algorithms for minimizing memory occupation of the solver for both the in-core (total memory minimization) and the out-of-core (active memory minimization) cases. We obtained good results with the algorithm that minimizes the active memory. With the other one the gains are smaller (we do not get any gain for the matrices where active memory is too small). The algorithms proposed are available in release 4.2 of the software package `MUMPS`.
We are now interested in studying the memory behaviour for the parallel case and design an out-of-core scheme to write the factors on disk. In the latter case, techniques to optimize the size of the stack will be extremely useful, and special care should be taken to efficiently balance the memory dynamically.

Acknowledgements

We are grateful to Patrick Amestoy for his comments on an earlier version of this report.

References

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

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

[3] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

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

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

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

[7] I. S. Duff and J. Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications*, 22(4):973–996, 2001.

[8] 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.

[9] 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.

[10] A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ., 1981.

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

[12] J. W. H. Liu. Modification of the minimum degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 11(2):141–153, 1985.

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

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

[15] E. Ng and P. Raghavan. Performance of greedy heuristics for sparse cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*, 20:902–914, 1999.

[16] F. Pellegrini. SCOTCH 3.4 User's guide. Technical Report RR 1264-01, LaBRI, Université Bordeaux I, November 2001.

[17] F. Pellegrini, J. Roman, and P. R. Amestoy. Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering. *Concurrency: Practice and Experience*, 12:69–84, 2000. Preliminary version published in *Proceedings of Irregular'99*, LNCS 1586, 986–995.

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

[19] Edward Rothberg and Stanley C. Eisenstat. Node selection strategies for bottom-up sparse matrix ordering. *SIAM Journal on Matrix Analysis and Applications*, 19(3):682–695, 1998.

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

Appendix A: Number of floating-point operations

	METIS	SCOTCH	PORD	AMF	AMD
3DTUBE	12703.4	*12220.3*	14509.5	36373.9	**38676.0**
M_T1	20922.8	20172.5	*16006.9*	18962.8	24398.6
OILPAN	3040.7	3035.7	2660.5	3615.7	**3848.6**
SHIP_003	83828.2	92614.0	*112519.6*	**96445.2**	155725.5
THREAD	35123.4	37142.3	*26959.8*	30203.0	43394.6
BCSSTK34	**5.6**	4.0	*3.3*	3.9	4.5
BCSSTK38	146.2	190.8	*113.2*	118.0	127.1
CFD2	33506.7	32263.8	28743.1	**80577.5**	*122500.9*
GUPTA1	536.7	**652.0**	529.0	446.4	434.7
GUPTA2	*2757.8*	4510.7	**4993.3**	2790.3	2663.9
GUPTA3	6346.5	**70527.4**	5924.0	20124.8	3229.4
NASA1824	7.0	**8.5**	4.3	5.1	5.6
NASA2910	26.0	27.0	*20.8*	20.9	22.2
NASA4704	40.6	41.7	*27.2*	31.2	35.4
STRUCT4	571.5	599.1	637.0	*1765.2*	2041.6
VIBROBOX	930.2	*1099.2*	671.8	804.7	**950.4**
AF23560	3106.8	3793.5	*1999.4*	**5006.3**	2580.3
BIG	34.3	32.3	30.3	36.4	**43.4**
CIRCUIT_4	15.9	18.0	16.7	10.2	**8.7**
EPB3	440.3	523.0	451.4	**784.9**	*1179.2*
GARON02	283.7	261.5	*238.6*	303.4	**353.5**
GRAHAM1	185.4	*142.2*	145.4	173.6	153.4
GRID48	4679.9	*4184.0*	4364.9	4787.5	**5207.7**
LI	16759.6	155434.3	21872.9	**61409.4**	*108784.0*
ONETONE1	**2819.5**	2108.4	1997.7	2493.0	2313.1
PRE2	189660.3	*172149.1*	364746.4	408845.5	**527223.8**
RMA10	1691.1	1466.1	*1172.3*	1232.6	1412.6
SAYLR1	0.077	0.075	0.064	*0.057*	0.062
THERMAL	16.3	*14.9*	16.5	**26.7**	18.8
TWOTONE	29120.3	*27764.7*	37167.4	29847.5	29552.9
VENKAT50	1834.2	1903.0	*1740.1*	2030.7	2333.7
WANG1	*33.5*	59.7	50.5	44.5	**62.1**
WANG3	4313.1	5801.7	5009.9	**6318.0**	*10492.2*
XENON2	**99273.1**	112213.4	126349.7	237451.3	298363.5

Table 3: Number of floating-point operations (millions).

Appendix B: Details on the shapes of the trees

	METIS	SCOTCH	PORD	AMF	AMD
3DTUBE	2504	*805*	**2860**	2809	2308
M_T1	4160	*2103*	**4452**	4294	4199
OILPAN	6296	*5873*	6582	**6699**	6073
SHIP_003	7454	*4294*	7798	**8253**	7634
THREAD	1138	*326*	**1478**	1387	1218
BCSSTK34	57	*33*	77	**80**	62
BCSSTK38	794	*464*	859	**889**	797
CFD2	14093	*5222*	17337	**18318**	14919
GUPTA1	19849	*8866*	**20441**	19777	19742
GUPTA2	30499	*11970*	**31731**	30311	30243
GUPTA3	413	*26*	1790	1300	**1898**
NASA1824	235	*177*	279	**280**	228
NASA2910	372	*136*	390	**393**	370
NASA4704	580	*457*	672	**695**	614
STRUCT4	355	*73*	456	**576**	444
VIBROBOX	1506	*948*	2016	**2022**	1375
AF23560	5700	*662*	6835	**8842**	8214
BIG	3608	*3339*	3945	**4019**	3496
CIRCUIT_4	44260	51618	**51899**	40273	*40263*
EPB3	*23018*	23665	24087	**24162**	23276
GARON02	1685	*1148*	**1925**	1858	1743
GRAHAM1	1818	*993*	**1972**	1151	1114
GRID48	3537	*2047*	**3590**	3574	3548
LI	1679	*266*	2175	**2405**	2058
ONETONE1	10428	9804	**11167**	9685	*8931*
PRE2	198571	*170053*	**215348**	205297	195812
RMA10	4587	*3465*	5109	**5325**	4524
SAYLR1	*100*	102	113	**118**	101
THERMAL	622	*354*	686	**700**	646
TWOTONE	36104	*27904*	41309	**41794**	39460
VENKAT50	6792	*5788*	7103	**7367**	6625
WANG1	**1452**	*589*	1322	1380	1336
WANG3	11911	*4519*	**12979**	12749	12559
XENON2	18887	*13130*	**20455**	20386	19043

Table 4: Number of nodes in the tree.

	METIS	SCOTCH	PORD	AMF	AMD
3DTUBE	1106	*416*	**1136**	1029	981
M_T1	1636	*917*	**1698**	1555	1491
OILPAN	3988	**4559**	4298	4164	*3805*
SHIP_003	3277	*2551*	**3392**	3193	3283
THREAD	431	*147*	**525**	445	401
BCSSTK34	24	*19*	36	**38**	24
BCSSTK38	360	*245*	**393**	367	339
CFD2	7043	*2367*	7561	**7881**	6572
GUPTA1	13745	*8796*	11825	**16518**	16504
GUPTA2	20574	*11832*	**22426**	20520	20420
GUPTA3	282	*7*	419	**439**	404
NASA1824	114	*76*	110	**116**	92
NASA2910	**168**	*59*	147	146	134
NASA4704	**285**	240	283	276	*239*
STRUCT4	146	*34*	195	**306**	217
VIBROBOX	728	*428*	**928**	806	554
AF23560	4534	*305*	5599	**7569**	7367
BIG	2169	2056	2341	**2375**	*1929*
CIRCUIT_4	34559	42561	**42793**	*30075*	30272
EPB3	12745	13044	**13250**	11053	*10855*
GARON02	873	*593*	**874**	780	765
GRAHAM1	1063	640	**1329**	449	*413*
GRID48	**1530**	*1134*	1288	1272	1246
LI	728	*138*	957	**1035**	895
ONETONE1	6589	6686	**7544**	5694	*5086*
PRE2	132706	*117317*	**159465**	134563	119457
RMA10	1987	*1485*	2183	**2227**	1790
SAYLR1	72	79	90	**95**	*71*
THERMAL	287	*164*	**304**	288	288
TWOTONE	23737	*19202*	**30087**	29988	26646
VENKAT50	3152	**3434**	3318	3296	*3063*
WANG1	**1268**	*413*	1101	1162	1127
WANG3	10187	*3107*	**11102**	10754	10614
XENON2	9139	9249	**10058**	9231	*8043*

Table 5: Number of leaves in the tree.

	METIS	SCOTCH	PORD	AMF	AMD
3DTUBE	44.2	**51.7**	39.7	*37.5*	42.5
M_T1	39.3	**43.6**	38.1	35.9	*35.5*
OILPAN	63.3	**77.6**	65.3	60.1	62.7
SHIP_003	43.8	**59.4**	43.5	*42.5*	43
THREAD	37.9	**45.1**	35.5	33.1	*32.9*
BCSSTK34	43.9	**57.6**	46.8	41.3	*38.7*
BCSSTK38	46.1	**52.8**	45.8	*41.6*	42.5
CFD2	50	45.3	43.6	*43*	44.1
GUPTA1	*56.9*	**99.2**	57.8	83.7	83.6
GUPTA2	69.3	**98.8**	70.7	67.7	*67.5*
GUPTA3	**95.9**	26.9	23.4	33.8	*21.3*
NASA1824	**48.9**	42.9	*39.4*	41.4	40.4
NASA2910	44.9	43.4	37.7	37.2	*36.2*
NASA4704	48.7	**52.5**	42.1	39.7	*38.9*
STRUCT4	*41*	46.6	42.8	**53.1**	48.9
VIBROBOX	48.3	45.1	46	*39.9*	40.3
AF23560	79.5	*46.1*	81.9	85.6	**89.7**
BIG	60.1	**61.6**	59.3	59.1	55.2
CIRCUIT_4	82.7	82.5	82.5	*74.7*	75.2
EPB3	**55.4**	55.1	55	*45.7*	46.6
GARON02	**51.9**	51.7	45.4	*42*	43.9
GRAHAM1	61.9	64.5	**67.4**	39	*37.1*
GRID48	43.3	**55.4**	35.9	35.6	*35.1*
LI	43.4	**51.9**	44	*43*	43.5
ONETONE1	64.9	68.2	67.6	58.8	*56.9*
PRE2	69	69	**74**	65.5	61
RMA10	43.2	42.9	42.7	41.8	*39.6*
SAYLR1	72.3	77.5	79.6	**80.5**	70.3
THERMAL	46.1	46.3	44.3	*41.1*	44.6
TWOTONE	68.2	68.8	72.8	71.8	*67.5*
VENKAT50	46.4	**59.3**	46.7	44.7	46.2
WANG1	**87.3**	*70.1*	83.3	84.2	84.4
WANG3	**85.5**	*68.8*	85.5	84.4	84.5
XENON2	48.3	**70.4**	49.2	45.3	*42.2*

Table 6: Percentage of leaves.

	METIS	SCOTCH	PORD	AMF	AMD
3DTUBE	1.5	*0.6*	9.6	**18.1**	1.9
M_T1	7.6	*1.7*	33.6	**73.9**	12.6
OILPAN	4.0	*0.8*	**41.4**	36.0	12.2
SHIP_003	14.0	*3.2*	321.4	**508.8**	25.4
THREAD	4.3	*1.8*	36.6	**1682.5**	12.8
BCSSTK34	2.1	*0.7*	4.8	**63.6**	5.4
BCSSTK38	6.3	*2.3*	19.8	**109.4**	24.3
CFD2	1.7	*0.7*	38.0	**105.3**	12.7
GUPTA1	6.2	*1.7*	15.3	**28.6**	17.6
GUPTA2	4.9	*1.5*	**136.4**	58.5	16.6
GUPTA3	106.0	3.8	*1.0*	**114.4**	5.7
NASA1824	*1.6*	1.6	3.9	**5.2**	2.1
NASA2910	*1.4*	1.5	**7.2**	6.7	5.2
NASA4704	3.5	*1.3*	9.9	**12.1**	2.2
STRUCT4	3.9	*0.5*	15.6	**525.3**	3.0
VIBROBOX	1.8	*0.6*	9.1	**15.3**	2.7
AF23560	3.8	*1.2*	77.3	**16662.6**	44.7
BIG	1.9	*0.9*	32.3	**44.4**	21.5
CIRCUIT_4	19.8	*3.0*	8.9	21.2	**27.0**
EPB3	1.1	*0.9*	**77.2**	34.0	7.2
GARON02	3.7	*1.1*	23.1	28.9	**72.1**
GRAHAM1	3.0	*0.7*	10.8	**15.7**	3.5
GRID48	2.2	*1.5*	21.8	**77.4**	8.4
LI	2.6	*1.4*	7.4	**2206.9**	8.9
ONETONE1	24.4	*7.0*	445.0	**952.3**	34.5
PRE2	66.1	*8.7*	**341.7**	265.0	21.5
RMA10	12.8	*7.4*	67.7	**2084.4**	1331.7
SAYLR1	*1.1*	1.4	1.7	**4.9**	1.7
THERMAL	1.6	*0.4*	25.0	**301.9**	28.4
TWOTONE	30.2	*5.2*	294.8	**458.5**	80.7
VENKAT50	5.9	*1.5*	13.9	**41.7**	6.0
WANG1	1.4	*1.3*	17.4	**67.6**	3.7
WANG3	2.3	*1.8*	**40.2**	38.6	4.0
XENON2	6.9	*1.7*	101.6	**144.7**	10.3

Table 7: Variance of the depth of leaves.

	METIS	SCOTCH	PORD	AMF	AMD
3DTUBE	16	*10*	20	**29**	15
M_T1	31	*14*	34	**51**	25
OILPAN	18	*14*	35	**53**	24
SHIP_003	27	*14*	75	**122**	32
THREAD	17	*10*	32	**155**	20
BCSSTK34	8	*5*	11	**27**	10
BCSSTK38	17	*9*	22	**45**	25
CFD2	17	*13*	49	**66**	31
GUPTA1	16	*7*	20	**30**	25
GUPTA2	15	*9*	**55**	40	26
GUPTA3	31	*8*	9	**41**	13
NASA1824	10	*9*	12	**14**	10
NASA2910	11	*9*	14	**17**	14
NASA4704	13	*10*	**20**	19	13
STRUCT4	20	*7*	22	**85**	12
VIBROBOX	12	*10*	19	**24**	13
AF23560	17	*11*	43	**485**	37
BIG	14	*13*	31	**36**	28
CIRCUIT_4	183	*11*	15	200	**617**
EPB3	17	*15*	**44**	37	26
GARON02	17	*16*	28	30	**51**
GRAHAM1	14	*10*	21	**22**	18
GRID48	18	*15*	31	**49**	23
LI	13	*8*	22	**149**	18
ONETONE1	26	*14*	77	**170**	34
PRE2	49	*18*	88	**99**	42
RMA10	*28*	40	43	**208**	165
SAYLR1	*6*	6	7	**10**	7
THERMAL	10	*8*	23	**62**	23
TWOTONE	44	*15*	77	**193**	47
VENKAT50	23	*15*	30	**42**	21
WANG1	10	*9*	18	**33**	12
WANG3	17	*12*	36	**37**	18
XENON2	25	*16*	54	**65**	26

Table 8: Maximum depth for a node.

	METIS	SCOTCH	PORD	AMF	AMD
3DTUBE	1.79	**2.07**	1.66	*1.58*	1.74
M_T1	1.65	**1.77**	1.62	1.57	*1.55*
OILPAN	2.73	**4.47**	2.88	*2.64*	2.68
SHIP_003	1.78	**2.46**	1.77	*1.63*	1.75
THREAD	1.61	**1.82**	1.55	*1.47*	1.49
BCSSTK34	1.70	**2.29**	1.85	1.88	*1.60*
BCSSTK38	1.83	**2.11**	1.84	*1.70*	1.74
CFD2	**2.00**	1.83	1.77	*1.75*	1.79
GUPTA1	3.25	**126.64**	*2.37*	6.07	6.10
GUPTA2	*3.07*	**86.73**	3.41	3.10	3.08
GUPTA3	**3.15**	1.32	1.30	1.51	*1.27*
NASA1824	**1.93**	1.74	*1.65*	1.70	1.67
NASA2910	**1.82**	1.75	1.60	1.59	*1.56*
NASA4704	1.96	**2.10**	1.73	1.66	*1.64*
STRUCT4	*1.69*	1.85	1.74	**2.13**	1.95
VIBROBOX	**1.93**	1.82	1.85	*1.66*	1.67
AF23560	4.89	*1.85*	5.53	6.95	**9.70**
BIG	2.51	**2.60**	2.46	2.44	*2.23*
CIRCUIT_4	3.20	4.24	**4.25**	*2.66*	2.71
EPB3	**2.24**	2.23	2.22	*1.84*	1.87
GARON02	**2.07**	2.07	1.83	*1.72*	1.78
GRAHAM1	2.41	2.81	**3.06**	1.64	*1.59*
GRID48	1.76	**2.24**	1.56	1.55	*1.54*
LI	1.76	**2.06**	1.78	*1.75*	1.77
ONETONE1	2.71	**3.14**	3.08	2.42	*2.32*
PRE2	3.02	3.22	**3.85**	2.90	*2.56*
RMA10	**1.76**	1.75	1.75	1.72	*1.65*
SAYLR1	3.54	4.39	4.87	**5.09**	*3.33*
THERMAL	1.85	**1.86**	1.79	*1.70*	1.80
TWOTONE	*2.92*	3.21	**3.68**	3.54	3.08
VENKAT50	1.87	**2.46**	1.88	*1.81*	1.86
WANG1	**7.89**	*3.34*	5.98	6.33	6.39
WANG3	6.91	*3.20*	**6.91**	6.39	6.46
XENON2	1.93	**3.37**	1.96	1.82	*1.73*

Table 9: Average number of children.

	METIS	SCOTCH	PORD	AMF	AMD
3DTUBE	*1398*	1398	1698	**2894**	2888
M_T1	**2004**	1710	*1377*	1824	1965
OILPAN	847	791	*763*	**952**	931
SHIP_003	3456	*3156*	3426	3408	**4038**
THREAD	**3096**	3078	*2274*	2502	2853
BCSSTK34	**154**	134	112	*111*	141
BCSSTK38	338	**468**	333	*332*	360
CFD2	2458	2332	*1958*	2808	**3619**
GUPTA1	705	706	*674*	727	**730**
GUPTA2	*1264*	1475	**1730**	1428	1454
GUPTA3	*827*	**5058**	1643	3028	1030
NASA1824	160	**187**	135	*133*	163
NASA2910	*190*	190	203	207	**231**
NASA4704	**297**	290	*251*	265	295
STRUCT4	*695*	702	834	1143	**1351**
VIBROBOX	938	981	*840*	912	**984**
AF23560	853	**872**	*488*	852	552
BIG	142	144	*120*	137	**158**
CIRCUIT_4	192	**246**	234	179	*139*
EPB3	356	461	*337*	507	**636**
GARON02	338	312	*293*	**356**	318
GRAHAM1	288	266	*254*	**314**	304
GRID48	*745*	765	845	825	**945**
LI	2463	3813	*2146*	2310	**3815**
ONETONE1	**1066**	928	*784*	796	986
PRE2	4252	*4204*	5514	6476	**7502**
RMA10	**466**	422	*378*	439	399
SAYLR1	26	**31**	27	*25*	26
THERMAL	87	75	*72*	**129**	108
TWOTONE	2382	*2316*	2561	2588	**2684**
VENKAT50	*484*	564	560	624	**676**
WANG1	*227*	275	279	268	**316**
WANG3	*1199*	1280	1221	1412	**1658**
XENON2	*2554*	2623	2743	3663	**4501**

Table 10: Max Front.

	METIS	SCOTCH	PORD	AMF	AMD
3DTUBE	178	**293**	*169*	171	190
M_T1	180	**220**	*157*	162	160
OILPAN	**68**	*60*	64	67	67
SHIP_003	**167**	153	165	163	*151*
THREAD	279	**490**	254	334	*251*
BCSSTK34	67	**73**	*53*	56	59
BCSSTK38	63	**70**	*59*	60	60
CFD2	80	**141**	*72*	76	84
GUPTA1	58	**95**	66	*48*	48
GUPTA2	88	**153**	89	*79*	79
GUPTA3	624	**1248**	365	338	*336*
NASA1824	35	**39**	*29*	30	34
NASA2910	49	**80**	*48*	48	49
NASA4704	42	**43**	*36*	37	40
STRUCT4	156	**306**	146	177	*130*
VIBROBOX	63	**75**	*50*	53	66
AF23560	43	**170**	43	57	*35*
BIG	14	**15**	*13*	13	15
CIRCUIT_4	*3*	3	3	3	3
EPB3	15	*14*	14	15	**16**
GARON02	46	**55**	*42*	43	47
GRAHAM1	39	46	*37*	**48**	47
GRID48	91	**110**	*86*	87	86
LI	151	**370**	*136*	206	171
ONETONE1	17	*13*	18	**22**	19
PRE2	**15**	*13*	15	14	14
RMA10	**60**	59	54	*53*	56
SAYLR1	**7**	7	*6*	6	7
THERMAL	*29*	**40**	31	32	29
TWOTONE	**21**	*18*	20	21	19
VENKAT50	**57**	*52*	55	54	55
WANG1	*12*	**23**	14	15	14
WANG3	17	**33**	*16*	17	17
XENON2	70	*69*	70	71	**76**

Table 11: Average front order.

	METIS	SCOTCH	PORD	AMF	AMD
3DTUBE	1470.6	2415.6	3365.6	**4397.0**	*786.3*
M_T1	2564.3	1245.4	1545.3	**3448.6**	*792.0*
OILPAN	846.3	*384.9*	911.0	**1467.9**	549.4
SHIP_003	8950.1	10723.7	**17148.5**	14507.4	*2114.2*
THREAD	6041.0	8900.5	6433.0	**24455.3**	*3336.5*
BCSSTK34	132.6	165.5	213.5	**471.7**	*123.6*
BCSSTK38	537.2	**659.2**	508.6	494.5	*202.7*
CFD2	831.0	689.5	1326.8	**3632.6**	*520.1*
GUPTA1	85.3	**728.2**	*18.1*	193.7	189.6
GUPTA2	516.0	**2999.8**	562.8	500.1	*440.4*
GUPTA3	245.6	1485.1	465.7	**3275.6**	*150.9*
NASA1824	**86.5**	51.5	63.9	86.2	*24.9*
NASA2910	158.0	**177.4**	113.2	118.6	*54.2*
NASA4704	**189.5**	113.5	133.6	171.2	*44.4*
STRUCT4	1236.8	1651.1	4275.5	**35314.5**	*948.3*
VIBROBOX	334.7	*185.8*	621.7	**754.6**	199.3
AF23560	819.1	*759.7*	1545.0	**11725.3**	1072.6
BIG	**29.2**	15.8	27.6	26.7	*13.2*
CIRCUIT_4	**1.1**	0.6	1.0	*0.6*	0.7
EPB3	21.0	12.6	**32.6**	25.2	*10.4*
GARON02	**280.9**	96.0	176.6	95.0	*85.8*
GRAHAM1	175.9	101.1	**203.5**	114.1	*23.0*
GRID48	**553.3**	252.0	301.1	511.6	*111.4*
LI	*2036.3*	41418.2	3123.3	**62500.4**	3076.5
ONETONE1	882.4	372.5	1592.8	**2046.6**	*308.0*
PRE2	925.2	*139.4*	2936.2	**3140.2**	762.1
RMA10	**369.8**	160.8	286.3	342.6	*81.6*
SAYLR1	8.7	7.1	10.7	**14.0**	*6.1*
THERMAL	43.3	*12.3*	**76.1**	44.4	26.9
TWOTONE	2329.4	*684.7*	2973.3	**6925.4**	2108.0
VENKAT50	346.1	150.2	294.1	**400.5**	*87.1*
WANG1	407.1	*179.7*	524.9	**860.9**	343.7
WANG3	1951.4	*1267.3*	2375.9	**2547.3**	2068.1
XENON2	930.8	*832.4*	2913.1	**6072.7**	2614.0

Table 12: Average variance of the order of matrices of brother nodes.

Appendix C: Details on the memory usage

	METIS	SCOTCH	PORD	AMF	AMD
3DTUBE	105.66	*67.20*	120.69	**167.26**	146.30
M_T1	**164.31**	*107.01*	134.03	151.97	144.53
OILPAN	41.63	*31.77*	39.35	**48.32**	42.02
SHIP_003	485.22	*249.64*	590.01	**656.21**	496.76
THREAD	135.56	*92.72*	152.25	**347.46**	144.84
BCSSTK34	0.25	*0.12*	0.21	**0.25**	0.20
BCSSTK38	4.10	*2.53*	4.06	**4.27**	3.45
CFD2	199.66	*165.85*	216.34	**400.99**	366.38
GUPTA1	102.15	95.35	**129.15**	72.34	*69.97*
GUPTA2	423.81	391.32	**510.79**	337.78	*327.00*
GUPTA3	144.78	*10.36*	**287.47**	231.41	236.74
NASA1824	**0.29**	0.26	*0.24*	0.26	0.25
NASA2910	**0.94**	*0.62*	0.91	0.93	0.89
NASA4704	**1.22**	*0.97*	1.05	1.17	1.14
STRUCT4	13.49	*5.38*	15.99	**53.41**	16.86
VIBROBOX	10.86	*9.87*	10.39	**12.96**	10.98
AF23560	24.13	*17.75*	27.45	**112.11**	23.22
BIG	0.91	*0.84*	0.87	0.97	**1.04**
CIRCUIT_4	**0.50**	0.34	0.46	0.33	*0.32*
EPB3	*7.38*	7.40	7.71	9.79	**11.28**
GARON02	4.19	*3.28*	3.91	4.01	**4.58**
GRAHAM1	**3.40**	*2.10*	3.27	2.85	2.50
GRID48	**35.71**	*24.17*	30.84	33.36	31.23
LI	*78.31*	91.29	93.37	**484.80**	237.14
ONETONE1	28.77	*12.61*	44.09	**60.67**	25.07
PRE2	822.67	*286.90*	**1854.25**	1267.99	623.00
RMA10	**20.55**	*13.15*	18.20	19.04	16.63
SAYLR1	**0.0043**	*0.0032*	0.0034	0.0033	0.0035
THERMAL	0.46	*0.39*	0.65	**0.80**	0.51
TWOTONE	242.16	*71.85*	272.93	**437.84**	214.58
VENKAT50	25.18	*17.65*	23.67	**25.77**	23.38
WANG1	*0.66*	0.71	0.87	**1.15**	0.87
WANG3	23.87	*20.61*	26.81	32.15	**33.57**
XENON2	271.04	*203.04*	348.95	**507.88**	446.99

Table 13: Stack memory traffic (millions of entries).

	METIS	SCOTCH	PORD	AMF	AMD
3DTUBE	1.98	2.43	*1.84*	4.05	**5.48**
M_T1	**3.90**	3.51	*1.33*	1.58	3.66
OILPAN	0.76	0.77	*0.51*	0.58	**0.83**
SHIP_003	6.35	**13.45**	6.43	*4.93*	8.57
THREAD	10.54	**12.08**	2.93	*2.19*	5.30
BCSSTK34	**0.03**	0.02	0.02	*0.01*	0.01
BCSSTK38	0.11	**0.27**	*0.10*	0.14	0.13
CFD2	7.03	6.01	*2.25*	6.02	**8.11**
GUPTA1	5.16	**22.84**	7.80	*1.73*	3.95
GUPTA2	23.59	**108.67**	17.35	*5.05*	19.35
GUPTA3	21.42	*1.60*	**43.59**	15.38	13.96
NASA1824	0.019	**0.035**	0.015	*0.015*	0.018
NASA2910	0.042	0.043	*0.028*	0.036	**0.049**
NASA4704	0.065	**0.095**	*0.037*	0.042	0.059
STRUCT4	*0.42*	0.55	0.46	0.59	**1.36**
VIBROBOX	0.63	**0.83**	*0.35*	0.37	0.60
AF23560	0.51	**1.11**	*0.21*	0.73	0.27
BIG	**0.022**	0.021	*0.013*	0.014	0.016
CIRCUIT_4	0.026	**0.088**	0.049	0.023	*0.016*
EPB3	0.12	0.24	*0.069*	0.13	**0.27**
GARON02	**0.14**	0.12	*0.059*	0.074	0.088
GRAHAM1	**0.094**	0.082	*0.052*	0.052	0.067
GRID48	0.70	**0.76**	0.54	*0.41*	0.54
LI	2.71	4.01	2.50	*2.12*	**8.46**
ONETONE1	**0.90**	0.52	0.45	*0.40*	0.58
PRE2	*12.51*	12.92	17.45	16.99	**37.05**
RMA10	0.24	0.23	**0.24**	*0.12*	0.19
SAYLR1	**0.0006**	0.0005	0.0005	*0.0003*	0.0003
THERMAL	*0.0078*	0.0093	0.0236	**0.0268**	0.0081
TWOTONE	5.11	**7.03**	*2.69*	3.55	4.32
VENKAT50	0.26	**0.36**	0.25	*0.23*	0.31
WANG1	0.040	**0.075**	0.033	*0.031*	0.056
WANG3	1.07	1.59	*0.69*	0.89	**1.69**
XENON2	6.87	8.85	*3.69*	6.71	**11.07**

Table 14: Average size of the stack (millions of entries).

	METIS	SCOTCH	PORD	AMF	AMD
3DTUBE	*5.31*	6.19	5.49	14.65	**19.80**
M_T1	7.00	6.92	*3.70*	6.09	**9.07**
OILPAN	1.92	1.94	*1.27*	1.88	**2.17**
SHIP_003	25.09	31.15	22.89	*20.77*	**32.02**
THREAD	**27.30**	26.52	*10.07*	11.54	18.18
BCSSTK34	**0.07**	0.05	0.05	*0.03*	0.04
BCSSTK38	0.31	**0.55**	*0.26*	0.27	0.34
CFD2	15.60	13.04	*6.96*	16.43	**28.52**
GUPTA1	15.04	**69.34**	35.10	*10.28*	16.77
GUPTA2	59.81	**289.67**	78.13	*33.61*	52.09
GUPTA3	44.59	*27.37*	**95.64**	34.04	34.72
NASA1824	0.057	**0.089**	*0.036*	0.039	0.054
NASA2910	0.11	**0.12**	*0.075*	0.088	0.11
NASA4704	0.18	**0.24**	*0.12*	0.13	0.17
STRUCT4	*1.19*	1.40	1.33	2.40	**4.54**
VIBROBOX	2.07	**2.46**	*1.27*	1.50	2.28
AF23560	1.54	**2.28**	*0.49*	1.90	0.73
BIG	**0.052**	0.047	*0.030*	0.040	0.049
CIRCUIT_4	0.09	**0.23**	0.17	0.09	*0.06*
EPB3	0.27	0.54	*0.24*	0.46	**0.75**
GARON02	**0.29**	0.27	*0.15*	0.24	0.22
GRAHAM1	**0.20**	0.17	*0.15*	0.19	0.19
GRID48	1.52	1.63	1.47	*1.24*	**1.67**
LI	12.06	**34.54**	*9.44*	9.53	32.34
ONETONE1	**2.73**	1.99	1.46	*1.21*	2.02
PRE2	*36.56*	37.12	55.01	84.30	**153.57**
RMA10	**0.52**	0.47	0.38	*0.35*	0.40
SAYLR1	0.0016	**0.0018**	0.0017	*0.0010*	0.0012
THERMAL	0.0234	0.0223	0.0479	**0.0662**	*0.0220*
TWOTONE	13.24	15.74	11.80	*11.63*	**17.59**
VENKAT50	*0.54*	0.80	0.63	0.73	**1.01**
WANG1	*0.12*	0.21	0.14	0.13	**0.23**
WANG3	3.28	4.49	*2.75*	3.62	**6.14**
XENON2	14.92	20.21	*13.15*	23.82	**37.82**

Table 15: Peak of the stack (millions of entries).

	METIS	SCOTCH	PORD	AMF	AMD
3DTUBE	2.2 / 2.2	**17.9** / 4.3	*0* / 12.9	0 / **16.7**	0 / *0*
M_T1	2.7 / 11.9	**16.6** / 21.1	5.5 / **50.6**	*0* / 5.4	0 / *0*
OILPAN	*0* / *12.7*	**23.9** / 28.6	17.6 / **47.2**	8.4 / 33.9	19.1 / 13.5
SHIP_003	*0* / *8.4*	**26.0** / 26.3	8.8 / **42.7**	0 / 24.7	0 / 39.7
THREAD	**30.4** / 25.4	24.1 / 22.0	*0* / **28.2**	0 / 20.0	0 / *5.9*
BCSSTK34	7.4 / 23.5	26.7 / 26.7	**26.9** / 26.9	*0* / **57.7**	0 / *20.2*
BCSSTK38	**22.6** / **31.9**	17.6 / 17.7	16.0 / 10.9	14.2 / *9.8*	*0* / 19.5
CFD2	**28.7** / **28.7**	15.4 / 27.0	*0* / 14.3	5.6 / 21.7	0 / *0*
GUPTA1	*0* / **59.5**	0 / *0*	0 / 48.0	0 / 42.4	0 / 0
GUPTA2	**2.5** / 67.2	*0* / *0*	0 / **74.9**	0 / 13.3	0 / 14.1
GUPTA3	**39.3** / **59.3**	*0* / *0*	1.8 / 16.3	26.0 / 57.4	8.6 / 39.7
NASA1824	*0* / 21.6	**5.7** / **25.9**	0 / 13.0	0 / 5.4	0 / *0*
NASA2910	11.7 / 23.7	**33.0** / 27.5	*0* / **30.2**	0 / *8.0*	0 / 11.5
NASA4704	*0* / *7.6*	**16.4** / 16.4	0 / **23.2**	0 / 20.4	0 / 23.2
STRUCT4	1.0 / 0.6	**15.6** / **15.6**	*0* / 8.5	0 / 11.1	0 / *0*
VIBROBOX	*0* / 8.0	**8.0** / 8.0	0 / **15.0**	0 / 6.3	0 / *0*
AF23560	*0* / *17.7*	28.8 / 23.1	1.6 / **63.6**	**32.2** / 39.8	1.6 / 59.9
BIG	**25.7** / 24.9	16.0 / *16.0*	*0* / 18.1	0 / 31.5	0 / **40.4**
CIRCUIT_4	*0* / **41.7**	0 / *21.5*	**3.0** / 41.3	0 / 38.2	0 / 41.4
EPB3	7.7 / 36.2	12.5 / **37.9**	**14.1** / 15.3	*0* / *9.6*	0 / 24.2
GARON02	**13.1** / 13.1	9.8 / 9.8	*0* / 31.7	0 / *5.9*	0 / **33.8**
GRAHAM1	**11.6** / 19.7	*0* / **25.5**	8.9 / *8.2*	0 / 12.7	0 / 18.1
GRID48	**8.1** / 19.5	4.3 / *4.3*	*0* / 4.4	0 / **38.3**	0 / 29.7
LI	*0* / *8.1*	**6.4** / 11.5	0 / 11.0	0 / **39.1**	0 / 14.9
ONETONE1	2.2 / *0.2*	**15.1** / 16.7	13.9 / 14.6	*0* / **35.8**	0 / 29.2
PRE2	6.3 / *0*	**9.4** / 9.0	*0* / 3.6	0 / **11.5**	0 / 1.2
RMA10	16.5 / 38.7	16.4 / *19.8*	**24.4** / 33.4	*0* / 51.1	17.6 / **71.2**
SAYLR1	16.5 / 5.0	*0* / 20.2	**29.3** / 27.4	0 / **38.1**	0 / *0.1*
THERMAL	*0* / 20.0	14.9 / 14.9	**73.5** / **73.5**	51.1 / *0*	0 / 0
TWOTONE	*0* / *0*	**14.0** / **28.3**	0 / 20.2	0 / 7.6	0 / 16.3
VENKAT50	*0* / 23.1	**7.3** / **26.2**	0 / 15.3	0 / 6.2	0 / *0*
WANG1	*0* / 10.3	**17.2** / **16.2**	0 / 9.3	0 / 11.4	0 / *0*
WANG3	*0* / *0*	**14.5** / 15.4	0 / **15.9**	0 / 7.0	0 / 1.6
XENON2	0.1 / 21.1	**24.7** / 26.1	*0* / **33.4**	0 / 22.3	0 / *0*

Table 16: Percentage of reduction of the peak of stack memory observed using Algorithm 1. On the left, with respect to `MUMPS`, on the right, with respect to a version of `MUMPS` where the mechanism used by `MUMPS` to put the largest node first has been switched off.

	METIS	SCOTCH	PORD	AMF	AMD
3DTUBE	7.9	**25.9**	2.0	*0*	*0*
M_T1	31.7	**33.9**	-0.5	*0*	*0*
OILPAN	8.6	**27.9**	24.5	*-16.6*	26.6
SHIP_003	9.7	**49.9**	32.9	*0.0*	*0.0*
THREAD	51.6	**70.4**	*-20.0*	0	0
BCSSTK34	34.1	**45.6**	25.5	*0*	*0*
BCSSTK38	35.4	**68.1**	*18.3*	37.5	20.0
CFD2	40.1	**45.9**	*0*	15.1	*0*
GUPTA1	**41.6**	*0*	*0*	*0*	10.2
GUPTA2	**63.1**	*0*	*0*	20.8	45.0
GUPTA3	**69.9**	*9.8*	11.2	52.5	39.8
NASA1824	11.5	**40.7**	0.05	*0*	*0*
NASA2910	10.8	0.277	9.2	**28.3**	*6.6*
NASA4704	21.5	**43.9**	*0*	4.2	10.7
STRUCT4	13.2	**31.6**	7.9	*0*	*0*
VIBROBOX	8.7	18.0	**20.4**	8.2	*0*
AF23560	11.8	**67.4**	13.7	61.0	*10.8*
BIG	**33.0**	16.4	4.5	*0.3*	0.4
CIRCUIT_4	4.8	**50.3**	26.7	22.1	*4.4*
EPB3	23.9	**30.1**	*-26.0*	0	0
GARON02	16.9	**25.6**	17.7	*0*	25.1
GRAHAM1	**34.1**	7.1	*-5.0*	0	0
GRID48	19.4	**29.6**	*0*	*0*	*0*
LI	**36.4**	*-1.3*	0.5	0	3.3
ONETONE1	**15.6**	14.5	*-1.5*	0	0
PRE2	35.1	**40.4**	1.3	25.6	*0*
RMA10	11.8	42.8	**59.8**	*0*	24.3
SAYLR1	**26.4**	*0*	17.9	*0*	8.4
THERMAL	*0*	15.6	**76.1**	66.0	*0*
TWOTONE	34.4	**51.5**	*0*	*0*	*0*
VENKAT50	13.9	**26.4**	1.5	7.9	*0*
WANG1	8.2	**38.4**	*-33.5*	7.5	-14.9
WANG3	19.2	**28.8**	2.7	*0*	3.2
XENON2	32.1	**44.7**	*0*	9.0	*0*

Table 17: Percentage of reduction of the average stack size using Algorithm 1.

	METIS	SCOTCH	PORD	AMF	AMD
3DTUBE	0.3	**1.5**	*0*	0	0
OILPAN	0.2	**1.2**	*0*	0	0
SHIP_003	*0*	**1.4**	0	0	0
THREAD	**6.8**	2.5	*0*	0	0
BCSSTK34	*0*	7.6	**8.2**	0	0
BCSSTK38	**1.2**	*0*	0	0	0
CFD2	**3.0**	1.5	*0*	0	0
GUPTA2	**6.2**	*0*	0	0	0
GUPTA3	**30.7**	*0*	1.4	3.2	6.8
NASA2910	2.2	**4.4**	*0*	0	0
NASA4704	*0*	**2.0**	0	0	0
STRUCT4	0.2	**2.6**	*0*	0	0
BIG	**0.7**	0.2	*0*	0	0
CIRCUIT_4	*0.7*	1.6	**4.6**	1.9	1.8
ONETONE1	*0*	**1.6**	0	0	0
THERMAL	**0.1**	*0*	0	0	0

Table 18: Percentage of reduction of the peak of total memory observed using Algorithm 2. Matrices with no gain at all have been removed from the table.

	METIS	SCOTCH	PORD	AMF	AMD
3DTUBE	5.20	*5.08*	5.49	14.65	**19.80**
M_T1	6.82	5.78	*3.50*	6.09	**9.07**
OILPAN	**1.92**	1.47	*1.04*	1.72	1.76
SHIP_003	**25.09**	23.06	20.86	*20.77*	32.02
THREAD	19.02	**20.14**	*10.07*	11.54	18.18
BCSSTK34	**0.06**	0.04	0.03	*0.03*	0.04
BCSSTK38	0.24	**0.45**	*0.22*	0.23	0.34
CFD2	11.12	11.04	*6.96*	15.51	**28.52**
GUPTA1	15.04	**69.34**	35.10	*10.28*	16.77
GUPTA2	58.33	**289.67**	78.13	*33.61*	52.09
GUPTA3	27.06	27.37	**93.96**	*25.21*	31.72
NASA1824	0.057	**0.084**	*0.036*	0.039	0.054
NASA2910	0.094	0.077	*0.075*	0.088	**0.113**
NASA4704	0.179	**0.202**	*0.120*	0.133	0.174
STRUCT4	*1.17*	1.18	1.33	2.40	**4.54**
VIBROBOX	2.07	2.26	*1.27*	1.50	**2.28**
AF23560	1.54	**1.62**	*0.48*	1.29	0.72
BIG	0.038	0.039	*0.030*	0.040	**0.049**
CIRCUIT_4	0.09	**0.23**	0.16	0.09	*0.06*
EPB3	0.25	0.47	*0.21*	0.46	**0.75**
GARON02	**0.25**	0.24	*0.15*	0.24	0.22
GRAHAM1	0.18	0.17	*0.13*	0.19	**0.19**
GRID48	1.40	1.56	1.47	*1.25*	**1.67**
LI	12.06	**32.34**	*9.44*	9.53	32.34
ONETONE1	**2.67**	1.69	1.26	*1.21*	2.02
PRE2	34.25	*33.64*	55.01	84.30	**153.57**
RMA10	**0.43**	0.40	*0.29*	0.35	0.33
SAYLR1	0.001	**0.002**	0.001	*0.001*	0.001
THERMAL	0.023	0.019	*0.013*	**0.032**	0.022
TWOTONE	13.24	13.54	11.80	*11.63*	**17.59**
VENKAT50	*0.54*	**0.74**	0.63	0.73	1.01
WANG1	0.12	0.18	0.14	*0.13*	**0.23**
WANG3	3.28	3.84	*2.75*	3.62	**6.14**
XENON2	14.89	15.21	*13.14*	23.82	**37.82**

Table 19: Peak of the stack after applying Algorithm 1 (millions of entries).

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

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