On the Memory Usage of a Parallel Multifrontal Solver

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

Laboratoire de l'Informatique du Parallélisme

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

On the Memory Usage of a Parallel Multifrontal Solver

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

November 2002

Research Report N° 2002-42

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

On the Memory Usage of a Parallel Multifrontal Solver

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

November 2002

Abstract

We are interested in the memory usage of sparse direct solvers. We particularly focus on the parallel multifrontal scheme. In the multifrontal approach two kinds of memory can be distinguished: a static one which corresponds to the result of the factorization process (ie, the factors), and a dynamic or active one, usually handled by a stack mechanism, which corresponds to the working space of the factorization process. For some problems the stack size may be as large as and even greater than the final factors. The size of the stack depends on the assembly tree and on how the computation is distributed. We present an extensive study of the impact of state-of-the-art sparse matrix reordering techniques on the assembly tree and on the memory occupation of the `MUMPS` solver in both sequential and parallel executions. The main observation of this study is that the stack of parallel multifrontal solvers does not scale well if a dynamic scheduling strategy based only on the balance of the workload is used.

Keywords: Sparse matrices, multifrontal method, assembly tree, reordering techniques, memory.

Résumé

Nous nous intéressons dans ce rapport à l'étude de l'occupation mémoire d'un solveur creux direct. Dans la méthode multifrontale, la mémoire est divisée en deux parties : une partie statique qui correspond au résultat du processus de factorisation (i.e. les facteurs), et une partie dynamique, gérée avec un mécanisme de pile, qui correspond à l'espace de travail du processus de factorisation. La taille de la mémoire dynamique dépend fortement de la topologie de l'arbre d'assemblage guidant le processus de factorisation ainsi que de la distribution des calculs. Pour certains problèmes, elle peut être du même ordre de grandeur, voire plus grande que la taille des facteurs. Nous présentons une étude de l'impact des techniques modernes de renumérotation sur la topologie de l'arbre d'assemblage ainsi que sur l'occupation mémoire du solveur `MUMPS` pour les cas séquentiels et parallèles. Le principal résultat de cette étude est que la mémoire dynamique pour les exécutions parallèles n'a pas une bonne scalabilité dans le cas de l'utilisation d'une stratégie d'ordonnancement basée uniquement sur l'équilibrage de charge.

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

1 Introduction

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: sometimes, large problems fail to be solved because of a lack of memory on the processors.
In this paper, we present a study of the memory usage of multifrontal solvers. Two types of memory can be distinguished in the process of solving sparse linear systems: a static memory needed to store the final factors of the sparse matrix; and an additional dynamic memory, also called active memory, needed to store temporary values used by the computation. In the case of multifrontal methods, this is handled by a stack mechanism. The size of the active memory can be large, and is sometimes larger than the factors.
Reordering (i.e., 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 significant 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 assembly trees resulting from the reordered matrix. Thus, we present an extensive study of the assembly tree shapes resulting from various combinations of sparse matrices and reorderings.
The active memory size also depends on the way the assembly tree is traversed during the factorization process and how the computation is distributed on the different processors. We experimentally study the memory usage of the parallel multifrontal solver `MUMPS`.
In Section 2, we recall some general mechanisms of the multifrontal method. Then we give in Section 3 a description of the reordering techniques used for our study. In Sections 4 and 5, we study the impact of these reordering techniques on both the shape of the assembly tree and on the evolution of the dynamic memory, respectively. After that, we study in Section 6 the influence of the reordering methods on the memory balance and occupation for parallel executions, then, we conclude.

2 The multifrontal method

Like other direct methods, the multifrontal method [9, 10] is based on the elimination tree [16], 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 [5]. We recall that a supernode is a contiguous range of columns (in the factor matrix) having the same nonzero structure.
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.
In the multifrontal approach, the factorization of the matrix is done by performing a succession of partial factorizations of small dense matrices called *frontal matrices*, and associated to each node of the tree. The order of the frontal matrix is given by the number of non-zeros below the diagonal in the first column of the supernode associated with the tree node. 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 complete, the contribution block is passed to the father node. When contributions from all children are available on the father, they can be assembled (i.e.

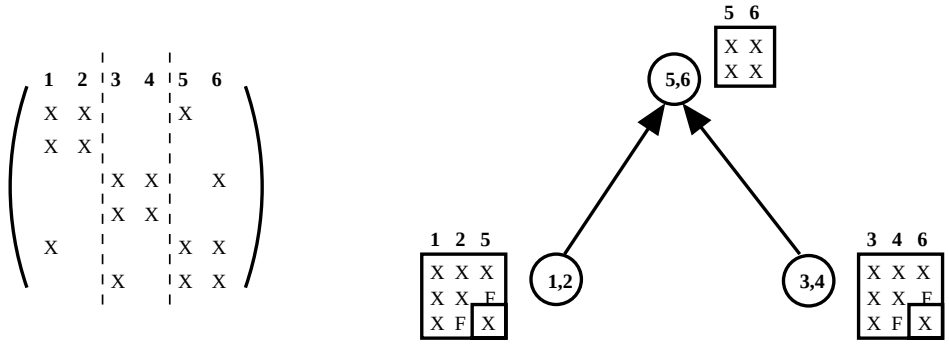


Figure 1: A matrix and the associated assembly tree.

summed with the values contained in the frontal matrix of the father). The elimination algorithm is a postorder traversal (we do not process father nodes before their children) [20] of the assembly tree.

The algorithm uses three areas of storage in a contiguous memory space, one for the factors, one to stack the contribution blocks, and another one for the current frontal matrix [2]. During the tree traversal, the memory space required by the factors always grows while the stack memory (containing the contribution blocks) varies depending on the operations made: when the partial factorization of a frontal matrix is processed, a contribution block is stacked which increases the size of the stack; in opposition, when the frontal matrix is formed and assembled, the contribution blocks of the children nodes are pushed out of the stack and its size decreases. Concerning stack memory evolution, it is very dependent on the assembly tree topology. Figure 2 gives an example of two trees and illustrates the influence of the tree topology on the stack memory. Indeed if we consider the deep tree (on the left), we have to store only two contribution blocks simultaneously whereas for the wide tree (on the right) we have five contribution blocks to store before processing the right-most child of the root node (see figure 2). This example illustrates the impact of the topology of the assembly tree on the stack memory.

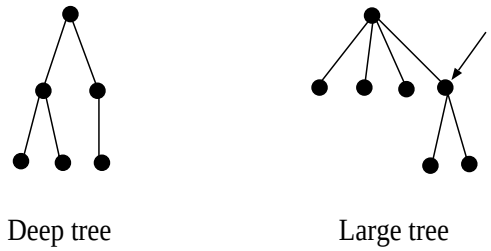


Figure 2: Importance of the tree topology for stack memory.

Note that in our description all the contribution blocks for children nodes in the sequential case are assembled at once. Another approach could be to perform the assembly of a contribution block on the fly, each time it is computed. 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 father is unpredictable when there is pivoting.

In the rest of the paper we only distinguish between two areas of storage: the factors, and the stack,

where the term stack includes the storage for the current frontal matrix.

3 Reordering techniques

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.
Here we only consider symmetric reordering techniques which 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 [8]).
Two popular schemes for symmetric reordering are bottom-up heuristics such as the minimum degree (AMD [1], MMD [15]) or minimum fill (MMF [17, 21]) and global or top-down heuristics based on partitioning the graph of the matrix, such as nested dissection [12]. A new class of algorithms has been developed recently that hybridize 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, top-down and hybrid heuristics:

- AMD: the Approximate Minimum Degree [1];
- AMF: the Approximate Minimum Fill, as implemented in `MUMPS`;
- PORD: a tight coupling of bottom-up and top-down sparse reordering methods [22];
- METIS: we use here the routine `METIS_NODEND` from the METIS package [14] which is an hybrid approach based on multilevel nested dissection and multiple minimum degree;
- SCOTCH: we use a modified version of `SCOTCH` [18] provided by the author that couples nested dissection and (halo) Approximate Minimum Fill (HAMF), in a way very similar to [19]. The switch to HAMF is done when the size of the subgraph obtained is 120.

In the following, we simply use the terms AMD, AMF, METIS, SCOTCH and PORD to refer to these heuristics. We should note that for AMD, AMF, SCOTCH and PORD, the assembly tree is returned directly from the reordering algorithm, while for METIS, only the permutation is returned and `MUMPS` rebuilds an assembly tree based from that permutation.
Finally, note that we had initially considered a pure nested dissection algorithm [12], but this one was competitive only in a few cases, and only for extremely regular problems, so that we decided to discard it.
Figures 4 gives the ratio between the peak of the stack and the final size of the factors in the sequential case. (Note that the final size of factors for every test problem and every reordering technique is given in Figure 3.) The matrices are from Table 1 and are extracted from either the Rutherford-Boeing collection [7], the collection from University of Florida[1] or the PARASOL collection[2]. We can see that the peak of the stack can be significant compared to the size of factors (the ratio is near to 1). Furthermore, for certain problems like the matrix GUPTA3, the peak of the stack is larger than the size of the factors. This illustrates the fact that the stack memory must be well managed for both sequential and parallel executions.

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

Matrix	Order	NZ	Type	Description
BMWCRA_1	148770	5396386	symmetric	Automotive crankshaft model with nearly 150000 TETRA elements (MSC-CRANKSHAFT-150K)
SHIP_003	121728	4103881	symmetric	Ship structure from production run
GRID	109744	1174048	symmetric	11-point discretization of the Laplacian on 3D grids (152*38*19)
GUPTA3	16783	4670105	symmetric	Linear programming matrix (A*A'), Anshul Gupta
MSDOOR	415863	10328399	symmetric	Medium size door
PRE2	659033	5959282	unsymmetric	AT&T,harmonic balance method, large example
RMA10	46835	2374001	unsymmetric	3D CFD model, Charleston harbor. Steve Bova, US Army Eng., WES
TWOTONE	120750	1224224	unsymmetric	AT&T,harmonic balance method, two-tone. More off-diag nz than one-tone
XENON2	157464	3866688	unsymmetric	Complex zeolite,sodalite crystals. D Ronis

Table 1: Description of the test problems.

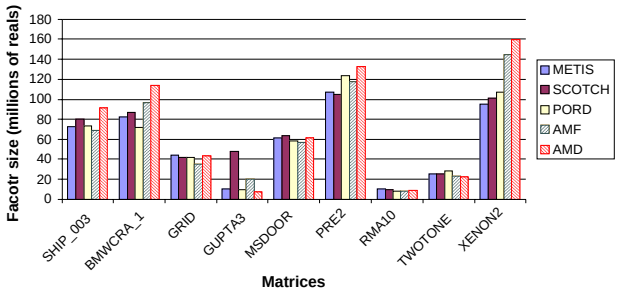
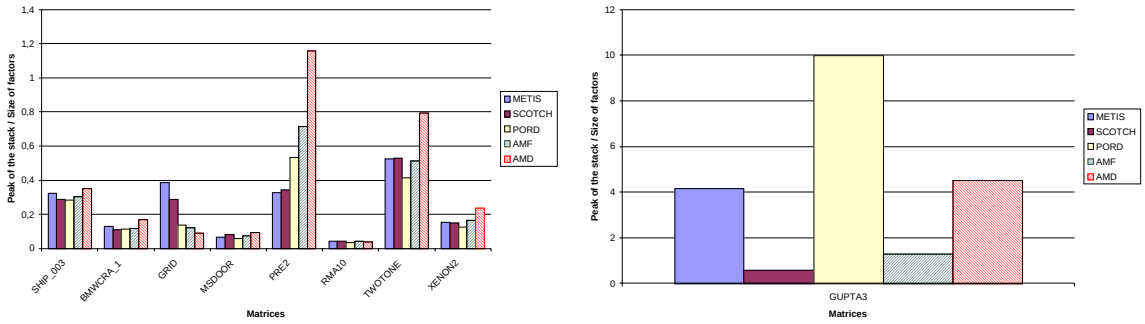


Figure 3: Size of the factors (millions of reals).



(a) (b)

Figure 4: Ratio between the size of the stack and the size of the factors.

4 Impact of reordering techniques on the assembly tree

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 from Table 1 and the reordering techniques METIS, SCOTCH, PORD, AMF and AMD introduced in Section 3.

We use the software package `MUMPS` (MUltifrontal Massively Parallel Solver) [4, 3], which implements parallel multifrontal solvers with threshold partial pivoting for both **LU** and $\mathbf{LDL}^T$ factorizations. For our purpose, we first experiment with the sequential version, and the tree is processed using a depth first search traversal. We have instrumented the code to obtain statistics on both the assembly

tree and the memory and be able to understand in better detail the evolution of the memory usage with time. Tests of `MUMPS` have been made on the IBM SP system of the CINES [3] which is composed of 29 nodes of 16 processors. Each node is equipped with 16 GB of memory shared among its 16 Power3+ (375MHz) processors.

The general shape of the assembly tree (width and depth) was estimated by the number of nodes (Table 2), and the percentage of leaves in the tree (Table 3). Regularity of the shape was estimated by the standard deviation of the depth of the leaves (Table 5), the maximum depth of a leaf (Table 4) and the average number of sons (Table 6). Because the stack size is influenced by the size of frontal matrices, we report for each tree the maximum size of a frontal matrix (Table 7), and the average front size (Table 8). In these tables, the largest value of a row is in bold, while the smallest value in italics.

General shape

We observe in Table 2 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 Table 3 that usually, SCOTCH and METIS generate trees with a large percentage of leaves when compared to the trees generated by AMF, AMD or PORD. Effectively, the trees generated by METIS and SCOTCH are rather wide (because of the global partitioning performed at the top), while the trees generated by AMD, AMF and PORD tend to be deeper.

	METIS	SCOTCH	PORD	AMF	AMD
SHIP_003	7474	*4294*	7798	**8253**	7634
BMWCRA_1	8767	*2833*	9268	**9902**	8320
GRID	24953	*10218*	24081	**29680**	28224
GUPTA3	413	*26*	1790	1300	**1898**
MSDOOR	31611	*28511*	32843	**33401**	31335
PRE2	204359	*169920*	**215403**	205297	195812
RMA10	4608	*3465*	5109	**5325**	4524
TWOTONE	35718	*27904*	41309	**41794**	39460
XENON2	18990	*13130*	**20455**	20386	19043

Table 2: Number of nodes in the tree.

	METIS	SCOTCH	PORD	AMF	AMD
SHIP_003	43.8	**59.4**	43.5	*38.7*	43.0
BMWCRA_1	39.6	**47.7**	38.1	*33.7*	38.0
GRID	58.8	**67.9**	*49.0*	49.1	51.2
GUPTA3	**95.9**	26.9	23.4	33.8	*21.3*
MSDOOR	55.0	**66.8**	53.9	*51.3*	54.2
PRE2	69.1	68.9	**74.0**	65.5	*61.0*
RMA10	**43.2**	42.9	42.7	41.8	*39.6*
TWOTONE	68.2	68.8	**72.8**	71.8	*67.5*
XENON2	48.3	**70.4**	49.2	45.3	*42.2*

Table 3: Percentage of leaves in the tree.

Regularity

On the other hand, according to Tables 5 and 4, 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 6, we can see that PORD, AMD and AMF have trees where the average number of sons for a node is smaller than for the METIS and SCOTCH cases; this also illustrates that the tree is not very wide (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.

[3] Centre Informatique National de l'Enseignement Supérieur

	METIS	SCOTCH	PORD	AMF	AMD
SHIP_003	29	*14*	75	**122**	32
BMWCRA_1	21	*14*	54	**100**	34
GRID	26	*14*	49	**188**	53
GUPTA3	*7*	8	9	**41**	13
MSDOOR	26	*17*	53	**80**	35
PRE2	56	*18*	**115**	99	42
RMA10	*26*	40	43	**208**	165
TWOTONE	55	*15*	77	**193**	47
XENON2	24	*16*	54	**65**	26

Table 4: Maximum depth for a node.

	METIS	SCOTCH	PORD	AMF	AMD
SHIP_003	19.083	*3.239*	321.386	**508.799**	25.375
BMWCRA_1	4.368	*1.139*	95.793	**374.991**	20.896
GRID	9.304	*0.953*	98.898	**2852.83**	149.809
GUPTA3	3.154	3.837	*1.022*	**114.437**	5.733
MSDOOR	3.997	*1.49*	**70.444**	53.629	10.901
PRE2	110.948	*9.164*	**815.96**	265.048	21.521
RMA10	11.13	*7.414*	67.729	**2084.394**	1331.673
TWOTONE	54.308	*5.172*	294.802	**458.468**	80.65
XENON2	7.058	*1.672*	101.599	**144.729**	10.295

Table 5: Variance of the depth of leaves.

	METIS	SCOTCH	PORD	AMF	AMD
SHIP_003	1.78	**2.463**	1.77	*1.631*	1.754
BMWCRA_1	1.656	**1.911**	1.615	*1.507*	1.612
GRID	2.427	**3.111**	*1.959*	1.966	2.051
GUPTA3	**24.235**	1.316	1.305	1.509	*1.27*
MSDOOR	2.222	**3.014**	2.168	*2.054*	2.185
PRE2	3.233	3.215	**3.852**	2.902	*2.564*
RMA10	**1.76**	1.749	1.746	1.719	*1.654*
TWOTONE	3.144	3.207	**3.681**	3.54	*3.079*
XENON2	1.93	**3.369**	1.962	1.823	*1.726*

Table 6: Average number of sons.

Front size analysis

According to Tables 7 and 8, 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. Note that AMD generates trees with big variations of the front size.

	METIS	SCOTCH	PORD	AMF	AMD
SHIP_003	3456	*3156*	3426	3408	**4038**
BMWCRA_1	2343	*2040*	2076	2496	**2835**
GRID	**2754**	2343	1721	1536	*1328*
GUPTA3	*827*	**5058**	1643	3028	1030
MSDOOR	1372	**1624**	*1358*	1491	1610
PRE2	*4290*	4334	5794	6476	**7502**
RMA10	**466**	422	*378*	439	399
TWOTONE	2382	*2316*	2561	2588	**2684**
XENON2	*2554*	2623	2743	3663	**4501**

Table 7: Maximal frontal matrix order.

	METIS	SCOTCH	PORD	AMF	AMD
SHIP_003	**170**	153	165	163	*151*
BMWCRA_1	177	**287**	*172*	187	183
GRID	41	**68**	42	*36*	38
GUPTA3	624	**1248**	365	338	*336*
MSDOOR	**74**	*67*	73	72	71
PRE2	**15**	*13*	14	14	14
RMA10	**60**	59	54	*53*	56
TWOTONE	**23**	*18*	20	21	19
XENON2	70	*69*	70	71	**76**

Table 8: Average front order.

5 Impact of reordering techniques on the memory evolution for sequential executions

After studying the shape of the assembly tree, we now focus on the impact of the reordering techniques on the memory occupation in `MUMPS`. Before analyzing the results, we should mention that we use a tree traversal that aims at optimizing the dynamic memory usage. Such techniques are described in more details in the forthcoming technical report [13].

Tables 9, 10 and 11 present the total amount of stack memory, the average stack size, and the peak of stack, respectively. For all these quantities, we did not take integer storage into account, so that the unit used is always the number of real entries.

Memory traffic

Table 9 gives the total amount of stack memory for each reordering technique. The total amount of stack memory represents the sum of the sizes for the contribution blocks for all the nodes of the tree. We can observe that the total amount of stack memory 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 PORD and AMF leads to the biggest global amount of stack memory.

	METIS	SCOTCH	PORD	AMF	AMD
SHIP_003	509.66	*249.64*	590.01	**656.21**	496.76
BMWCRA_1	432.23	*286.21*	425.46	**709.76**	541.35
GRID	**252.53**	*147.23*	204.81	233.06	201.2
GUPTA3	144.83	*10.36*	**287.47**	231.41	236.74
MSDOOR	230.75	*175.43*	**247.32**	244.51	213.60
PRE2	1186.45	*280.019*	**1557.68**	1267.98	623.01
RMA10	**20.74**	*13.15*	18.2	19.03	16.63
TWOTONE	305.18	*71.85*	272.93	**437.84**	214.58
XENON2	274.28	*203.04*	348.95	**507.88**	446.99

Table 9: Total amount of stack memory (millions of reals).

Average stack size

Table 10 gives the average size of the stack during execution, defined as the average stack sizes for all variations observed in the traces generated. We see that for PORD the average size of stack memory is smaller than for the other reorderings. This is because PORD has deep trees (as shown in the previous sections) where we have not to store a lot of contribution blocks at the same time. Compared with reorderings that give deep trees like AMF or AMD, its tree has fewer nodes and smaller frontal matrices which explains the difference in terms of the average size between AMD, AMF and PORD. We can also observe that SCOTCH has a good average size because the number of nodes of its tree is smaller than the one for the other reordering techniques.

	METIS	SCOTCH	PORD	AMF	AMD
SHIP_003	5.76	6.74	*4.31*	7.01	**10.67**
BMWCRA_1	3.03	3.38	*2.16*	3.41	**6.11**
GRID	**3.18**	2.56	2.03	1.65	*1.21*
GUPTA3	15.2	*1.44*	**38.78**	7.31	8.41
MSDOOR	1.62	**2.42**	*1.15*	1.64	2.12
PRE2	8.9	*8.2*	23.52	12.64	**54.22**
RMA10	0.16	0.13	*0.097*	**0.17**	0.14
TWOTONE	*2.97*	3.41	3.24	4.63	**5.24**
XENON2	4.69	4.89	*3.9*	6.11	**11.27**

Table 10: Average size of the stack (millions of reals).

Peak stack size

Finally, Table 11 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 the wide trees given by SCOTCH or METIS. We can also observe that the peak of stack memory for AMD (which has a deep tree) tends to be greater than for other reorderings and particularly PORD and AMF (which also have deep trees). The first property of AMD's tree that can help us to explain this phenomenon is that we have observed that the nodes on the top of the tree for AMD are larger than the ones for other reorderings. When these large nodes start to be processed, the stack memory will contain large contribution blocks which will increase the size of the stack for the remaining subtrees.

	METIS	SCOTCH	PORD	AMF	AMD
SHIP_003	23.42	23.06	20.86	*20.77*	**32.02**
BMWCRA_1	10.69	9.53	*8.16*	11.26	**19.32**
GRID	**17.08**	11.91	5.83	4.17	*3.79*
GUPTA3	44.44	27.37	**93.96**	*25.21*	31.72
MSDOOR	4.12	5.22	*3.49*	4.18	**5.82**
PRE2	*34.95*	36.16	65.60	84.29	**153.57**
RMA10	**0.43**	0.39	*0.28*	0.34	0.33
TWOTONE	13.23	13.54	11.8	*11.63*	**17.59**
XENON2	14.39	15.21	*13.14*	23.82	**37.82**

Table 11: Peak of the stack (millions of reals).

The second property is that we saw that the tree of AMD is usually better balanced than those of AMF and PORD (see Table 5). Also, the global effect of the sorting of brother nodes in `MUMPS` (described at the beginning of this section) is that it puts the subtree with the biggest stack at the left-most position, so that the biggest subtree is processed first.

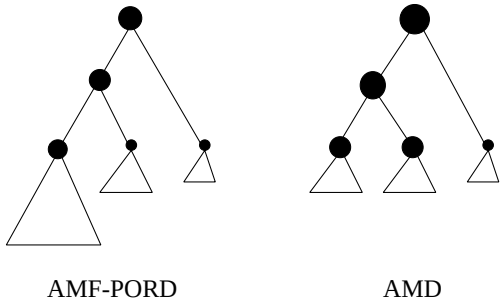


Figure 5: Structural difference between AMF's tree and AMD's tree.

Figure 5 illustrates the structural difference between AMD's and PORD-AMF's trees. 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 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.

Summary

To summarize this Section, we have seen that reordering techniques have a strong impact on both the shape of the assembly tree and the memory usage in the factorization. Table 12 summarizes

	Shape of the tree				Stack memory	
	Shape	**Balance**	**Number of nodes**	**Frontal matrices size**	**Peak of the stack**	**Average size of the stack**
METIS	wide	++	+	+	+	+
SCOTCH	wide	+++	–	++	+	+
PORD	deep	-	++	–	–	–
AMF	deep	–	++	-	-	-
AMD	deep	+	+	+	++	++

Table 12: Characteristics of the assembly tree and stack memory for different reordering techniques. The symbol "+" means a bigger value, the symbol "-" means a lower value.

the general observations made for the different reorderings on the assembly tree and on the stack. Concerning 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. In addition, METIS and SCOTCH give trees with bigger frontal matrices than the ones generated by other reorderings. From the memory point of view, we have seen that PORD and AMF are the reorderings that use the smallest stack size (peak and average). This should of course be related to the sizes of the factors, for which the following has been observed (see, for example, [13]): 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.

6 Memory usage for parallel executions

Because memory evolution depends on distribution of nodes of the assembly tree on processors, we first describe the current scheduling strategy used in `MUMPS`. Then, we show the memory behavior for parallel executions for different combination of matrices and ordering.

6.1 Scheduling strategy used in MUMPS

`MUMPS` use a combination of static and dynamic scheduling strategy. This is described in details in [4] and [3]. The computation is driven by the assembly tree and to each node is assigned one type of parallelism. Figure 6 summarizes the different types of parallelism available in `MUMPS`:

- The first type uses the intrinsic parallelism induced by the assembly tree: each branch of the tree can be treated in parallel. A type one node is statically assigned to one processor which processes it when processors assigned to children nodes have communicated the contribution blocks. Leave subtrees are a set of type 1 nodes all assigned to the same processor. Those are determined using a top-down algorithm [11] and a subtree-to-process mapping is used to balance the computational work of the subtrees onto the processors.

- The second type corresponds to a 1D parallelism of the frontal matrices. For some nodes in the assembly tree, the front is so big that is must be treated in parallel for an adequate granularity. The front is then distributed by blocks of rows. A *master* processor is chosen statically during the symbolic preprocessing step, all the others (*slaves*) are chosen dynamically

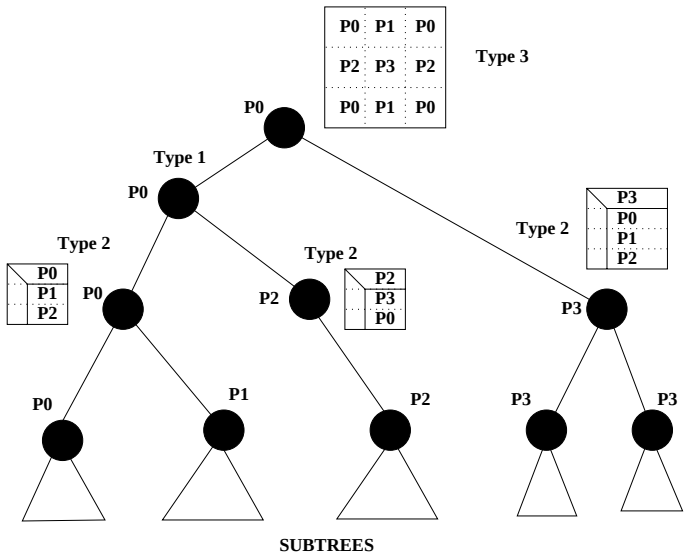


Figure 6: Example of distribution of a multifrontal assembly tree over four processors.

based on load balance considerations. The *master* processor is responsible for the eliminations of the fully summed pivot block. The master processor dynamically chooses its slave processors according to their load and assigns them new tasks. The load metric is the number of floating-point operations still to be done, where only the operations corresponding to the elimination process are taken into account (those are an order of magnitude larger than the operations for assembly).

- The third type of parallelism, which is a 2D parallelism, concerns the root node, which is processed by all processors using ScaLAPACK [6]: we use a 2D block cyclic distribution.

The choice of the type of parallelism depends on the position in the tree, and on the size of the frontal matrices. For the top of the tree the mapping of type 1 nodes and masters of type 2 nodes is static and only aims at balancing the memory of the corresponding factors. Usually, type 2 nodes are high in the assembly tree (fronts are bigger), and on large numbers of processors, about 80% of the floating-point operations are done in type 2 nodes.

6.2 Parallel Results

Tables 15 and 13 (resp. 16 and 14) shows the average stack peak, and the maximum of stack peak on 16 (resp. 32) processors for different matrices and reorderings.
We can observe that METIS and SCOTCH tend to have a more balanced peak of the stack among all the processors than other reorderings. This can be explained by the fact that these reordering techniques generate well-balanced trees where all the subtrees are approximatively of the same size. Concerning AMF and PORD, we can see that the stack memory is very unbalanced (particularly for AMF). This is due to the shape of AMF's and PORD's trees which are very unbalanced. Indeed, for unbalanced trees, the subtrees described in the previous section are not of the same size. As a result, some processors will begin to treat type 2 nodes where other ones are still processing the subtrees. Thus, these ones can be chosen as slaves which will cause an increase of their stack memory and an increase of the stack memory unbalance.

Concerning the scheduling strategy, only floating-point operations for the factorizations of frontal matrices are taken into account, the memory of the processors is not considered. We observe that, although the scheduling strategy gives good results from the point of view of load balancing, the memory load balancing is not perfect and the difference between the maximum peak and the average peak is usually significant.

	METIS	SCOTCH	PORD	AMF	AMD
SHIP_003	10.078	7.907	5.719	*5.033*	**11.014**
BMWCRA_1	6.655	7.137	*5.934*	7.969	**13.16**
GRID	4.123	**4.194**	3.384	*1.804*	3.184
GUPTA3	11.26	4.269	8.842	**16.368**	*3.459*
MSDOOR	**3.029**	2.716	1.685	*1.659*	2.562
PRE2	*10.162*	13.032	10.905	10.992	**23.18**
RMA10	0.404	**0.411**	0.407	*0.347*	0.36
TWOTONE	3.65	*2.758*	3.022	2.944	**4.104**
XENON2	5.093	5.173	*4.631*	7.107	**9.495**

Table 13: Max peak of the stack on 16 processors (millions of reals).

	METIS	SCOTCH	PORD	AMF	AMD
SHIP_003	5.484	4.296	2.992	*2.621*	**6.154**
BMWCRA_1	*3.681*	3.709	3.708	4.161	**6.691**
GRID	2.371	1.897	2.029	*1.088*	**2.465**
GUPTA3	10.936	*1.669*	7.89	**16.07**	3.256
MSDOOR	-	1.522	1.275	*1.19*	**1.74**
PRE2	7.83	13.45	10.376	*6.971*	**14.826**
RMA10	**0.404**	0.365	0.359	0.347	*0.313*
TWOTONE	3.233	2.35	2.244	*2.183*	**3.278**
XENON2	4.161	3.973	*3.628*	4.65	**8.908**

Table 14: Max peak of stack on 32 processors (millions of reals).

	METIS	SCOTCH	PORD	AMF	AMD
SHIP_003	6.698	6.477	4.283	*3.485*	**8.377**
BMWCRA_1	5.327	5.557	*3.826*	5.369	**8.774**
GRID	**3.504**	3.409	2.645	*1.38*	2.774
GUPTA3	4.297	3.266	3.068	**5.58**	*2.149*
MSDOOR	1.599	**1.75**	1.186	*1.158*	1.641
PRE2	7.042	*6.855*	8.15	8.438	**15.691**
RMA10	0.252	0.247	*0.203*	0.225	**0.271**
TWOTONE	2.213	2.093	2.093	*1.761*	**2.564**
XENON2	3.718	3.951	*3.289*	4.527	**7.405**

Table 15: Average peak of the stack on 16 processors (millions of reals).

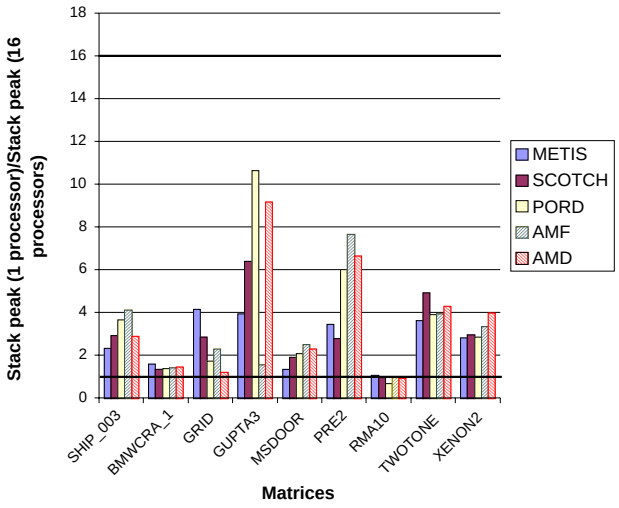
	METIS	SCOTCH	PORD	AMF	AMD
SHIP_003	2.994	2.835	2.245	*1.877*	**3.586**
BMWCRA_1	2.765	2.748	*2.428*	2.878	**5.088**
GRID	1.407	1.411	1.549	*0.779*	**1.84**
GUPTA3	1.715	*1.601*	1.86	**3.034**	2.138
MSDOOR	-	0.941	0.752	*0.728*	**1.009**
PRE2	4.328	*3.413*	4.306	5.24	**8.217**
RMA10	**0.221**	0.198	*0.169*	0.172	0.215
TWOTONE	1.512	1.409	1.354	*1.196*	**1.581**
XENON2	2.299	2.429	*2.048*	2.801	**4.308**

Table 16: Average peak of the stack on 32 processors (millions of reals).

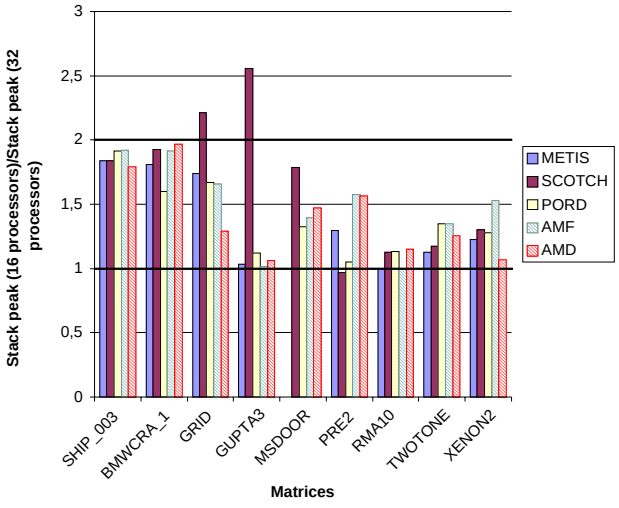
For matrices where the stack size is significant, if we compare the peak of stack memory measured in the sequential execution (Table 11) to the maximum peak of stack on 16 (Table13) and 32 processors (Table 14) , we don't observe any linear improvement in the memory usage: in parallel doubling the number of nodes, i.e. the memory size, doesn't mean we are able to treat a problem twice larger! Figures 7(a) and 7(b) illustrate this last point better. The first one gives the ratio between stack memory peak on 1 and 16 processors. We can see that we never reach the bold line that represents a perfect scalability; the best scalability observed is 11 but is in most cases between 2 and 4. This illustrates that the stack memory does not scale well. Figure 7(b) gives a comparison between the peak of the stack on 16 and 32 processors. Again, we observe that the memory usually does not decrease linearly with the number of processors (except for some cases like GUPTA3 with SCOTCH).

7 Conclusion

In this paper we have presented an experimental study of the memory usage for a parallel multi-frontal solver. This memory usage can be divided into two parts: a static one which corresponds to



(a) Ratio between stack memory peak on 1 and 16 processors.



(b) Ratio between stack memory peak on 16 and 32 processors.

Figure 7: Comparison of stack memory peak for different number of processors.

the factors, 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 results from the reordering technique applied. Reordering techniques have a significant impact on the memory usage: the amount of fill-in for the static memory (factors), but also 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 active memory usage of parallel multifrontal solvers, and in particular links between the reordering technique and the stack evolution. We also observed that the stack evolution not only depends on the shape of the tree but also on the distribution of the nodes on the processors during the factorization, while the size of the stack may be significant compared to the factor size.

The main result of our experiments is that the stack of parallel multifrontal solvers does not scale well from the memory point of view: if a problem doesn't fit in a p nodes parallel machine because the active memory is, say 50% larger than the main memory of a node, doubling the number of nodes does not guarantee that the problem will be solved !

In this parallel multifrontal scheme, the workload is not directly related to the memory occupation: workload is mainly done in the factorization part, whereas the dynamic memory usage is more related to the size of the contributions blocks needed in the assembly part. From the workload point of view, the assembly part is one order of magnitude less than the factorization part. So a scheduling strategy based only on the workload fails to balance the dynamic memory usage.

We are currently studying new scheduling strategies which are memory aware: these strategies try to balance the work on the processors taking into account the memory constraints. We are designing and experimenting such strategies within the dynamic scheduling part of `MUMPS`.

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] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. Technical Report LAPACK Working Note 95, CS-95-283, University of Tennessee, 1995.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] A. Geist and E. Ng. Task scheduling for parallel sparse Cholesky factorization. *Int J. Parallel Programming*, 18:291–314, 1989.

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

[13] A. Guermouche, J.-Y. L'Excellent, and G. Utard. Analysis and improvments of the memory usage in a multifrontal solver. Technical Report to appear, INRIA, 2002.

[14] 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.

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

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

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

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

[19] 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.

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

[21] 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.

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