

Impact of reordering on the memory of a multifrontal solver

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

*INRIA ReMaP Project, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon,
UMR CNRS-ENS Lyon-INRIA 5668, 46 allée d'Italie, 69364 Lyon Cedex 07, France*

Received 15 January 2003; received in revised form 23 May 2003; accepted 30 May 2003

Abstract

This paper is concerned with the memory usage of sparse direct solvers, which depends on the ordering of the unknowns and the scheduling of the computational tasks. We study the influence of state-of-the-art sparse matrix reordering techniques on the memory usage of a multifrontal solver. Concerning the scheduling, the memory usage depends on the tree traversal and how the tasks are assigned to the processors. We analyze the memory scalability when a dynamic scheduling strategy mainly based on the balance of the workload is used. Finally we give hints to improve the parallel memory behaviour.

© 2003 Elsevier B.V. All rights reserved.

Keywords: Sparse direct solvers; Parallel multifrontal method; Reordering; Assembly tree; Memory

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 multifrontal solvers two types of memory areas can be distinguished in the process of solving sparse linear systems: a static memory needed to store the final factors

* Corresponding author.

E-mail addresses: abdou.guermouche@ens-lyon.fr (A. Guermouche), jean-yves.l.excellent@ens-lyon.fr (J.-Y. L'Excellent), gil.utard@ens-lyon.fr (G. Utard).

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, the latter is handled by a stack mechanism. The size of the active memory can be large, and is sometimes larger than the factors.

The contribution of this paper is an extensive study of the memory usage of parallel multifrontal solvers. We show that the memory depends on both the reordering technique applied and the scheduling of the computational tasks and give hints on how to optimize the memory usage for sequential and parallel cases.

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 processors. We experimentally study the memory usage of the parallel multifrontal MUMPS.

This paper is organized as follows. In Section 2, we recall some general mechanisms of the multifrontal method. Then we give in Section 3 a description of the reordering techniques and test problems 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. Section 5.2 presents a variant of the algorithm by Liu [17] 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 depending on the reordering technique used. After that, we analyze in Section 6 the influence of reordering on the memory balance and consumption for parallel executions and study the main factors that limit the memory scalability. Finally, we draw conclusions.

2. The multifrontal method

Like other direct methods, the multifrontal method [9,10] is based on the elimination tree [18], 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 lower diagonal nonzero structure.

Fig. 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.

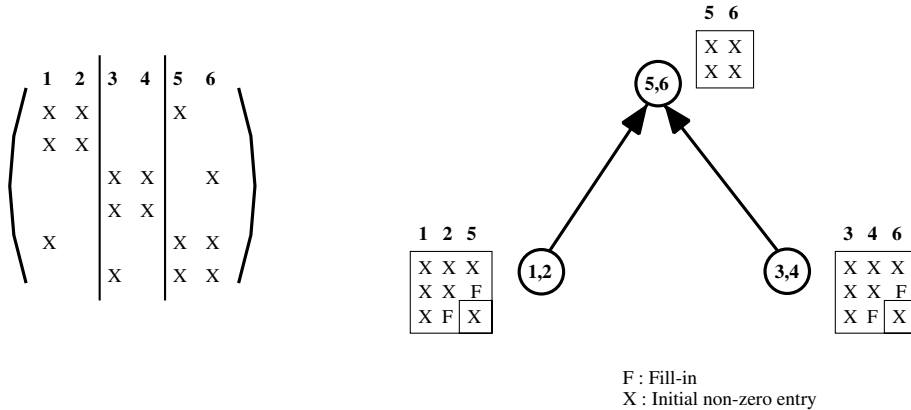


Fig. 1. A matrix with three supernodes ($\{1,2\}$, $\{3,4\}$, $\{5,6\}$) and the associated assembly tree.

In the multifrontal approach, the factorization of a matrix is done by performing a succession of partial factorizations of small dense matrices called *frontal matrices*, which are associated to the nodes of the tree. The order of a frontal matrix is given by the number of nonzeros below the diagonal in the first column of the supernode associated with the tree node. Each frontal matrix is divided into two parts: the *factor block*, also called *fully summed block*, which corresponds to the variables factorized when the elimination algorithm processes the frontal matrix; and the *contribution block* which corresponds to the variables updated when processing the frontal matrix. Once the partial factorization is complete, the contribution block is passed to the parent node. When contributions from all children are available on the parent, they can be assembled (i.e. summed with the values contained in the frontal matrix of the parent). The elimination algorithm is a postorder traversal (we do not process parent nodes before their children) of the assembly tree [22]. It 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; on the other hand, when the frontal matrix is formed and assembled, the contribution blocks of the children nodes are popped out of the stack and its size decreases. The stack memory is thus very dependent on the assembly tree topology.

To illustrate our observations, we give in Fig. 2 two examples of assembly trees. The corresponding memory evolution for the factors, the stack and the current frontal matrix is given in Fig. 3. First storage for the current frontal matrix is reserved (see “Allocation of 3” in Fig. 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

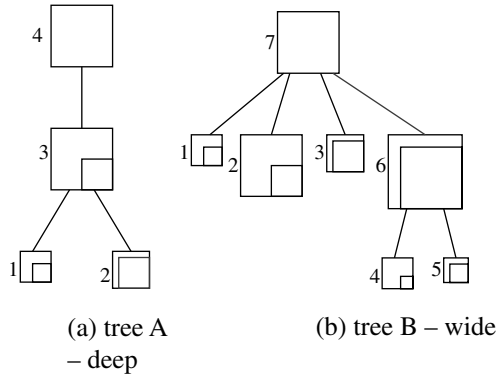


Fig. 2. Examples of assembly trees.

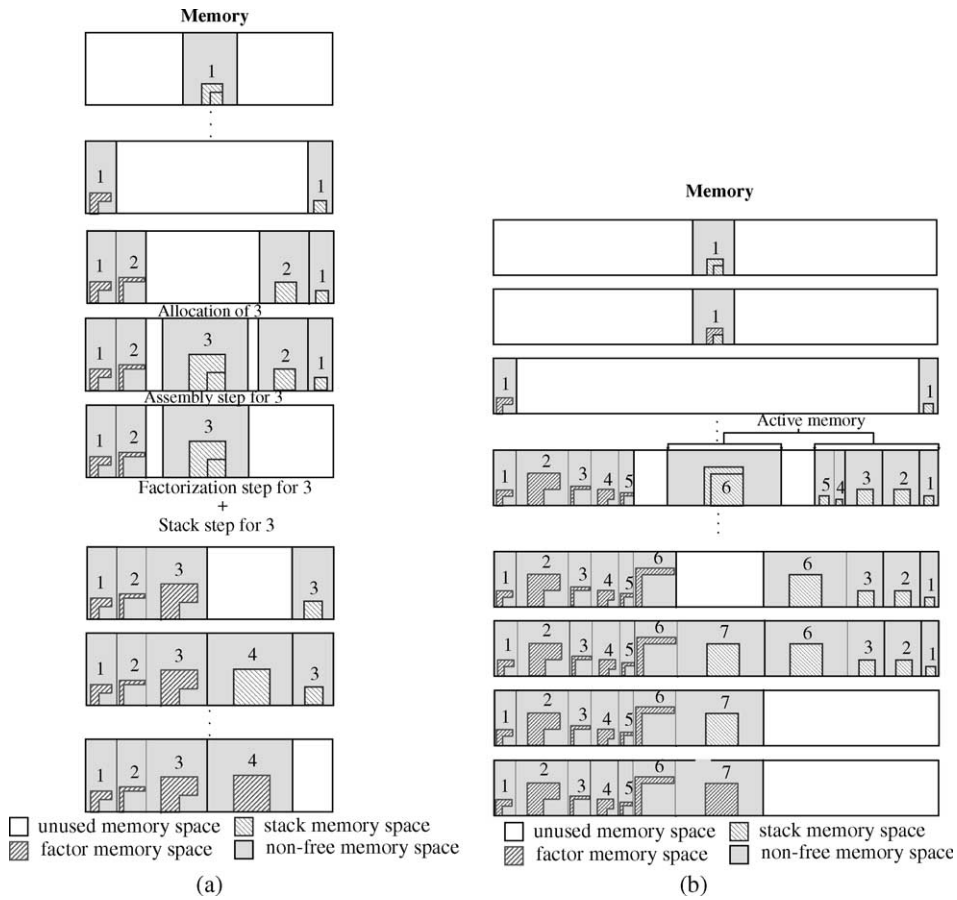


Fig. 3. Memory evolution for the trees given in Fig. 2. (a) Memory usage for tree A. (b) Memory usage for tree B.

figure and the contribution block is stacked (“Stack step for 3”). The process continues until the complete factorization of the root node(s). We can observe the different memory behaviours between the wide tree (Fig. 3(b)) and the deep tree (3(a)): the peak of active memory (see Fig. 3(b)) is larger for the wide tree.

Note that in our description all the contribution blocks for children nodes are assembled at once (in the sequential case). Another approach could be to preallocate the frontal matrix of the parent node and perform an assembly step each time a contribution block 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 parent is unpredictable when there is pivoting. Also, except for very wide trees, this is not necessarily a more efficient memory scheme because it implies storing several frontal matrices (each containing all the future contribution blocks of the subtree).

In the rest of the paper we only distinguish between two areas of storage: the factors, and the stack, where the 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 (while keeping numerical stability under control), 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 [16]) or minimum fill (MMF [19,23]) and global or top-down heuristics based on partitioning the graph of the matrix, such as nested dissection [12]. A class of algorithms has also been developed 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 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;¹
- PORD: a tight coupling of bottom-up and top-down sparse reordering methods [24];

¹ Available from <http://www.enseiht.fr/apo/MUMPS>.

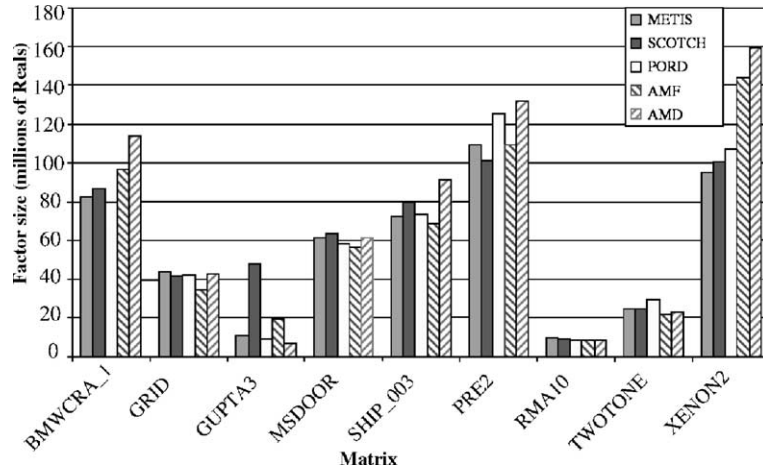


Fig. 4. Size of the factors (millions of reals).

- METIS: we use here the routine `METIS_NODEND` from the METIS package [15] which is an hybrid approach based on multilevel nested dissection and multiple minimum degree;
- SCOTCH: we use a modified version of SCOTCH [20] provided by the author that couples nested dissection and (halo) Approximate Minimum Fill (HAMF), in a way very similar to [21]. 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 must 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 is used to build an assembly tree based on this 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.

Fig. 5 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 Fig. 4.) The matrices are from Table 1 and are extracted from either the Rutherford–Boeing collection [7], the collection from University of Florida ² or the PARASOL collection. ³ We can see that the peak of the stack can be significant compared to the size of factors (the ratio is near to 1).

² Available from <http://www.cise.ufl.edu/~davis/sparse/>.

³ Available from <http://parallab.uib.no/parasol>.

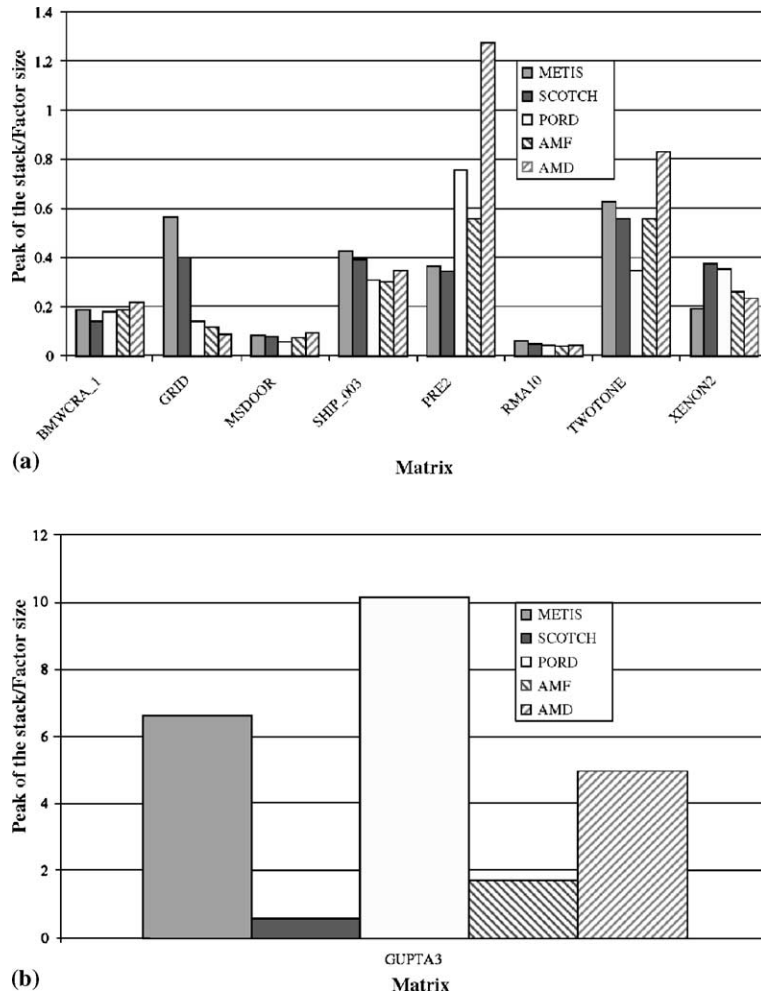


Fig. 5. Ratio between the size of the stack and the size of the factors.

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.

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.

Table 1
Description of the test problems

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

“SYM” stands for symmetric, “UNS” for unsymmetric.

We use the software package MUMPS (Multifrontal Massively Parallel Solver) [3,4], which implements parallel multifrontal solvers with threshold partial pivoting for both **LU** and **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 CINES⁴ which is composed of 29 nodes of 16 processors. Each node is equipped with 16 GB of memory shared among its 16 Power3+ (375 MHz) 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 children (Table 6). Because the stack size is influenced by the size of frontal matrices, we report for each tree the maximum and average sizes of a frontal matrix (Tables 7 and 8). In these tables, the largest value of a row is in bold, while the smallest value in italics.

As previously noticed AMD, AMF, PORD and SCOTCH directly return an assembly tree and METIS only provides a permutation that is used by MUMPS to build

⁴ Centre Informatique National de l'Enseignement Supérieur.

Table 2
Number of nodes in the tree

	METIS	SCOTCH	PORD	AMF	AMD
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
SHIP_003	7474	4294	7798	8253	7634
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 3
Percentage of leaves in the tree

	METIS	SCOTCH	PORD	AMF	AMD
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
SHIP_003	43.8	59.4	43.5	38.7	43.0
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 4
Maximum depth for a node

	METIS	SCOTCH	PORD	AMF	AMD
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
SHIP_003	29	14	75	122	32
PRE2	56	18	115	99	42
RMA10	26	40	43	208	165
TWOTONE	55	15	77	193	47
XENON2	24	16	54	65	26

an assembly tree. Therefore in the following, remarks concerning to METIS actually apply to the tree obtained by METIS followed by MUMPS symbolic factorization.

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

Table 5
Variance of the depth of leaves

	METIS	SCOTCH	PORD	AMF	AMD
BMWCR1_1	4.37	1.14	95.79	375.00	20.90
GRID	9.30	0.95	98.90	2852.83	149.81
GUPTA3	3.15	3.84	1.02	114.48	5.73
MSDOOR	3.99	1.49	70.44	53.63	10.90
SHIP_003	19.08	3.24	321.38	508.80	25.37
PRE2	110.95	9.16	815.96	265.05	21.52
RMA10	11.13	7.41	67.73	2084.39	1331.67
TWOTONE	54.31	5.17	294.80	458.47	80.65
XENON2	7.06	1.67	101.60	144.73	10.29

Table 6
Average number of children

	METIS	SCOTCH	PORD	AMF	AMD
BMWCR1_1	1.66	1.91	1.62	1.51	1.61
GRID	2.43	3.11	1.96	1.96	2.05
GUPTA3	24.24	1.32	1.31	1.51	1.27
MSDOOR	2.22	3.01	2.17	2.05	2.18
SHIP_003	1.78	2.46	1.77	1.63	1.75
PRE2	3.23	3.21	3.85	2.90	2.56
RMA10	1.76	1.75	1.75	1.72	1.65
TWOTONE	3.14	3.21	3.68	3.54	3.08
XENON2	1.93	3.37	1.96	1.82	1.73

Table 7
Maximal frontal matrix order

	METIS	SCOTCH	PORD	AMF	AMD
BMWCR1_1	2343	2040	2076	2496	2835
GRID	2754	2343	1721	1536	1328
GUPTA3	827	5058	1643	3028	1030
MSDOOR	1372	1624	1358	1491	1610
SHIP_003	3456	3156	3426	3408	4038
PRE2	4290	4334	5794	6476	7502
RMA10	466	422	378	439	399
TWOTONE	2382	2316	2561	2588	2684
XENON2	2554	2623	2743	3663	4501

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 (see also Table 4).

Regularity: According to Tables 4 and 5, we can see that PORD and AMF generate more unbalanced trees (where depth of leaves varies a lot depending on the

Table 8
Average front order

	METIS	SCOTCH	PORD	AMF	AMD
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
SHIP_003	170	153	165	163	151
PRE2	15	13	14	14	14
RMA10	60	59	54	53	56
TWOTONE	23	18	20	21	19
XENON2	70	69	70	71	76

branches) while SCOTCH and METIS generate much better balanced trees. Finally, we can see in Table 6 that PORD, AMD and AMF have trees where the average number of children 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.

Front size analysis: According to Tables 7 and 8, we can say that in most cases, SCOTCH and METIS generate trees with frontal matrices that are 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.

Summary: To summarize this section, we have seen that reordering techniques have a strong impact on the shape of the assembly tree. Fig. 6 summarizes the general observations made for the different reorderings on the assembly tree. 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.

5. Sequential memory usage of the multifrontal method

Given an assembly tree, an 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 processed 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 Fig. 7, and assuming that the depth-first search is used with nodes on the left processed first, the best case in terms of memory usage is the tree on the left where we

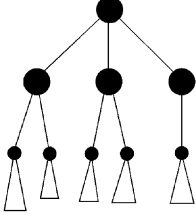
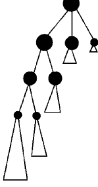
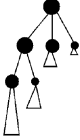
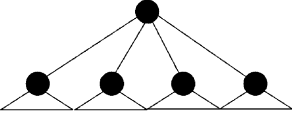
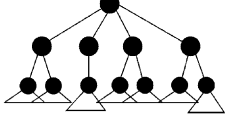
Reordering technique	Shape of the tree	Observations
AMD		<ul style="list-style-type: none"> • Deep well-balanced tree • Large frontal matrices on top of the tree
AMF		<ul style="list-style-type: none"> • Very deep unbalanced tree • Small frontal matrices • Very large number of nodes
PORD		<ul style="list-style-type: none"> • Deep unbalanced tree • Small frontal matrices • Large number of nodes
SCOTCH		<ul style="list-style-type: none"> • Very wide well-balanced tree • Large frontal matrices • Small number of nodes
METIS		<ul style="list-style-type: none"> • Wide well-balanced tree • Large number of nodes • Smaller frontal matrices (than SCOTCH)

Fig. 6. Shape of the trees resulting from various reordering techniques.

need to store the contribution blocks of at most two nodes simultaneously. On the other hand, the tree on the right-hand-side corresponds to the worst case because the contribution blocks of all leaves must be stored simultaneously.

Having as purpose to factorize large problems, we are interested in an out-of-core scheme either implicit (relying on system paging) or explicit. In both cases, since factors are not reaccessed once computed, they can be saved to disk. Therefore we focus in the following on the stack memory usage. We begin by studying the impact of reordering techniques on the stack memory. Then, we study how the memory consumption can be improved by using an optimal tree traversal.

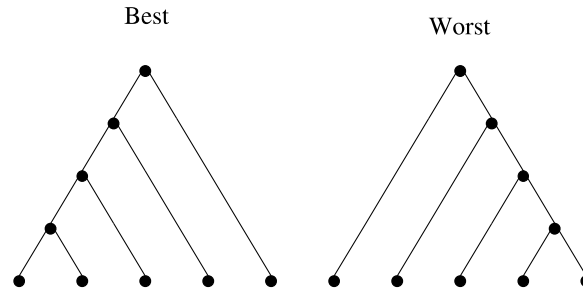


Fig. 7. Importance of the tree traversal.

5.1. Impact of reordering techniques on the memory

After studying the shape of the assembly tree, we now focus on the impact of the reordering techniques on the memory consumption in MUMPS.

Tables 9–11 present the stack memory traffic, the average stack size, and the peak of stack, respectively. For all these quantities, we neglected integer storage, so that the unit used is always the number of real entries. We observed that the storage needed by the integers is small compared to the one needed by the reals. For example, if we consider the matrix SHIP_003 with METIS, the total integer storage (factors + stack) for a sequential execution represents 2.3% of the storage needed by the reals. This ratio can slightly increase for small problems with limited fill-in like RMA10 where it reaches 4.4%.

Memory traffic: Table 9 gives the the sum of the sizes of contribution blocks for all the nodes of the tree. 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 PORD and AMF lead to the biggest global stack memory traffic.

Average stack size: Table 10 gives the average size of the stack during execution, defined as the average stack sizes for all variations observed. We see that for PORD the average size of stack memory is smaller than for the other reorderings. This is

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

	METIS	SCOTCH	PORD	AMF	AMD
BMWCR1_1	432.23	286.21	425.46	709.76	541.35
GRID	252.53	147.23	204.81	233.06	201.20
GUPTA3	144.83	10.36	287.47	231.41	236.74
MSDOOR	230.75	175.43	247.32	244.51	213.60
SHIP_003	509.66	249.64	590.01	656.21	496.76
PRE2	1186.45	280.02	1557.68	1267.98	623.01
RMA10	20.74	13.15	18.20	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 10
Average size of the stack (millions of reals)

	METIS	SCOTCH	PORD	AMF	AMD
BMWCR1_1	3.03	3.38	2.16	3.41	6.10
GRID	3.18	2.56	2.03	1.65	1.21
GUPTA3	15.20	1.44	38.72	7.31	8.41
MSDOOR	1.62	2.42	1.14	1.64	2.12
SHIP_003	5.76	6.74	4.32	7.01	10.67
PRE2	16.54	6.07	25.04	12.69	47.31
RMA10	0.16	0.13	0.09	0.17	0.14
TWOTONE	4.08	3.79	2.62	2.71	5.62
XENON2	4.69	4.89	3.90	6.11	11.27

Table 11
Peak of the stack (millions of reals)

	METIS	SCOTCH	PORD	AMF	AMD
BMWCR1_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
SHIP_003	23.42	23.06	20.86	20.77	32.02
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	11.80	11.63	17.59
XENON2	14.39	15.21	13.14	23.82	37.82

because PORD has deep trees (as shown in the previous sections) where we do not have to store a lot of contribution blocks at the same time. Compared with the other reorderings that also give deep trees like AMD, its tree often has fewer nodes and smaller frontal matrices which explains the difference in terms of the average size between AMD and PORD. Concerning AMF, we can see that its average is generally greater than the one of PORD. This is because the tree of AMF has larger branches (where there are a lot of memory operations) than the one of PORD. We can also observe that SCOTCH has a good average size because the number of nodes of its tree is smaller than for the other reordering techniques.

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 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

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). We also observed that MUMPS chooses to process the largest node first, and the largest node normally tends to have the deepest subtree.

Summary: To summarize this section, we have seen that since reordering techniques have a strong impact on the shape of the assembly tree, they also have a strong impact on the memory usage in the factorization. Table 12 summarizes the memory usage according to the reordering techniques. We have seen that PORD and AMF are the reorderings that use the smallest stack size (peak and average). For in-core executions, this should of course be related to the amount of work and the size of the factors, for which the following has been observed (see, for example, [13] as well as Figs. 4 and 5): 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.

In the previous section, we measured the influence of reordering techniques on the dynamic memory usage. 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 dynamic memory usage. In this section we derive an algorithm which



Table 12
Characteristics of the stack memory for different reordering techniques

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

The symbol “++” means a very big value, “+” means a big value, “-” a small value, and “--” a very small value.

determines for a given assembly tree what is the optimal tree traversal for minimizing the dynamic memory usage.

We first define some notation 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 denoted as $c_{i,j}$ where j varies between 1 and $nb_children(i)$. Finally, let cb_i and $factor_i$ be the memory requirement to store the contribution block and the factors (respectively) of the frontal matrix i (as shown in Fig. 9).

In [17], Liu proposes an algorithm that finds the best traversal of the tree in terms of peak stack size for a sequential multifrontal approach such as the code MA27 (available in the Harwell Subroutine Library). Based on his work, we present here a variant which is more appropriate to a distributed memory multifrontal solver such as 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 contribution block. In the case of MUMPS this optimization is not available because it cannot be implemented simply in a distributed memory parallel context. Thus, the application of Liu’s algorithm on a distributed memory code does not always give the best traversal. Indeed, if we consider the tree given in Fig. 10 (with no overlap between factors and contribution blocks), 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

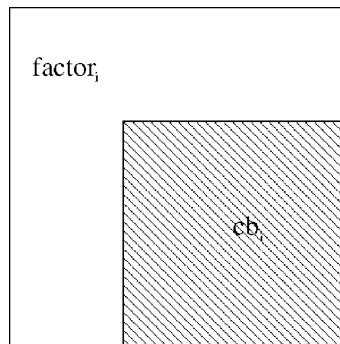


Fig. 9. Structure of a frontal matrix.

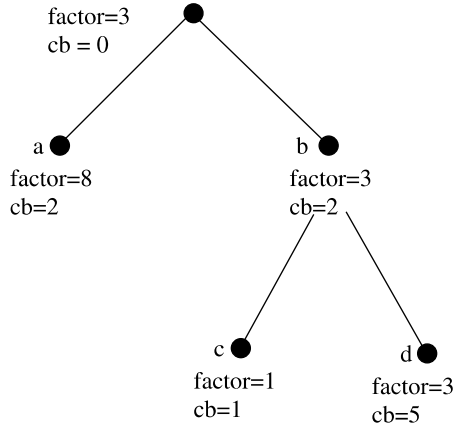


Fig. 10. Example of the application of LIU's algorithm.

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 just apply Liu's algorithm in our case. Note also that Liu's initial algorithm is restricted to elimination trees where only one pivot is eliminated at each node. In our case we work on assembly trees (with amalgamated nodes).

Let M_i be the maximum amount of stack memory necessary to process the complete subtree rooted at 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 $c_{i,j}$ (if any) and the current frontal matrix; thus the assembly step requires a storage:

$$store_i + \sum_{j=1}^{nb_children(i)} cb_{c_{i,j}}$$

When processing a child node $c_{i,j}$ the stack will contain the first $j - 1$ contribution blocks of the brothers of $c_{i,j}$ that have already been processed. The result is that the storage requirement at the time of factorizing the frontal matrix associated with $c_{i,j}$ is:

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

Thus, the storage requirement to process node *i* is recursively defined as:

$$M_i = \max \left(\max_{j=1, nb_children(i)} \left(M_{c_{i,j}} + \sum_{k=1}^{j-1} cb_{c_{i,k}} \right), store_i + \sum_{j=1}^{nb_children(i)} cb_{c_{i,j}} \right) \quad (1)$$

Since we want to minimize the peak of the stack, we should reduce the value of M for the root node(s). Adopting a theorem from [17] which says that the minimum of

```

Tree_Reorder (T):
  Begin
    for all  $i$  in the set of root nodes do
      Process_Child( $i$ );
    end for
  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( $c_{i,j}$ );
      end for
      Reorder the children  $c_{i,j}$  of  $i$  in decreasing order of  $(M_{c_{i,j}} - cb_{c_{i,j}})$ ;
      Compute  $M_i$  using the formula (1);
    end if
  End

```

Fig. 11. Optimal tree reordering for minimizing stack memory peak.

$\max_j(x_j + \sum_{i=1}^{j-1} y_j)$ 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 $M_{c_{i,k}} - cb_{c_{i,k}}$.

Considering a tree T , based on this result, the algorithm given in Fig. 11 gives an optimal traversal of the tree in terms of the peak of the stack. This algorithm consists in sorting the children nodes of a node i in descending order of $M_{c_{i,j}} - cb_{c_{i,j}}$ and compute the new value M_i for the parent node i , using (1).

Finally note that we have implemented variants of this algorithm: one for minimizing the global memory (stack + factors) and one for minimizing the average stack size during execution (when the peak is not changed). A complete analysis of all variants is available in [14].

5.3. Experimental results

We performed the experiments of Section 5.1 again, after algorithm given in Fig. 11 has been applied to the tree. In Table 13, we report on the gain in stack memory usage after applying algorithm given in Fig. 11. The gain is computed between the value of peak of stack memory of standard MUMPS and MUMPS where we postprocess the tree using the algorithm of Fig. 11. We can observe that the algorithm gives good results with METIS and SCOTCH. This can be explained by the fact that these reorderings generate wide trees where the traversal is very important in terms of memory usage. For AMD, we can see that the algorithm does not provide much gain. This is due to the shape of the tree of AMD. Indeed, it is deep and well-balanced. In addition, the brother nodes are not very different in terms of frontal matrix size. This implies that the order of the nodes does not have a strong impact for AMD (relatively well-balanced tree with balanced frontal matrices for brother nodes). Finally, for AMF and PORD we can see that the algorithm does not always give large gains. However, it gives very good results for some matrices like BMW_CRA_1. The reason

Table 13
Percentage of reduction of the peak of stack memory observed using Algorithm 11

	METIS	SCOTCH	PORD	AMF	AMD
BMWCRA_1	31.3	22.2	37.3	37.4	22.8
GRID	31.4	28.8	1.4	0	0
GUPTA3	37.3	0	1.8	26	8.6
MSDOOR	20.9	0	0	3.4	0
SHIP_003	24.8	26	8.8	0	0
PRE2	0.1	0	0	0	0
RMA10	31.9	16.4	24.4	0	17.6
TWOTONE	21.8	23.9	0	6	0
XENON2	21.1	24.7	0	0	0

for these different results is in the shape of the tree and the order of the brothers already implemented in MUMPS. We recall that MUMPS provides a basic sorting algorithm for the tree that processes the biggest child first. Generally, the biggest child roots the biggest subtree so this should be good for memory. Thus, for cases where the algorithm does not work well like matrix PRE2, or more generally with AMD and PORD, the order of MUMPS is in fact already optimal (thanks to the biggest node being processed before its brothers). To better illustrate the potential of Algorithm of the Fig. 11, we switched off the sorting mechanism of MUMPS; we observed that the gains are in that case much larger. For example, gains for SHIP_003 were 47.2%, 24.7% and 39.7% (instead of 8.8%, 0% and 0%), with PORD, AMF and AMD, respectively. Finally, for matrices like BMWCRA_1, and when the tree is better balanced, the order from MUMPS is not that good, and it is worth using the optimal tree traversal.

6. Memory usage for parallel executions

In this section we mainly focus on the size of the stack memory as a function of the number of processors and of the reordering technique used, when algorithm given in Fig. 11 is first applied to the tree. Because memory evolution depends on the distribution of nodes of the assembly tree onto the processors, we first describe the current scheduling strategy used in MUMPS. Then, we study the memory behaviour for parallel executions for different combinations of matrices and reorderings and analyze the factors that limit memory scalability.

6.1. Scheduling strategy used in MUMPS

MUMPS use a combination of static and dynamic mapping with distributed dynamic scheduling of the computational tasks. This is described in detail in [3,4]. The computation is driven by the assembly tree and a certain type of parallelism is assigned to each node. Fig. 12 summarizes the different types of parallelism available in MUMPS:

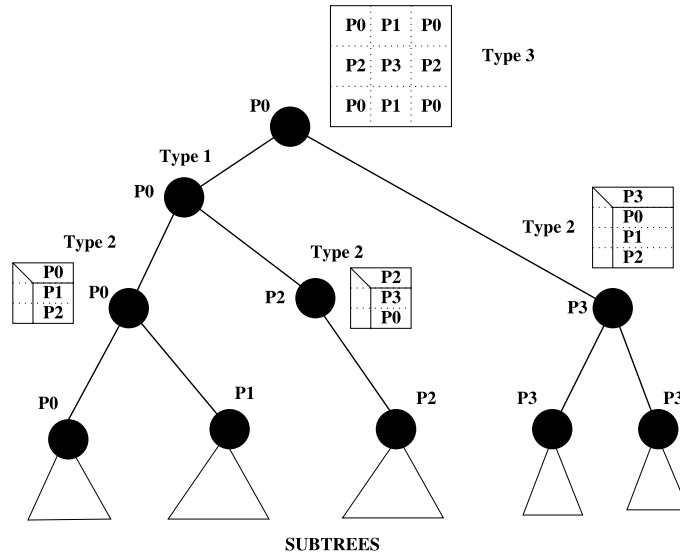


Fig. 12. Example of distribution of a multifrontal assembly tree over four processors.

- The first type uses the intrinsic parallelism induced by the assembly tree: each branch of the tree can be treated in parallel. A type 1 node is statically assigned to one processor which treats it when processors assigned to children nodes have communicated the contribution blocks. Leaf 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 it 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 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 workload (rather than memory usage) 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). Note that the slave selection strategy is different between the symmetric and the unsymmetric cases. Indeed, the granularity is smaller for the symmetric case with more slaves chosen in the symmetric case [4].
- 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 14 and 15 (respectively 16 and 17) show the maximum and average stack peak on 16 (respectively 32) processors for different matrices and reorderings.

Balance of the peak across processors: If we consider the difference between the maximum peak and the average peak, we observe that the balance is not perfect and that a better balance is obtained for METIS and SCOTCH. 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, the stack memory is very unbalanced. This is due to the shape of AMF's trees which are also very irregular and unbalanced. For such trees, the subtrees described in the previous section are also irregular. Some processors may for example begin to treat type 2 nodes when other ones are still processing subtrees and this can perturb the memory behaviour. This is also related to the mapping of the nodes of the tree and will be further discussed in Section 6.3.

Note that in the MUMPS scheduling strategy, only floating-point operations for the factorizations of frontal matrices are taken into account, the memory of the processors is not considered. Although this leads to a good balance of the workload, the memory load balancing is not perfect with a difference between the maximum peak and the average peak that can be significant.

Scalability of the stack peak: 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 (Table 14) and 32 processors (Table 16), we do not observe a linear improvement of the memory usage: in parallel doubling the

Table 14
Max peak of the stack on 16 processors (millions of reals)

	METIS	SCOTCH	PORD	AMF	AMD
BMWCR1_1	6.75	7.71	5.90	7.78	12.63
GRID	4.12	3.97	3.38	1.93	3.52
GUPTA3	9.27	4.99	8.84	16.34	4.88
MSDOOR	3.03	2.78	1.62	1.80	2.62
SHIP_003	10.02	7.91	5.72	5.01	11.01
PRE2	9.72	9.96	12.83	8.67	20.46
RMA10	0.41	0.36	0.42	0.35	0.36
TWOTONE	3.80	3.64	2.80	3.58	3.65
XENON2	6.32	5.00	4.63	6.29	9.75

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

	METIS	SCOTCH	PORD	AMF	AMD
BMWCRA_1	5.30	5.54	3.90	5.32	8.39
GRID	3.52	3.42	2.61	1.44	2.85
GUPTA3	6.12	3.37	3.05	5.63	2.92
MSDOOR	1.55	1.72	1.13	1.13	1.61
SHIP_003	6.67	6.42	4.29	3.47	8.24
PRE2	6.90	6.13	9.27	6.71	16.14
RMA10	0.25	0.24	0.21	0.20	0.25
TWOTONE	2.36	1.75	2.14	2.43	2.84
XENON2	3.94	4.11	3.26	4.39	7.66

Table 16
Max peak of stack on 32 processors (millions of reals)

	METIS	SCOTCH	PORD	AMF	AMD
BMWCRA_1	3.71	3.77	3.44	4.26	6.69
GRID	2.48	1.79	2.05	1.25	2.29
GUPTA3	7.73	3.40	8.85	16.07	3.57
MSDOOR	1.41	1.44	1.56	1.18	1.74
SHIP_003	5.48	4.29	3.15	2.63	6.15
PRE2	7.08	5.92	10.71	6.95	10.93
RMA10	0.40	0.36	0.36	0.35	0.31
TWOTONE	2.78	1.93	2.77	2.47	2.67
XENON2	3.92	3.52	3.45	4.64	7.97

Table 17
Average peak of the stack on 32 processors (millions of reals)

	METIS	SCOTCH	PORD	AMF	AMD
BMWCRA_1	2.84	2.70	2.39	2.92	5.08
GRID	1.46	1.34	1.55	0.84	1.86
GUPTA3	3.43	1.76	1.93	3.10	2.17
MSDOOR	0.85	0.93	0.72	0.70	1.01
SHIP_003	3.01	2.87	2.26	1.92	3.58
PRE2	3.80	3.25	5.483	3.88	8.36
RMA10	0.22	0.20	0.16	0.16	0.20
TWOTONE	1.55	1.17	1.68	1.79	1.62
XENON2	2.10	2.46	2.00	2.94	4.21

number of nodes, i.e., the memory size, does not mean we are able to treat a problem twice larger.

Figs. 13 and 14 illustrate this 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 many cases between 2 and 6. This illustrates that the stack memory does not scale

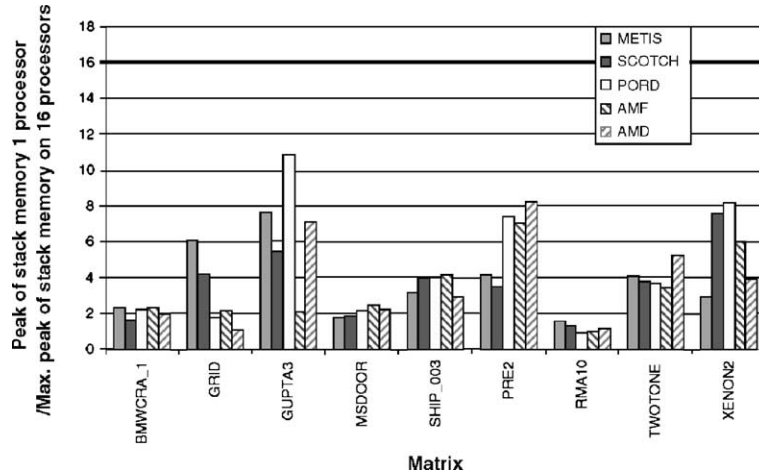


Fig. 13. Ratio between stack memory peak on 1 and 16 processors.

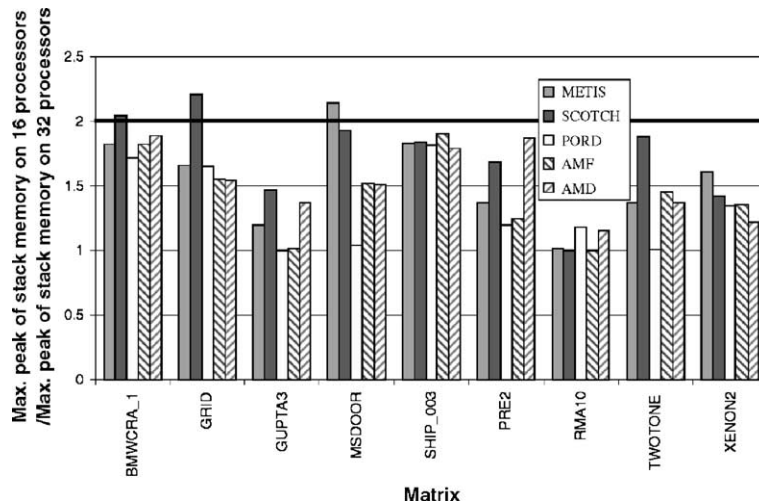


Fig. 14. Ratio between stack memory peak on 16 and 32 processors.

well (except for some cases like GUPTA3 with PORD). Fig. 14 gives a comparison between the peak of the stack on 16 and 32 processors. This time, the stack starts to scale better, although not linearly with the number of processors. The scalability is generally better for the symmetric case because more processors are used for each type 2 node than in the unsymmetric case (see Section 6.1, and [4] for more details).

Scalability of the total memory: Decreasing the stack memory is especially interesting in the case of an out-of-core approach. For an in-core solver like MUMPS, one is limited by the total memory (stack and factors). The ratio between the maximum peak of total memory on 16 and 32 processors is given in Fig. 15.

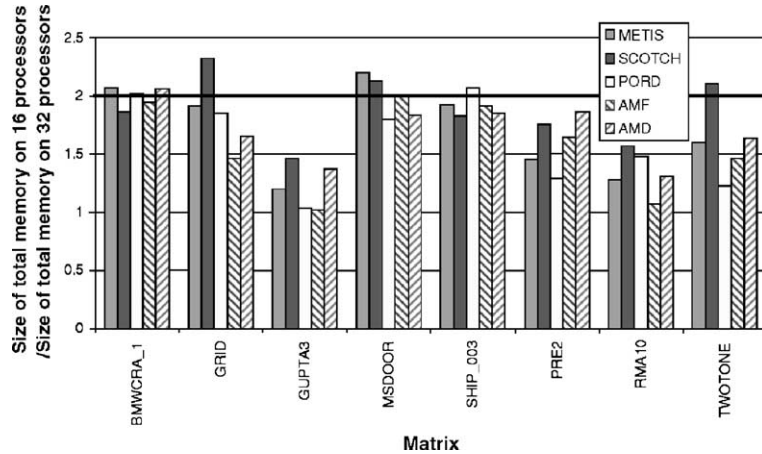


Fig. 15. Comparison of total memory peak for 16 and 32 processors.

We can observe that the scalability of the total memory is significantly better than for the stack memory and is even rather good for symmetric matrices. This is because the slave selection strategy aiming at balancing the flops will tend to provide a good balance of the factors on the processors. Furthermore contrarily to the stack size, the factors have a fixed size, independent of the number of processors. One consequence is that the size of the factors per processor decreases faster than the peak of the stack, and thus, the ratio stack/factors increases with the number of processors. So for problems where the stack is significant compared to the factors and/or for very large numbers of processors, the stack will play an important role even for an in-core parallel solver like MUMPS.

6.3. Factors impacting the memory scalability

In this section we give some remarks about the parallel memory behaviour of MUMPS and aim at finding factors that limit the scalability reported in the previous section. We illustrate this by analyzing some examples of typical situations where the peak of memory is reached and propose approaches that could improve both balance and scalability by avoiding such situations. We will particularly focus on the definition and assignment of subtrees and on dynamic scheduling. Note that in this section, when we say peak of memory, we mean the largest peak of memory across the processors.

- *Small matrices:* We observed in Section 6.2 that for a very small matrix like RMA10 the stack memory scalability is not good. In fact such small problems hardly exhibit any parallelism (no type 2 or type 3 parallelism) and this explains that the memory will not scale; independently of the mapping and of the number of processors, the peak observed is the same and is obtained during the sequential assembly of a node with the contribution blocks of its children.

- *Peak of memory inside a subtree:* We illustrate here the impact of the size of the subtrees on the memory behaviour of the solver. For example, considering the execution on 16 processors of matrix SHIP_003 with METIS, we have observed that the peak of stack memory is reached inside the first subtree treated sequentially by processor 12. In addition, processor 12 has not received any additional task from other processors. This shows that in that case the peak of stack memory is due to the static definition of the subtrees. A possible improvement would consist in splitting critical subtrees and distribute the resulting subtrees among several processors. Since the peak of the stack for a subtree can be determined statically, a strategy to avoid the lack of scalability due to that situation could be to split large subtrees until conditions such as $peak(subtree) < \frac{peak(whole\ tree\ in\ sequential)}{number\ of\ processors}$ and $peak(subtree) < \alpha \times memory\ on\ the\ processor, \alpha < 1$, are satisfied for all subtrees.

Fig. 16 illustrates the subtree splitting. We can see that the subtrees are smaller which is better for memory (but can be worst for performance). This simple example shows that the stack memory must be taken into account in the analysis (static) phase.

- *Slave selection:* An example that shows the importance of taking stack memory into account in the dynamic slave selection strategies of MUMPS is the execution on 32 processors of matrix SHIP_003 with PORD. For this execution the peak of stack memory is reached when processor 4, that has not finished one of its subtrees, is chosen as slave by processor 0. Since priority is given to the work received from other processors (slave work), the amount of memory needed by such tasks add up to the memory of the subtree being treated and increase the peak. We performed the following experiment: we put a synchronization barrier such that all processors wait for all subtrees to be processed. Even if processors store all contribution blocks of subtree roots, we observed a diminution of the maximum stack peak. This shows that this situation can be avoided by changing the slave selection strategy by giving preference to processors not involved in a subtree. This example illustrates that the memory should be taken into account in the selection strategy and a solution (not limited to the subtrees) is to design a general memory-aware dynamic scheduling strategy.

- *Order of subtrees:* A crucial point to obtain good performance and memory behaviour is the order in which the tasks are processed (particularly subtrees). Fig. 17 gives an example that illustrates the impact of the order of the initial pool of subtrees on both the performance and memory behaviour. In MUMPS children are recursively

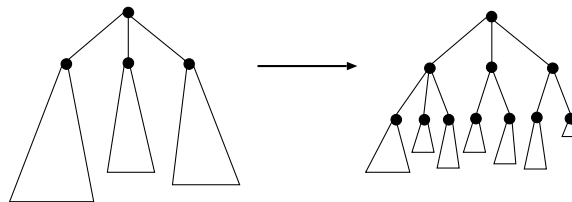


Fig. 16. Static improvement of the memory behaviour.

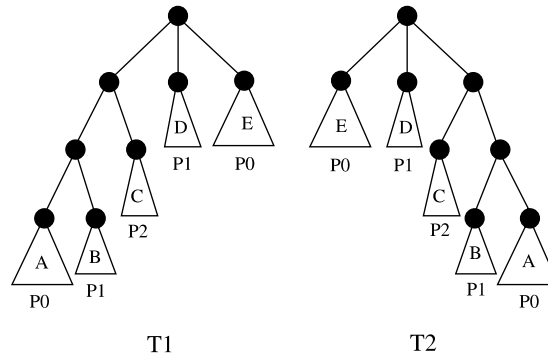


Fig. 17. Impact of the order on which subtrees are treated (processors are labeled P0, P1, P2 and subtrees are labeled A, B, C, D, E).

processed from left to right so that in the tree T1 (on the left), each processor will begin by its deepest (farthest to the root node) subtree. The bottom-up process will be time effective since the processors begin by the deepest parts of the tree. On the contrary, in the tree T2, each processor will begin by the subtree closest to the root. The bottom-up process will not exploit well the parallelism of the tree and be less time effective than T1. Concerning the memory, for tree T1, the first contribution blocks computed by P0 and P1 (corresponding to subtrees A and B) will be consumed quickly. On the other hand, for tree T2, processor P0 (respectively P1) will have to store the contribution blocks of the root node of subtree E (respectively D) until the root node of T2 can be activated. This leads to a larger memory usage for T2 compared to T1.

This simple example shows the great impact of the subtree sequence on each processor. It is important to note that Algorithm of Fig. 11 described in Section 5.2 will help to avoid the situation shown in the example because it tends to begin by the deepest parts of the tree. However, the application of the algorithm is not sufficient to ensure a good memory tree traversal for parallel cases since it does not take the mapping of the upper layers of the tree into account. A more sophisticated strategy to define the order of the subtrees on each processor should be based on both the tree topology and the mapping of the upper layers.

7. Conclusions

Whereas there are a lot of studies on the impact of reordering on fill-in, this paper provides an original study of the memory aspects of parallel multifrontal solvers, and in particular links between the reordering technique and the stack memory usage. We began our study with the impact of reordering techniques on the assembly tree and have observed that reordering techniques like METIS and SCOTCH give wide well-balanced trees while reordering techniques like AMF and PORD (respectively AMD) give very deep unbalanced (respectively balanced) trees with a large number

of nodes. From these results, we showed that deep unbalanced trees are better in terms of memory occupation than wide well-balanced ones. We have also seen how the stack memory evolution not only depends on the shape of the tree but also on the tree traversal during the factorization and have experimented with a variant of the algorithm by Liu to find the best tree traversal (in terms of memory occupation) in a distributed memory multifrontal solver such as MUMPS.

In the parallel case, the stack memory not only depends on the shape of the tree but also on the distribution of the computational tasks onto the processors. Our experiments show that the stack does not scale perfectly with the MUMPS default scheduling strategy based on workload. We analyzed some limitations and presented some ideas that can help improving this behaviour. We believe that optimizing and balancing the stack memory usage for parallel executions requires new scheduling strategies that are memory-aware. Furthermore, the static mapping of the subtrees and the order in which subtrees assigned to the same processor are treated is of great importance for the stack memory. This will be the object of future work.

References

- [1] P.R. Amestoy, T.A. Davis, I.S. Duff, An approximate minimum degree ordering algorithm, *SIAM Journal on Matrix Analysis and Applications* 17 (1996) 886–905.
- [2] P.R. Amestoy, I.S. Duff, Memory management issues in sparse multifrontal methods on multiprocessors, *International Journal of Supercomputer Applications* 7 (1993) 64–82.
- [3] P.R. Amestoy, I.S. Duff, J. Koster, J.-Y. L'Excellent, A fully synchronous multifrontal solver using distributed dynamic scheduling, *SIAM Journal on Matrix Analysis and Applications* 23 (1) (2001) 15–41.
- [4] P.R. Amestoy, I.S. Duff, J.-Y. L'Excellent, Multifrontal parallel distributed symmetric and unsymmetric solvers, *Computer Methods in Applied Mechanics and Engineering* 184 (2000) 501–520.
- [5] C. Ashcraft, R.G. Grimes, J.G. Lewis, B.W. Peyton, H.D. Simon, Progress in sparse matrix methods for large linear systems on vector computers, *International Journal of Supercomputer Applications* 1 (4) (1987) 10–30.
- [6] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petit, K. Stanley, D. Walker, 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, 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, J. Koster, On algorithms for permuting large entries to the diagonal of a sparse matrix, *SIAM Journal on Matrix Analysis and Applications* 22 (4) (2001) 973–996.
- [9] I.S. Duff, J.K. Reid, The multifrontal solution of indefinite sparse symmetric linear systems, *ACM Transactions on Mathematical Software* 9 (1983) 302–325.
- [10] I.S. Duff, J.K. Reid, The multifrontal solution of unsymmetric sets of linear systems, *SIAM Journal on Scientific and Statistical Computing* 5 (1984) 633–641.
- [11] A. Geist, E. Ng, Task scheduling for parallel sparse Cholesky factorization, *International Journal of Parallel Programming* 18 (1989) 291–314.
- [12] A. George, 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, G. Utard, On the memory usage of a parallel multifrontal solver, Technical Report RR-4617, INRIA, 2002. Also LIP Research Report RR(2002)-42.
- [14] A. Guermouche, J.-Y. L'Excellent, G. Utard, Analysis and improvements of the memory usage of a multifrontal solver, Technical Report RR-4829, INRIA, 2003. Also LIP Report RR(2003)-08.
- [15] G. Karypis, 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.
- [16] J.W.H. Liu, Modification of the minimum degree algorithm by multiple elimination, *ACM Transactions on Mathematical Software* 11 (2) (1985) 141–153.
- [17] J.W.H. Liu, On the storage requirement in the out-of-core multifrontal method for sparse factorization, *ACM Transactions on Mathematical Software* 12 (1986) 127–148.
- [18] J.W.H. Liu, The role of elimination trees in sparse factorization, *SIAM Journal on Matrix Analysis and Applications* 11 (1990) 134–172.
- [19] E. Ng, P. Raghavan, Performance of greedy heuristics for sparse Cholesky factorization, *SIAM Journal on Matrix Analysis and Applications* 20 (1999) 902–914.
- [20] F. Pellegrini, **SCOTCH** 3.4 user's guide, Technical Report RR 1264-01, LaBRI, Université Bordeaux I, November 2001.
- [21] F. Pellegrini, J. Roman, P.R. Amestoy, Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering, *Concurrency: Practice and Experience* 12 (2000) 69–84. (Preliminary version published in *Proceedings of Irregular'99 LNCS 1586*, pp. 986–995.)
- [22] E. Rothberg, R. Schreiber, Efficient methods for out-of-core sparse Cholesky factorization, *SIAM Journal on Scientific Computing* 21 (1) (1999) 129–144.
- [23] E. Rothberg, Stanley C. Eisenstat, Node selection strategies for bottom-up sparse matrix ordering, *SIAM Journal on Matrix Analysis and Applications* 19 (3) (1998) 682–695.
- [24] J. Schulze, Towards a tighter coupling of bottom-up and top-down sparse matrix ordering methods, *BIT* 41 (4) (2001) 800–841.