Memory-based scheduling for a parallel multifrontal solver.

Abdou Guermouche, Jean-Yves L'Excellent

Laboratoire de l'Informatique du Parallélisme

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

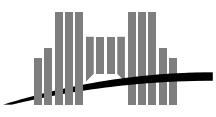
Memory-based scheduling for a parallel multifrontal solver

Abdou Guermouche,
Jean-Yves L'Excellent

April 2004

Research Report N° 2004-17

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`



Memory-based scheduling for a parallel multifrontal solver

Abdou Guermouche, Jean-Yves L'Excellent

April 2004

Abstract

The memory usage of sparse direct solvers can be the bottleneck to solve large-scale problems. This paper describes dynamic scheduling strategies that aim at reducing the memory usage of a parallel direct solver. Combined to static modifications of the tasks dependency graph, experiments show that such techniques have a good potential to improve the memory usage of a parallel multifrontal solver, MUMPS.

Keywords: Sparse matrices, multifrontal method, scheduling, memory

Résumé

L'occupation mémoire des méthodes directes peut être critique pour la résolution parallèle de systèmes linéaires creux de grande taille. Ce rapport propose des stratégies d'ordonnancement dynamique ayant pour objectif de réduire l'occupation mémoire de ces méthodes. En les combinant avec des modifications statiques du graphe de dépendence des tâches, nous montrons le potentiel qu'ont ces stratégies d'ordonnancement pour améliorer l'occupation mémoire d'un solveur parallèle basé sur la méthode multifrontale, MUMPS.

Mots-clés: matrices creuses, méthode multifrontale, ordonnancement, 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. It is thus crucial to optimize the memory scalability.
We studied in [12] the memory behaviour of the parallel multifrontal solver `MUMPS` [3, 4] that uses both static and dynamic scheduling approaches for the management and the distribution of the tasks onto the processors. We have seen that the active memory occupation of the solver for parallel executions can be improved. Thus we propose in this paper new memory-based dynamic scheduling mechanisms to improve the memory behaviour of the solver. The first one consists in a memory-based slave selection strategy for the parallel tasks during the factorization. The second one concerns the dynamic scheduling of the tasks assigned to a processor. These mechanisms aim at reducing the memory occupation of the solver and improving the memory balance between the processors. This work is a first step towards a more global solution that will take both workload and memory constraints into account. As outlined later in this paper, the design of more memory-aware static scheduling strategies is also very important.
This paper is organized as follows. In Section 2, we recall some general mechanisms of the multifrontal method. In Section 3 we give a description of the management of parallelism in `MUMPS`. In Section 4, we present a new memory-based slave selection strategy. After that, in Section 5.1, we describe some improvements that we made to the slave selection strategy given in Section 4. We present in Section 5.2 a dynamic memory-aware task management strategy for the tasks statically assigned to the processors. In Section 6, we present some experimental results illustrating the impact of our new strategies on the memory behaviour of the solver and show how static parameters can be modified to improve that impact. Finally, we conclude.

2 The multifrontal method

In the multifrontal approach [8, 9], the factorization of the matrix is done by performing a succession of partial factorizations of small dense matrices called *frontal matrices*, that are associated to each node of a so-called assembly tree (Figure 1) representing the dependency between tasks. The frontal matrix is divided into two parts: the *factor* block, also called *fully summed* block, which corresponds to the variables that are factorized when the elimination algorithm processes the frontal matrix; and the *contribution block*, which corresponds to the variables that are 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 processor parent nodes before their children) [16] 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

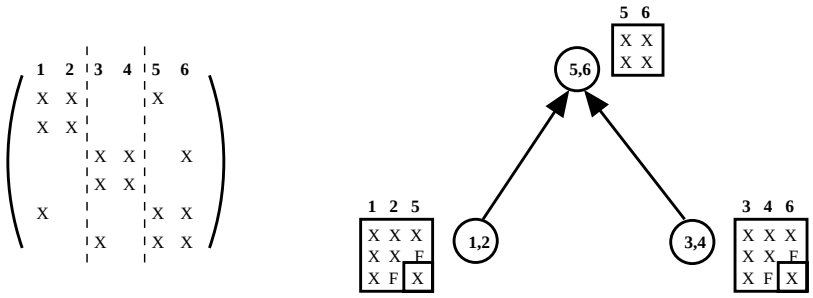


Figure 1: A matrix and the associated assembly tree.

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 removed from the stack and its size decreases. The stack memory evolution is very dependent on the assembly tree topology, as observed in [12].

3 Scheduling strategy used in MUMPS

MUMPS uses a combination of static and dynamic scheduling strategies. This is described in details in [3] and [4]. The computation is driven by the assembly tree and to each node is assigned one type of parallelism. Figure 2 summarizes the different types of parallelism available in MUMPS:

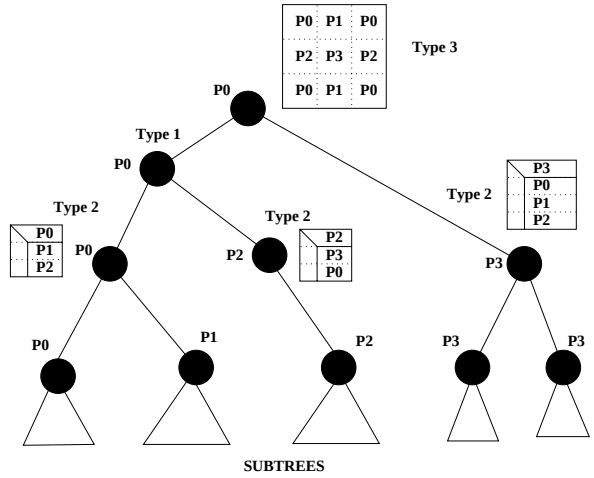


Figure 2: Example of distribution of a multifrontal assembly tree over four processors.

- The first type uses only 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 contribution blocks from children nodes have been communicated. Leave subtrees (simply called *subtrees* in the rest of this paper) are a set of type 1 nodes all assigned to the same processor. Those are determined using

a top-down algorithm [10] and a subtree-to-processor 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. Large fronts in the assembly tree are treated in parallel and are 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 workload 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 and assigns them new tasks. The corresponding slave tasks are then activated as soon as they are received on the slave side. The workload 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 [5]: a 2D block cyclic distribution is applied for the factorization.

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. Each processor has a pool of ready tasks and informs others of the corresponding number of floating-point operations. Initially, a processor has as its initial workload the cost of all its subtrees. Concerning the slave selection strategy for type 2 nodes, each (master) processor tries to choose only the processors less-loaded than itself, with some granularity constraints. In addition, the selection is done such that the amount of work given to the slaves is as balanced as possible with the workload of the corresponding task on the master.

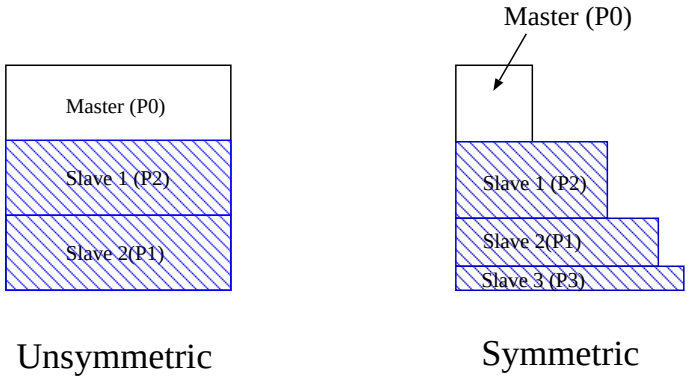


Figure 3: Type 2 nodes blocking with the default strategy.

The blocking used for the type 2 nodes is regular for the unsymmetric case and irregular for the symmetric case (see Figure 3), in order to balance the work between slave subtasks.

We showed in [13] that the active memory behaviour of MUMPS is not very good for parallel executions. Thus we present in the next sections some memory-aware dynamic scheduling strategies to improve the memory behaviour of the solver. We first present a memory-based slave selection strategy. Then we describe some improvements to the strategy. Finally, we give a new dynamic task scheduling that aims at improving the memory behaviour of MUMPS.

4 Memory-based slave selection strategy

In this section we describe a new memory-based scheduling approach that aims at reducing the size of the stack memory during the execution. It is a fully dynamic algorithm that helps a processor to choose its co-workers or slaves with memory constraints. This algorithm requires information about the memory occupation of each processor so that the calling processor (master of type 2 node) can choose its slaves according to their memory load. It is given below:

Algorithm 1 Memory-based slave selection strategy.

Begin
$nfront$=order of the current frontal matrix;
Sort the potential slave processors in growing order of their memory occupation;
find the biggest i, number of processors, so that $\sum_{j=1}^{i}(MEM[i]-MEM[j])$ is smaller than the surface of the frontal matrix;
for $j = 1$ to i **do**
give $(MEM[i] - MEM[j])/nfront$ rows of the matrix to processor j;
end for
assign the remaining rows equitably to each processor 1..i;
number_of_slaves=i;
return;
End

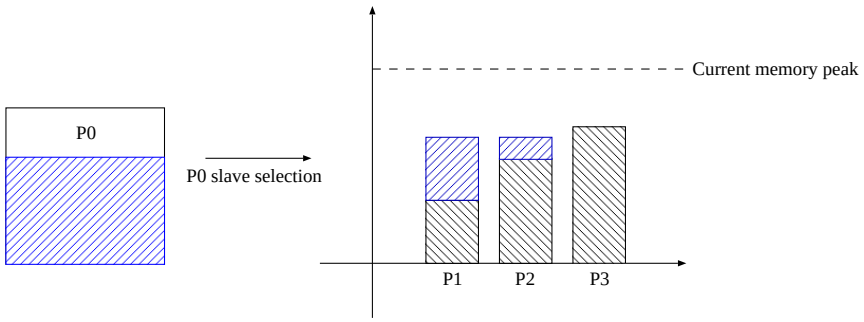


Figure 4: Memory-based slave selection strategy.

The algorithm chooses slave processors according to their memory load. It tends to choose the smallest set of processors that globally balances the memory without increasing the current peak, as shown in Figure 4. When a processor has to choose slaves it first sorts the

others in growing order of their memory load. Then it distributes the rows of the frontal matrix on the processors in a way that both balances the memory and preserves the current peak (if possible). The 1D-blocking produced by the algorithm for both the symmetric and unsymmetric cases will have the same shape as in Figure 3 except that the number of rows in each partition will be far more irregular.
Algorithm 1 is designed for both the symmetric and unsymmetric cases. In the following parts we will concentrate on the unsymmetric case since the symmetric one is more complex with more irregular blocks.
Note that the mechanism used to broadcast information relative to the memory occupation of other processors is the object of a forthcoming technical report. In this mechanism, each processor broadcasts the variation of its memory when it occurs. Thus, the others can accumulate the increments, that can be positive or negative, sent by the processor. In addition, a mechanism ensures that the choices done by master processors are known as quickly as possible by the others so that they can take them into account in their slave selection. Such a mechanism is necessary for each processor to have a coherent and as up-to-date view as possible of the system.

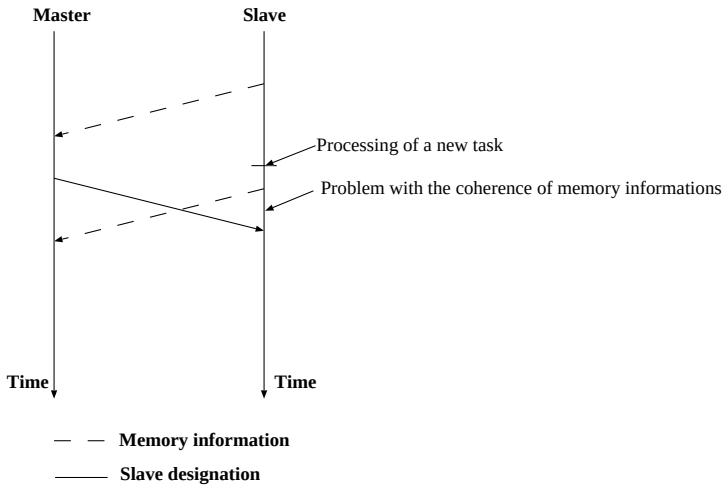


Figure 5: Example illustrating the necessity of predicting the activation of ready tasks.

As described above, Algorithm 1 chooses the slaves based on the instantaneous memory information exchanged by the processors. This can cause some critical situations as shown in Figure 5: when a processor chooses another one as slave, it must be aware of the next task(s) that will be treated on the slave. In that figure, the new task treated by the slave processor can be very costly in terms of memory occupation and this can lead to an increase of the peak of memory. Such situations have been observed in preliminary experiments (not reported here) that showed the necessity of taking into account information about ready tasks. This is the object of the following section.

5 Improvements to the dynamic memory-based strategies

In this section we present some improvements to the dynamic memory-based slave scheduling strategy presented in the previous section. We present two mechanisms aiming at injecting

static information in the dynamic scheduling to make it more efficient. In a second part, we present a memory-aware task selection scheme that concerns the nodes assigned statically to a processor.

5.1 Injecting static knowledge to the memory-based slave-selection strategy

As shown in the previous section, the dynamic memory-aware strategies must take into account the tasks that will be processed (in the near future). Such tasks are characterized by the fact that they have been statically assigned to a given processor. To ensure a good memory behaviour, we must provide the information relative to these tasks before they are treated. We distinguish the tasks that are inside the subtrees (type 1 nodes) and the tasks of the upper layers of the tree (type 1,2 or 3 nodes).

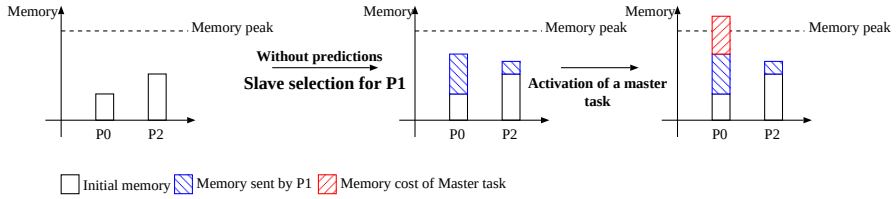
Subtrees

The subtrees can be very critical in terms of memory occupation [12]. Thus, a memory-based dynamic strategy must be aware of the fact that a processor is treating a subtree to avoid a memory occupation increase (the memory of a slave task would add to the peak of the subtree, not reached yet). Note that this would be most beneficial if combined to a static splitting of subtrees with a large memory cost, as noted in [12]. The mechanism we design is based on messages: once a processor starts a subtree, it broadcasts the cost (memory peak) of the concerned subtree. This choice (sending the memory peak of the subtree) is motivated by the fact that the active memory can have great variations inside a subtree and that the tasks belonging to subtrees are relatively small in terms of processing time. Thus each processor has a view of the cost of the subtrees being treated on the others. Such information will be used to select the slaves for type 2 nodes.

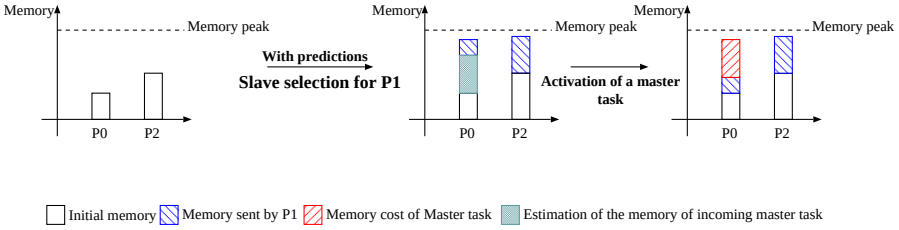
Upper part of the tree

Concerning the upper part of the tree, we must be able to anticipate the activation of a node. The motivation is that we should avoid giving slave work to a processor that will soon start a master task (that can be large). Indeed, if we consider the situation of Figure 6 where processor 0 will treat in the near future a large type 1 node, an increase of the memory peak will occur if processor P_0 is selected as slave (see Figure 6(a) – the dashed line represents the global memory peak observed since the beginning of the factorization). The selection of P_0 as slave will occur with the memory-based slave selection strategy described in the previous section. To avoid such a critical situation, we must predict the activation of such tasks. This can be done in a simple way using the assembly tree. Note that a task becomes ready once all its children have been processed. Thus, if every processor treating a child sends a message to the one in charge of the parent node, the processor in charge of the parent knows that this task will become ready in a relatively small amount of time. It can then send the cost of the corresponding task to all the processors. In addition, if more than one task becomes ready, the processor concerned sends the cost of the task, in the set of ready nodes, that has the largest memory requirements. Finally, once a task is activated, the processor concerned sends the new cost that corresponds to the largest task that can be activated. An example of the impact of the mechanism described here is given in Figure 6(b). We can see that the information about the incoming master tasks is helpful to avoid an increase of the memory peak in this

situation. The informations given by the previous mechanisms to inform the others about the



(a) Without prediction



(b) With prediction

Figure 6: Critical situation with a node of the upper part of the tree.

subtrees and the nodes of the upper layers of the tree can be injected easily in Algorithm 1. Indeed, if we replace the metric used for the slave selection, which was the instantaneous memory, by the sum of the instantaneous memory, the amount of memory needed by the current subtree and the memory cost of the next upper layers task, the algorithm will select the slaves and respect all the constraints introduced by the new mechanisms. In the rest of the paper we will use Algorithm 1 to denote Algorithm 1 where we use the metric described above.

5.2 A memory-aware task selection strategy

In this section we present a memory-aware task selection strategy. We begin by describing the behaviour of `MUMPS` in terms of tasks scheduling. Here we mean by task the type 1 or type 2 master tasks statically assigned to the processors; we remind that slave tasks are activated as soon as they are received by the chosen slave processor. Then we present our new task management scheme.

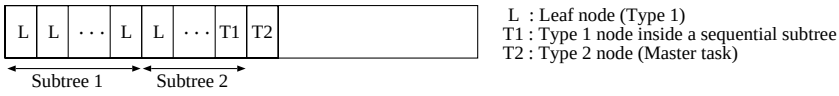


Figure 7: Example of a pool of tasks.

`MUMPS` has a *pool* of ready tasks, where each task corresponds to a node of the assembly tree. It is important to note that the pool is local to each processor and it contains only the ready

tasks statically assigned to the processor. The pool is managed with a stack mechanism. In the default strategy, the task that will be selected is the one on the top of the stack. In addition, when a node of the tree becomes ready (all its children have been processed) it is pushed on the top of the stack. This simple mechanism ensures a "depth-first" traversal of the assembly tree, which generally limits the amount of temporary memory needed by the factorization, although this is not always applicable in the parallel case. Initially, the pool of tasks contains all the leaf nodes of the tree statically assigned to the processor, sorted so that the leaves of a subtree are all in a contiguous set (and minimize the memory of each subtree using a variant of the algorithm by [15]). A representation of the pool is given in Figure 7. Note that for the workload-based dynamic scheduling, the load of the processor is updated as soon as a task becomes ready (i.e. is inserted in the pool) whereas for the memory-based dynamic scheduling the memory information is updated when the task is activated (.i.e when it consumes memory).

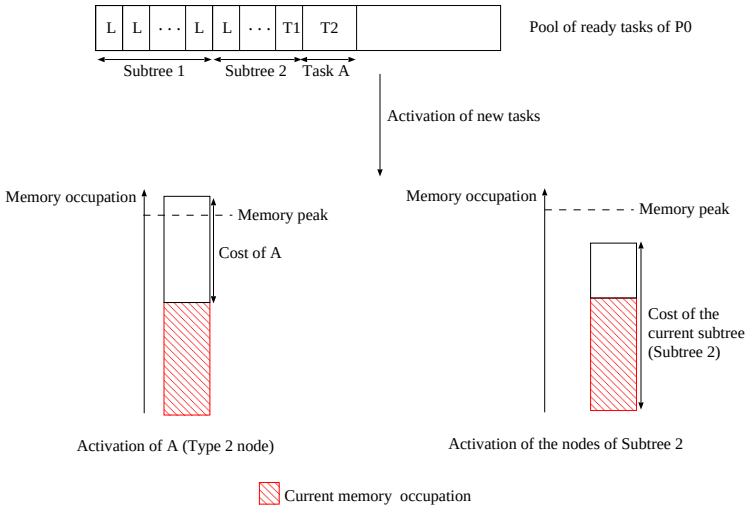


Figure 8: Example of a critical situation for the task selection.

The problem with the basic strategy described in the previous paragraph is that it does not take the memory needed by the task into account for the selection of the node that will be activated. Indeed if we consider the situation given in Figure 8 where a processor is treating a subtree and has a large type 2 node that becomes ready, the basic strategy will extract the large node even if the memory occupation is already critical (situation on the left). Thus the cost of this node will add up to the cost of the subtree and this can lead to a dramatical increase of the memory occupation. On the other hand, if the processor delays the activation of the type 2 node until the situation becomes less critical (situation on the right), the memory peak will not increase. With the purpose of avoiding such a situation, we designed a more sophisticated task selection algorithm (Algorithm 2). It chooses the task that will be activated taking memory constraints into account. First, if a processor has started a subtree and if the task located on the top of the pool is belonging to the subtree it activates it systematically. This is because subtrees can be very expensive in terms of memory usage. If the node at the top of the pool belongs to the upper part of the tree (outside a subtree), the algorithm selects it only if it does not increase the memory peak observed since the beginning

of the factorization. Otherwise, it tries to select another task by giving priority to the nodes belonging to the subtrees, trying to respect the depth-first traversal as much as possible. (Going too far from the depth-first-search traversal could tend to increase the number of branches of the tree active simultaneously and increase the global memory usage.)

Algorithm 2 Memory-aware task selection algorithm.

```
Begin
  if the node of the top of the pool is inside a subtree then
    return the node of the top of the pool;
  else
    for i in the pool of ready tasks (starting from the top of the pool) do
      if memory_cost(i) + current memory (including peak of subtree) ≤ memory peak
      observed since the beginning of the factorization then
        return i;
      else
        if i belongs to a subtree then
          return i;
        end if
      end if
    end for
    return the node of the top of the pool;
  end if
End
```

6 Experimental results

We should first mention that the design of the algorithms of Sections 4, 5.1 and 5.2 are already the result of a large amount of experimentation and we focus here on showing their global effects.

To study the impact of the dynamic strategies proposed in the previous sections, we experiment our strategies on several problems (see Table 1) extracted from either the Rutherford-Boeing collection [7], the collection from University of Florida[1] or the PARASOL collection[2]. The tests have been performed on the IBM SP system of IDRIS [3] composed of several nodes of 32 processors each. Each node is equipped with a minimum of 64 GB of memory shared among its 32 Power4 (1,3 GHz) processors.

We have tested our strategies on 32 processors of the above-described platform using several reordering techniques: AMD (Approximate Minimum Degree) [1], AMF (approximate Minimum Fill) as implemented in `MUMPS`, PORD [17] and METIS [14]. These reordering techniques were used to study the behaviour of our strategies on different topologies of assembly trees, since the tree topology is very influenced by the reordering technique used [12]. The results obtained are given in Table 2.

We observe that the dynamic memory-based strategies from the previous sections are beneficial for some combinations of problems/orderings.

[1] `http://www.cise.ufl.edu/~davis/sparse/`
[2] `http://www.parallab.uib.no/parasol`
[3] Institut du Développement et des Ressources en Informatique Scientifique

Matrix	Order	NZ	Type	Description
BMWCRA_1	148770	5396386	SYM	Automotive crankshaft model
GUPTA3	16783	4670105	SYM	Linear programming matrix (A*A')
MSDOOR	415863	10328399	SYM	Medium size door
SHIP_003	121728	4103881	SYM	Ship structure
PRE2	659033	5959282	UNS	AT&T,harmonic balance method
TWOTONE	120750	1224224	UNS	AT&T,harmonic balance method.
ULTRASOUND3	185193	11390625	UNS	Propagation of 3D ultrasound waves generated by X. Cai (Simula Research Laboratory, Norway) using Diffpack.
XENON2	157464	3866688	UNS	Complex zeolite,sodalite crystals.

Table 1: Test problems.

	METIS	PORD	AMD	AMF
BMWCRA_1	3	*0*	0.6	**4.1**
GUPTA3	**5.6**	*0*	0	0
MSDOOR	**14.3**	*0*	2	0
SHIP_003	2	*-1*	**2.1**	0.2
PRE2	**10.3**	1	8.8	*-10.5*
TWOTONE	-0.3	*-4.9*	10.9	**50.6**
ULTRASOUND3	**16.5**	3.5	*-2*	3.9
XENON2	3.5	*0*	12	**12.4**

Table 2: Percentage of decrease of the maximum stack memory peak obtained by using dynamic memory strategies.

For symmetric cases, we have observed that most entries with a zero in the table correspond to the situation where the peak of the stack is obtained inside a subtree. Although our strategy will avoid to choose a slave processor treating a subtree with a significant peak, the peak of the subtree itself is too large and corresponds to the peak obtained. It is the same with or without the application of our dynamic memory strategies.
This remark shows that in the symmetric cases, the definition of the subtrees should be revised and take memory constraints into account, as subtree peaks are the limiting factor of memory scalability. The order in which subtrees are treated is also important and some preliminary work is available in [11].
For unsymmetric matrices, there are some gains that can be significant, but in many cases the gain is small compared to the initial scheduling strategy based on workload. We could observe that each time the gain is negative, the peak is reached when a master of a large type 2 node (see Figure 4, left) is allocated. For example, on PRE2 with AMF, there is a master of a type 2 node that represents 3.6 millions entries while the peak of the stack was of order 5.4 millions.
To overcome such problems, we modify statically the assembly tree and split some tasks of the dependency tree into a chain of several tasks (see [3]), thus avoiding nodes with a large master part. We used a threshold value of 2 million entries to define the splitting strategy; in other words we do not allow master nodes to have more than 2 million entries.

	METIS	PORD	AMD	AMF
PRE2	11	**16.9**	4.3	*0.8*
TWOTONE	9.2	*0*	14.1	**51.4**
ULTRASOUND3	5.9	13.4	*-2.8*	**14.1**
XENON2	**12.9**	0	*-3.3*	9

Table 3: Percentage of decrease of the maximum stack memory peak obtained by using dynamic memory strategies. The tree has been statically modified to avoid large type 2 masters.

The dynamic memory strategies are less limited by huge masters of type 2 nodes and work better with such a modified task graph. Table 3 gives the results obtained by the dynamic memory strategies on trees with splitted nodes against the dynamic workload strategy (from `MUMPS`) on the same tree with same splitted nodes. We can see that the gains are globally more significant in that case.
For TWOTONE with AMF, the gain is approximatively the same as in Table 2 because no splitting occurred in that case (no nodes with a master part larger than 2 millions). The little difference on the gains measured between Table 2 and Table 3 is due to the non-deterministic execution scheme of `MUMPS`. Note that these observations also mean that the choice of the threshold for splitting may be improved and should be more matrix-dependent.
There is also a loss in some cases with the static modification of the tree (splitting of large nodes). For ULTRASOUND3 matrix with AMD, the loss is mainly due to the mechanisms managing the subtrees and the prediction of incoming master nodes. Indeed, in this case, the peak of active memory is reached when a processor (P_i) starts a slave task sent by another one (P_j). This normally tends to be avoided, but at this moment of the factorization, all the processors have approximatively the same memory occupation and most processors (but not P_i) have sent a prediction information. Thus, when P_j selects its slaves, it will select by priority processors that do not have an incoming task.
Concerning XENON2 and AMD, the peak is reached before any slave selection has been

done. It is a side effect of the mechanism of task selection presented in Section 7. Indeed, in this example, the processor is treating a subtree when a type 1 node becomes ready. Thus, using Algorithm 2, the processor chooses to delay this type 1 task (that does not belong to the current subtree). It then treats the subtree until its end. The peak is reached when it activates the type 1 node. At this moment, it only has contribution blocks, that correspond to the root nodes of the treated subtrees, and the type 1 node. This illustrates some limitations of the task selection strategy given in Algorithm 2: either type 1 nodes should be avoided in the top of the tree, or this local strategy should be improved by using more global information. For example the selection of a task should not only be based on the memory of the processor concerned but also on the memory that will be freed (contribution blocks) on others by the selection.

In order to illustrate separately the gains due to the splitting and to the dynamic strategy, we report the peak of stack memory obtained for two cases where we observed significant gains in Table 4. We observe that both the static splitting and the new memory-aware dynamic strategies are useful to decrease memory.

	ULTRASOUND3 – METIS		XENON2 – AMF	
	No splitting	Splitting	No splitting	Splitting
`MUMPS` dynamic strategy	7.56	6.09	3.14	3.14
memory-based dynamic strategy	6.13	5.73	1.55	1.52

Table 4: Maximum peak (over the processors) of stack memory (millions of entries) on two illustrative cases.

Finally, the gains between original `MUMPS` and `MUMPS` with both dynamic memory strategies and static additional splitting are reported in Table 5. Except for the case of TWOTONE with METIS and PORD (bad result due to Algorithm 2 – same situation as XENON2 with AMD discussed earlier), these results show the potential of combining dynamic and static approaches.

	METIS	PORD	AMD	AMF
PRE2	12.5	**31**	24.5	*1*
TWOTONE	-1.3	*-3*	14.1	**51.4**
ULTRASOUND3	24.2	*5.1*	31.6	**39.5**
XENON2	13.8	*0*	18	**32.7**

Table 5: Percentage of decrease of the maximum stack memory peak when both static and dynamic approaches are applied, compared to the original strategy of MUMPS.

	METIS	PORD	AMD	AMF
SHIP_003	3.0	94.3	21.2	36.8
PRE2	-4.5	0.1	8.5	-3.2
ULTRASOUND3	8.5	3.7	9.0	49.8

Table 6: Loss of performance (percentage) between the original MUMPS strategy and the strategy where memory is optimized.

7 Conclusion

In this paper we have proposed some dynamic scheduling strategies to improve the memory behaviour of a parallel sparse direct solver. We have observed some static limitations to the impact of the dynamic approaches and showed that by modifying the granularity some tasks (splitting them), the behaviour of the dynamic strategy can be improved.
The work presented on static aspects can still be improved, by exploiting more the potential of splitting large tasks, as well as by splitting subtrees with large memory peaks (especially for symmetric matrices). We would also benefit from a more global strategy in the task selection scheme given in Algorithm 2. However, the global results are promising and show the potential of these strategies.
We have observed (see results from Table 6 for three large test problems) that the factorization time (which we did not especially try to preserve) does not increase by a too large factor by applying memory-based strategies. This gives good hope to still improve the memory behaviour while keeping the execution time good. For that, hybrid strategies well adapted at both balancing the workload and the memory need to be designed.
By minimizing the stack memory and improving the memory scalability, we will be able to treat larger problems since the scalability of the stack is currently a limiting factor of the factorization. Furthermore, such strategies can also be coupled to out-of-core approaches (explicit or implicit): since factors are not reaccessed before the solve phase once computed, they can be stored on disk, and it is crucial to minimize the remaining part of the memory (that is, the stack). Such techniques could also be coupled to implicit out-of-core techniques where I/O on disks are controlled at the system-level with directives given by the application (similarly to [6]).

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] 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.
[6] O. Cozette, A. Guermouche, and G. Utard. Adaptative paging for a parallel multifrontal solver. Technical report, LIP/INRIA, 2004. In preparation.
[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. K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.
[9] I. S. Duff and J. K. Reid. The multifrontal solution of unsymmetric sets of linear systems. *SIAM Journal on Scientific and Statistical Computing*, 5:633–641, 1984.

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

[11] A. Guermouche. Impact de l'ordonnancement sur l'occupation mémoire d'un solveur multifrontal parallèle. In *15ieme Rencontres Francophones en Parallélisme, La Colle sur Loup, France*, pages 37–45, 2003.

[12] A. Guermouche, J.-Y. L'Excellent, and G. Utard. Impact of reordering on the memory of a multifrontal solver. *Parallel Computing*, 29(9):1191–1218, 2003.

[13] A. Guermouche, J.-Y. L'Excellent, and G. Utard. On the memory usage of a parallel multifrontal solver. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS'03)*, 2003.

[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, Sept. 1998.

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

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

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