

# Hybrid scheduling for the parallel solution of linear systems

Patrick R. Amestoy, Abdou Guermouche, Jean-Yves L'Excellent and Stéphane Pralet

Technical Report TR/PA/04/140

December 2004

## Abstract

In this paper, we consider the problem of designing a dynamic scheduling strategy that takes into account both workload and memory information in the context of the parallel multifrontal factorization. The originality of our approach is that we base our estimations (work and memory) on a static optimistic scenario during the analysis phase. This scenario is then used during the factorization phase to constrain the dynamic decisions. The task scheduler has been redesigned to take into account these new features. Moreover performance have been improved because the new constraints allow the new scheduler to make optimal decisions that were forbidden or too dangerous in unconstrained formulations. Performance analysis show that the memory estimation becomes much closer to the memory effectively used and that even in a constrained memory environment we decrease the factorization time with respect to the initial approach.

**Keywords:** sparse matrices, parallel multifrontal method, dynamic scheduling, memory.

## 1 Introduction

We consider the direct solution of large sparse systems of linear equations  $Ax = b$  on distributed memory parallel computers using multifrontal Gaussian elimination. For an unsymmetric matrix, we compute its  $LU$  factorization; if the matrix is symmetric, its  $LDL^T$  factorization is computed. Because of numerical stability, pivoting may be required.

The multifrontal method was initially developed for indefinite sparse symmetric linear systems [8] and was then extended to unsymmetric matrices [9]. It belongs to the class of approaches which separates the factorization into two phases. The symbolic factorization looks for a permutation of the matrix that will reduce the number of operations in the subsequent phase, and then computes an estimation of the dependency graph associated with the factorization. Finally, in an implementation for parallel computers, this phase partially maps the graph onto the target multiprocessor computer. The numerical factorization phase computes the matrix factors. It exploits the partial mapping of the dependency graph and performs dynamic task creation and scheduling to balance the work performed on each processor [1, 2, 4]. The work in this paper is based on the solver MUMPS, a MUltifrontal Massively Parallel Solver [1]. For an overview of the multifrontal method we refer to [7, 8, 16].

The work presented in [12] has shown how to use memory-based dynamic scheduling to improve the memory management of a parallel multifrontal approach. However, the authors also noticed

that they can significantly improve the memory behaviour but at the cost of an increase in the factorization time. Another important issue concerns the overestimation of the memory needed for parallel factorization. Indeed, even if in [4] the authors have shown that with the concept of candidate processors the memory estimates can be significantly reduced, there is still an important and unpredictable gap between real and estimated memory. Hence another target will be to decrease the memory estimates of the analysis and to respect them during the factorization.

In this paper, we propose a scheduling approach that uses both memory and workload information in order to obtain a better behaviour in terms of estimated memory, memory used and factorization time in the context of the parallel symmetric and unsymmetric factorization algorithms. The main principle of our approach is to use an optimistic scenario during the analysis that is then relaxed to offer flexibility for the factorization phase.

This paper is organized as follows. In Section 2, we briefly describe the parallelism involved in MUMPS. In Section 3, we then describe the constraints and objectives of our work. Section 4 introduces the quantities that will influence the dynamic decisions. Section 5 describes our dynamic scheduling algorithm in the context of unsymmetric matrices. Section 6 explains why the symmetric case is more complicated and shows how we extended our algorithms to this case. In Section 7 we present experimental results on large symmetric and unsymmetric matrices on 64 and 128 Power 4 processors of an IBM machine.

## 2 Scheduling and parallelism in the sparse solver

In this section, we describe the tasks arising in the factorization phase of a multifrontal algorithm and how parallelism can be exploited. The so called *elimination tree* [8, 15] represents the order in which the matrix can be factored, that is, in which the unknowns from the underlying linear system of equations can be eliminated. This graph is in the most general case a forest, but we will assume in our discussions, for the sake of clarity, that it is a tree. One central concept of the multifrontal approach [8] is to group (or *amalgamate*) columns with the same sparsity structure to create bigger *supervariables* or *supernodes* [8, 17] in order to make use of efficient dense matrix kernels. The amalgamated elimination tree is called the *assembly tree* (see Figure 2). The work associated with an individual node of the assembly tree corresponds to the factorization of a so called *frontal matrix*, or *front*. Frontal matrices can be partitioned as shown in Figure 1.

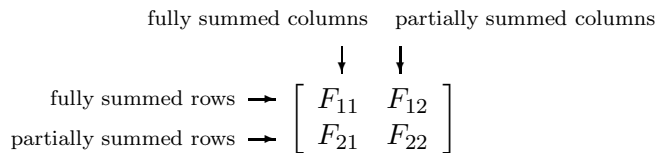


Figure 1: A frontal matrix.

Here, pivots can be chosen only from within the block of fully summed variables  $F_{11}$ . Once all eliminations have been performed, the Schur complement matrix  $F_{22} - F_{21}F_{11}^{-1}F_{12}$  is computed and used to update later rows and columns of the overall matrix which are associated with the parent nodes. We call this Schur complement matrix the *contribution block* of the node.

The notion of child nodes which send their contribution blocks to their parents leads to the following interpretation of the factorization process. When a node of the assembly tree is being processed, it assembles the contribution blocks from all its child nodes into its frontal matrix. Afterward, the pivotal variables from the fully summed block are eliminated and the contribution block computed. The contribution block is then sent to the parent node to be

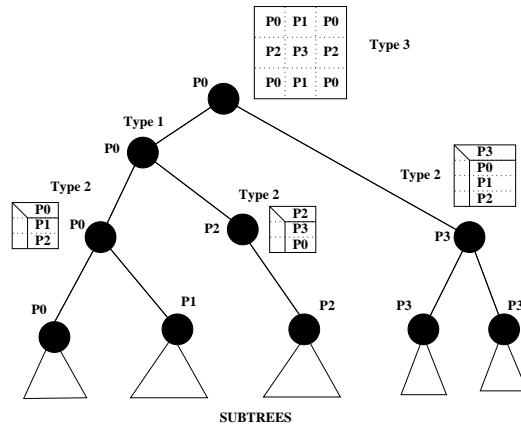


Figure 2: Different types of parallelism in the assembly tree.

assembled once all children of the parent (which are the siblings of the current node) have been processed. If some variables are not eliminated because of numerical issues, they are delayed and sent to the parent node.

A pair of nodes of the assembly tree where neither is an ancestor of the other can be factored independently from each other, in any order or in parallel. Consequently, independent branches of the assembly tree can be processed in parallel, and we refer to this as *tree parallelism* or *type 1 parallelism*. It is obvious that, in general, tree parallelism can be exploited more efficiently in the lower part of the assembly tree than near the root node.

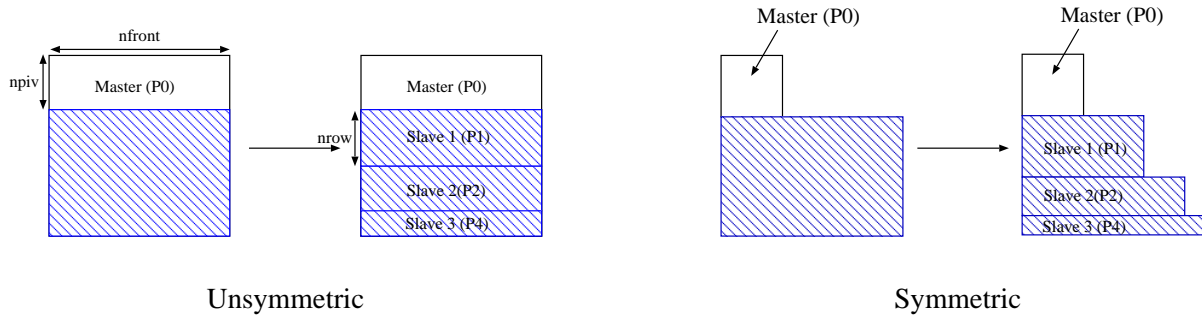


Figure 3: Distribution of slave tasks in symmetric and unsymmetric case.

Additional parallelism is then created using distributed memory versions of blocked algorithms to factor the frontal matrices (see, for example, [2, 6]). The contribution block is partitioned and each part of it is assigned to a different processor. The so called *master processor* is responsible for the factorization of the block of fully summed variables and will also decide (only during the numerical phase) how many and which processors (the so called *slave processors*) will be involved in the parallel activity associated with this node (see Figure 3). A slave receives the computation of a part of the *factors* (entries that intersect the fully summed column) that it will store and of a part of the *contribution block* (entries that intersect the partially summed column) that it will send for the computation of the parent node. We refer to this approach as *type 2 parallelism* and call the nodes concerned *type 2 nodes* (see Figure 2). The work distribution depends on the asymmetry of the matrix. This point explains why our algorithms are simpler in the unsymmetric case than in the symmetric case (see Section 6). Note also that in the

unsymmetric case, all communications are performed from the master to the slaves whereas in the symmetric case slaves have to communicate between each other (see [1]).

Of course, if the node is not large enough, it will not be split and will be a type 1 node. Finally, the factorization of the dense root node can be treated in parallel with ScaLAPACK [5]. The root node is partitioned and distributed to the processors using a 2D block cyclic distribution. This is referred to as *type 3 parallelism* (see Figure 2).

## 2.1 Partial task mapping during the symbolic factorization phase

The selection of slaves for type 2 nodes during the factorization phase is an attempt to detect and adjust a possible imbalance of the workload between the processors at runtime. However, it is necessary to carefully control the freedom given to dynamic scheduling (see [4] for a detailed analysis). Our sparse solver, MUMPS, addresses these issues by using the concept of *candidate processors*. This concept originates in an algorithm presented in [18, 19] and has also been used in the context of static task scheduling for sparse Cholesky factorization [13]. Each type 2 node is associated, during the symbolic factorization phase, with a limited set of candidate processors from which the slaves can be selected during numerical factorization. The candidate concept can be thought of as an intermediate step between fully static and fully dynamic scheduling. While we leave some freedom for dynamic decisions at runtime, this is directed by static decisions on the candidate assignment.

The assignment and the choice of the candidate processors is guided by a relaxed *proportional mapping* (see Pothén and Sun [18]). It consists of a recursive assignment of processors to subtrees according to their associated computational work. The assembly tree is processed top-down, starting with the root node. Each node gets its set of so called *preferential* processors which guides the selection of the candidate processors. A second bottom-up step maps not only the master tasks of type 2 nodes but also chooses the candidates for slave tasks of type 2 nodes using the previously computed preferential processors.

## 2.2 Dynamic task scheduling during the factorization phase

During the factorization, each processor maintains a local pool of ready tasks that corresponds to nodes of the tree statically assigned to it (type 1 and type 2 master tasks). Each time all children of a given parent node have been factored, the parent is inserted in the pool of the processor on which it was statically mapped. Tasks are then extracted from the pool and activated.

We associate to each processor, say  $p_i$ , its workload, referred to as  $load_i$ , that corresponds to the computational work (number of floating-point operations) associated with:

- each task in the pool of ready tasks, and
- each ongoing task (active tasks not yet finished).

For a better balance of the actual computational work during factorization, both the number and the choice of the slaves of type 2 nodes are determined *dynamically*. In an approach that is purely based on the workload (see [1, 2]), the master decides of a regular (balanced) distribution of the slave tasks (ie, each slave is assigned approximatively the same amount of work). The slaves involved in the factorization are selected based on their current workload, the least loaded processors being chosen from among the candidate processors of the node.

In [12], the authors developed an approach with irregular partitions to decrease the memory usage. We will also use this capability (actually we need it) to offer more flexibility to our new scheduling strategy. Remark that this flexibility can be exploited only if the memory

constraints guide us to control the size of the overall memory (communication buffers, factors, working memory) per processor. If we wanted to use irregular partition without any memory constraint, the overall memory per processor would have to be severely overestimated.

### 2.3 The four zones

The mapping algorithm by Geist and Ng [10] allows us to find a layer, so called *layer 0* or  $L_0$ , in the assembly tree so that the subtrees rooted at the nodes of this layer can be mapped onto the processors with a good balance of the floating-point operations associated. Initially, the assembly tree was separated into two zones, the upper part of the tree (above layer 0) and the bottom part of the tree (below layer 0) where each subtree is mapped onto a unique processor (type 1 parallelism).

We decided to separate the tree into 4 zones instead of 2 (see Figure 4). Zone 4 corresponds to the bottom of the tree. It was suggested in the conclusion of [4] that the mapping of the upper part of the assembly tree could be separated into two zones. The first zone (zone 1) would correspond to a relaxed proportional mapping whereas the second zone (zone 2) would correspond to a stricter proportional mapping. Hence the flexibility offered at the top of the tree would enable the master processors to correct the mistakes or the unbalance due to the variations of the load of the machine. Guided by this remark, zones 1 and 2 have been implemented.

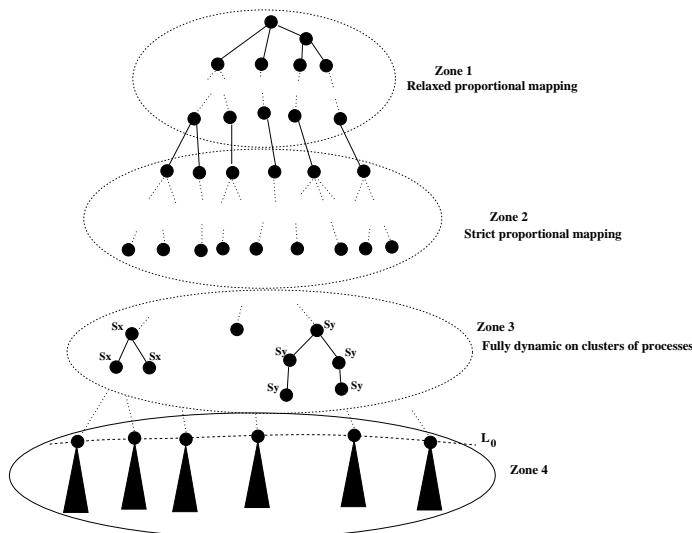


Figure 4: The four zones of the assembly tree.  $S_x$  and  $S_y$  are sets of preferential processors.

Moreover we decided to add another zone (zone 3) in which each child inherits all the preferential processors from its parent. This choice was motivated by the following experimental observations:

- on a small number of processors the fully dynamic code is very competitive,
- increasing the number of candidate processors near layer 0 makes the memory management easier (if a node with few candidates and a large contribution block appears, memory problems can occur),
- with more candidates above zone 4 we have more freedom to balance the work between processors while respecting the proportional mapping on layer 0,

- on clusters of SMPs, this will naturally take into account the memory locality.

The limit of zone 3 depends on a parameter  $proc_{max}$  which corresponds to a number of processors. During the top-down approach of the proportional mapping, if the number of preferential processors of a node  $x$  is smaller than or equal to  $proc_{max}$  then  $x$  and all its descendants (above zone 4) belong to zone 3 and have the same set of preferential processors (see sets  $S_x$  and  $S_y$  in Figure 4).

The extra freedom given in zone 1 does not perturb too much the memory estimation of the standard version of MUMPS (version 4.3), although it is based on a worst case scenario. It is not the case for zone 3. In this zone, the memory estimates behave like the fully dynamic code which has shown to severely overestimates the required space. That is why the four zones approach cannot be considered for the standard version of the experimental section (Section 7). We will give more details about memory estimates in Section 4.1.

### 3 Our constraints and objectives

When the target is time reduction, the master processor of each node determines a partition of the frontal matrix in order to “balance as much as possible the workload” between the processors. In our context, we have the same objective, but also have to respect additional constraints. In this section, we first present the memory constraints taken into account during scheduling. Then, we give a generic formulation of this new constrained problem.

The different criteria used to estimate the memory constraints are presented below:

- **Amount of available memory.** It corresponds to the remaining memory that can be used to store the contribution blocks and the factors. It varies during the factorization. For each processor  $p_i$ ,  $mem_i$  will refer to this quantity (see Section 4.1 for more details).
- **Maximum factor size.** It corresponds to the maximum size of the factors that a master can assign to a slave processor  $p_i$ . It will be denoted by  $fact_i$ . This quantity is related both to the static scenario used to estimate the memory during analysis and to the dynamic information obtained during factorization (see Section 4.2).
- **Maximum buffer size.** Let us consider a type 2 node. In the worst case, each slave will receive/send a block corresponding to the size of its share of the front. Assuming that the send and receive buffers are of the same size, it is therefore sufficient to ensure that the size of the slave task (front size multiplied by number of rows) is smaller than the size of the buffer. For each processor  $p_i$ ,  $buf_i$  will denote the size of its buffer.

We now define some of the notations used to describe our algorithms. Let us consider a node of the assembly tree of order  $nfront$  with  $npiv$  variables to eliminate (see Figure 3, left). The expressions below correspond to the unsymmetric case where the memory size of a slave task is given by the number of rows times  $nfront$  (see Section 6 for the symmetric case). For each slave processor  $p_i$  and for each corresponding constraint  $buf_i$ ,  $mem_i$ ,  $fact_i$ , we define a function  $nb\_row$  by

$$nb\_row(buf_i) = \frac{buf_i}{nfront}, \quad nb\_row(mem_i) = \frac{mem_i}{nfront}, \quad \text{and} \quad nb\_row(fact_i) = \frac{fact_i}{npiv},$$

respectively. Furthermore for a maximum number of floating-point operations  $flop_i$  that we want to assign, we define

$$nb\_row(flop_i) = \frac{flop_i}{npiv(2 \times nfront - npiv)}.$$

(The number of operations to factor a strip of  $nbrow$  rows is  $nbrow \times npiv \times (2 \times nfront - npiv)$ .) If the master of a node is aware of the above constraints, it can determine the maximum number of rows that it can assign to each candidate. During the factorization, each master has to “balance as much as possible the workload” between its “slave” candidates. For each slave  $p_i$ , the master processor gives  $n_i$  rows such that  $n_i \leq \min\{nb\_row(buf_i), nb\_row(mem_i), nb\_row(fact_i)\}$ . Even if the problem of finding the best workload balance which respects the memory constraints is easy to solve theoretically using linear programming, in practice, communication schemes, granularity and the topology of the assembly tree need to be considered to answer this question. That is why we do not give, in this section, more details about the meaning of “balance as much as possible the workload”. Note also that the above problem may not have a solution because the memory constraints are too restrictive. These points will be examined in Section 5.

## 4 Static and dynamic estimates

This section describes the metrics used by our scheduler. Some of these quantities are computed during the analysis (the memory estimations, the size of the buffers, the size of the area reserved for factors), some are also adjusted during the factorization (the memory available to store contribution blocks, the memory available to store factors). We have already described in Section 3 how the buffers are estimated; as for the memory estimation, a relaxation parameter is then used to increase freedom and allow for numerical pivoting.

### 4.1 Memory estimates and available memory

We first explain how we decrease the total memory allocated compared to the standard version of MUMPS. After choosing the candidates, the memory estimates are computed thanks to a bottom-up, depth-first traversal of the assembly tree, that simulates the actual factorization. (Note that the depth-first traversal may differ from the traversal occurring during the actual factorization.) For each processor, master or candidate, involved in the computation associated to a node, the memory estimate for the processor is decreased when a contribution block is assembled and discarded, and it is increased when assemblies, activation of tasks, or storage of factors occur. For type 2 nodes our new estimates, computed during the analysis phase, are based on an average optimistic scenario instead of the worst case as in [4]. In both cases, the estimation assumes regular partitions of the contribution block, that is, each slave is assigned the same amount of work.

In the worst case scenario corresponding to the standard version of MUMPS, we first compute the minimum number of slaves,  $min\_needed$ , to perform all the work for the slaves (this number depends on internal parameters and algorithmic aspects that fix the maximum granularity and it is smaller than the number of candidates). Then, for our simulation, each candidate receives a block of size  $nfront(nfront - npiv)/min\_needed$ , where  $nfront$  and  $npiv$  are respectively the size of the front and the number of variables to eliminate, as defined earlier.

In the optimistic case, we now assume that work can be assigned to all available slaves and so each candidate receives a block of size  $nfront(nfront - npiv)/ncand$  where  $ncand$  is the number of candidates of the type 2 node. Since  $nfront(nfront - npiv)/min\_needed \geq nfront(nfront - npiv)/ncand$ , the estimates will be smaller in this optimistic scenario.

For example, let us consider a type 2 node that has 5 candidates and that has a block of 200 MBytes to be distributed over the slaves. We suppose that at least 2 slaves are needed. Then the worst case will give 100 MBytes to each candidate processor whereas the optimistic scenario will give 40 MBytes to each slave. So we save 60 MBytes per processor.

These estimates are then relaxed by a percentage (by default equal to 20%) to offer more flexibility to the scheduler. The resulting supplementary memory enables us to take into account extra fill-in due to numerical pivoting and to offer more freedom to the dynamic decisions. It will also reduce the amount of data compressions involved in a parallel environment because of the irregular access to the contribution blocks (garbage collection). The memory available on each processor is then dynamically updated as proposed in [11].

## 4.2 Maximum size of factors

The maximum size of factors is used by the scheduler to determine the largest portion of the factors that can be given to a candidate processor. It is composed of two terms. For each node  $J$  of the assembly tree, the first term  $fact\_anal_i(J)$  is estimated during the analysis. It corresponds to the size of the factors given to processor  $p_i$  with the optimistic scenario (with the convention that if a processor  $p_i$  is neither the master nor a candidate of node  $J$ ,  $fact\_anal_i(J) = 0$ ). Thus, according to the analysis scenario, a processor  $p_i$  will store the quantity  $fact\_anal_i = \sum_J fact\_anal_i(J)$  of factors.

---

**Algorithm 1** Update of the supplementary memory for the factors.

---

**Initialization on processor  $p_i$ :**

Let  $\Delta_i$  be the initial supplementary memory for the factors.  
 Include  $(p_i, \Delta_i)$  in a message *msg\_sup\_mem*.  
 Asynchronous send of *msg\_sup\_mem* to the other processors.

---

**After the selection of a set  $\mathcal{S}$  of slaves by processor  $p_i$  for the node  $J$ :**

**for all** candidate processor  $p_k$  **do**  
   **if**  $p_k \in \mathcal{S}$  **then** /\* $p_k$  has been selected\*/  
      $\delta_k = fact\_anal_k(J) - \text{Actual size of the factor assigned to } p_k$   
   **else**  
      $\delta_k = fact\_anal_k(J)$   
   **end if**  
   Include  $(p_k, \delta_k)$  in a message *msg\_sup\_mem*.  
**end for**  
 Asynchronous send of *msg\_sup\_mem* to the other processors.

---

**At the reception of a message *msg\_sup\_mem*:**

**for all**  $(p_i, \delta_i)$  included in the message **do**  
    $\Delta_i = \Delta_i + \delta_i$   
**end for**

---

The second term, the flexibility  $\Delta_i$ , is initialized to the supplementary memory given to store the factors. It enables the dynamic decisions to deviate from the optimistic analysis scenario. Using Algorithm 1,  $\Delta_i$  is adjusted dynamically during the factorization phase. Hence, after having updated information about workload and memory during the selection of the slaves of a node  $J$ , the master knows that it should not give more than  $fact_i(J) = fact\_anal_i(J) + \Delta_i$  factors to the candidate  $p_i$ . Obviously, if there are no numerical problems and if  $\Delta_i = 0$  for each processor  $p_i$ , then the factorization will respect the same partition of the factors that was predicted during the analysis phase.

## 4.3 Workload and anticipation

The dynamic scheduling decision taken by the master processor of a type 2 task is guided by its view of the workload of its candidate processors. The workload for a processor  $p_i$  is referred to as  $load_i$  and it represents the sum of the computational cost of all its ready and active tasks (see Section 2.2). For each processor  $p_i$ ,  $load_i$  and its variations are made available

to master processors thanks to the asynchronous communication mechanism described in [11]. Experiments in [12] have shown the positive effects of anticipating the memory variations. Algorithm 2 describes this mechanism for the workload. The basic idea is to anticipate the arrival of a costly task and take its workload into account slightly before the task is effectively inserted in the pool of ready tasks. Note that a task becomes ready once all its children have been processed. Thus, if every processor treating a child sends (when it starts the task) 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 to make them aware of this new load just arriving. Note that when a predicted task effectively becomes ready (*i.e.* is inserted in the pool of ready tasks), the workload of the processor is not updated since this has already been done.

---

**Algorithm 2** Anticipation of the tasks.

---

Initialization on processor  $p_i$

**for all** nodes  $J$  for which  $p_i$  is the master **do**  
    Set  $nb\_children(J)$  to the number of children of  $J$ .  
**end for**

---

Emission of a message  $child\_OK$  when task  $J$  starts on the master  $p_i$ :

Let  $K$  be the parent of node  $J$ .  
Let  $p_k$  be the master in charge of task  $K$ .  
Include  $K$  in a message  $child\_OK$ .  
Asynchronous send of  $child\_OK$  to the processor  $p_k$ .

---

At the reception of a message  $child\_OK$  on processor  $p_i$ :

Extract task  $K$  from the message  $child\_OK$ .  
 $nb\_children(K) = nb\_children(K) - 1$ .  
**if**  $nb\_children(K) = 0$  **then**  
    Let  $W_K$  be the work associated with the task of the master of node  $K$ .  
    Include  $(p_i, W_K)$  in a message  $msg\_load\_update$ .  
    Asynchronous send of  $msg\_load\_update$  to the other processors.  
**end if**

---

At the reception of a message  $msg\_load\_update$  containing  $(p_i, W)$ :

$load_i = load_i + W$

---

## 5 Hybrid dynamic scheduling

In this section, we describe the algorithms used to balance the workload while taking into account the memory constraints. Let us first define the notations used in our algorithms. For a node  $J$  just extracted from the local pool of ready tasks (by the master processor of  $J$ ), we define:

- $ncand$ : the number of candidates,
- $\{p_1, \dots, p_{ncand}\}$ : the set of candidate processors initially sorted by increasing workload,
- $W_{master}$ : the computational cost (number of floating-point operations) of the master task,
- $W_{slaves}$ : the computational cost associated with the sum of all the slave tasks.

For each candidate processor  $p_i$ , we know/compute:

- the quantities  $mem_i$ ,  $buf_i$  and  $fact_i$  (see Section 3),

- its workload  $load_i$  (see Sections 2.2 and 4.3).

Algorithm 3 presents the main steps of our hybrid dynamic scheduling approach. Note that since violating a constraint related to  $buf_i$  or  $mem_i$  would lead to a failure we never relax them during the algorithm. At Step 1, we compute an advised maximum number of slaves  $nlim$  used during a first attempt to balance the work over a subset of the  $nlim$  least loaded processors.  $nlim$  is designed to limit the number of slaves by considering that the work given to each slave should be related to the master’s work (see Algorithm 4). Step 2 of our algorithm is performed only when Step 1 did not succeed in distributing all work because of memory constraints. Processors are then added one by one during Step 2 with the objective of mapping the remaining work on up to  $ncand$  processors, while saturating the memory constraints. At Step 3, we then suppress the memory constraint relative to  $fact_i$  and redistribute the remaining work.

---

**Algorithm 3** Main steps of our hybrid scheduling strategy.

---

Receive information related to workload and memory.

- 1 Try to balance the workload on a maximum of  $nlim$  processors (see Algorithm 4).
  - 2 If Step 1 did not succeed (the  $nlim$  processors are saturated in memory and work remains to be mapped) then add new slaves one by one.
  - 3 If Steps 1 and 2 did not complete the mapping then suppress the memory constraint on the size of the factors,  $fact_i$ , and try to balance the remaining work onto the candidates.
- 

All steps, although based on different algorithms, use similar techniques. In this section, we focus on Step 1, described in Algorithm 4, since it is the most complex and critical one for performance. Algorithm 4 is iterative. Starting from an initial value of  $ntry$ , we try to find a partition of the frontal matrix (inner loop in Algorithm 4) on a maximum of  $ntry$  slaves. If this attempt fails, we increase  $ntry$  and repeat the previous process to map the remaining work  $W_r$  until  $nlim$  is reached (outer loop in Algorithm 4). Furthermore note that in the algorithm we use the convention that  $load_{nlim+1} = +\infty$ , so that if  $ntry = nlim$ , only memory constraints may prevent us from finding a mapping of the slave tasks on the  $nlim$  slave processors. Obviously, during Step 2, the real value of  $load_{nlim+1}$  will be used.

The  $r_i$  term represents the number of rows assigned to processor  $p_i$  at each iteration of the algorithm. It corresponds to the minimal value between the number of rows necessary to reach the workload of the current *reference* processor ( $p_{ntry+1}$ ), the memory constraints of  $p_i$ , and another term ( $W_b = nb\_row(W_r / (ntry - nsat))$ ). This last term corresponds to a balanced distribution of the remaining work among unsaturated processors. Moreover, since at each point of the algorithm the workloads of the already selected slaves  $(p_i)_{1 \leq i \leq ntry}$  have been well-balanced (in the previous iterations of the algorithm), the order in which the  $ntry$  slaves are processed is not important. The  $ntry$  slaves are then sorted from the most constrained processor with respect to memory bounds to the least constrained one. Doing so we ensure that a single pass is sufficient to produce a balanced partition of the matrix that respects all the memory constraints. Indeed each time the memory constraints of a processor are saturated, the remaining load  $W_r$  is updated together with the number of saturated processors  $nsat$ . These two quantities are then used to reevaluate the term  $W_b = W_r / (ntry - nsat)$  and the corresponding constraint.

One can also notice that, in Algorithm 4, the initial values of  $ntry$  and  $nlim$  depend on the location in the assembly tree. Since in zone 3 the tree parallelism is sufficient, we expect to improve the performance and limit the volume of communications by allowing the algorithm to select less slaves. This is done by setting both  $nlim = ncand$  and  $ntry = ncand - 1$ . The workload of each processor will then be adjusted to the most loaded candidate processor during the first attempt of the “while” loop. Since the reference load  $load_{ntry+1}$  is equal to  $load_{ncand}$  and since  $\alpha = +\infty$  inhibits the limitation relative to the  $nb\_row(W_b)$  term, more work will be assigned to the first slaves and less slaves will normally be chosen. Furthermore, when the node

---

**Algorithm 4** Step 1: Slave task mapping for a node *Inode*

---

OUTPUT:

```
Wr: workload not yet mapped;
assigned_rows: number of rows assigned per processor;

nsat = 0 (number of saturated processors);
(assigned_rowsi)i=1,...,ncand = 0;
(buf_loci = bufi)i=1,...,ncand;
if Inode in zone 3 then
   $\alpha = +\infty$  (to inhibit the constraint on nb_row(Wb), see below);
  ntry = ncand - 1; nlim = ncand;
else
   $\alpha = 1$ ; ntry = 1; nlim = min(ncand, max( $\frac{W_{slave}}{\rho W_{master}}$ , 1));
end if
Wr = Wslaves; loadnlim+1 =  $+\infty$ 
while ntry  $\leq$  nlim do
  Wb = Wr / (ntry - nsat) (balanced distribution of the remaining work among first ntry - nsat unsaturated processors)
  if Inode not in zone 3 then
    sort the sublist of the ntry least loaded processors in increasing order of
      min{nb_row(memi), nb_row(buf_loci), nb_row(facti)};
  end if
  for i = 1 to ntry do
    if processor pi not already saturated then
      ri = min{nb_row(loadntry+1 - loadi),  $\alpha \times$  nb_row(Wb), nb_row(memi),
        nb_row(buf_loci), nb_row(facti)};
      Let w, m, f be the workload, the memory, the size of the factors (respectively) corresponding to ri rows;
      Assign ri rows to processor pi and update its configuration:
        loadi = loadi + w, memi = memi - m, facti = facti - f, assigned_rowsi = assigned_rowsi + ri,
        and buf_loci = buf_loci - m;
      Wr = Wr - w;
      if processor pi saturated then nsat = nsat + 1 and Wb = Wr / (ntry - nsat)
      if Wr = 0 then return;
    end if
  end for
  ntry = ntry + 1;
end while
```

---

belongs to zone 3 we do not sort the list of *ntry* processors in the inner loop because we do not expect, in that case, the workload between the slaves to be balanced.

To illustrate our discussion let us assume that the workload is very well balanced on entry of Algorithm 4 among the processors and consider a relatively small type 2 node not in zone 3. For the sake of simplicity, we assume that *W<sub>b</sub>* will be large enough and is not the constraining factor. In that case, the role of *nlim* is to maintain a minimum granularity relatively to the work of the master. Indeed, the first loops with *ntry* < *nlim* - 1 do nearly nothing since *load<sub>nlim</sub>*  $\approx$  *load<sub>i</sub>* for all candidate processors *p<sub>i</sub>*. When *ntry* becomes equal to *nlim*, the reference load *load<sub>ntry+1</sub>* becomes infinity so that we will try to saturate the memory before to consider adding new slaves at Step 2 of Algorithm 3. We now consider the example of Figure 5 to illustrate the behaviour of the first steps of Algorithm 3. Let us assume that at Step 1 the advised maximum number of processors *nlim* is set to 3. Work is assigned at Step 1 to these three processors. At the end of the while loop for *ntry* = 2, we show in Figure 5(b) that all the work has not been fully distributed. At the end of the next iteration (*ntry* = *nlim* = 3) we see in Figure 5(c) that the memory constraints of the three first processors are saturated. This is due to the fact that *load<sub>nlim+1</sub>* =  $+\infty$ . (Note that in our example, the constraint on *W<sub>b</sub>* is never attained.) Finally, in Figure 5(d), the remaining work can be assigned at Step 2 when *p<sub>4</sub>* is added to our set of

slaves.

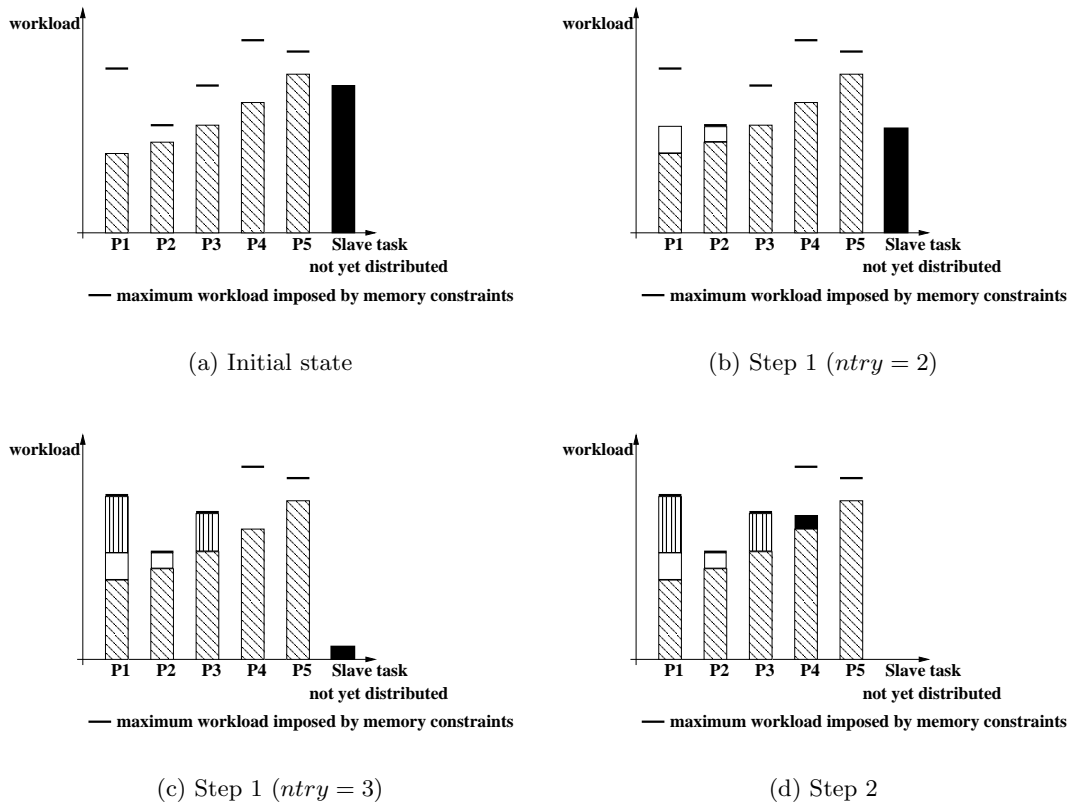


Figure 5: Example of hybrid scheduling on 5 processors of a node not belonging to zone 3.

## 6 Extension to symmetric matrices

The hybrid scheduling strategies have also been adapted to the symmetric case. The additional difficulties come from the fact that in the symmetric case, the relations between the memory, the computational cost (number of operations) and the number of rows inside a front is not straightforward; these relations actually depend on the choices done for the previous slaves of the node (see the partition of a type 2 node in Figure 3). In our dynamic approach to hybrid scheduling, the slave tasks are mapped in increasing order of the workload of the candidates. Hence, the evaluation of the function  $nb\_row$  of the  $i^{th}$  candidate depends on the rows assigned to the  $i - 1$  previous candidates. In particular, let us consider the  $i^{th}$  and the  $j^{th}$  candidate with  $i < j$ . An equal memory size corresponds to more rows given for  $p_i$  than to  $p_j$ . In this context, linear programming no more provides a theoretical answer to our problem.

Algorithm 4 needs to be revisited. In particular each time we increase  $ntry$ , the partition of the matrix needs to be fully recomputed. Also, if we want to add rows to the  $i^{th}$  candidate processor, then the assignment needs to be revised for all the candidates with position  $j > i$ . The  $nb\_row$  functions are then difficult to evaluate and are not explicitly known since the relation between memory and floating-point operations depends on the position in the frontal matrix of the first row assigned to processor  $p_i$ .

## 7 Experimental results

In this section we analyse the effects of our hybrid approach. We first describe our test environment. In Section 7.2, we analyse the behaviour of our algorithms in terms of estimated memory and memory effectively used during factorization. The influence on the factorization time is discussed in Section 7.3.

### 7.1 Experimental environment

In this section, we focus on four large symmetric and unsymmetric matrices, described in Table 1:

- AUDIKW\_1 comes from Automotive crankshaft model with over 900,000 TETRA elements and is available at <http://www.parallab.uib.no/parasol/data.html>,
- CONESHL comes from a 3D finite element problem (cone with shell and solid element connected by linear constraints with Lagrange multiplier technique). It was created by SAMCEF and provided by the SAMTECH company. It is available on request.
- CONV3D64 has been provided by CEA-CESTA and was generated using AQUILON (<http://www.enscpb.fr/master/aquilon>),
- ULTRASOUND80 comes from propagation of 3D ultrasound waves and has been provided by Masha Sosonkina.

We use METIS [14] during the reordering phase for all our experiments. Note that for the matrix CONESHL, the numerical behaviour of the current release of MUMPS is sensitive, because of numerical pivoting, to the number of processors. This results in a number of floating-point operations and a size for the factors that varies (although only slightly) when the number of processors changes. Since we do not want to be perturbed by such effects in the experiments of this paper, the matrix was made diagonal dominant to have the factorization cost independent of the number of processors. All results have been obtained with this modified matrix, referred to as CONESHL\_mod.

	Order	nnz	$nnz(L U) \times 10^6$	Ops $\times 10^9$
<b>Symmetric matrices</b>				
AUDIkw_1	943695	39297771	1368.6	5682
CONESHL_mod	1262212	43007782	790.8	1640
<b>Unsymmetric matrices</b>				
CONV3D64	836550	12548250	2693.9	23880
ULTRASOUND80	531441	33076161	981.4	3915

Table 1: Test set. nnz: number of nonzeros in the matrix.  $nnz(L|U)$ : number of nonzeros in the factors. Ops: total number of operations during factorization.

Our target machine is the IBM SP from IDRIS<sup>1</sup>. It is composed of clusters of SMP nodes. In our experiments, we use a maximum of 2 clusters of 16 SMP nodes of 4 processors each (Power 4/1.7Ghz P655). Each node shares 8 GBytes of memory. A Federation switch interconnects the SMP nodes. We will compare the following versions of MUMPS on 64 and 128 processors:

- The standard version with proportional mapping and candidates will be referred to as MUMPS\_cand. It corresponds to the version used in [4] except that the mechanism described in Algorithm 2 to anticipate the workload has also been included. A master selects its slaves among the candidate processors and balances the workload using regular partitions.

---

<sup>1</sup>Institut du Développement et des Ressources en Informatique Scientifique

- The hybrid version will be referred to as `MUMPS_hyb`. It corresponds to a candidate version implementing all the algorithms described in the previous sections. In particular, the separation of the tree in four zones (see Section 2.3), the estimation of the supplementary available memory for the factors (see Algorithm 1) and the hybrid scheduling (see Section 5) are included. To compute the advised number of processors  $nlim$  in Algorithm 3 we set  $\rho = 50\%$  for unsymmetric matrices and  $\rho = 70\%$  for symmetric matrices. The fact that the relative cost of a master with respect to a slave is larger on unsymmetric matrices justifies this difference in the setting.

## 7.2 Estimated and effective memory

In this section, we analyse the memory behaviour of both versions of our solver. We look at both the predicted memory peak and the memory actually used. We are interested in both the average memory per processor and the peak between processors.

Matrix		MUMPS_cand		MUMPS_hyb	
		Estim	Real	Estim	Real
AUDIKW_1	Max	118.96	50.08	74.80	62.53
	Avg	76.24	31.19	49.32	33.54
CONESHL_mod	Max	64.05	37.04	31.81	24.22
	Avg	25.20	16.79	22.05	16.82
CONV3D64	Max	102.94	93.04	88.73	87.46
	Avg	68.66	60.95	61.24	62.41
ULTRASOUND80	Max	42.98	38.02	34.17	32.11
	Avg	26.19	22.82	24.01	22.65

Table 2: Estimated and effective memory (millions of reals) for the factorization on 64 processors. Max: maximum amount of memory. Avg: average memory per processor. Memory in millions of entries. Memory allocated is 20% more than estimated.

We recall that with the `MUMPS_cand` version memory estimates are based on a worst case scenario whereas the `MUMPS_hyb` version uses a more optimistic scenario. Note that for all cases we relax the memory estimated by 20% to run the factorization (except for the `MUMPS_cand` version on `CONV3D64` and `AUDIKW_1` with 64 processors for which this percentage is reduced because of memory limitations on the machine). This leads anyway in all cases to a much larger memory allocated for `MUMPS_cand` than with `MUMPS_hyb` strategy.

Matrix		MUMPS_cand		MUMPS_hyb	
		Estim	Real	Estim	Real
AUDIKW_1	Max	107.09	33.49	59.54	29.22
	Avg	48.37	15.60	27.74	16.92
CONESHL_mod	Max	40.23	16.44	17.32	14.52
	Avg	14.90	8.44	12.16	8.69
CONV3D64	Max	74.30	56.17	49.86	47.35
	Avg	39.14	31.93	35.02	33.20
ULTRASOUND80	Max	45.95	23.90	21.26	17.47
	Avg	17.47	12.55	13.44	11.84

Table 3: Estimated and effective memory for the factorization (millions of reals) on 128 processors. Max: maximum amount of memory. Avg: average memory per processor. Memory in millions of entries. Memory allocated is 20% more than estimated.

Tables 2 and 3 show the memory estimated and the memory used on 64 and 128 processors respectively. We notice that the hybrid version significantly reduces the estimated memory (both average and peak). We can also see that the gap between the hybrid strategy and the standard one grows with the number of processors. This is due to the fact that on a larger number of processors we have less memory limitations so that we can offer more freedom to dynamic decisions. The worst case memory estimates of the standard strategy then converts this freedom into memory overestimations which is not the case with the new strategy. The same observation explains why on the large CONV3D64 matrix, the gap between the hybrid strategy and the standard one is relatively small on 64 processors. Indeed there is no over-estimation, even with worst-case memory estimates, for the large type 2 nodes because both strategies have to consider all the candidate processors as slaves to avoid too large slave tasks.

Concerning the effective memory occupation, we can observe that the hybrid strategy gives a maximum memory peak that is generally smaller than the one of the standard strategy (except for the AUDIKW\_1 problem on 64 processors). In addition, we can see that the average effective memory size over the processors is close between the two strategies and that the hybrid one tends to give slightly higher average memory occupation over the processors. Finally, notice that the gap between the estimated memory size and the effective one is smaller with the hybrid strategy and tends to be very small for most problems. The difference is generally due to the fact that the estimate is computing with the tree processed in a special order (depth-first traversal) which is not the one occurring during factorization.

### 7.3 Factorization time

In this section, we analyse the factorization time. We expect two different effects. First, irregular partitions offer the flexibility to balance the workload better, and this should improve the factorization time. Second, memory constraints may prevent the master from making a perfect decision in terms of balancing the workload, and this should moderate the benefits from irregular partitions.

Matrix	64 processors		128 processors	
	MUMPS_cand	MUMPS_hyb	MUMPS_cand	MUMPS_hyb
AUDIKW_1	99.95	78.50	59.51	43.51
CONESHL_mod	49.31	24.80	20.98	14.98
CONV3D64	304.90	239.62	240.02	168.59
ULTRASOUND80	46.02	42.52	35.96	33.85

Table 4: Factorization time in seconds with 64 and 128 processors.

Table 4 shows the impact of our new strategy on the factorization time. For all cases, the hybrid approach improves the factorization time on both 64 and 128 processors. In other words the memory constraints do not prevent the scheduling from balancing the workload well. We can see for example on problems like CONV3D64 or AUDIKW\_1 a very significant improvement in terms of factorization time (near to 30% reduction for matrix CONV3D64). Even, for problems requiring less computation, we can see that the new strategy is very efficient in comparison to the standard one (see CONESHL\_mod), except for ULTRASOUND80 for which gains are relatively minor. Our results show that thanks to the new hybrid dynamic scheduler and to the fact that it can be combined with a modified and improved static approach, we have freedom to schedule/manage better parallel tasks even under much tighter memory constraints.

Finally, concerning the scalability of the factorization time, we can observe that for symmetric problems, the relative speed-up between 64 and 128 processors is not too far from 2 (1.80 for

AUDIkw\_1 and 1.65 for CONESHL\_mod). Furthermore, we could factorize the symmetric matrices on one processor (3401 seconds for AUDIKW\_1 and 1195 seconds for CONESHL\_mod) and obtained ratios between the sequential and the parallel factorization time on 128 processors of 78.18 and 79.80 for respectively AUDIKW\_1 and CONESHL\_mod. This illustrates the good scalability of our scheduling strategies with symmetric matrices. Concerning unsymmetric problems, the speed-ups between 64 and 128 processors are not as good (1.45 for CONV3D64 and 1.25 for ULTRASOUND80). This is mainly due to the fact that for unsymmetric problems, the size of master tasks is relatively bigger than for symmetric ones (see Figure 3) and may become a bottleneck, since increasing the number of processors does not decrease the size of master tasks. Even if a splitting mechanism [1] is available in MUMPS to reduce the size of such tasks, it induces a lot of extra communications and some stronger synchronisation that affect the performance. Improving the splitting strategy is possible but is out of the scope of this study.

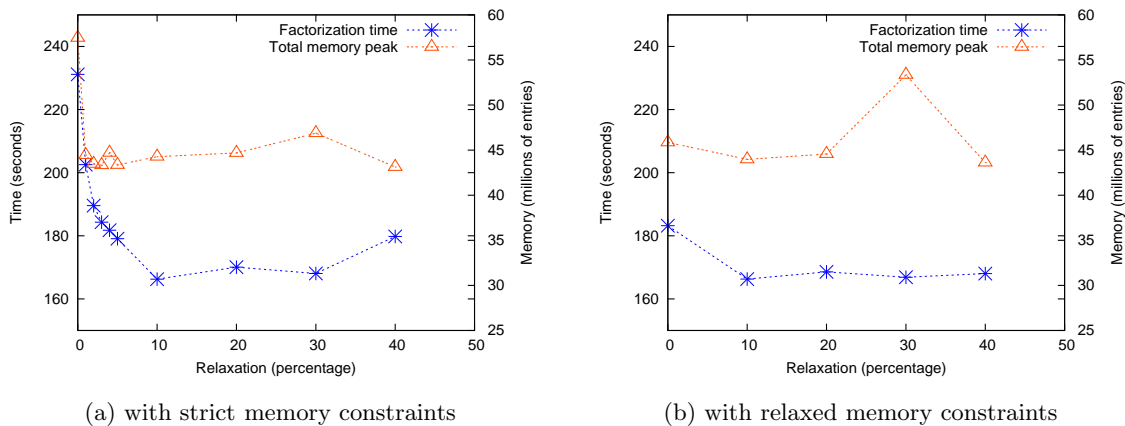


Figure 6: Influence of the memory relaxation on the factorization time and on the memory effectively used for matrix CONV3D64 on 128 processors.

To conclude this section, we report in Figure 6 the influence of relaxing the memory allocated (memory estimated  $\times$  percentage of relaxation) on the factorization time and on the memory effectively used to factor the matrix CONV3D64 on 128 processors. This study will also give us the opportunity to comment on the effect of an algorithmic tuning not yet introduced but already used in all previously presented results. The algorithmic modification results from the following observation. The memory constraint relative to the deviation of the size of the factors (see Section 4.2) makes sense only when the processor still has master tasks to process. Otherwise, all tasks that will be treated by the processor are slave tasks managed by our dynamic scheduler, and if there is enough redundancy between the candidate processors, the dynamic scheduler will be able to avoid choosing a processor that has no more memory available. It is then possible to deviate from the analysis for those processors and assign them more factors than authorized by our memory bounds.

Figure 6(a) shows that with strict memory constraints the memory relaxation parameter (whose default value is 20 %) influences both factorization time and the memory effectively used. We can observe that with no relaxation (*i.e.*, relaxation parameter equal to 0), both the factorization time and the memory peak increase because the scheduler has no freedom for its slave selection: all candidates processors are chosen and are assigned a task of the same size, following exactly the predictions from the analysis. It is interesting to notice that, in this case, the factorization time

(230 seconds) is comparable with the time obtained with the `MUMPS_cand` version (240 seconds). We also see in Figure 6(a) that for all values of this relaxation greater than 5 %, the scheduler has enough freedom since the curves have no significant variations. Note that the memory peak is even less sensitive to the value of the relaxation parameter.

Figure 6(b) shows that relaxing the memory constraints improves the performance obtained with small values of the relaxation parameter. Although a strategy with no memory relaxation at all should not be advised, we see that the behaviour is quite good. This is mainly because, in that case, several processors have not been assigned any master task initially. Thus, the dynamic hybrid scheduler can give more work to those processors, creating in this way extra freedom for the other processors. Furthermore, the number of processors with no incoming master tasks increases while processing the tree and will provide extra freedom to the schedulers. The curves of Figure 6(b) thus shows that with this simple modification of the management of memory constraints it is possible to make the factorization less sensitive to the memory relaxation parameter.

## 8 Concluding remarks and future work

We presented in this paper a hybrid approach to dynamic scheduling that takes into account information about both the workload and the memory availability of the processors in the context of the parallel  $LU$  and  $LDL^T$  multifrontal factorizations. We proposed modifications concerning the static mapping of computational tasks, as well as a new scheduler combining workload and memory constraints for its dynamic decisions. We have shown the benefits of our approach on four large test cases (symmetric and unsymmetric) on 64 and 128 processors. For our future work, we plan to further improve the parallel behaviour of our approach following two directions.

Firstly, the candidate version of MUMPS has been adapted to clusters of SMPs in [3]. We want to adapt our hybrid approaches to better exploit this feature of the computer architecture. In our context, it seems already natural that the size of the SMP nodes will give a criterion to define the size of the clusters in zone 3. The information about the SMP nodes could then be used to influence the dynamic scheduler in its decisions (*i.e.*, try to select as slaves processors belonging to a same SMP node).

Secondly the task selection strategy that manages the ready nodes on each processor (see Section 2.2) can be improved. Indeed, the current strategy is local and greedy. It can be improved by designing more sophisticated strategies based on workload, topological and memory criteria. An example of such a strategy could be to select among all ready tasks on the most costly (in terms of computation) branches of the assembly tree the one that is best at reducing the current global memory peak.

## References

- [1] 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.
- [2] 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.
- [3] P. R. Amestoy, I. S. Duff, S. Pralet, and C. Vömel. Adapting a parallel sparse direct solver to architectures with clusters of smps. *Parallel Computing*, 29(11-12):1645–1668, 2003.

- [4] P. R. Amestoy, I. S. Duff, and C. Vömel. Task scheduling in an asynchronous distributed memory multifrontal solver. *SIAM Journal on Matrix Analysis and Applications*, 2004.
- [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. *Computer Physics Communications*, 97:1–15, 1996. (also as LAPACK Working Note #95).
- [6] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. SIAM Press, Philadelphia, 1998.
- [7] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, London, 1986.
- [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 and J.-Y. L’Excellent. Coherent load information mechanisms for distributed dynamic scheduling. Technical Report RR2004-25, LIP, 2004. Also INRIA report RR5178.
- [12] A. Guermouche and J.-Y. L’Excellent. Memory-based scheduling for a parallel multifrontal solver. In *18th International Parallel and Distributed Processing Symposium (IPDPS’04)*, 2004.
- [13] P. Hénon, P. Ramet, and J. Roman. PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. *Parallel Computing*, 28(2):301–321, January 2002.
- [14] G. Karypis and V. Kumar. METIS – *A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices – Version 4.0*. University of Minnesota, September 1998.
- [15] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11:134–172, 1990.
- [16] J. W. H. Liu. The multifrontal method for sparse matrix solution: Theory and Practice. *SIAM Review*, 34:82–109, 1992.
- [17] J. W. H. Liu, E. G. Ng, and W. Peyton. On finding supernodes for sparse matrix computations. *SIAM Journal on Matrix Analysis and Applications*, 14:242–252, 1993.
- [18] A. Pothen and C. Sun. A Mapping Algorithm for Parallel Sparse Cholesky Factorization. *SIAM Journal on Scientific Computing*, 14(5):1253–1257, 1993.
- [19] P. Raghavan. *Distributed sparse matrix factorization: QR and Cholesky decompositions*. Ph.D. thesis, Department of Computer Science, Pennsylvania State University, 1991.