A study of various load information exchange mechanisms for a distributed application using dynamic scheduling.

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

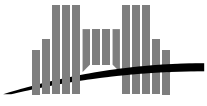
A study of various load information exchange mechanisms for a distributed application using dynamic scheduling

Abdou Guermouche,
Jean-Yves L'Excellent

January 2005

Research Report N° 2005-02

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`



A study of various load information exchange mechanisms for a distributed application using dynamic scheduling

Abdou Guermouche, Jean-Yves L'Excellent

January 2005

Abstract

We consider a distributed asynchronous system where processes can only communicate by message passing and need a coherent view of the load (e.g., workload, memory) of others to take dynamic decisions (scheduling). We present several mechanisms to obtain a distributed view of such information, based either on maintaining that view or demand-driven with a snapshot algorithm. We perform an experimental study in the context of a real application, an asynchronous parallel solver for large sparse systems of linear equations.

Keywords: Snapshot, distributed system, dynamic scheduling, load balancing, message passing

Résumé

Nous considérons un système distribué et asynchrone où les processus peuvent seulement communiquer par passage de messages, et requièrent une estimation correcte de la charge (travail en attente, mémoire utilisée) des autres processus pour procéder à des décisions dynamiques liées à l'ordonnancement des tâches de calcul. Nous présentons plusieurs types de mécanismes pour obtenir une vision distribuée de telles informations. Dans un premier type d'approches, la vision est maintenue grâce à des échanges de messages réguliers ; dans le deuxième type d'approches (mécanismes à la demande ou de type *snapshot*), le processus demandeur des informations émet une requête, et reçoit ensuite les informations de charge correspondant à sa demande. Nous expérimentons ces approches dans le cadre d'une application réelle utilisant des ordonnanceurs dynamiques distribués.

Mots-clés: *Snapshot*, système distribué, ordonnancement dynamique, équilibrage de charge, passage de messages

Introduction

Scheduling tasks in distributed systems is crucial for many applications. The scheduling process can be either static or dynamic (based on dynamic information), distributed or centralized. Here we are interested in a distributed asynchronous system where processes can only communicate by message passing and need an as exact as possible view of the state (e.g., workload, memory) of others to take dynamic scheduling decisions. Therefore, mechanisms need to be designed to provide that view when needed to proceed to a dynamic decision. Those mechanisms can be divided into two classes.
The first class (discussed in Section 2) consists in maintaining the view of the load information during the computation: when quantities vary significantly, processes exchange information and maintain an approximate view of the load of the others.
The second class of approaches (Section 3) is more similar to the distributed snapshot problem of [4] and is demand-driven: a process requiring information (to proceed to a scheduling decision) asks for that information to the others. Although less messages are involved, there is a stronger need for synchronization. In this paper, we discuss possible algorithms for those two classes of approaches in our context, and compare their impact on the behaviour of a distributed application using dynamic scheduling strategies.

1 Context

We consider a distributed asynchronous system of N processes that can only communicate by message passing. An application consisting of a number of (dependent or independent) tasks is executed on that system. From time to time, any process P (called *master*) needs to send work to other processes. The choice of the processes (called *slaves*) that will receive work from P is based on an estimate that P has of the *load* (workload, memory, ...) of others. For that, the estimates of the loads should be as accurate and coherent as possible. Note that load information on a process P varies in the following cases: (i) when P processes some work (less work waiting to be done, temporary memory freed at the end of a task), or (ii) when a new task appears on P (that can either come from the application or from another process).
In our case, we also have the property that the quantities we need to estimate are very much linked to the dynamic decisions taken. The algorithms presented in this paper aim at providing state information about the system that will be used to take distributed dynamic scheduling decisions. Furthermore, we assume that a process cannot treat a message and compute simultaneously. To fix the ideas, a simplified model for our asynchronous distributed application is given by Algorithm 1.

Algorithm 1 Main algorithm of the considered application.

```
1: while Global termination not detected do
2:   if a message of type state information is ready to be received then
3:     Receive and process the message (load information, load increment, demand for snapshot,
       ...);        (1)
4:   else if another message is ready to be received then
5:     Receive and process the message (task, data, ...);
6:   else
7:     Process a new local ready task. If the task is parallel, proceed to a slave selection (i.e. dynamic
       scheduling decision) and send work to other processes;
8:   end if
9: end while
```

The mechanisms we study/propose in this paper are based on message passing. In the first approach, each process broadcasts information when its state changes. Thus, when a process has to take a dynamic decision (we call this type of dynamic decisions a *slave selection* in the rest of this paper), it already has a view of the state of the others. Indeed the goal is to maintain an approximative snapshot of the load information. A condition to avoid a too incoherent view is

to make sure that all pending messages related to load information are received before taking a decision of sending work to others. This is the case in the context of Algorithm 1 (see (1) in the algorithm).
The second solution to this problem is close to the distributed snapshot approach [4, 8], where the snapshot is demand-driven and initiated by the process that requires information from the others. This approach avoids the cost of maintaining the view during the computations, but loses some of the asynchronous properties of the application. Indeed, when a process requires information from the others, it has to wait for all others to be ready to send that information. Furthermore, since in our case the information is strongly linked to the dynamic scheduling decisions taken, two simultaneous snapshots should be sequentialized so that the second one takes into account the slave selection resulting from the first one. This will be discussed in more detail in Section 3.
Coming back to Algorithm 1, note that all messages discussed in this paper are of type *state information*, and they are processed in priority compared to the other messages. In practice a specific channel is used for those messages.

2 Maintaining a distributed view of the load

2.1 Naive mechanism

In this mechanism, described by Algorithm 2, each process P_i is responsible of knowing its own load; for each significant variation of the load, the absolute value of the load is sent to the other processes, and this allows them to maintain a global view of the load of the system. A threshold mechanism ensures that the amount of messages to exchange load information remains reasonable.

Algorithm 2 Naive mechanism to exchange load information.

Initialization
1: $last\_load\_sent = 0$;
2: Initialize(my_load);
When my_load **has just been modified:**
3: **if** $|my\_load - last\_load\_sent| > threshold$ **then**
4: **send** (in a message of type *Update*, asynchronously) my_load to the other processes;
5: $last\_load\_sent = my\_load$;
6: **end if**
At the reception of load l_j **from** P_j **(message of type *Update*):**
7: $load(P_j) = l_j$;

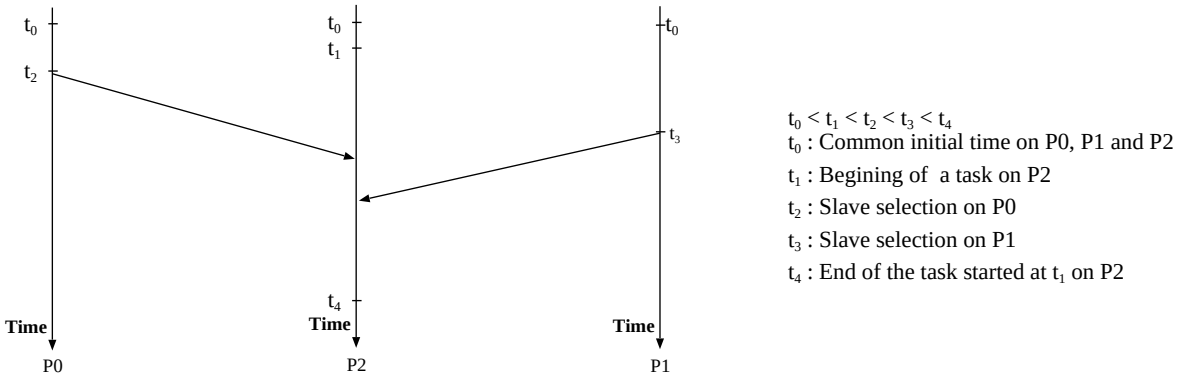


Figure 1: Example using the naive mechanism that illustrates the problem of the correctness of load information.

The local load l_i should be updated on the local process regularly, at least when work is received from another process, when a new local task becomes ready (case of dependent tasks), and when a significant amount of work has just been processed.

Limitations

Some problems can arise with the mechanism described above for the dynamic scheduling parts of our system. Indeed, with this mechanism, if several successive slave selections occur, there is nothing to ensure that a slave selection has taken into account the previous ones. Thus, a slave selection can be done based on invalid information and this can lead to critical situations (in practice, large imbalance of the workload or critical increase of the memory).

Figure 1 gives an illustration of the problem. In this example, P_2 is chosen twice as a slave (first by P_0, then by P_1). In addition, P_2 has started a costly task at time t_1. Thus P_2 might not be able to receive the subtask from P_0 before the end of that task. As a result, P_2 that does not know yet that it has been chosen by P_0, cannot inform the others. P_1, which is the second process that has to select slaves, will then select P_2 without taking into account the amount of work already sent by P_0. This simple example exhibits the problem of the coherence of the information exchanged by the processes.

2.2 Mechanism based on load increments

In this section we present another mechanism based on load increments to improve the correctness of the load information during the execution, and avoid situations like in Figure 1. Each time a process selects slaves, it sends (to all processes) a message of type *Master_To_All* containing the identity of the slaves and the amount of workload/memory assigned to each of them (it is a kind of reservation mechanism). At the reception of a message of this type, each process updates its local information on the processes concerned with the information contained in the message.

A formal description of the mechanism is given in Algorithm 3. For each variation of the workload on a process P_i, P_i broadcasts the increment representing the variation in a message of type *update*. Again, a threshold mechanism is applied to avoid too many messages: $\Delta load$ accumulates smaller $\delta load$ increments and is sent when larger than the threshold.

Note that when a (slave) process starts a task that was sent by another, it need not broadcast a message of type *Update* if the increment is positive: the master has already sent the information relative to its selected slaves (see (1) in Algorithm 3).

2.3 Reducing the number of messages

To control the number of messages, the threshold should be chosen adequately. For example it is consistent to choose a threshold of the same order as the granularity of the tasks appearing in the slave selections. The number of messages will increase with the number of processes, since we basically broadcast a message to all processes for each load variation in the system. However, some processes may never be master and never send work to others; this information may be known statically. Those processes do not need any knowledge of the workload/memory of the others. More generally, if at some point a process P_i knows that it will not proceed to any further slave selection in the future, it can inform the others. After P_i has performed its last slave selection, it can thus send a message of type *No_more_master* to the other processes (including to processes which are known not be master in the future). On reception of a message of type *No_more_master* from P_i by P_j, P_j stops sending load information to P_i. Note that the experiments presented later in this paper use this mechanism. Typically, we observed that the number of messages could be divided by 2 in the case of our test application, `MUMPS`.

Algorithm 3 Mechanism based on load increments.

Initialization
1: $my\_load = 0$;
2: $\Delta load = 0$;
When my load varies of $\delta load$:
3: **if** $\delta load$ concerns a task where I am slave **then**
4: **if** $\delta load > 0$ **return**; **(1)**
5: **end if**
6: $my\_load = my\_load + \delta load$;
7: $\Delta load = \Delta load + \delta load$;
8: **if** $\Delta load > threshold$ **then**
9: **send** $\Delta load$ (in a message of type *Update*, asynchronously) to the other processes;
10: $\Delta load = 0$
11: **end if**
At the reception of load increment Δl_j from processor P_j (message of type *Update*):
12: $load(P_j) = load(P_j) + \Delta l_j$;
At each slave selection on the master side:
13: **for all** P_j **in** the list of selected slaves **do**
14: Include in a message of type *Master_To_All* the load δl_j assigned to P_j;
15: **end for**
16: **send** (asynchronously) the message *Master_To_All* to the other processes;
At the reception of a message of type *Master_To_All*:
17: **for all** $(P_j, \delta l_j)$ **in** the message **do**
18: **if** $P_j \neq$ myself **then**
19: $load(P_j) = load(P_j) + \delta l_j$;
20: **else**
21: $my\_load = my\_load + \delta l_j$
22: **end if**
23: **end for**

3 Exact Algorithm

In this section we present another way to provide the information needed by the processes to take their scheduling decisions. This scheme is demand-driven and based on a classical distributed snapshot mechanism, coupled to a distributed leader election algorithm. Each time a process has to take a dynamic decision that can modify the state of the others, it initiates a snapshot. After the completion of the snapshot, it can take its dynamic decision, inform the others about its choice (message *master_to_slave* to the processes that have been selected as slaves) and finally restart the others. A more formal description of this scheme is given in Algorithm 4. Note that on reception of a message *master_to_slave*, a processor updates its state information (load) with the information contained in that message, so that the result of a first slave selection is taken into account if another snapshot is initiated from another process. Apart from that particular case, a processor is responsible for updating its own load information regularly.

Algorithm 4 Context in which the snapshot algorithm is applied.

1: Initiate a snapshot (see below)
2: Proceed to a dynamic slave selection
3: **for all** *islave* slave chosen **do**
4: Send a message of type *master_to_slave* to *islave* containing information to update its state (typically share of the work
5: **end for**
6: Finalize the snapshot (see below)

The algorithm we use to build the snapshot of the system is similar to the one proposed by Chandy and Lamport [4]. In addition, since we are in a distributed system, several snapshots may

be initiated simultaneously. They are in that case "sequentialized" to ensure that each process needing a snapshot takes into account the variation of the state (i.e. workload, available memory, etc ...) of the processes chosen during the previous dynamic decision. For that, a distributed leader election [6, 11], based for example on process ranks, is performed. The process elected is the one that will complete its snapshot first. After the termination of the snapshot of the leader, a new leader election is done within the set of processes having already initiated a snapshot. The algorithm given here is based on message passing between the processes. A first step consists in initializing the data structures that will be used during the execution to manage the snapshot mechanism:

Initialization:

1: *leader* =*undefined /*current leader*/*
2: $nb\_snp = 0$ */*number of concurrent snapshots except myself*/*
3: *during_snp = false /*flag saying if I think that I am the current leader*/*
4: *snapshot = false /*flag saying if there is an active snapshot for which I am not leader*/*
5: **for** $i = 1$ **to** *nprocs* **do**
6: $request(P_i) = 0$ */*request identifier*/*
7: $snp(P_i) = false$ */*array of flags saying if a processor has initiated a snapshot*/*
8: $delayed\_message(P_i) = false$ */*array of flags saying if I delayed the sent of a message to a processor*/*
9: **end for**

The rest of the algorithm uses principally three types of messages: *start_snp*, *snp* and *end_snp*. When a process initiates a snapshot, it broadcasts a message of type *start_snp*. Then it waits for the information relative to the state of all the others. Note that if there are several snapshots initiated simultaneously, a "master" (i.e. process that initiates a snapshot) may have to broadcast a message of type *start_snp* several times with different request identifiers to be able to gather a correct view of the system, in the case where it was not the leader among the "master" processes.

Initiate a snapshot:

1: *leader* = *myself*
2: $snp(myself) = true$
3: *during_snp* = *true*
4: **while** $snp(myself) == true$ **do**
5: $request(myself) = request(myself) + 1$
6: **send** asynchronously a message of type *start_snp* containing $request(myself)$ to all the others
7: $nb\_msgs = 0$
8: **while** $nb\_msgs \neq nprocs - 1$ **do**
9: **receive** and treat a message
10: **if** $during\_snp == false$ **then**
11: *during_snp* = *true*
12: $nb\_msgs = 0$
13: $request(myself) = request(myself) + 1$
14: **break**
15: **end if**
16: **end while**
17: **if** $nb\_msgs == nprocs - 1$ **then**
18: $snp(myself) = false$
19: **end if**
20: **end while**

After receiving the load information from all other processes, the process that initiated the snapshot can proceed to a scheduling decision (see Algorithm 4), and update the load information resulting from that decision. After that (see the algorithm below), it informs the other processes that its snapshot is finished (message of type *end_snp* and waits for other snapshots in the system to terminate.

Finalize the snapshot:

1: **send** asynchronously a message of type *end_snp* to all other processes
2: *leader* = *undefined*
3: **if** $nb\_snp \neq 0$ **then**

```
4:      snapshot = true
5:      for i=1 to nprocs do
6:        if snp(P_i) == true then
7:          leader = elect(P_i, leader)
8:        end if
9:      end for
10:     if delayed_message(leader) == true then
11:       send asynchronously my state and request(leader) to leader in a message of type snp
12:       delayed_message(leader) = false
13:     end if
14:     while nb_snp ≠ 0 do
15:       receive and treat a message
16:     end while
17:   end if
```

When a process P_j receives a message of type *start_snp* from a process P_i (see the algorithm below), it can either ignore the message (if P_j is the current leader, see lines 7-10), either send a message of type *snp* that contains its state (lines 14 or 20), or delay the message to avoid an inconsistency in the snapshot. This last case can occur if P_j detects that P_i is not the leader (line 17) or because of asynchronism.

To give an example showing how asynchronism can be managed, consider a distributed system with three processes P_1, P_2, P_3, where P_1 receives a message *start_snp* from P_3 and P_2 in that order. P_1 first answers to P_3 and then to P_2 which is the leader (we assume that the leader is the process with smallest rank). When P_2 completes its snapshot, suppose that P_3 receives *end_snp* from P_2 before P_1. In addition, suppose that P_3 reinitiates a snapshot (sending a message of type *start_snp*) and that P_1 receives the *start_snp* message from P_3 before *end_snp* from P_2 arrives. Then P_1 will not answer to P_3 until it receives the message *end_snp* from P_2. This ensures that the information sent from P_1 to P_3 will the variation of the state information induced by the dynamic decision from P_2. Such a situation may occur in case of heterogeneous links between the processes.

Note that the algorithm is recursive. After the first reception of a message of type *start_snp*, the process does not exist from the algorithm until all snapshots have terminated (lines 25-27 in the algorithm below and lines 14-16 in the previous one). Note that we avoid more than one level of recursivity.

At the reception of a message *start_snp* **from** P_i **with request number** *req*:

```
1:    leader = elect(P_i, leader)
2:    request(P_i) = req
3:    if snp(P_i) == false then
4:      nb_snp = nb_snp + 1
5:      snp(P_i) = true
6:    end if
7:    if leader == myself then
8:      delayed_message(P_i) = true
9:      return
10:   end if
11:   if snapshot == false then
12:     snapshot = true
13:     leader = P_i
14:     send asynchronously my state and request(P_i) to P_i in a message of type snp
15:   else
16:     if leader ≠ P_i or delayed_message(P_i) == true then
17:       delayed_message(P_i) = true
18:       return
19:     else
20:       send asynchronously my state and request(P_i) to P_i in a message of type snp
21:     end if
22:   end if
```

```
23:  if nb_snp == 1 then /*loop on receptions for the first start_snp message (if nb_snp is greater than
     1, I am already waiting for the completion of all the snapshots)*/
24:    during_snp = false
25:    while snapshot == true do
26:      receive and treat a message
27:    end while
28:  end if
```

On the other hand, when a process receives a message of type *end_snp*, it checks if there is another active snapshot in the sytem (different from the sender of the message). If not, the receiving process exits and continues its execution. Otherwise, it sends its state information only to the process viewed as the leader (*leader*) of the remaining set of processes that have initiated a snapshot. It stays in snapshot mode (*snapshot* =*true*) until all ongoing snapshots have completed.

At the reception of a message of type *end_snp* **from** P_i**:**

```
1:   leader =undefined
2:   nb_snp = nb_snp − 1
3:   snp(P_i) = false
4:   if nb_snp == 0 then
5:     snapshot = false
6:   else
7:     for i=1 to nprocs do
8:       if snp(P_i) == true then
9:         leader = elect(P_i, leader)
10:      end if
11:    end for
12:    if leader == myself then
13:      return
14:    end if
15:    if delayed_message(leader) == true then
16:      send asynchronously my state and request(leader) to leader in a message of type snp
17:      delayed_message(leader) = false
18:    end if
19:  end if
```

Finally, when a "master" process receives a message of type *snp* from another one, it first checks that the request identifier contained in the message is equal to its own. In that case, it stores, the state of the sender. Otherwise, the message is ignored since there is in that case no guaranty about the validity of the information received.

At the reception of a message of type *snp* **from** P_i **with request id** *req***:**

```
1:   if  req == request(myself) then
2:     nb_msgs = nb_msgs + 1
3:     Extract the state/load information from the message and store the information for P_i
4:   end if
```

4 Application to a distributed sparse matrix solver

In this section, we suppose that the target platform is dedicated to a single application. However things could be extended to the case of several applications sharing the same platform and/or to heterogeneous platforms by using load quantities nearer to the operating system load measurements or dynamic information on the processor current speed. We focus here on exchanging memory and workload information.

In Section 4.1 we present the software package `MUMPS` [1, 2] and show how it fits with the distributed system presented earlier. Both workload-based and memory-based strategies are described (Section 4.2), aiming at respectively optimizing the time of execution of the complete graph of tasks or balancing the memory over the processors.

In Section 4.3 we compare the behaviour of that application for the three algorithms presented earlier to exchange load and memory information.

4.1 Task graph within MUMPS

MUMPS uses a combination of static and dynamic approaches. The tasks dependency graph is indeed a tree (also called *assembly tree*), that must be processed from the leaves to the root. Each node of the tree represents the partial factorization of a dense matrix called *frontal matrix* or *front*. The shape of the tree and costs of the tasks depend on the problem solved and on the reordering of the unknowns of the problem. Furthermore tasks are generally larger near to the root of the tree where parallelism of the tree is limited. Figure 2 summarizes the different types of parallelism available in MUMPS:

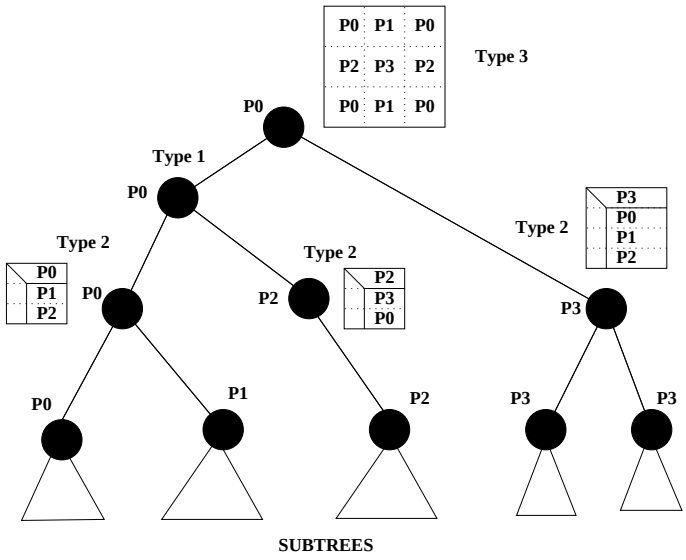


Figure 2: Example of distribution of a multifrontal assembly tree over four processors.

The first type only uses the intrinsic parallelism induced by the tree (since each branch of the tree can be treated in parallel). A type one node is a sequential task, that can be activated when results from children nodes have been communicated. Leave subtrees are a set of tasks all assigned to the same processor. The second type corresponds to parallel tasks; a 1D parallelism of large frontal matrices is applied: the front is distributed by blocks of rows. A *master* processor is chosen statically during the symbolic preprocessing step, all the others (*slaves*) are chosen dynamically by the master based on load balance considerations, which can be either the number of floating-point operations still to be done, or the memory usage. Note that in the partial factorization done, the *master* processor is eliminating the first block of rows, while slaves perform the updates on the remaining Schur complement. Finally, the task corresponding to the root of the tree uses a 2D parallelism, and does not require dynamic decisions: ScaLAPACK [5] is applied, with a 2D block cyclic static distribution.
The choice of the type of parallelism is done statically and depends on the position in the tree, and on the size of the frontal matrices. The mapping of the masters of parallel tasks is static and only aims at balancing the memory of the corresponding factors. Usually, parallel tasks are high in the dependency tree (fronts are bigger), and on large enough numbers of processors, about 80% of the floating-point operations are performed in slave tasks. During the execution, several slave selection strategies can be made independently by different master processors.

4.2 Dynamic scheduling strategies

The two following scheduling heuristics will be used to illustrate the behaviour of the load information exchange mechanisms. We chose them because there offer more freedom to the schedulers

and might be more sensible to the accuracy of load information than the approach available in the public version of MUMPS.

4.2.1 Case 1: memory-based scheduling strategy

We presented in [7] memory-based dynamic scheduling strategies for the parallel multifrontal method as implemented in MUMPS. These strategies are a combination of a memory-based slave selection strategy and a memory-aware task selection strategy. The slave processors are selected with the goal to obtain the best memory balance, and we use an irregular 1D-blocking by rows for both symmetric and unsymmetric matrices. Concerning the task selection strategy, the management is also memory-aware in the sense that we do not select a ready task if memory balance will suffer too much from this choice.
These dynamic strategies need to have a view as correct as possible of the state of each process taking part to the factorization. Indeed, the slave selection strategy chooses slaves based on the information provided by the mechanisms described above. The task selection strategy depends on the mechanism that provides the information about the system to compute the memory constraints that will be used during the slave selection.

4.2.2 Case 2: workload-based scheduling strategy

This strategy [3] is based on the floating-point operations still to be done. Each processor takes into account the cost of a task once it can be activated. In addition, each processor has as initial load the cost of all its subtrees.
The slave selection for parallel tasks (Type 2 nodes), is done such that the selected slaves give the best workload balance. The matrix blocking for these nodes is an irregular 1D-blocking by rows . In addition, there are granularity constraints on the sizes of the subtasks for issues related to either performance or size of some internal communication buffers. Furthermore, this strategy dynamically estimates and uses information relative to the amount of memory available on each processor to constrain the schedulers. More details on this strategy will be given in a future technical report.

4.3 Experimental study of the load exchange mechanisms

We should first mention that the mechanisms described in Sections 2.1, 2.2 and 3 have been implemented inside the MUMPS package. In fact, the mechanism from Section 2.1 used to be the one available in MUMPS, while the mechanism of Section 2.2 is the default one since MUMPS version 4.3. In order to study the impact of the proposed mechanisms, we experiment them on several problems (see Tables 1 and 2) extracted from various sources including Tim Davis's collection at 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 either 4 processors at 1.7 GHz or 32 processors at 1.3 GHz.

Matrix		Order	NZ	Type	Description
BMWCRA_1	(PARASOL)	148770	5396386	SYM	Automotive crankshaft model
GUPTA3	(Tim Davis)	16783	4670105	SYM	Linear programming matrix (A*A')
MSDOOR	(PARASOL)	415863	10328399	SYM	Medium size door
SHIP_003	(PARASOL)	121728	4103881	SYM	Ship structure
PRE2	(Tim Davis)	659033	5959282	UNS	AT&T,harmonic balance method
TWOTONE	(Tim Davis)	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	(Tim Davis)	157464	3866688	UNS	Complex zeolite,sodalite crystals.

Table 1: First set of test problems.

[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
AUDIKW_1 (PARASOL)	943695	39297771	SYM	Automotive crankshaft model
CONV3D64	836550	12548250	UNS	provided by CEA-CESTA; generated using AQUILON (http://www.enscpb.fr/master/aquilon)
ULTRASOUND80	531441	330761161	UNS	Propagation of 3D ultrasound waves, provided by M. Sosonkina, larger than ULTRASOUND3

Table 2: Set of larger test problems.

We have tested the algorithms presented in the previous sections (naive, based on increments and based on snapshot) on 32, 64 and 128 processors of the above-described platform. By default, we used the METIS package [9] to reorder the variables of the matrices. The results presented in the following sections have been obtained using the dynamic memory-based strategy and dynamic workload-based scheduling strategy presented in Sections 4.2.1 and 4.2.2, respectively. This is motivated by the fact that a memory-based scheduling strategy is very sensitive to the correctness of the view. The workload-based dynamic scheduling strategy (also sensitive to the correctness of the view) will be used to illustrate the cost of each mechanism in terms of time.

Matrix	32 processors	64 processors	128 processors
BMWCRA_1	41	96	-
GUPTA3	8	8	-
MSDOOR	38	81	-
SHIP_003	70	152	-
PRE2	92	125	-
TWOTONE	55	57	-
ULTRASOUND3	49	116	-
XENON2	50	65	-
AUDIKW_1	-	119	199
CONV3D64	-	169	274
ULTRASOUND80	-	122	218

Table 3: Number of dynamic decisions for 32, 64 and 128 processors.

For the memory-based strategy, we measure the memory peak observed on the most memory consuming process. The tests using the memory-based scheduling have been made on 32 and 64 processors which are enough for our study. For the workload-based scheduling strategy, we measure the time to factorize the matrix on the largest test problems on 64 and 128 processors. It is important to note that each set of results (test problem/number of processors) is performed on the same configuration of computational nodes. However, when going from one test problem to another, the configuration can change: 64 processors can either be 16 nodes of quadri-processors, either 2 nodes of 32 processors, or some intermediate configuration, including cases where some processors are not used in some nodes. Given this characteristic of the platform, results presented in this section should also not be used to get an idea of speed-ups between 64 and 128 processors. Finally, we give in Table 3 the number of dynamic decisions that will occur during the execution.

4.4 Memory-based scheduling strategy

In Table 4, we give the peak of active memory (maximum value over the processors) required to achieve the factorization. We compare the influence of the naive mechanism introduced in Section 2.1, of the mechanism based on increments introduced (see Section 2.2), and of the algorithm presented in Section 3 on the dynamic memory-based scheduler (see Section 4.2.1).
On 32 processors (Figure 4(a)), we observe that the peak of memory is generally larger for the naive mechanism than for the others. This is principally due to the limitation discussed in Section 2.1 for that mechanism: some dynamic scheduling decisions are taken by the schedulers with a view that does not include the variations of the memory occupation caused by the previous decisions. In addition, we observe that the snapshot algorithm given in Section 3 gives in most cases the best memory occupation and that the mechanism based on increments is not too far from the algorithm based on distributed snapshots. Note that for the GUPTA3 matrix, the algorithm based on snapshots provides the worst memory peak. In that case, we observed that there is

	Increments based	Snapshot based	naive
BMWCRA_1	3.71	3.71	3.71
GUPTA3	3.88	4.35	3.88
MSDOOR	1.51	1.51	1.51
SHIP_003	5.52	5.52	5.52
PRE2	7.88	7.83	8.04
TWOTONE	1.94	1.89	1.99
ULTRASOUND3	7.17	6.02	10.69
XENON2	2.83	2.86	2.93

(a) 32 processors.

	Increments based	Snapshot based	naive
BMWCRA_1	2.30	2.30	3.55
GUPTA3	2.70	2.70	2.70
MSDOOR	1.01	0.84	0.84
SHIP_003	2.19	2.19	2.19
PRE2	7.66	7.87	7.72
TWOTONE	1.86	1.86	1.88
ULTRASOUND3	3.59	3.40	5.24
XENON2	2.45	2.41	3.61

(b) 64 processors.

Table 4: Peak of active memory (millions of real entries) on 32 and 64 processors as a function of the exchange mechanism applied. The memory-based scheduling strategy is applied.

a side effect of doing snapshots on the schedule of the application. The asynchronous and non-deterministic nature of the application explain such possible exceptions to the more important general tendency.
On 64 processors, we can observe a similar behaviour: the naive mechanism gives in most cases worse results than the other mechanisms. For the largest problems in this set (e.g. matrix ULTRASOUND3), the algorithm based on snapshots gives the best results, followed by the mechanism based on increments and finally the naive mechanism.
The results of this section illustrate that when we are interested in a metric that has great variations (such as the memory), the algorithm based on snapshots is well-adapted, although costly. (We will discuss this in the next section.) We also see that in terms of quality of the information, the mechanism based on increments is never far from the one based on snapshots.

4.5 Workload-based scheduling strategy

	Increments based	Snapshot based
AUDIKW_1	94.74	141.62
CONV3D64	381.27	688.39
ULTRASOUND80	48.69	85.68

(a) 64 processors.

	Increments based	Snapshot based
AUDIKW_1	53.51	87.70
CONV3D64	178.88	315.63
ULTRASOUND80	35.12	66.53

(b) 128 processors.

Table 5: Time for execution (seconds) on 64 and 128 processors as a function of the exchange mechanism applied. The workload-based scheduling strategy is used.

We compare in Table 5 the factorization time from `MUMPS` with a workload-based scheduling strategy (see Section 4.2.2) when using the algorithm based on snapshots and the one based on increments. We can observe that the mechanism based on snapshots is less performant than the one based on increments. This is principally due to the fact that the snapshot operation requires a strong synchronization that can be very costly in terms of time. In addition, when there are several dynamic decisions that are initiated simultaneously, there are serialized to ensure the correctness of the view of the system on each processor. Thus, this can increase the duration of the snapshots. Finally, the synchronization of the processors may have unneeded effects on the behaviour of the

whole system. For example, if we consider the CONV3D64 matrix on 128 processors, the total time spent to perform all the snapshot operations is of 100 seconds. In addition, there were at most 5 snapshots initiated simultaneously. This illustrates the cost of the algorithm based on snapshots especially when the processors cannot compute and communicate simultaneously. (A long task involving no communication will delay all the other processes.) Furthermore, we remark that if we measure the time spent outside the snapshots for CONV3D64, we obtain $315.63 - 100.00 = 215$ seconds, which is larger than the 178.88 seconds obtained with the increments-based mechanism (see Table 5(b)). The reason is that after a snapshot, all processors restart their computation and data exchanges simultaneously. The data exchanges can saturate the network. Another aspect could be the side-effect of the leader election on the global behaviour of the distributed system, where the sequence of dynamic decisions imposed by the criterion for the leader election (processor rank in our case) has no reason to be good strategy. Finding a better strategy is a schedule issue and is out-of-scope in this study.

	Increments based	Snapshot based
AUDIKW_1	302715	11388
CONV3D64	386196	16471
ULTRASOUND80	208024	12400

(a) 64 processors.

	Increments based	Snapshot based
AUDIKW_1	1386165	39832
CONV3D64	1401373	57089
ULTRASOUND80	746731	50324

(b) 128 processors.

Table 6: Total number of messages related to the load exchange mechanisms on 64 and 128 processors.

Concerning the number of messages exchanged during the factorization, the results are given in Table 6. Note that the size of each message is larger for the snapshot-based algorithm since we can send all the metrics required (workload, available memory,...) in a single message. On the other hand, for the increments based mechanism, we send a message for each sufficient variation of a metric. We can observe, that the algorithm based on snapshots uses less messages than the mechanism based on increments that tries to maintain a view of the system on each process. The communication cost of these messages had no impact on our factorization time measurement since we used a very "high bandwith/low latency" network. For machines with high latency networks, the cost of the mechanism based on increments could become large and have a bad impact on performance. In addition, the scalability of such an approach can be a problem if we consider systems with a large number of computational nodes (more than 512 processors for example).
To study the behaviour of the snapshot mechanism in a system where processors can compute and communicate at the same time, we slightly modified our solver to add a thread that periodically checks for messages related to snapshots and/or load information. The algorithm executed by this second thread is given below:

```
1: while not end_of_execution do
2:   sleep(period)
3:   while there are messages to be received do
4:     receive a message
5:     if the received message is of type start_snp then
6:       block the other thread (if not already done)
7:     end if
8:     treat the received message
9:     if the received message is of type end_snp and there is no other ongoing snapshot then
10:      restart the other thread
11:    end if
12:  end while
13: end while
```

It is based on POSIX threads and only manages messages corresponding to state information.

Also, we fixed the sleep period arbitrarily to 50 microseconds. Furthermore since our application is based on MPI [10], we have to ensure that there is only one thread at a time calling MPI functions using locks. Finally, the interaction between the two threads can be either based on signals or locks. One way to block the other thread is to send a special signal to block it. Another way, which is the one used here, is to simply get the lock that protects the MPI calls and to release it only at the end of the snapshot.

	Increments based	Snapshot based
AUDIKW_1	79.54	114.96
CONV3D64	367.28	432.71
ULTRASOUND80	49.56	69.60

(a) 64 processors.

	Increments based	Snapshot based
AUDIKW_1	41.00	59.19
CONV3D64	189.47	237.69
ULTRASOUND80	35.91	52.00

(b) 128 processors.

Table 7: Workload-based scheduling: Impact of the threaded load exchange mechanisms on the factorization time on 64 and 128 processors.

We tested this threaded version of the application on 64 and 128 processors. The results are given in Table 7. Note that we also measured the execution time for the threaded increments mechanism with the intention to evaluate the cost of the thread management. We observe that using a thread has a benefic effect on the performance in most cases for the mechanism using increments (compare the left columns of Tables 5 and 7). We believe that this is because the additionnal thread treats the messages more often and thus avoids to saturate the internal communication buffers of the communication library (and from the application). Concerning the algorithm based on snapshots, the execution time is greatly reduced compared to the single-threaded version, thus illustrating the fact that processors spend less time performing the snapshot. For example if we consider the CONV3D64 problem on 128 processors, the total time spent to perform all the snapshot operations has decreased from 100 seconds to 14 seconds. However, we can observe that this threaded version of the snapshot algorithm is still less performant than the one based on increments. This is principally due to the stronger synchronization points induced by the construction of a snapshot (even in the threaded version), as well as the possible contention when all processors restart their other communications (not related to state/snapshot information).

5 Conclusion

In this paper, we have discussed different mechanisms aiming at obtaining a view as coherent and exact as possible of the load/state information of a distributed asynchronous system under the message passing environment. We distinguished between two principal algorithms achieving this goal: maintaining a view as correct as possible during the execution, and building a correct distributed snapshot.
We have shown that broadcasting periodically messages that update the load/state view of the other processes, with some threshold constraints and some optimization in the number of messages, could provide a good solution to the problem, but that this solution requires the exchange of a large number of messages. On the other hand, the demand-driven approach based on distributed snapshot algorithms is also of interest and provides more accurate information, but is also much more complex to implement in the context of our type of asynchronous applications: we had to implement a distributed leader election followed by a distributed snapshot; also, we had to use a dedicated thread (and mutexes to protect all MPI calls) in order to increase reactivity. In addition, this solution appears to be costly in terms of execution time and might not be well-adapted for high-performance distributed asynchronous applications. It can however represent a good solution in the case of applications where the main concern is not execution time but another metric to which the schedulers are very sensitive (e.g. the memory usage). We also observed that

this approach reduces significantly the number of messages exchanged between the processes in comparison to the first one; it could still be well adapted for distributed systems where the links between the computational nodes have high latency/low bandwidth.
Some perspectives of this work are as follows. Having observed that the dynamic scheduling strategies are very sensitive to the approach used, it would be interesting to study some issues such as the criterion used to elect the leader, which propably have a significant impact on the overall behaviour. In addition, for applications where only a subset of the processes may be candidate in each dynamic decision, it would be useful to study how snapshot algorithmes involving only part of the processes can be implemented, with the double objective of reducing the amount of messages and having a weaker synchronization.

Acknowledgements

Discussions with Stéphane Pralet and Patrick Amestoy have been very helpful in the design of Algorithm 3.

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, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling strategies for the parallel multifrontal method. In *3rd International workshop on Parallel Matrix Algorithms and Applications (PMAA'04)*, October 2004.

[4] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[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] H. Garcia-Molina. Election in distributed computing system. *IEEE Transactions on Computers*, pages 47–59, 1982.

[7] 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.

[8] Letian He and Yongqiang Sun. On distributed snapshot algorithms. In *Advances in Parallel and Distributed Computing Conference (APDC '97)*, 1997. 291–297.

[9] 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.

[10] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, Cambridge, Massachusetts, 1996.

[11] S. D. Stoller. Leader election in asynchronous distributed systems. *IEEE Transactions on Computers*, pages 283–284, 2000.