Coherent load information mechanisms for distributed dynamic scheduling

Abdou Guermouche, Jean-Yves l'Excellent

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Coherent load information mechanisms for distributed dynamic scheduling

Abdou Guermouche (ENS Lyon) — Jean-Yves L'Excellent (INRIA)

N° 5178

May 2004

THÈME Num

ISSN 0249-6399 ISRN INRIA/RR--5178--FR+ENG

Coherent load information mechanisms for distributed dynamic scheduling

Abdou Guermouche (ENS Lyon), Jean-Yves L'Excellent (INRIA)

Thème Num — Grilles et calcul haute-performance
Projet GRAAL

Rapport de recherche n° 5178 — May 2004 — 18 pages

Abstract: We consider a distributed 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 distributed estimates of such information and experiment them in the context of a real application, an asynchronous parallel solver for large sparse systems of linear equations.

Key-words: snapshot, distributed system, dynamic scheduling, load balancing, message passing

This text is also available as a research report of the Laboratoire de l'Informatique du Parallélisme `http://www.ens-lyon.fr/LIP`.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Cohérence des informations de charge pour des ordonnanceurs dynamiques distribués

Résumé : Nous considérons un système distribué où les processus peuvent seulement communiquer par passage de messages, et requièrent une estimation cohérente de la charge des autres processus (travail, mémoire utilisée) pour procéder à des décisions dynamiques liées à l'ordonnancement des tâches de calcul. Nous présentons plusieurs mécanismes pour maintenir une vision distribuée de telles informations et les expérimentons 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, distributed or centralized. Here we are interested in 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 scheduling decisions. Several mechanisms may be designed to obtain distributed estimates of such information.
The paper is organized as follows. In Section 1, we present in more detail the context of the problem. In Sections 2, 3 and 4, we analyse three different mechanisms to exchange such load information, from the most naive to the most sophisticated one. We study in Section 5 the impact of these mechanisms on the behaviour of a real-life application that uses distributed dynamic scheduling strategies: a parallel asynchronous solver for sparse systems of linear equations. Then, in Section 6, we present a simple mechanism to reduce the number of messages for load information. Finally we conclude.

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).
We should note that this problem is close to the distributed snapshot problem [3, 7], although we cannot afford a complete snapshot that would be costly in terms of performance whereas we want to spend the shortest possible time to get the load information. Indeed, in classical snapshot algorithms, the snapshot is initiated by the process that requires it. In our case, especially for systems with large numbers of processes, this kind of snapshot algorithms becomes costly. Since we are interested in load information that varies with the activity of the process, we would have to ensure

that the load does not vary during the snapshot process; this would be equivalent to stopping the activity of a process at the reception of a message related to the snapshot mechanism.
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 here aim at providing information about the system that will be used to take distributed dynamic scheduling decisions. Our mechanisms are based on message passing where each process broadcasts when its state changes. Thus, when a process has to take a dynamic decision, 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. In the following, we call this type of dynamic decisions a *slave selection*.

2 Naive mechanism

In this mechanism, described by Algorithm 1, 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 others. A threshold mechanism ensures that the amount of messages to exchange load information remains reasonable.

Algorithm 1 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: $load(P_j) = l_j$;

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.

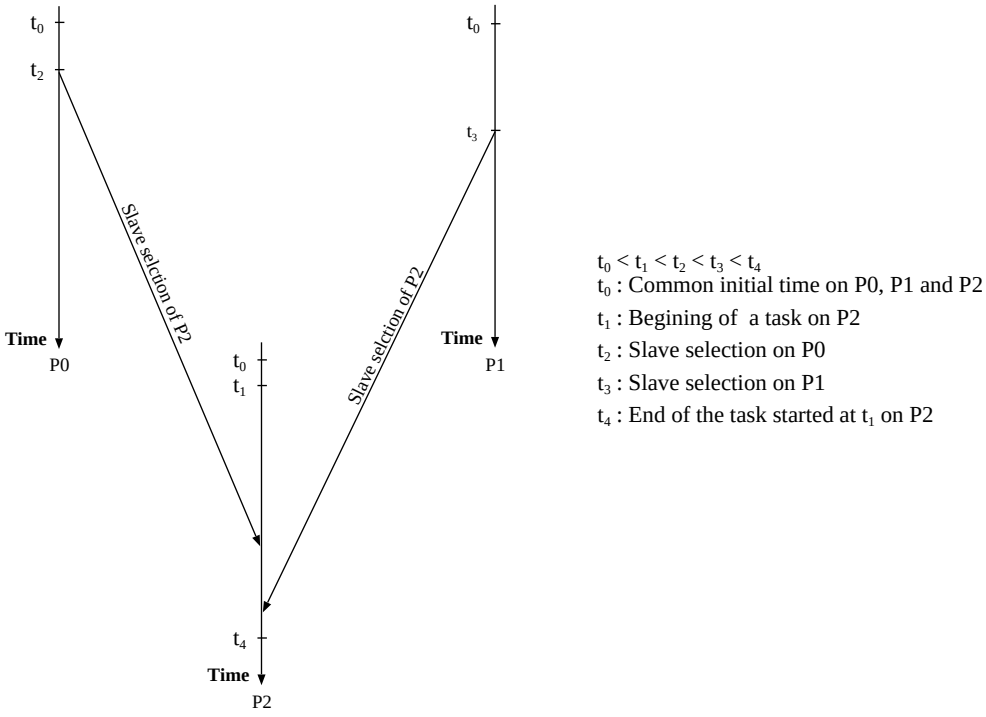


Figure 1: Example illustrating the problem of the coherence of load information with the naive mechanism.

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 that can lead to critical situations (in practice, large imbalance for 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 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 of this situation, P_2 that does not know yet that it has been chosen as slave by P_0, cannot inform 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 shows the problem of the coherence of the information exchanged by the processors.

3 Intermediate mechanism

The mechanism we describe in this section is a mechanism designed to avoid situations like the one given in Figure 1. The main goal of such a mechanism is to make the slave selection of a processor visible by the others as early as possible.
Each time a process selects slaves, it broadcasts (to all processes) the list of selected slaves and the load (workload, memory,...) assigned to each of them. A message of type *Master_To_All* is used for such messages. Each time a process receives a message of this type, it aggregates the information sent by the master to the one it already has for each slave except himself if it is selected as slave. Concerning the selected slaves, their behaviour is not changed in this new strategy. A more formal description of the mechanism is given in Algorithm 2. It is applied on top of Algorithm 1.

Algorithm 2 Processing of *Master_To_All* messages (intermediate mechanism).

At each slave selection on the master side:
1: **for** P_j **in** the list of selected slaves **do**
2: Include in a message of type *Master_To_All* the load δl_j assigned to P_j;
3: **end for**
4: **send** (asynchronously) the message *Master_To_All* to the other processes;
At the reception of a message of type *Master_To_All*:
5: **for all** $(P_j, \delta l_j)$ **in** the message **do**
6: **if** $P_j \neq$ myself **then**
7: $load(P_j) = load(P_j) + \delta l_j$;
8: **end if**
9: **end for**

Limitations

The problem with this mechanism is that the information sent by the master processes concerning the slave selections may be overwritten (lost) when the slave sends a message of type *Update*. An example of this situation is given in Figure 2. In this configuration, there are five processes taking part to the computation. In addition,

P_0 is treating a possibly costly task in terms of execution time whereas P_1 and P_2 are in the process of selecting slaves. The problem occurs at time t_1 in Figure 2, when P_0 just finishes a task. At this time, P_0 will broadcast a new workload and memory information. This will overwrite other information about P_0 related to the slave selection of P_0. Information relative to the slave selection of P_0 on the others are thus lost. Finally, if P_3 has a slave selection to perform at a time $t_2 > t_1$, it will choose its slaves without any information about the previous slave selections by P_1 and P_2, thus giving P_0 too much work. This can clearly be critical, with scheduling decisions based on inadequate load information.

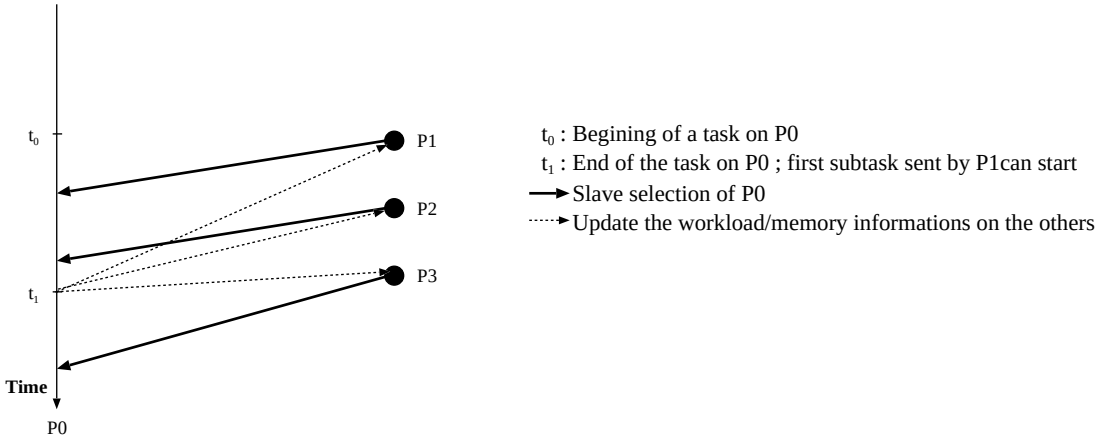


Figure 2: Example illustrating the problem of the coherence of load information with the intermediate mechanism.

4 Mechanism based on load increments

In this section we present another mechanism based on load increments to improve the coherence of load information during execution. Similarly to the mechanisms described in the previous sections, the load estimates are obtained thanks to a set of messages exchanged by the processes.
Algorithm 3 presents the mechanism based on increments. For each variation of the workload on a process P_i, P_i broadcasts the increment representing the variation. Again, a threshold mechanism is applied to avoid too many messages. These messages are of type *Update*. In addition, we use the same mechanism as the one

Algorithm 3 Mechanism based on load increments.

Initialization

1: $my\_load = 0$;
2: $\Delta load = 0$;

When my load varies of $\delta load$**:**

3: $my\_load = my\_load + \delta load$;
4: **if** $\delta load$ concerns a task where I am slave **then**
5: **if** $\delta load > 0$ **return**; **(1)**
6: **end if**
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**

described in Section 3 for the slave selection information. Indeed each time a process selects slaves, it broadcasts 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. Note that when a (slave) process starts a task that was sent by another, it does not broadcast a message of type *Update* if the increment is positive: since the master has already sent the information relative to its selected slaves, there is no need for the slave processes to send that information again (see (1) in Algorithm 3).

5 Application to a distributed sparse matrix solver, MUMPS

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 5.1 we present the software package `MUMPS` and show how it fits with the distributed system presented earlier. Both workload-based and memory-based strategies are described (Section 5.2), aiming at respectively optimizing the time of execution of the complete graph of tasks or balancing the memory over the processors. In Section 5.3 we compare the behaviour of that application for the three algorithms presented earlier to exchange load and memory information.

5.1 Task graph in MUMPS

`MUMPS` uses a combination of static and dynamic approaches. Those are described in details in [1] and [2]. 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 3 summarizes the different types of parallelism available in `MUMPS`:

- 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

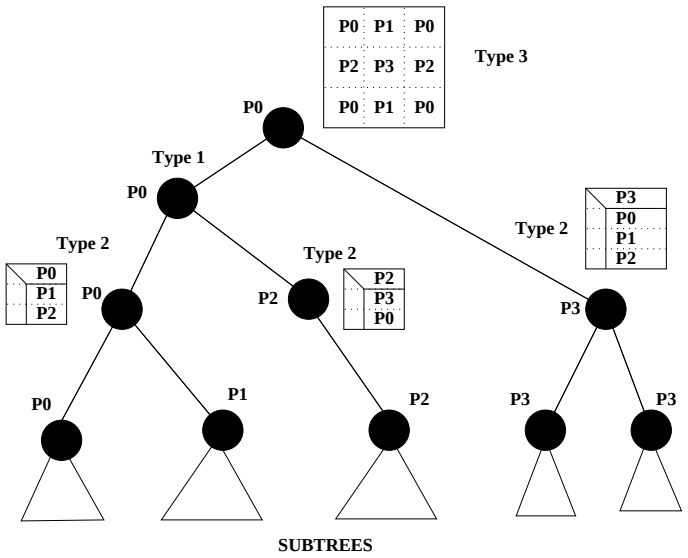


Figure 3: Example of distribution of a multifrontal assembly tree over four processors.

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. Those are determined using a top-down algorithm [5] and a subtree-to-process mapping is used to balance the computational work of the subtrees onto the processors.

- 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 [4] 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.

5.2 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`.

5.2.1 Case 1: memory-based scheduling strategy

We presented in [6] 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 (see Figure 4). 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.

5.2.2 Case 2: workload-based scheduling strategy

This strategy 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 ir-

regular 1D-blocking by rows (Figure 4). 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.

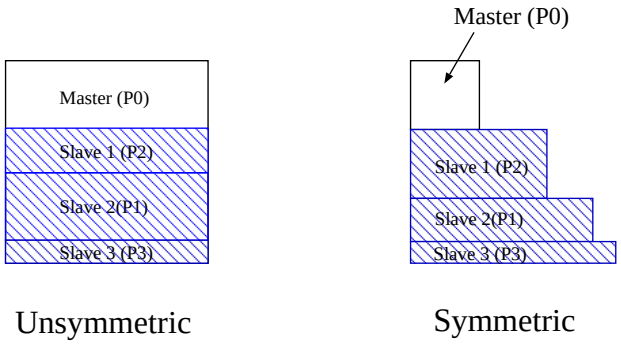


Figure 4: type 2 nodes blocking with the default strategy.

5.3 Experimental study of the load exchange mechanisms

We should first mention that the mechanisms described in Sections 2, 3 and 4 have been implemented inside the `MUMPS` package. In fact, the mechanism from Section 2 used to be the one available in `MUMPS`, while the mechanism of Section 4 is now the default in `MUMPS` 4.3.

To study the impact of the proposed mechanisms, we experiment them on several problems (see Table 1) 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. Each node is equipped with a minimum of 2 GBytes per processor.

We have tested the three load exchange mechanisms on 32 and 64 processors of the above-described platform. By default, we used the METIS package [8] to reorder the

[1] `http://www.cise.ufl.edu/~davis/sparse/`

[2] `http://www.parallab.uib.no/parasol`

[3] Institut du Développement et des Ressources en Informatique Scientifique

Matrix		Order	NZ	Type	Description
BMWCRA_1	(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
CONV3D64		836550	12548250	UNS	provided by CEA-CESTA; generated using AQUILON (http://www.enscpb.fr/master/aquilon)
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.

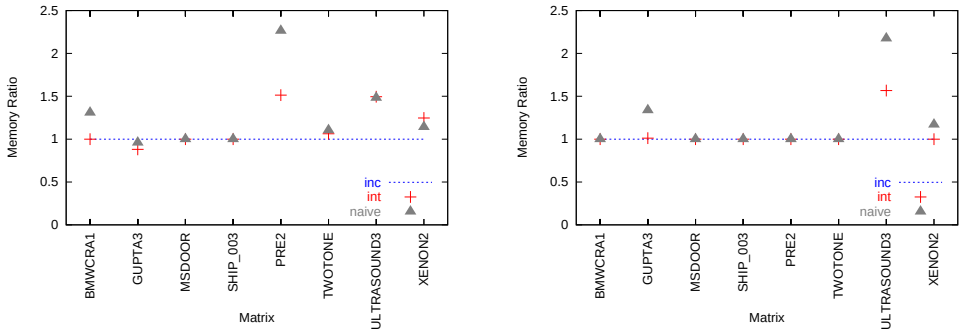
Table 1: Test problems.

variables of the matrices. We give in Figures 5 and 6 the results obtained using the three mechanisms with a dynamic memory-based strategy and a dynamic workload-based scheduling strategy, respectively. The test problems are the ones from Table 1 except for the matrix CONV3D64 which was too large to run on 32 processors. For the memory-based strategy, we measure the memory peak observed on the most memory consuming process. For the workload-based scheduling strategy, we measure the time to factorize the matrix. In both cases the metric used is divided by the result (memory or time for factorization) obtained with the mechanism based on increments. This allows to normalize the measures, with the mechanism based on increments always giving a metric equal to 1.

On 32 processors (Figure 5(a)), we observe that the memory is generally smaller for the mechanism based on increments except for the GUPTA3 matrix. In addtition, we observe that there is no gain for matrices 2, 3 and 4. For these symmetric matrices, the memory on 32 processors is dominated by the memory of subtrees treated sequentially on some processors, and there is indeed not much to be gained from the scheduling strategy.

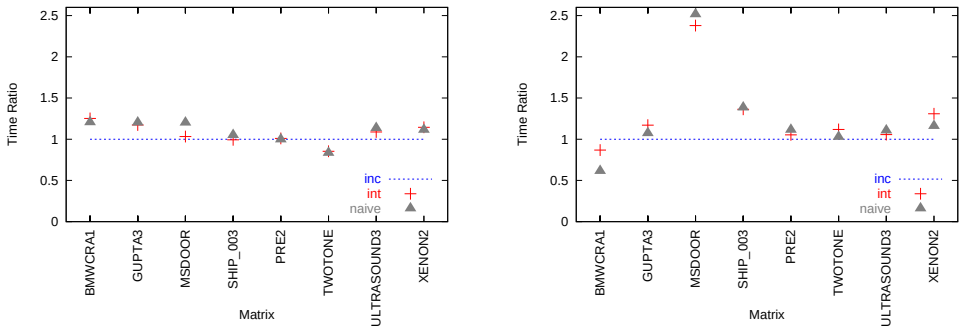
Comparing the intermediate and naive mechanisms, the intermediate mechanism is sometimes better, sometimes worse than the naive mechanism.

On 64 processors, the results are slightly less significant, while we would have expected a better gain since more dynamic decisions are taken in that case. However, for the cases where there are no gains (for example PRE2 on 64 processors and for many symmetric matrices on both 32 and 64 processors), the peak is reached before any dynamic decision has been taken. In addtion, we can observe that for the matrix ULTRASOUND3 (the largest of this set), the ratios obtained with respect to the mechanism based on increments reach 1.5 and 2.1 for the intermediate and the naive mechanism respectively.



(a) 32 processors (b) 64 processors

Figure 5: Memory-based scheduling: impact of the load exchange mechanisms on the active memory peak on 32 and 64 processors (results are normalized with respect to the mechanism based on increments).



(a) 32 processors (b) 64 processors

Figure 6: Workload-based scheduling: impact of the load exchange mechanisms on the factorization time on 32 and 64 processors (results are normalized with respect to the mechanism based on increments).

Considering the influence of the mechanisms on the time for factorization (Figure 6), the mechanism based on increments nearly always gives the best execution time

(except for matrix TWOTONE matrix on 32 processors, and matrix BMWCRA_1 on 64 processors). This illustrates the fact that the dynamic decisions (slave selections) are taken using more coherent information.

Finally, we experimented the three strategies on our largest test problem, CONV3D64, on 64 processors. The time for factorization for the three mechanisms is given in Table 5.3. We observe, for this large test problem, the impact of using the increments algorithm. Indeed, we can see that we obtain a reduction of 46 and 54 seconds in comparison with the intermediate and the naive mechanism respectively. Notice that for this test problem, there are 161 slave selections during the factorization. Table 5.3 gives the number of dynamic decisions taken with a coherent view of the system for the CONV3D64 problem. We consider that a processor P_i has a coherent view of the system if it satifies the following conditions:

- all the messages sent concerning the load of all the processors have been received by P_i, and
- there is no incoherence due to the slave selections (as in Figures 1 and 2).

We remind that before a slave selection, P_i processes all the messages related to load information that are present in its reception queue. The results given in Table 5.3 show that the mechanism based on increments provides a better view of the system compared to the other ones.

Note that in the case where we only consider the incoherence due to slave selections (of the type of Figures 1 and 2), we obtain that for the intermediate mechanisms there are 30 decisions based on coherent information and for the naive mechanism there are 25 decisions; for the mechanism based on increments all the decisions are taken with valid information. This shows that the mechanism based on increments is well-adapted to solve the first type of incoherence. It does provide an incoherent view only if there are some messages that have been sent and that have not yet been received. This situation could be avoided by doing a real snapshot but since we have a strong constraint on performance, we cannot afford such kinds of snapshot algorithms. Another improvement would consist, after a slave selection is performed, in checking again for load messages in the reception queue in order to update the view of the system. If the view has changed significantly, we can imagine to redo the slave selection based on the new information. This should of course be done only a small number of times.

naive	intermediate	increment based
330.62	322.28	276.15

Table 2: Execution time using the three mechanisms with the CONV3D64 matrix on 64 processors.

naive	intermediate	increment based
15	18	135

Table 3: Number of slave selections done with a coherent view of the system for the CONV3D64 matrix on 64 processors.

6 Reducing the number of messages

In some types of applications, some processes may never be master and never send work to others and this information may be known statically. Thus, those processes do not need any knowledge of the workload/memory of the others. More generally speaking, if at some point, a process knows that it will not proceed to any further slave selection in the future, it can inform others. After a process P_i has performed its last slave selection, it can thus send a message of type *No_more_master* to 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 this mechanism has been applied in the results reported in Section 5.3. It allows to decrease signigicantly the number of messages related to load information. For example with our test application `MUMPS` on the matrix CONV3D64, there are 101085 messages received with the mechanism based on increments, whereas the number of messages received without the mechanism to reduce the number of messages is 171860.

7 Conclusion

In this paper, we presented different mechanisms aiming at providing distributed information (workload, memory) to the schedulers in the context of a distributed asynchronous system.

The problem of providing a correct view of a distributed system is known as the snapshot problem and several solutions were proposed to provide a correct snapshot. However, these algorithms can be costly and thus are not well-adapted to a high-performance application (or to an application where time spent in scheduling or building a view of the system has to be minimal). Thus, we aim at maintaining a view of the system as correct as possible during the execution. The main advantage

of such an approach is that the process that has to take a dynamic decision already has the view of the system.
We experimented the different mechanisms in the context of an asynchronous parallel solver for large sparse systems of linear equations. Experiments showed that a mechanism based on increments (load variations) provides an acceptable view of the system to the schedulers and results in an improved behaviour of the application in several cases. In addition, in that mechanism, the incoherence can only come from messages that have been sent without being received when a dynamic scheduling decision is taken. Finally, some improvements to the increment mechanism have been proposed to reduce the number of messages.

Acknowledgements

Discussions with Stéphane Pralet and Patrick Amestoy have been very helpful in the design of Algorithms 2 and 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] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[4] 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.

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

[6] 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. To appear.

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

[8] 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.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399