A new aproach to parallel sparse cholesky factorization on distributed memory parallel computers

Mounir Hahad, Jocelyne Erhel, Thierry Priol

To cite this version:

Mounir Hahad, Jocelyne Erhel, Thierry Priol. A new aproach to parallel sparse cholesky factorization on distributed memory parallel computers. [Research Report] RR-2081, INRIA. 1993. inria-00074590

HAL Id: inria-00074590

https://inria.hal.science/inria-00074590

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A New Approach to Parallel Sparse Cholesky Factorization on Distributed Memory Parallel Computers

Mounir HAHAD, Jocelyne ERHEL, Thierry PRIOL

N° 2081

Octobre 1993

PROGRAMME 1

Architectures parallèles,
bases de données,
réseaux et systèmes distribués

1993

ISSN 0249-6399

A New Approach to Parallel Sparse Cholesky Factorization on Distributed Memory Parallel Computers

Mounir HAHAD*, Jocelyne ERHEL, Thierry PRIOL

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet PAMPA

Rapport de recherche n° 2081 — Octobre 1993 — 17 pages

Abstract: Nowadays, programming distributed memory parallel computers (DMPCs) evokes the "no pain, no gain" idea. That is, for a given problem to be solved in parallel, the message passing programming model involves distributing the data and the computations among the processors. While this can be easily feasible for well structured problems, it can become fairly hard on unstructured ones, like sparse matrix computations. In this paper, we consider a relatively new approach to implementing the Cholesky factorization on a DMPC running a shared virtual memory (SVM). The abstraction of a shared memory on top of a distributed memory allows us to introduce a large-grain factorization algorithm, synchronized with events. Several scheduling strategies are compared, and experiments conducted so far show that this approach can provide the power of DMPCs and the ease of programming with shared variables.

Key-words: KOAN, Shared Virtual Memory, Cholesky factorization, task scheduling, sparse matrices.

(Résumé : tsvp)

*e-mail :hahad@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

Une nouvelle approche pour la factorisation de Cholesky de matrices creuses sur les machines parallèles à mémoire distribuée

Résumé : L'exploitation efficace de machines parallèles à mémoire distribuée nécessite des efforts importants de la part des utilisateurs. En effet, la mise en œuvre d'applications parallèles dans le modèle de programmation par envoi de messages requiert une distribution des données ainsi que des calculs sur les différents processeurs. Cette tâche peut se révéler complexe sur des problèmes irréguliers tels que les calcul sur matrices creuses. Dans cet article, nous considérons une nouvelle approche de la factorisation de Cholesky de matrices creuses sur les machines à mémoire distribuée dotées d'une mémoire virtuelle partagée. L'abstraction d'une mémoire partagée au dessus d'une mémoire physique distribuée nous permet d'introduire un algorithme de factorisation à grain large synchronisé par événements. Plusieurs stratégies d'ordonnancement sont comparées. Les résultats actuels montrent que nous pouvons aboutir à une utilisation efficace des machines parallèles à mémoire distribuée, tout en conservant l'avantage d'une programmation aisée grâce à la communication par variables partagées.

Mots-clé : KOAN, Mémoire Virtuelle Partagée, factorisation de Cholesky, ordonnancement, matrices creuses.

1 Introduction

In many computational kernels, the solution to a sparse linear system is required. This system may arise in various problems, such as the discretization of partial differential equations, encountered for instance in computational fluid dynamics or structural analysis. Quite often, the matrix of order n is symmetric and positive definite, so that a direct method involving two steps can be used to solve the system $Ax = b$:

1. factorize the matrix A in the form $A = LL^T$, where L is a lower triangular matrix;
2. solve the two triangular systems $Ly = b$ and $L^T x = y$.

This paper deals with the first step which is called the Cholesky factorization. We consider a new approach to computing the numerical factorization in parallel on a distributed memory parallel computer (DMPC) running a shared virtual memory (SVM). The SVM is a paged-demand shared address space built on a physically distributed memory. We believe that it is an interesting alternative to conventional DMPCs programming approaches. An SVM allows us to implement a large-grain algorithm on a DMPC, reducing processor interactions. Indeed, a large grain task in the Cholesky factorization requires several column accesses, which may be non local accesses. In a distributed environment, where columns are mapped onto local memories, a processor cannot read/write data stored on other processors. Thus, it cannot achieve its task without their contribution. We conclude that no large grain Cholesky factorization is possible on conventional local memory multiprocessors. Benefiting from the SVM, we have designed a large grain algorithm synchronized with events. Experiments were conducted on a software based SVM, called KOAN.

The layout of this paper is as follows: in section 2, we introduce the sequential Cholesky factorization. In section 3, existing parallel algorithms, mainly the Fan-In and the Fan-Out, are briefly overviewed. In section 4, we introduce our approach consisting of the KOAN shared virtual memory, the numerical factorization algorithm and the scheduling strategies. In section 5, we provide experimental results for an intel IPSC/2 running KOAN SVM, focusing on the SVM behavior.

2 Sparse Cholesky Factorization

The left hand side of figure 1 shows a dense Cholesky factorization algorithm. Consecutive columns of L are computed by performing a series of updates to each column. Note that if the element L_{jk} is zero, which is often the case if the matrix is sparse, the loop L_1 does not involve computations. In other words, if the element L_{jk} is zero, the column k has no effect on column j. Taking into

```
    For j := 1 to n do
       For k := 1 to j - 1 do
L1        For i := j to n do
             L_ij = L_ij - L_ik * L_jk
          Endfor
       Endfor

       L_jj = sqrt(L_jj)
L2     For i := j + 1 to n do
          L_ij = L_ij / L_jj
       Endfor
    Endfor
```

```
    For j := 1 to n do
       For k ∈ Struct(L_j*) ∪ {j} do
L1        For i ∈ Struct(L_*j) do
             L_ij = L_ij - L_ik * L_jk
          Endfor
       Endfor

       L_jj = sqrt(L_jj)
L2     For i ∈ Struct(L_j*) do
          L_ij = L_ij / L_jj
       Endfor
    Endfor
```

Figure 1: Column-Cholesky for dense matrices and for sparse ones.

account this particular point leads to a sparse Cholesky factorization algorithm. The right hand side of figure 1 shows such an algorithm where $Struct(L_{j*})$ is the set of column subscripts of non zero elements in the jth row of L, and $Struct(L_{*j})$ is the set of row subscripts of non zero elements in the jth column.

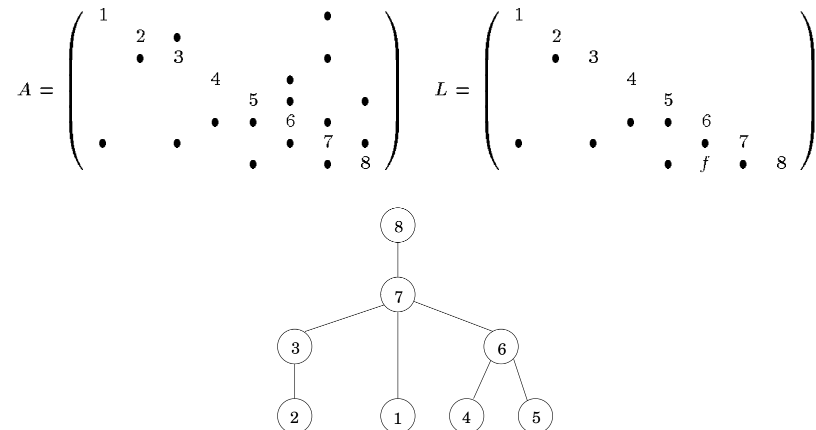

Figure 2: Sparse matrix structure and its Cholesky factor and elimination tree

During the factorization process, a fill-in phenomenon occurs : some zero elements in A become non zero in L. Figure 2 shows a sample matrix and its Cholesky factor (a "f" in L matches a fill). In order to save computations and storage, it is worthwhile to reduce that fill-in, which depends on the ordering of the columns in the matrix A. Therefore, the Cholesky factorization is performed

in three steps : 1- ordering ; 2- symbolic factorization ; 3- numerical factorization. Orderings are based on heuristics to reduce the fill-in and for some of them, to enhance the parallelism. If not, a post-ordering technique that enhances the parallelism with an equivalent fill-in may be desirable for parallel factorization. Symbolic factorization sets up some data structures and estimates the amount of storage required. One of these data structures is the elimination tree of L. Liu [12] describes the role of this structure in both sequential and parallel factorization. Finally, the numerical factorization performs the evaluation of the elements of L. Since this last phase is the most time consuming, we focus on its parallelization.

3 Overview of Parallel Approaches

We do not address frontal and multifrontal methods [2, 3] in this paper. These originally aimed at providing an *out-of-core* solution to huge sparse systems. They are based on a submatrix assembly at each step of the factorization and a more efficient inner loop which avoids indirections. We are interested in less sophisticated direct methods. Consequently, we will emphasize the two well known Fan-In and Fan-Out approaches[1] and their implementation on both shared memory and distributed memory parallel computers. The two most relevant points in these algorithms are the task partitioning and the scheduling.

3.1 Task Partitioning

In [11], Liu reviews some computational models, generally well suited to column oriented factorizations. We briefly describe the medium grain and large grain task models. The fine grain model is of less interest, since it is fruitless to exploit such a parallelism on DMPCs.

A fairly simple way to generate medium grain tasks is to group all the operations within an inner loop into a single task. Two such tasks can be built in the Cholesky factorization from the loops L_1 and L_2 :

- *Cmod(j,k)* : update column j by column k ($k \in Struct(L_{j*})$) ,
- *Cdiv(j)* : scale column j by the square root of its diagonal element.

From this, a precedence relation graph can be built on the basis of the following dependency constraints :

- *Cmod(j,k)* cannot be executed before *Cdiv(k)* ,
- *Cdiv(j)* cannot be executed before all *Cmod(j,k)* $k \in Struct(L_{j*})$.

[1] sometimes referred to as left-looking and right-looking approaches

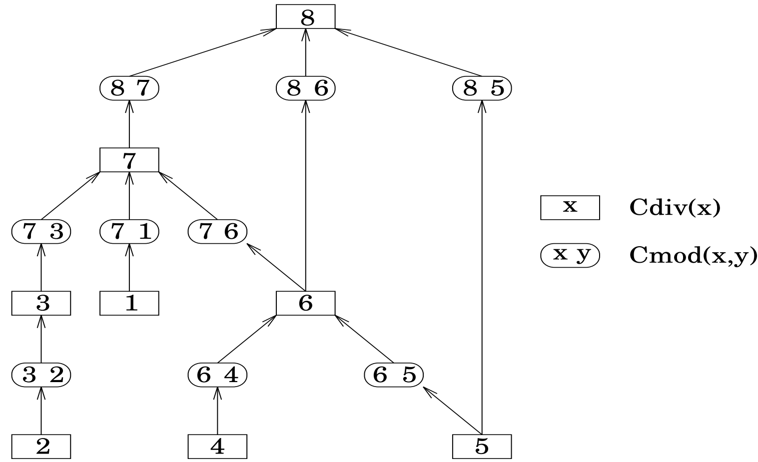

Figure 3: Task precedence graph for matrix of figure 2

Note that for each column j there is a corresponding *Cdiv(j)* task, and for each off-diagonal non zero element L_{jk}, there is a corresponding *Cmod(j,k)* task. Hence, the task precedence graph depicted in figure 3 has $\eta(L)$ vertices and $2(\eta(L) - n)$ edges, where $\eta(L)$ is the number of nonzero elements in L.

In order to obtain a coarser task grain, the medium grain tasks *Cdiv* and *Cmod* are packed together in such a way as to simplify the precedence graph and to minimize synchronization. Two relevant large grain task models have been introduced for shared memory machines (surveyed by Liu in [11]) :

- in the Fan-In approach, a single task is defined by :

$$T_j = \{Cmod(j,k)\, k \in Struct(L_{j*})\} \ \cup \ Cdiv(j)\,;$$

- in the Fan-Out algorithm, a single task is defined by :

$$J_k = Cdiv(k) \ \cup \ \{Cmod(j,k)\, j \in Struct(L_{*k})\}$$

Since all subtasks of a single task T_j or J_k are scheduled in sequence, there is no need to keep the precedence relation between them in the task precedence graph. Furthermore, we note that some precedence relations between vertices are redundant and may be discarded too. This graph reduction leads to a tree which is merely the elimination tree of the matrix to be factored. It is easy to verify that T_j has $\eta(L_{j*})$ medium grain subtasks and J_k has $\eta(L_{*k})$ such subtasks. Recall that $\eta(L_{j*})$ is the number of nonzero elements in row j of L and $\eta(L_{*k})$ is the number of nonzero elements in column k. Moreover, the reduced precedence graph has only n vertices and $n - 1$ edges.

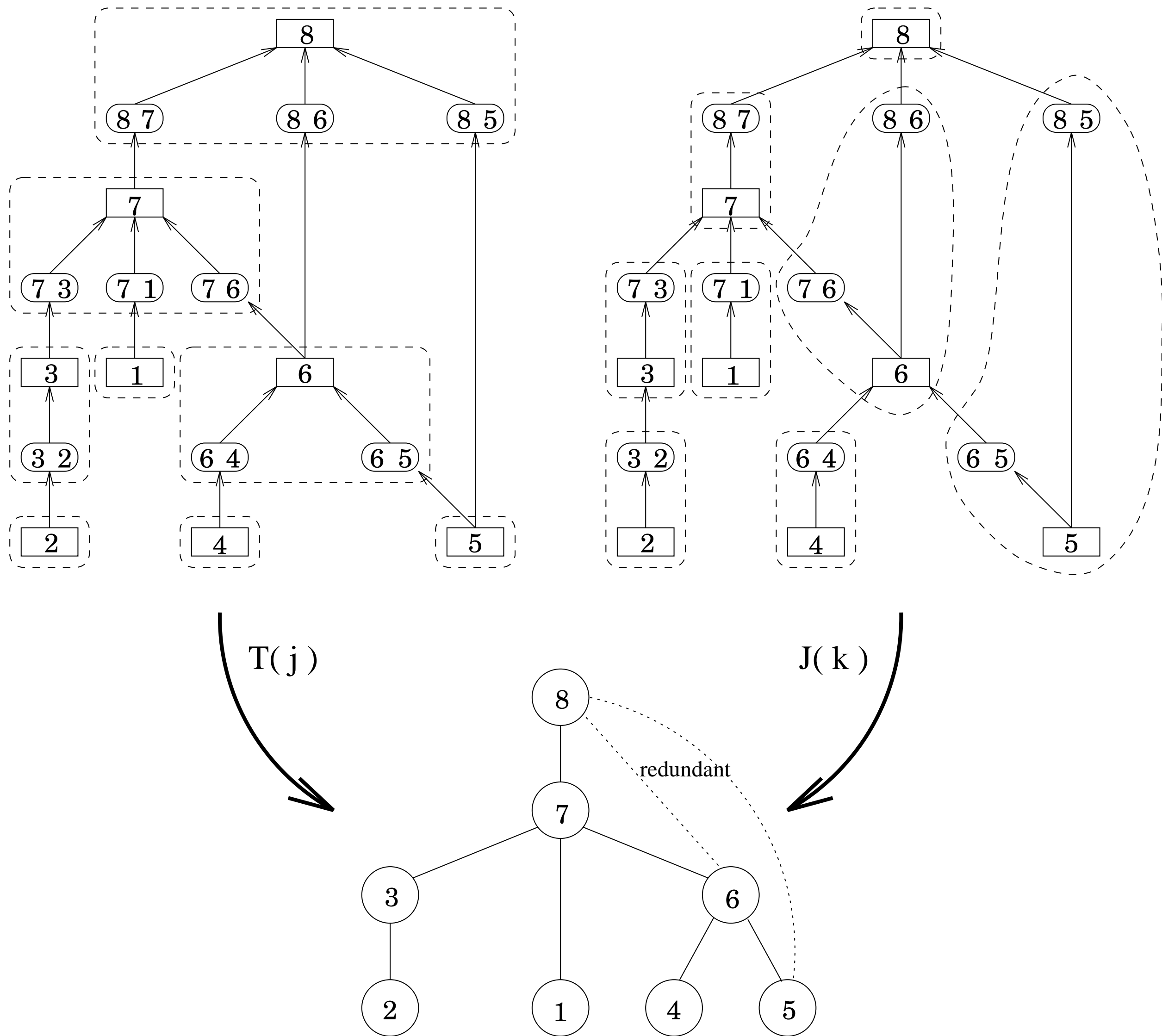

Figure 4: Task precedence graph and elimination tree.

3.2 Scheduling

On DMPCs, columns are spread among the processors prior to any computation. The task scheduling obviously depends on this initial column mapping. In both the distributed Fan-In [1] and the distributed Fan-Out [5] algorithms, the medium grain task model is used with different column assignment strategies. By contrast, there is no such constraint (column mapping) to schedule the tasks on a shared memory machine and so, a large grain model can be used.

For distributed memory machines, the first experiments on the Fan-In algorithm [1] were carried out on matrices arising from nine-point finite difference operators on rectangular grids. The nested dissection ordering [6] and the subtree-to-subcube mapping [7] are used, achieving low fill-in and good load balance for the problems under consideration. Thus, the performances obtained are better than for the distributed Fan-Out algorithm. Indeed, the latter is used to solve finite elements problems arising from L-shaped triangular meshes. A simple wrap-around task assignment is used after a nested dissection ordering of the test matrices [5].

For shared memory machines, George et al. [4] use straightforwardly the large grain task model in the Fan-In algorithm. However, they do not enhance the parallelism by a post ordering of the elimination tree. In a quite different way, Rothberg et al. [16] take advantage of a larger task grain size : a task is the completion of a set of contiguous columns in the Cholesky factor that share essentially the same sparsity structure. However, it turns out that these so called supernodes are often too large to allow a good load balancing. Therefore, they are split according to a simple heuristic introduced in [15]. Rothberg et al. do not use a dynamic load balancing technique even though the machine they experiment with has a shared memory. Using up to eight processors, this algorithm exhibits better performances than the Fan-In for shared memory machines.

4 SVM Approach

With the advent of shared virtual memory machines (like Kendall Square Research KSR1), a new programming paradigm is provided : computers with physically distributed memory can be programmed using a shared variable programming model. Obviously, the goal is to bypass the lack of programming environments for DMPCs, which makes it difficult to achieve high performances. We take advantage of this new concept to present an algorithm that uses a large grain task model upon a distributed memory architecture, and that ensures its synchronization by means of events.

4.1 KOAN : a Shared Virtual Memory

KOAN is a SVM implemented on an Intel *iPSC/2* hypercube [9]. It is embedded in the operating system, the *NX/2*. Our machine configuration uses up to 32

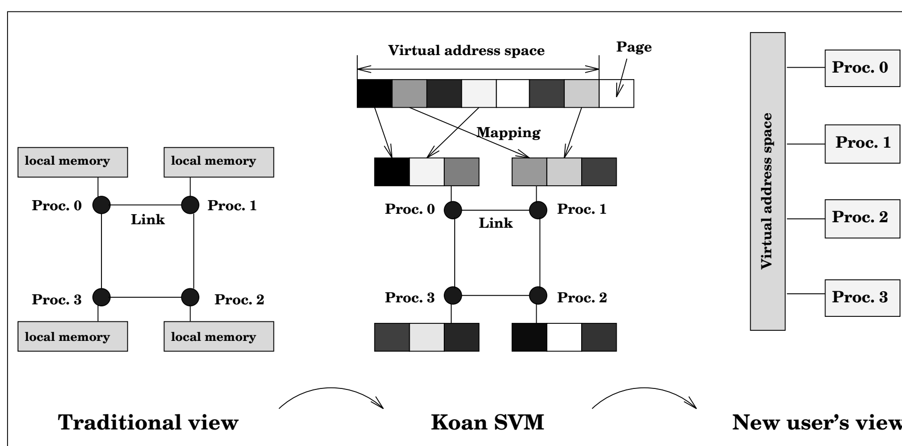

Figure 5: The Shared Virtual Memory abstraction

nodes. Each node includes an *i80386* (16 Mhz), augmented by an *i80387* floating point processor (300 kflops peak), and 4 Mbytes of main memory partitioned into pages of 4096 Bytes each. These nodes are interconnected in a hypercube topology. The minimum time to send a small message between two nodes is 0.3ms. To send a message of one page size, 2.7 ms are necessary.

KOAN provides the user with a virtual address space, shared by all the processors. It is partitioned into pages of equal size (4096 bytes, afforded by the MMU) which are spread among the processors' local memories. Furthermore, the shared space can be split into a set of regions of arbitrary size, each of them managed with a different coherence protocol [14]. The data stored in a read/write mode (the sparse matrix) is managed with a strong coherency : several *read* copies of a page may coexist on different processors, but they are invalidated as soon as a *write* operation occurs. Synchronization mechanisms are provided to manage mutual exclusion. *KOAN* is also supplied with performance measurement tools (all statistics and timings about an execution), and a trace generator facility.

4.2 Choices and motivations

Ideally, one would use a shared virtual memory machine just like a physically shared memory machine. We implemented the supernodal Fan-Out algorithm supplied with the SPLASH benchmark [17], which is written for a shared memory parallel computer [8]. However, results of this experiment show that a shared virtual memory machine should not be used as a real shared memory machine. The main problems encountered with this algorithm are :

- scheduling : a dynamic task scheduling, using a task queue in the shared address space, is not efficient due to multiple access conflicts ;
- critical sections : this algorithm requires $\eta(L) - n$ column lockings and as many task queue lockings as there are supernodes in the matrix L ; this is too much on a machine on which locks are not efficiently implemented ;
- as a processor executes a task T_k, it may write to almost all the right part of the matrix (pages containing columns in $Struct(L_{*k})$) causing page invalidations and high data traffic.

Efficient dynamic load balancing with low overhead on DMPCs seems hard to achieve. Therefore, we opt for a static task assignment on the basis of a level numbering of the elimination tree, so that we suppress the access control to a task queue. In order to minimize page conflicts and to suppress column locks, left-looking methods are much more interesting. Recall that we use a large grain task model. Hence, a column is modified by only one processor. The remaining page conflicts are due to false sharing, since padding is not realistic with too large a page size.

4.3 Events-based Fan-In algorithm

As stated earlier, our algorithm is straightforwardly derived from the sequential left looking algorithm. It can also be seen as a Fan-In algorithm since all updates to a column are gathered into a single "atomic" task. All these updates are computed by one processor whereas in the distributed Fan-In they are distributed on several processors. When a task j is completed, the column j is ready to modify columns in $Struct(L_{*j})$. Since $Cmod(k, j)$ are executed on different processors depending on the scheduling of the tasks T_k, these processors must be informed that the task T_j is completed. There are several solutions to this synchronization problem, including shared arrays and events. In our opinion, events lead to better performance, since they can be easily and efficiently handled at the system kernel level by means of messages. Two function calls, namely `signal(e)` and `wait(e)`, allow posting an event and waiting for an event to be signaled. In our implementation, `signal(e)` just broadcasts a message to all processors and `wait(e)` is a blocking function call. Each processor maintains a local copy of all events that occurred during the execution and no event is destroyed when accessed. The algorithm we introduce is depicted in figure 6. Note that this algorithm does not take advantage of any special scheduling technique : the outer loop is simply distributed on the available processors in a cyclic way. Recall that the optimal scheduling of a task graph (even a tree) on several processors is NP-complete. However, it is interesting to see the effects of different scheduling heuristics on both performances and memory reference locality. In [11], Liu discusses the use of a self-scheduling loop driven by a global variable, but as explained earlier, we avoid all kinds of dynamic scheduling.

```
Do j = Proc_number, n step Number_of_processors

   Do k ∈ Struct(L_j*)
      Wait-event(k)
      Cmod(j, k)
   Enddo

   Cdiv(j)
   Post-event(j)
Enddo
```

Figure 6: Large-grain events-based algorithm

The problem we address now is how to find a good static ordering of n tasks T_i of unequal duration. Dependence constraints between these tasks are expressed by a task graph (directed acyclic graph) which is rooted at T_n. For each node i of this graph, we define :

w_i : task weight which is proportional to the task T_i duration ;

h_i : height of the node defined as :

$$h_i = \begin{cases} 0 & \text{if } T_i \text{ has no predecessor} \\ \max\limits_{succ(k) \ni i} (h_k) + 1 & \text{otherwise} \end{cases}$$

d_i : depth of the node defined as :

$$d_i = \begin{cases} 0 & \text{if } i = n \\ \max\limits_{k \in succ(i)} (d_k) + 1 & \text{otherwise} \end{cases}$$

d_i^w : weighted depth of the node defined as :

$$d_i^w = \begin{cases} w_i & \text{if } i = n \\ \max\limits_{k \in succ(i)} (d_k^w) + w_i & \text{otherwise} \end{cases}$$

For our algorithm, the task graph is a tree, and the tasks are weighted with the number of floating point operations they perform to complete. Table 1 gives the weight, height, depth and weighted depth of each task for the matrix of figure 2 .

In order to introduce the experimented scheduling strategies we define levels of tasks : tasks in a low level have higher priority in the scheduling. These levels are defined as follows :

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
w	3	3	5	3	4	10	11	8
h	0	0	1	0	0	1	2	3
d	2	3	2	3	3	2	1	0
d^w	22	27	24	32	33	29	19	8

Table 1: Weight, height, depth, weighted depth of the sample problem tasks

1. simple order (**SO**) : a particular level numbering assigns each task T_i to the level i.
2. Bottom-Up browsing of the elimination tree (**BU**)[2] : each task T_i is assigned to a level which is its height h_i.
3. Weighted Bottom-Up (**WBU**) : each task T_i is assigned to a level which is its height h_i, and tasks within the same level are ordered with respect to their weighted depth d^w : T_i before $T_j \Longleftrightarrow d_i^w \geq d_j^w$.
4. Top-Bottom browsing of the elimination tree (**TB**)[3] : each task T_i is assigned to a level which is its depth d_i.
5. Weighted Top-Bottom (**WTB**) : each task T_i is assigned to a level which is $\max_{k=1}^{n}(d_k) - d_i$, and tasks within the same level are ordered with respect to their weighted depth d^w : T_i before $T_j \Longleftrightarrow d_i^w \geq d_j^w$.
6. Critical path (**CP**) : tasks are assigned to levels according to : $level(T_i) < level(T_j) \Longleftrightarrow d_i^w \geq d_j^w$.

The simple order is the easiest strategy to implement. The bottom-up strategy schedules a task as soon as possible whereas the top-bottom one schedules a task as late as possible. For these two techniques, ordering the tasks within the same level by their weighted depth may reduce processor stall time, while keeping a simple order makes a better use of locality. Finally, the critical path strategy, also known as the *longuest path* or *largest processing time*, is known to produce near-optimal schedules [10].

5 Experiments

Up till now, we defined scheduling strategies that may influence load balancing. In this section, we report on our first results and we discuss issues closely related to the SVM concept. Two matrices among the three we used are from the

[2] Also known as forward scheduling.
[3] Also known as backward scheduling.

Boeing-Harwell collection and the third one (GH1-01) is a randomly generated matrix. All of them are reordered using the minimum degree heuristic. More details are provided in table 2.

matrix	*order*	$\eta(A)$	$\eta(L)$	# pages
LSHP3466	3 466	23 896	86 582	170
BCSSTK14	1 806	32 630	112 267	220
GH1-01	1 024	5 754	109 701	215

Table 2: Test matrices

Tables 3 and 4 show the timing results and the efficiency of the parallel code obtained with a hypercube running KOAN. They are computed on the basis of a pure sequential code, without any parallel construct.

Some primary conclusions arise from these results :

- a predictable result is that using levels in the elimination tree greatly improves performances (from 3% to 325% on the test matrices) ,
- efficiency is much better with the GH1-01 matrix than with the two others, and the scheduling has almost no effect with the latter matrix ,
- the critical path strategy does not give results as good as one would have expected.

Matrix	Sequential code	16 processors					
		SO	BU	WBU	TB	WTB	CP
BCSSTK14	88.16	44.21	12.67	12.53	14.77	14.38	27.36
LSHP3466	37.14	53.57	13.74	13.45	14.74	14.72	33.44
GH1-01	281.85	26.73	24.93	25.15	25.70	25.50	26.30

Table 3: Execution times (in seconds) with different scheduling strategies

In order to explain these two last points, we need a detailed analysis on data access. We should first point out that in our implementations, all column modifications are written in a private vector. After the completion of the *Cdiv* operation, the resulting vector is then copied to the shared virtual memory address space. In other words, the SVM is used as frequently as needed for read operations, but only necessary and useful write operations are done in it. Thus, for a matrix of order n, only n *vector-writes* are done in the SVM regardless of the number of processors contributing to the computation. We used this

Matrix	16 processors										
	SO	*BU*		*WBU*		*TB*		*WTB*		*CP*	
	e	*e*	*% i*	*e*	*% i*	*e*	*% i*	*e*	*% i*	*e*	*% i*
BCSSTK14	0.12	0.43	258	0.44	267	0.37	208	0.38	217	0.20	66
LSHP3466	0.04	0.17	325	0.17	325	0.16	300	0.16	300	0.07	75
GH1-01	0.66	0.71	7.5	0.70	6	0.69	4.5	0.69	4.5	0.67	3

Table 4: Efficiency *(e)* of different scheduling strategies and improvement *(i)* relative to SO

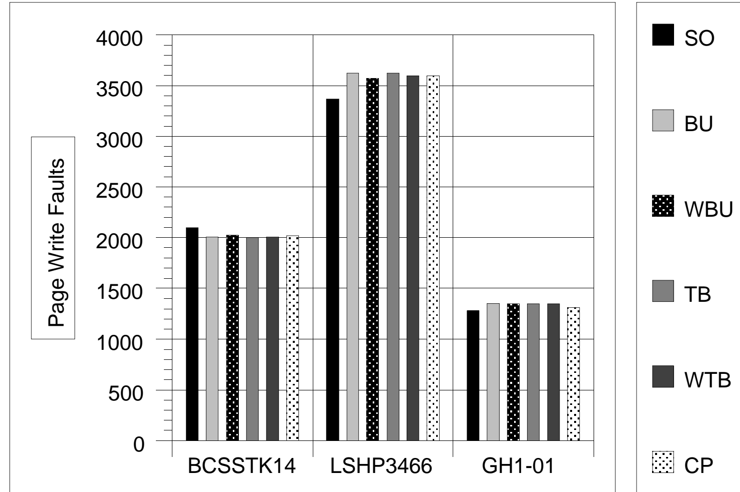

Figure 7: Write faults for different scheduling strategies (16 procs).

technique in consequence to previous studies [13] which proved the usefulness of buffered writes.

The availability of accessed data in the local memory is an important issue for this study. It is strongly related to the page sharing of columns. As can be deduced from table 2, the average number of columns that share the same memory page is 8.2 for the BCSSTK14 matrix, 20.3 for the LSHP3466 and 4.7 for the GH1-01. This leads to a false sharing problem since dependencies are between columns while coherency is at a page level. The results that we present hereafter do not take into account this difference, and read/write misses may be due to false sharing.

As shown in figure 7, the number of write misses is nearly identical for all the scheduling strategies on 16 processors. This is due to the fact that this number of page write faults is close to the maximum number of page write faults that may occur. Indeed, if we suppose that a processor can write a vector of less than one page long without being interrupted by another processor needing

that page, then it will generate at worst two page faults: the worst case is when the column is mapped astride two pages, and none of them is present in the local cache of the writing processor. As we can see, the number of page faults that really occurred during the execution is in a margin of less than 8% of our estimation of the worst case. On the other hand, figure 8 show that even with a small number of processors, the amount of write misses is very high. In addition, using more processors increases data traffic on the connection network and increases also the probability that several processors conflict with each other. This has a straightforward repercussion on the write fault service time, as shown on figure 8.

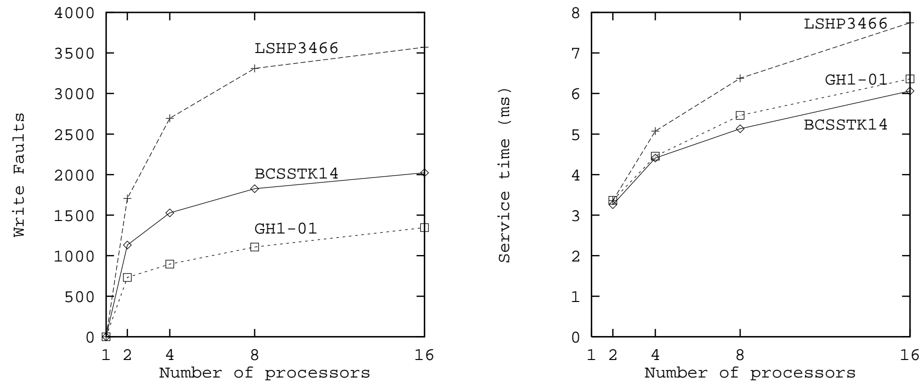

Figure 8: Evolution of write faults and mean service time of a write fault for WBU scheduling.

Apart from the write misses, the false sharing and the need for communicating updated data causes read misses. Indeed, a write operation invalidates all existing copies of the written page. Later read accesses in that page on a different processor from the writing one generates a read fault. Figure 9 shows the amount of page read faults that occurred during the execution of the various scheduling strategies on 16 processors. Furthermore, it shows the maximum amount of time spent by one processor in the system waiting for a page delivery (either read or write fault). From both of these figures, we can see the reason for the poor performance when scheduling with *SO* and *CP*. These two scheduling techniques involve access patterns that exhibit poor data reuse (locality) and more read/write conflicts. This is confirmed by table 5 which gives the average number of pages that must be read or written by each processor to complete all of its tasks (among the sixteen processors used) and the corresponding standard deviation.

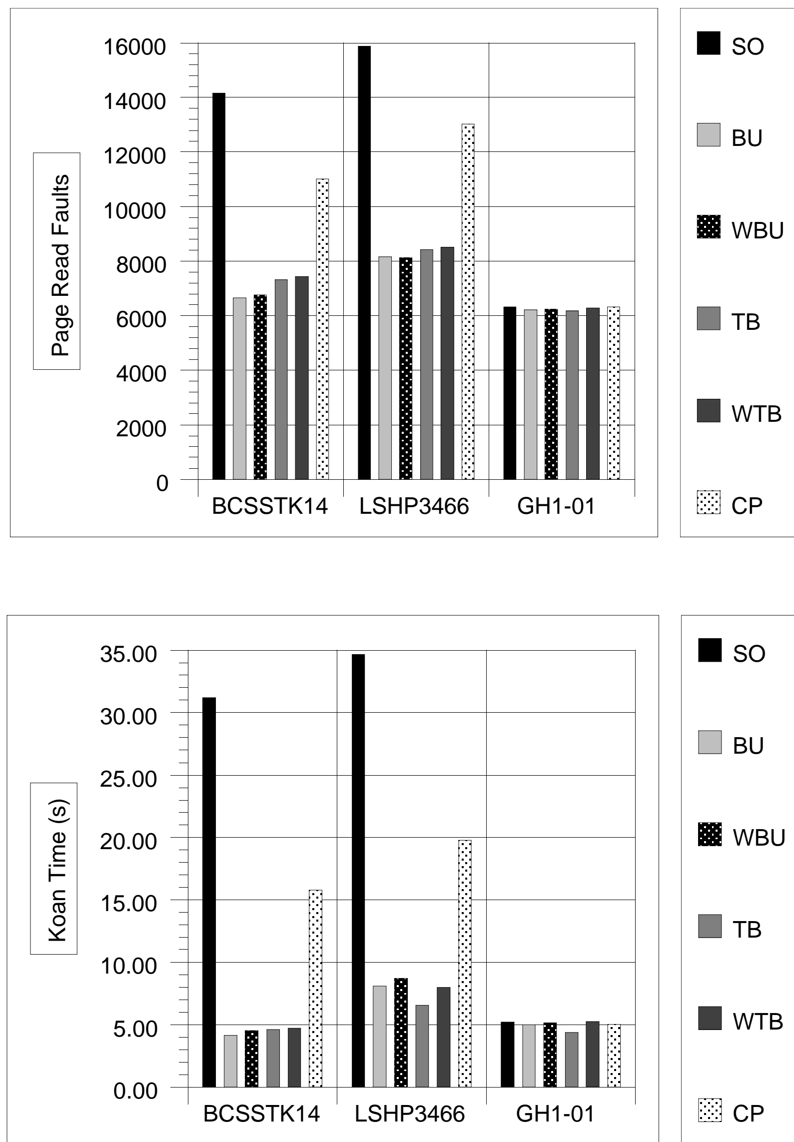

Figure 9: Read faults and KOAN time for various scheduling strategies (16 procs).

		SO	BU	WBU	TB	WTB	CP
BCSSTK14	Read	220	219.06	219.5	219.43	219.18	219
	$\sqrt{\sigma}$	0	0.96	0.61	0.86	0.95	1
	Write	119.81	91.75	91.93	94.18	96.25	106.37
	$\sqrt{\sigma}$	2.89	3.36	3.88	3.18	4.42	2.71
LSHP3466	Read	169.56	167.93	168.25	167.81	167.75	169.18
	$\sqrt{\sigma}$	0.49	1.47	1.39	1.01	1.82	0.88
	Write	133.62	119.31	117.62	112.75	112.87	124.5
	$\sqrt{\sigma}$	1.93	3.21	3.37	3.56	4.44	3.12
GH1-01	Read	214.93	214.93	214.93	214.93	214.93	214.93
	$\sqrt{\sigma}$	0.24	0.24	0.24	0.24	0.24	0.24
	Write	58.06	56.06	57.18	56.68	56.93	57.25
	$\sqrt{\sigma}$	2.63	4.29	2.29	4.25	2.38	3.11

Table 5: working sets

6 Conclusion

In this paper, a large grain algorithm for the Cholesky factorization using synchronization by events and a shared variables is introduced. The main novelty relies on its implementation on a DMPC running the KOAN software-based shared virtual memory. The benefit of a large grain algorithm is the reduced number of necessary synchronization points. The usefulness of a globally addressable memory is that it makes the parallelization phase simpler, and as shown, leads to an interesting performance. However, the efficiency of the parallel machine can be poor because of the sparse nature of the data : depending on the degree of sparsity, more or fewer columns will be stored in the same page of memory ; if the algorithm exhibits some locality, it is worthwhile initiating a page transfer on a vector read access since the columns stored in that page will be used. Otherwise, several columns will be brought, but only one of them will be used and thus, the page transfer mechanism is not cost-effective. The problem is more intricate in producer/consumer schemes : a processor needs to bring back the same page if it is invalidated while it is reading in it. We can see this situation on the bar charts showing page read faults and invalidations.

The other problem related to page access conflicts is the increase of page faults service time when several processors need the same page at the same time. If we consider a small number of processors, the conflicts do not really affect the KOAN response time. By contrast, this problem has to be taken into account as soon as the number of processors increases.

For our matrices test set, the false sharing has a greater influence on performance than load imbalance. Consequently, the *CP* scheduling strategy, which is

known to be near-optimal, leads to poor processor utilization because of memory management problems.

To solve the false sharing problem, or at least to minimize its drawbacks, one can imagine using smaller memory pages. On the one hand, fewer columns will be stored in the same page, reducing the false sharing. On the other hand, the latency will be paid more frequently and the communication network may become overloaded.

Another partial solution to the false sharing problem may rely on an adequate coherency protocol. The weak coherence afforded by KOAN showed itself to be very interesting in similar cases, especially on algorithms that exhibit a synchronized succession of 1 producer/ several consumer phases. Relying on the synchronization by events as introduced in this paper, the weak coherency protocol can be extended to support several producers/several consumers in asynchronous phases.

References

[1] Cleve ASHCRAFT, Stanley EISENSTAT, and Joseph LIU. A fan-in algorithm for distributed sparse numerical factorization. *Siam journal of scientific and statistical computations*, 593–599, 1990.

[2] Iain DUFF. Parallel implementation of multifrontal schemes. *Parallel computing*, 193–204, 1986.

[3] Iain DUFF, Danny SORENSEN, Henk VAN DER VORST, and Jack DONGARRA. *Solving Linear Systems on Vector and Shared-Memory Computers*, chapter Direct Solution of Sparse Linear Systems, pages 109–142. Siam, 1991.

[4] Alan GEORGE, Michael HEATH, Joseph LIU, and Esmond NG. Solution of sparse positive definite systems on a shared memory multiprocesor. *International journal of parallel programming*, 309–325, 1986.

[5] Alan GEORGE, Michael HEATH, Joseph LIU, and Esmond NG. Sparse cholesky factorization on a local-memory multiprocesor. *Siam journal of scientific and statistical computations*, 327–340, 1988.

[6] Alan GEORGE and Joseph LIU. An automatic nested dissection algorithm for irregular finite element problems. *Siam journal of numerical analysis*, 1053–1069, October 1978.

[7] Alan GEORGE, Joseph LIU, and Esmond NG. Communication reduction in parallel sparse cholesky factorization on a hypercube. pages 576–586, Hypercube 1987.

[8] Mounir HAHAD. *Parallel Sparse Cholesky Factorization : Performace Study on the KOAN System.* DEA thesis (in french), IFSIC/IRISA, 1992.

[9] Zakaria LAHJOMRI and Thierry PRIOL. Koan : a shared virtual memory for an ipsc/2 hypercube. In *CONPAR/VAPP*, September 1992.

[10] Ted G. LEWIS and Hesham EL-REWINI. *Introduction to Parallel Computing*, chapter Scheduling Parallel Programs, pages 245–282. Prentice-Hall, 1992.

[11] J.W.H. LIU. Computational models and task scheduling for parallel sparse cholesky factorization. *Parallel computing*, 327–342, 1986.

[12] J.W.H. LIU. The role of elimination trees in sparse factorization. *Siam journal of matrix analysis applications*, 134–172, 1990.

[13] Thierry MONTAUT and François BODIN. Performance analysis of a shared virtual memory: communication costs experiments. 1993. Unpublished.

[14] Thierry PRIOL and Zakaria LAHJOMRI. Experiments with shared virtual memory and message-passing on an ipsc/2 hypercube. In *International Conference on Parallel Processing*, pages 145–149, August 1992.

[15] Edward ROTHBERG and Anoop GUPTA. *A Comparative Evaluation of Nodal and Supernodal Parallel Sparse Matrix Factorization: Detailed Simulation Results.* Technical Report, Department of computer science-Stanford university, 1990.

[16] Edward ROTHBERG and Anoop GUPTA. Techniques for improving the performance of sparse matrix factorization on multiprocessor workstations. In *Supercomputing 90*, pages 232–241, November 1990.

[17] Jaswinder SINGH, Wolf-Dietrich WEBER, and Anoop GUPTA. Splash: stanford parallel applications for shared-memory. Computer systems laboratory, Stanford. 1991.

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105,
78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
(France)
ISSN 0249-6399