ELSEVIER

Parallel Computing 28 (2002) 301–321

PARALLEL COMPUTING

www.elsevier.com/locate/parco

PaStiX: a high-performance parallel direct solver for sparse symmetric positive definite systems

P. Hénon, P. Ramet *, J. Roman

LaBRI, UMR CNRS 5800, Université Bordeaux 1 et ENSEIRB, 351, cours de la Libération, F-33405 Talence, France

Abstract

Solving large sparse symmetric positive definite systems of linear equations is a crucial and time-consuming step, arising in many scientific and engineering applications. The block partitioning and scheduling problem for sparse parallel factorization without pivoting is considered. There are two major aims to this study: the scalability of the parallel solver, and the compromise between memory overhead and efficiency. Parallel experiments on a large collection of irregular industrial problems validate our approach. © 2002 Elsevier Science B.V. All rights reserved.

Keywords: High performance computing; Parallel sparse Cholesky factorization; Static partitioning and scheduling

1. Introduction

Solving large sparse symmetric positive definite systems $Ax = b$ of linear equations is a crucial and time-consuming step, arising in many scientific and engineering applications. Consequently, many parallel technics for sparse matrix factorization have been studied and implemented; see [12] for a complete survey on high performance sparse factorization.

* Corresponding author.
E-mail address: ramet@labri.u-bordeaux.fr (P. Ramet).

0167-8191/02/$ - see front matter
PII: S0167-8191(01)00141-7

From a practical point of view, we have mainly worked on the parallelization of an industrial vectorized code for structural mechanics, which is a 2D and 3D finite element code and nonlinear in time. This computational finite element code solves plasticity (or thermo-plasticity, possibly coupled with large displacements) problems. Since the matrices of these systems do not have good properties, classical iterative methods do not behave well. Therefore, as we want to obtain an industrial software tool that would be robust and versatile, we must use high-performance direct sparse solvers; moreover, as we need to solve very large sparse systems (more than one million unknowns for 3D problems), parallelism is necessary for reasons of memory capabilities and acceptable solving time [17, and included references].

In this paper, we focus on the block partitioning and scheduling problem for high performance sparse LDL^T factorization without pivoting on parallel architectures; in fact, our strategy is suitable for general distributed heterogeneous architectures whose computation and communication performances are predictable in advance. We use LDL^T factorization in order to solve some electro-magnetism problems that lead to symmetric nonsingular systems with complex coefficients.

In order to achieve efficient parallel sparse factorization, we perform three sequential pre-processing phases:

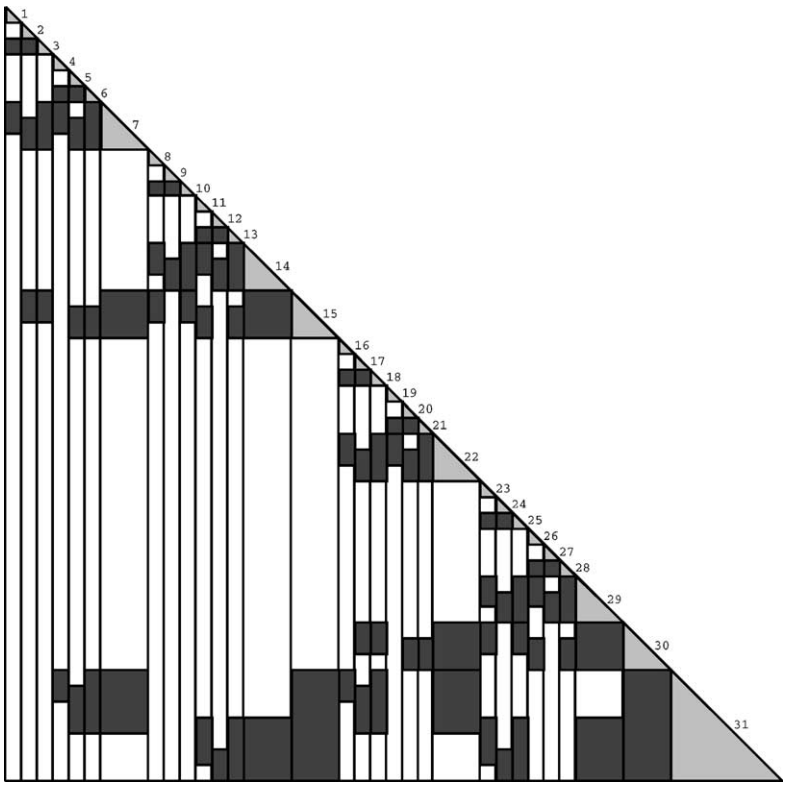

Fig. 1. Block data structure of a factorized matrix.

- The *ordering* phase, which computes a symmetric permutation of the initial matrix A such that factorization process will exhibit as much concurrency as possible while incurring low fill-in. In this work, we use a tight coupling of the Nested Dissection and Approximate Minimum Degree (AMD) algorithms [1,28]. The partition of the original graph into supernodes is achieved by merging the partition of separators computed by the Nested Dissection algorithm and the supernodes amalgamated for each subgraph ordered by Halo Approximate Minimum Degree.
- The *block symbolic factorization* phase, which determines the block data structure of the factorized matrix L associated with the partition resulting from the ordering phase. This structure consists of N column-blocks, each of them containing a dense symmetric diagonal block and a set of dense rectangular off-diagonal blocks (see Fig. 1). One can efficiently perform such a block symbolic factorization in quasi-linear space and time [10]. From the block structure of L, we can deduce the weighted elimination quotient graph that describes all dependencies between blocks, as well as the supernodal elimination tree. The block algorithm that we use is highly cache-friendly; block data structures are much more compact than column-compressed storage [28]. The blocks of the factorized matrix are computed and appended, column-block by column-block, to a growing block array, such that the blocks of the already-computed contributing column-blocks are merged to build the blocks of the current column-block.
- The *block repartitioning and scheduling* phase, which refines the previous partition by splitting large supernodes in order to exploit concurrency within dense block computations, and which maps the resulting blocks onto the processors of the target architecture (see Section 2.2).

According to the properties of matrix A, we classically distinguish two types of factorization:

- If matrix A is symmetric positive definite, a Cholesky factorization ($A = \mathrm{LL}^{\mathrm{T}}$) or Cholesky–Crout factorization ($A = \mathrm{LDL}^{\mathrm{T}}$), with or without symmetric numerical pivoting, can be used.
- If matrix A is unsymmetric, one must use a LU factorization with unsymmetric pivoting.

There are two main approaches for numerical factorization algorithms: the *multifrontal* approach [2,11,13,14,18,31], and the *supernodal* [19,30,32,33] with *fan-in* or *fan-out* variations [5–8]. Both can be described by a computational tree whose nodes represent computations and whose edges represent transfer of data. In the case of the multifrontal method, at each node, some steps of Gaussian elimination are performed on a dense frontal matrix and the remaining Schur complement, or contribution block, is passed to the parent node for assembly. In the case of the supernodal method, the distributed memory version uses a right-looking formulation which, having computed the factorization of a column-block corresponding to a node of the tree, then immediately sends the data to update the column-blocks corresponding to ancestors in the tree. In a parallel context, we can locally aggregate contributions to the same block before sending the contributions. This can significantly reduce the number of messages.

Independently of these different methods, a static or dynamic scheduling of block computations can be used. For homogeneous parallel architectures, it is useful to find an efficient static scheduling [15,20,29]. In this context, this scheduling can be induced by a fine cost computation/communication model.

The PSPASES solver [22] is based on a multifrontal approach without pivoting for symmetric positive definite systems. It uses METIS [23] for computing a fill-reducing ordering which is based on a multilevel nested dissection algorithm; when the graph is separated into p parts, one subgraph per processor, a multiple minimum degree (MMD [26]) is then used. A "subtree to subcube" algorithm is applied to build a static mapping before the numerical factorization.

In [4] the performances of MUMPS [3] and SUPERLU [25] are compared for non-symmetric problems. MUMPS uses a multifrontal approach with dynamic pivoting for stability while SUPERLU is based on a supernodal technique with static pivoting. The standard ordering used by MUMPS is the AMD [1] ordering, while SUPERLU uses the multiple minimum degree (MMD [26]) ordering. In both cases, a pivot order is defined by the symbolic factorization stage, but numerical considerations might prevent strict adherence to this order during numerical factorization. MUMPS can choose pivots off of the diagonal: the modulus of the prospective pivot is compared with the largest modulus of an entry in the column and the pivot is only accepted if this modulus is greater than a threshold value. In the SUPERLU approach, a static pivoting strategy is used and kept rigorously to the pivot sequence chosen in the symbolic analysis.

The PARDISO solver [34] is an efficient implementation of a parallel sparse direct factorization for unsymmetric and symmetric matrices in the context of shared memory architectures. Parallelism is exploited with a combination of left-looking and right-looking supernodal algorithms; the parallel pivoting methods use either complete pivoting or Bunch and Kaufmann supernode pivoting [9] in order to reach a compromise between numerical stability and scalability during the factorization process.

In this paper, we focus on the block partitioning and scheduling problem for high performance sparse supernodal LDL^T factorization without pivoting for symmetric positive definite systems. Thus, our algorithmic framework is close to the PSPASES one [22]. We presented in [20,21] a preliminary version of this work describing a mapping and scheduling algorithm based on a combination of 1D and 2D block distributions. This algorithm computes an efficient static schedule of the block computations for a parallel solver based on a supernodal approach such that the parallel solver is fully driven by this scheduling. This can be done by very precisely taking into account the computational costs of the BLAS 3 primitives, the communication cost and the cost of local aggregations. Our study is suitable for heterogeneous parallel/distributed architectures whose performances are predictable by using a cost model for computations and communications. This paper extends this preliminary work by presenting a complete description of our mixed 1D/2D distribution strategy, a description and an analysis of partial aggregation technique (for memory constraints), and finally complete numerical experiments for case studies with more than 1.5×10^6 unknowns on IBM SP2 architecture.

The paper is organized as follows. In Section 2, we introduce the algorithmic framework for parallel sparse symmetric factorization before to describe our block repartitioning and scheduling algorithm. Section 3 provides many numerical experiments on an IBM SP2 for a representative class of large sparse matrices from industrial problems, including performance results and analysis. According to these results, our PaStiX software appears to be a good competitor with the current reference PSPASES software [22] for symmetric matrices. Finally, we conclude with remarks concerning the benefits of this study, with preliminary results for the two biggest case studies on an IBM SP3, and finally with some prospects of our future work.

2. Description of algorithms

We first introduce the numerical factorization algorithm in Section 2.1 in order to facilitate the description of our static partitioning and mapping algorithm in Section 2.2. We conclude this part by presenting some important extensions in Section 2.3.

2.1. Parallel factorization algorithm

Let us consider the block data structure of the factorized matrix L computed by the block symbolic factorization. Recall that each of the N column-blocks holds one dense diagonal block and some dense off-diagonal blocks. From this block data structure, we can introduce the boolean function $off\_diag(k,j)$, $1 \leqslant k < j \leqslant N$, that returns *true* if and only if there exists an off-diagonal block in column-block k facing column-block j (in what follows, we will say "block j in the column-block k"). Then we define the two sets:

$$BStruct(L_{k*}) := \{i < k | off\_diag(i,k) = true\},$$
$$BStruct(L_{*k}) := \{j > k | off\_diag(k,j) = true\}.$$

Thus, $BStruct(L_{k*})$ is the set of column-blocks that update column-block k, and $BStruct(L_{*k})$ is the set of column-blocks updated by column-block k. In Fig. 1 we have $BStruct(L_{7*}) := \{1,2,3,4,5,6\}$ and $BStruct(L_{*7}) := \{15,31\}$.

Let us now consider a parallel supernodal version of sparse LDL^T factorization with total local aggregation: all nonlocal block contributions are aggregated locally in block structures. This scheme is close to the Fan-In algorithm [8] as processors communicate using only aggregated update blocks. The proposed algorithm can yield 1D (column-block) or 2D (block) distributions. If memory is a critical issue, an aggregated update block can be sent with partial aggregation to free memory space. So, a block $j \geqslant k$ in column-block k will receive an aggregated update block only from every processor in the set

$$Procs(L_{jk}) := \{map(j,i) \,|\, i \in BStruct(L_{k*}) \text{ and } j \in BStruct(L_{*i})\},$$

where the $map(,)$ operator is the 2D block mapping function. These aggregated update blocks, denoted in what follows by AUB_{jk}, can be built from the block symbolic factorization. These contributions are locally aggregated before being sent.

The pseudo-code of $\mathrm{LDL}^{\mathrm{T}}$ factorization can be expressed in terms of dense block computations or *tasks*; these computations are performed, as much as possible, on compacted sets of blocks for BLAS efficiency.

Let us introduce some notations:

- P : total number of processors;
- N_p : total number of local tasks for processor p;
- $K_p[n]$: nth local task for processor p;
- S_q : set of all AUBs to be sent to processor q;
- for the column-block k, symbol $\bigstar$ means $\forall j \in BStruct(L_{*k})$;
- let $j \geqslant k$; sequence $[j]$ means $\forall i \in BStruct(L_{*k}) \cup \{k\}$ with $i \geqslant j$.

Block computations can be classified in four types denoted by `COMP1D`, `FACTOR`, `BDIV` and `BMOD`. The associated tasks are defined as follows (p is the local processor number):

- `COMP1D(k)`: *factorize the column-block k and compute all the contributions for the column-blocks in* $BStruct(L_{*k})$
  Factorize A_{kk} into $L_{kk}D_kL_{kk}^{\mathrm{T}}$
  Solve $L_{kk}F_{\bigstar}^{\mathrm{T}} = A_{\bigstar k}^{\mathrm{T}}$ and $D_kL_{\bigstar k}^{\mathrm{T}} = F_{\bigstar}^{\mathrm{T}}$
  For $j \in BStruct(L_{*k})$ **Do**
    Compute $C_{[j]} = L_{[j]k}F_j^{\mathrm{T}}$
    If $map([j], j) == p$ **Then** $A_{[j]j} = A_{[j]j} - C_{[j]}$
    Else $\mathrm{AUB}_{[j]j} = \mathrm{AUB}_{[j]j} + C_{[j]}$
- `FACTOR(k)`: *factorize the diagonal block k*
  Factorize A_{kk} into $L_{kk}D_kL_{kk}^{\mathrm{T}}$
- `BDIV(j,k)`: *update the off-diagonal block j in column-block k*
  Solve $L_{kk}F_j^{\mathrm{T}} = A_{jk}^{\mathrm{T}}$ and $D_kL_{jk}^{\mathrm{T}} = F_j^{\mathrm{T}}$
- `BMOD(i,j,k)`: *compute the contribution of the block i in column-block k for block i in column-block j*
  Compute $C_i = L_{ik}F_j^{\mathrm{T}}$
  If $map(i, j) == p$ **Then** $A_{ij} = A_{ij} - C_i$
  Else $\mathrm{AUB}_{ij} = \mathrm{AUB}_{ij} + C_i$

Then, the pseudo-code of $\mathrm{LDL}^{\mathrm{T}}$ factorization for processor p is shown in Fig. 2.

On each processor p, K_p is the vector of tasks for local computations (lines 2 on Fig. 2), ordered by priority (see Section 2.2). Each task should have received all its contributions and should have updated associated local data before any new contribution is computed. When the last contribution is aggregated in the corresponding AUB, this aggregated update block is said to be "completely aggregated" and is ready to be sent. To achieve a good efficiency, the sending of AUB have to match the static scheduling of tasks on the destination processor (Fig. 3). That is why we define, in the `Sending_Phase` (lines 5 and 14 in Fig. 2), the property $\mathscr{P}$ that is true when this matching is verified (the property is introduced in Section 2.2).

```
1. For n = 1 to N_p Do
2.   Switch ( Type(K_p[n]) ) Do
3.     COMP1D: Receive all AUB_[k]k for A_[k]k
4.       COMP1D(k)
5.       Sending_Phase()
6.     FACTOR: Receive all AUB_kk for A_kk
7.       FACTOR(k)
8.       send L_kk D_k to map([k], k)
9.     BDIV: Receive L_kk D_k and all AUB_jk for A_jk
10.      BDIV(j,k)
11.      send F_j^T to map([j], k)
12.    BMOD: Receive F_j^T
13.      BMOD(i,j,k)
14.      Sending_Phase()
```

Fig. 2. Outline of the parallel factorization algorithm.

2.2. Partitioning and mapping phase

Before running the general parallel algorithm presented above, we must perform a step consisting of partitioning and mapping the blocks of the symbolic matrix onto the set of processors. The partitioning and mapping phase aims at computing a static regulation that balances workload and enforces the precedence constraints imposed by the factorization algorithm; the block elimination tree structure must be used there. Different levels of parallelism can be exhibited:

- The first level is induced by the sparseness of the matrix and corresponds to independent branches in the block elimination tree.
- The second level is induced by dense computation on large full blocks. The partitioning phase is assigned to split such blocks.
- The third level exploits instruction-level parallelism at the processor level by using BLAS subroutines.

The performance of the partitioning and mapping phase can be viewed in terms of its ability to exploit these three levels of parallelism.

The existing approaches for block partitioning and mapping give rise to several problems, which can be divided into two categories: on the one hand, problems due to the measure of workload, and on the other hand those due to the runtime schedule of block computations in the solver.

```
Sending_Phase():
. For q = 1 to P, q ≠ p, Do
.   If S_q ≠ ∅ Then
.     E_q = {AUB ∈ S_q / P(AUB) is true}
.     Send E_q to q; S_q = S_q \ E_q
```

Fig. 3. Outline of the sending phase.

The measure of workload is usually very rough because only numbers of operations are taken into account. However, in order to be efficient, solver algorithms are block-oriented to take advantage of BLAS subroutines, whose efficiencies are very far from being linear in terms of numbers of operations. Moreover, workload encompasses block computations but does not take into account all of the other phenomena that occur in parallel factorization, such as extra workload generated by the local aggregation of contributions and idle waits due to the latency generated by message passing.

The obtaining of high performance at runtime requires computational efficient solutions to several scheduling problems that define the orders in which one

- processes tasks that are locally ready, which is crucial for minimizing idle time (vector K_p in line 2 of Fig. 2);
- sends and processes the receiving of blocks and of aggregate update blocks used by `BDIV` or `BMOD` tasks, which determines what block will be ready next for local computation (line 3, 5, 6, 8, 9, 11, 12 and 14 of Fig. 2).

We tackle all of these problems by a static regulation led by a time simulation during the mapping phase. Thus, the partitioning and mapping step generates a fully ordered schedule used in the parallel factorization. This schedule aims at statically regulating all of the issues that are classically managed at runtime. To make our scheme very reliable, we estimate the workload and message passing latency by using a BLAS and communication network time model, which is automatically calibrated on the target architecture.

Unlike usual algorithms, our partitioning and distribution strategy is divided in two distinct phases. The partition phase splits column-blocks associated with large supernodes, builds, for each column-block, a set of candidate processors for its mapping, and determines if it will be mapped using a 1D or 2D distribution. Once the partitioning step is over, the task graph is built. In this graph, each task is associated with the set of candidate processors of its column-block. The mapping and scheduling phase then optimally maps each task onto one of these processors.

The partitioning algorithm is based on a recursive top–down strategy over the block elimination tree provided by block symbolic factorization. Pothen and Sun presented such a strategy in [29]. It starts by splitting the root and assigning it to a set of candidate processors Q that is the set of all processors. Given the number of candidate processors and the size of the supernodes, it chooses the strategy (1D or 2D) that the mapping and scheduling phase will use to distribute this supernodes. Then each subtree is recursively assigned to a subset of Q proportionally to its workload (Fig. 4).

Two points are important to notice in our method:

- Since a candidate processor is only a suggestion for the mapping and scheduling phase, we avoid any problem of rounding to integral numbers of processors by allowing a candidate processor to be in two sets of candidate processors for two subtrees having the same parent in the block elimination tree (for example processor 3 in Fig. 4).
- The column-blocks corresponding to large supernodes are split using the blocking size suitable to achieve good BLAS efficiency.

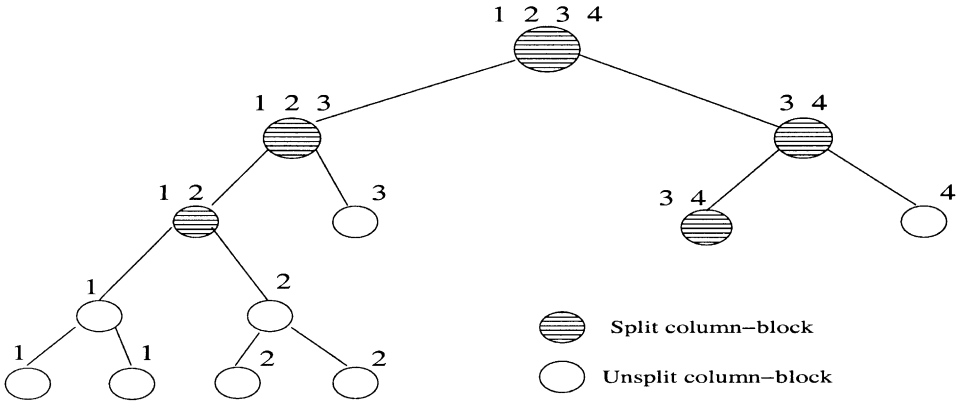

Fig. 4. Partitioning and sets of candidate processors.

Once the partitioning phase has built a new partition and the set of candidate processors for each task, the election of an owner processor for each task falls to the mapping and scheduling phase. The idea behind this phase is to simulate parallel factorization as each mapping comes along. Thus, for each processor, we define a timer that will hold the current elapsed computation time, and a ready task heap. At a given time, this task heap will contain all tasks that are not yet mapped, that have received all of their contributions, and for which the processor is a candidate. The algorithm then starts by mapping the leaves of the elimination tree (those which have only one candidate processor). When a task is mapped onto a processor, the mapper

- updates the timer of this processor according to our BLAS model;
- computes the time at which any contribution from this task to another is ready to be sent;
- puts into the heaps of their candidate processors all tasks for which all of the contributions have been computed.

After a task has been mapped, the next task to be mapped is selected as follows: we take the first task of each ready task heap and choose the one that comes from the lowest node in the elimination tree.

Now, we have to decide on which candidate processor a task is to be mapped. As shown in Fig. 5, the communication pattern of all the contributions for a task depends on the already mapped tasks and on the candidate processor for the ownership of this task. We therefore compute for each of the candidate processors, the time at which it will have completed the task if it is mapped onto it as a function of

- the processor timer;
- the time at which all contributions to this task have been computed (taking into account the overhead due to aggregation of local contributions);
- the communication cost model that gives the time needed to send the contributions.

The task is mapped onto the candidate processor that will be able to compute it the soonest.

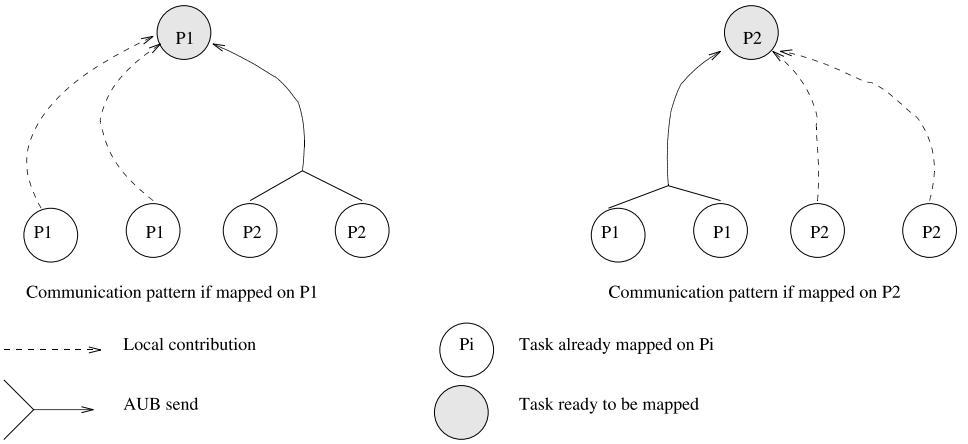

Fig. 5. Communication pattern of the contributions updating a task.

After this phase has ended, the computations of each task are ordered with respect to the rank at which the task have been mapped. Thus, for each processor p, we obtain a vector K_p of the N_p local task numbers fully ordered by priority (line 2 in Fig. 2). Thus, the vector K_p stands as the task computation scheme for processor p during the parallel computation.

On a given processor p, as defined in Section 2.1, we denote by S_q the set of all the aggregated contributions to be sent to the processor q. In addition, we suppose that all AUBs of S_q are ordered by the priority of their matching tasks on processor q. Thus, we denote by $S_q[i]$ the AUB $\in S_q$ with the ith rank. In this way, we define the property $\mathscr{P}$ introduced in Section 2.1 by

$$\mathscr{P}(S_q[i]) \text{ is true } \iff \begin{cases} S_q[i] \text{ is completely aggregated,} \\ \forall j < i,\ S_q[j] \text{ is completely aggregated.} \end{cases}$$

This communication scheme is fully compatible with the task computation scheme since, by construction, the sending of all the AUBs in S_q are compatible with the order of the task computations on q.

As a conclusion about the partitioning and mapping phase, we can say that we obtain a strategy that allows us to take into account, in the mapping of task computations, all the phenomena that occur during the parallel factorization. Thus we achieve a block computation and communication scheme that drives the parallel solver efficiently.

Another important point is that our strategy can take into account heterogenous architecture. For example, in the case of an architecture based on SMP nodes, the time to send an AUB from a processor p_1 to a processor p_2 is estimated using the intra-node communication model if p_1 and p_2 belong to the same SMP node, or using the extra-node communication model if p_1 and p_2 belong to different SMP nodes.

2.3. Static regulation under memory constraint

We have explained in the previous paragraph our strategy to improve the parallel factorization in terms of time. Another important problem that occurs in industrial problems is the performance in terms of space complexity; in particular, the memory overhead due to the extra-structures needed to manage distributed data and communication exchanges must be as small as possible. In the case of the LDL^T supernodal factorization with total local aggregation of contributions, this overhead cost is mainly caused by the local aggregation of contributions. Indeed, an aggregated update block AUB_{ij} on a given processor consists of an overlapping block structure of all the contributions from this processor to the block (i, j) mapped on another processor. Thus any AUB needs to be allocated in local memory, since its first contribution has been computed until it is updated by its last contribution and sent according to the communication scheme. The total amount of simultaneously allocated memory for AUBs can be very significant with respect to the amount of memory needed to store the distributed matrix. A solution to reduce this extra-memory requirement is to reduce the number of simultaneously allocated AUBs by sending some of them before they are completed. That is to say that we allow some of the AUBs to be sent before they have been updated by their last contribution, and to be reallocated when the next contribution following the sending of the partial AUB is computed. This method is called partial local aggregation.

Thanks to the task computation scheme, it is very simple to deduce the data structure accesses. This way, as shown in Fig. 6, we can know the exact amount of extra-memory used when a new allocation of an AUB is needed. Fig. 6 shows how the memory allocations and disallocations can be predicted from the order in which the local `BMOD` tasks are computed. Indeed from this order, we deduce the order in which the AUB are updated by the `BMOD` tasks. When an AUB is updated by a `BMOD` task for the first time it means that it has to be allocated and when it is updated for the last time (and sended) it means that it has to be disallocated. For example, in Fig. 6, we can see that the peak of extra-memory is reached when task 4 needs to allocate AUB 3. As a consequence, *given a memory constraint*, we can know exactly which allocation of a new AUB will violate this constraint. If we denote by AUB_r such a new AUB and by AUB_s a previously allocated AUB that has been partially updated, then an AUB_s should have been sent before the allocation of the new AUB_r to enforce the memory constraint.

A good choice of the partially updated AUB_s to be sent must satisfy two criteria:

- the memory size of AUB_s has to be greater than or equal to the memory size of AUB_r;
- the next contribution to AUB_s must be computed as late as possible after the allocation of AUB_r.

It is clear that the best AUB_s corresponding to these criteria is easy to find thanks to the data allocation and disallocation scheme deduced from the task computation and communication schemes. Thus, the method consists of finding the best AUB_s according to these criteria whenever the allocation of an AUB violates the memory

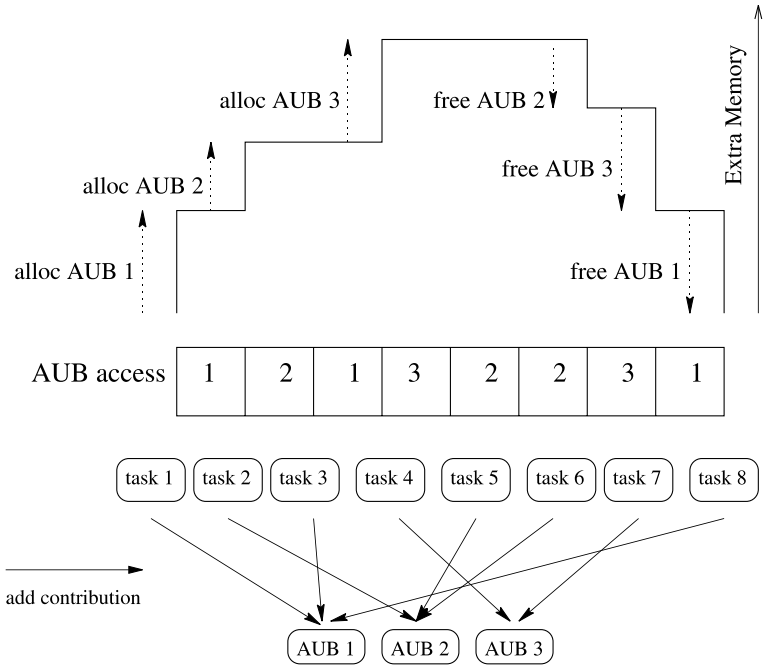

Fig. 6. Extra-memory evolution during task computings.

constraint. For example, in Fig. 6, if AUB_r is AUB 3 then the best choice for AUB_s is AUB 1.

In addition to the reduction of extra-memory, we are currently working on a more general version of our strategy that is suitable for heterogeneous architectures and more particularly those based on SMP nodes like the IBM SP3. As described at the end of the Section 2.2, our previous model for communication is extended to take into account the (less costly) data exchanges by shared memory, and the (more costly) data exchanges performed by the network. Numerical experiments are in progress to validate this extension. More generally, our strategy is suitable for heterogeneous architectures with predictable performances.

3. Numerical experiments

In Section 3.1, we first introduce our experimental environment. Then we present the time performances for the pre-processing steps in Section 3.2. Before the analysis of the parallel factorization times in Section 3.4, we describe some experiments for finding the threshold for mixed 1D/2D distributions (see Section 3.3). We next compare the softwares PaStiX and PSPASES in Section 3.5. Finally, Section 3.6 gives some results for the partial aggregation technique.

3.1. Experimental environment and test cases

All of the algorithms described in this paper have been integrated in the PaStiX software [20,21], based on libraries that make use of version 3.4 of the Scotch static mapping and sparse matrix ordering software package [27], both developed at LaBRI.

The parallel experiments were run on an 192 node IBM SP2 at CINES (Montpellier, France); the processors are 120 MHz Power2SC thin nodes (480 MFlops peak performance) having 256 MB of physical memory each. Switch interrupts are enabled with default delay to perform nonblocking communication efficiently. All computations are performed in double precision and all time results are given in seconds. The sequential experiments for the pre-processing steps (see Section 3.2) were run on one node of this IBM SP2. In all the following tables, the symbol "–" is used when the time measurements are not significant due to memory swapping.

Our experiments are performed on a collection of sparse matrices from the Rutherford–Boeing Collection, from the PARASOL ESPRIT Project and from CEA/CESTA (3D Cologne, Coupole). The almost part of these matrices are structural mechanics and CFD matrices. The values of the associated measurements in Table 1 come from scalar column symbolic factorization.

3.2. Partitioning and mapping results

Table 2 presents the sequential times to compute the three pre-processing steps and more particularly the times spent in the partitioning and mapping step for a number of processors varying from 2 to 64.

We notice that our partitioning and mapping algorithm is inexpensive in time; its complexity grows as $\Theta(M\log(P))$, where M is the number of blocks and P the number of processors.

3.3. Experimental threshold for mixed 1D/2D distribution

Table 3 presents some experiments on the threshold to shift between a 2D distribution and a 1D distribution during the first step of the partitioning and mapping algorithm (cf. Section 2.2). At the moment, our threshold criterion only takes into account the number of candidate processors for a node. We plan to improve this threshold criterion by taking into account the computation cost of the subtree rooted at the node in addition to this number of candidate processors. As shown in Table 3, a good experimental approximation gives a threshold around 12 processors on the IBM SP2 architecture. We will use this criterion to perform all the tests in Sections 3.4–3.6.

3.4. Parallel factorization results

All floating-point arithmetic is performed with BLAS routines as said previously. Profiled executions confirm that BLAS level 3 block computations (`GEMM` and `TRSM`)

Table 1
Description of our test problems

Name	Columns	NNZ_A	NNZ_L	OPC	NNZ_L/OPC	Description
GRID1023	1 046 529	4 179 980	5.615708e + 07	2.083481e + 10	2.69e − 3	Regular 2D mesh
CUBE39	59 319	730 778	2.210534e + 07	2.240674e + 10	0.99e − 3	Regular 3D mesh
CUBE47	103 823	1 290 898	4.828456e + 07	6.963850e + 10	0.69e − 3	Regular 3D mesh
BCSSTK32	44 609	985 046	5.239146e + 06	1.162900e + 09	4.50e − 3	Rutherford–Boeing
BBMAT	38 744	1 274 141	1.716094e + 07	1.250040e + 10	1.37e − 3	Rutherford–Boeing
TOOTH	78 136	452 591	1.031143e + 07	6.267094e + 09	1.64e − 3	3D element mesh
OCEAN	143 437	409 593	2.029997e + 07	1.301477e + 10	1.56e − 3	3D element mesh
M14B	214 765	1 679 018	6.236747e + 07	6.112540e + 10	1.02e − 3	3D element mesh
OILPAN	73 752	1 761 718	8.912337e + 06	2.984944e + 09	2.98e − 3	PARASOL
QUER	59 122	1 403 689	9.118592e + 06	3.280680e + 09	2.78e − 3	PARASOL
INVEXTR1	30 412	906 915	7.256566e + 06	3.766788e + 09	1.93e − 3	PARASOL
SMDOOR	162 610	3 873 534	2.541937e + 07	1.530774e + 10	1.66e − 3	PARASOL
SHIP001	34 920	2 304 655	1.427916e + 07	9.033767e + 09	1.58e − 3	PARASOL
X104	108 384	5 029 620	2.634047e + 07	1.712902e + 10	1.54e − 3	PARASOL
MT1	97 578	4 827 996	3.114873e + 07	2.109265e + 10	1.48e − 3	PARASOL
BMW3_2	227 362	5 530 634	4.420244e + 07	3.007981e + 10	1.47e − 3	PARASOL
MIXTANK	29 957	982 542	9.280247e + 06	7.316933e + 09	1.26e − 3	PARASOL
BMWCRA_1	148 770	5 247 616	6.597301e + 07	5.701988e + 10	1.16e − 3	PARASOL
CRANKSG1	52 804	5 280 703	3.142730e + 07	3.007141e + 10	1.05e − 3	PARASOL
SHIPSEC8	114 919	3 269 240	3.572761e + 07	3.684269e + 10	0.97e − 3	PARASOL
CRANKSG2	63 838	7 042 510	4.190437e + 07	4.602878e + 10	0.91e − 3	PARASOL
SHIPSEC5	179 860	4 966 618	5.649801e + 07	6.952086e + 10	0.81e − 3	PARASOL
SHIP003	121 728	3 982 153	5.872912e + 07	8.008089e + 10	0.73e − 3	PARASOL
THREAD	29 736	2 220 156	2.404333e + 07	3.884020e + 10	0.62e − 3	PARASOL
COLOGB30	20 373	1 394 292	7.849464e + 06	4.359803e + 09	1.80e − 3	3D cologne
COLOGB75	50 208	3 473 292	2.248613e + 07	1.532035e + 10	1.47e − 3	3D cologne
COUP1500T	994 983	70 303 275	5.219997e + 08	3.79847e + 11	1.38e − 3	3D coupole
COUP2000T	1 326 483	93 734 775	6.888813e + 08	5.00924e + 11	1.37e − 3	3D coupole

NNZ_A is the number of off-diagonal terms in the triangular part of matrix A, NNZ_L is the number of off-diagonal terms in the factorized matrix L and OPC is the number of operations required for the factorization. Matrices are sorted in decreasing order of NNZ_L/OPC which is a measure of the potential data reuse [24].

represent more than 75% of total BLAS computations. A multivariable polynomial regression has been used to build an analytical model of these routines. This model and the experimental values obtained for communication startup and bandwidth are used by the partitioning and scheduling algorithm. It is important to note that the

Table 2
Sequential times in seconds for the scheduling algorithm, for the Symbolic Factorization step (SF_Time) and for the Ordering step (O_Time)

Name	Number of processors						SF_Time	O_Time
	2	4	8	16	32	64		
GRID1023	1.15	1.12	1.77	2.20	3.27	10.27	3.31	
CUBE39	0.28	0.38	0.47	0.80	1.63	3.11	0.58	
CUBE47	0.94	1.17	1.39	2.70	4.72	8.92	1.35	
BCSSTK32	0.13	0.14	0.16	0.19	0.28	0.52	0.55	6.83
BBMAT	0.49	0.58	0.69	0.87	1.06	1.69	0.69	65.75
TOOTH	1.82	1.94	2.16	2.50	3.41	7.22	0.96	38.10
OCEAN	3.89	4.11	4.24	4.86	6.43	12.55	1.29	63.76
M14B	5.89	6.73	7.52	8.42	10.06	16.32	2.43	150.81
OILPAN	0.14	0.14	0.14	0.19	0.31	0.92	0.88	3.77
QUER	0.10	0.10	0.10	0.14	0.29	0.62	0.69	2.98
INVEXTR1	0.87	0.96	1.17	1.39	1.69	2.40	0.62	47.27
SMDOOR	0.31	0.34	0.39	0.51	0.98	2.90	1.93	9.27
SHIP001	0.05	0.05	0.08	0.11	0.19	0.49	0.88	5.51
X104	0.13	0.19	0.22	0.31	0.78	2.15	1.95	10.86
MT1	0.08	0.12	0.18	0.26	0.84	1.46	1.80	12.26
BMW3_2	0.57	0.73	0.83	1.00	1.80	5.13	3.12	27.67
MIXTANK	0.57	0.85	0.97	1.16	1.70	3.72	0.61	46.61
BMWCRA_1	0.33	0.39	0.58	0.87	1.08	1.56	2.59	45.37
CRANKSG1	0.09	0.13	0.26	0.41	0.89	2.02	1.99	17.08
SHIPSEC8	0.51	0.75	1.02	1.43	2.85	7.88	1.62	8.52
CRANKSG2	0.11	0.17	0.31	0.52	0.71	1.91	2.55	20.84
SHIPSEC5	0.88	0.94	1.38	1.76	3.28	8.00	2.43	12.90
SHIP003	0.70	0.95	1.32	1.85	3.36	8.53	2.05	14.25
THREAD	0.11	0.51	0.56	0.71	2.35	6.20	0.87	11.07
COLOGB30	0.03	0.05	0.06	0.80	0.17	0.33	0.10	6.90
COLOGB75	0.08	0.11	0.14	0.20	0.29	0.88	0.24	17.57
COUP1500T	1.46	1.55	1.69	1.99	2.65	5.19	1.17	–
COUP2000T	2.10	2.13	2.27	2.70	3.57	6.77	2.05	–

For 2D and 3D grids we do not use the SCOTCH software package but the optimal nested dissection algorithm of George and Liu [16].

number of operations actually performed during factorization is greater than the OPC value because of amalgamation performed by the ordering and of block computations.

Table 4 reports the performances of our parallel block factorization for our distribution strategy and shows that a good scalability is achieved for all the test problems. On both moderate and large size grid and irregular problems, the measured performances vary between 1.08 and 3.24 Gigaflops on 16 nodes and between 2.36 and 11.62 on 64 nodes, the better scalability being achieved on the largest tests. For large enough problems, Gigaflop rate reaches half max `GEMM` rate which is rather good for a sparse factorization.

Table 3
Influence of the threshold between 2D and 1D distributions on factorization time

Name	Threshold	Number of processors			
		8	16	32	64
CUBE47	6	39.59	27.17	17.82	14.77
	8	**38.61**	25.08	**17.05**	13.53
	12		**24.08**	17.15	**12.59**
	16		25.14	19.54	15.21
BMWCRA_1	6	**31.03**	18.91	9.59	7.01
	8	31.12	19.06	9.62	6.56
	12		**18.75**	**9.55**	**5.94**
	16		19.11	9.55	6.11
COLOGB75	6	10.18	5.75	4.16	3.25
	8	**8.99**	5.84	4.09	3.34
	12		**5.69**	**3.89**	3.14
	16		5.85	4.02	**3.13**

3.5. Comparisons with PSPASES

We compare factorization times performed in double precision real by our PaStiX software and by PSPASES version 1.0.3 based on a multifrontal approach [22]. PaStiX makes use of version 3.4 of the Scotch static mapping and sparse ordering software package developed at LaBRI. PSPASES uses MeTiS version 4.0 as its default ordering library. In both cases, blocking size is set to 64 and the IBM ESSL library is used. We describe experiments performed on a collection of sparse matrices in the RSA format; the values of the metrics in Table 5 come from scalar column symbolic factorization. Table 6 reports the performances of our parallel block factorization for our distribution strategy and the ones of the PSPASES software.

An important remark to facilitate comparisons is that PSPASES uses a Cholesky (LL^T) factorization, intrinsically more BLAS efficient and cache-friendly than the LDL^T one used by PaStiX (the `SYRK` routine cannot be used in LDL^T factorization, and moreover we have to compute diagonal matrix D with BLAS 1 routines). For instance, for a dense 1024×1024 matrix on one Power2SC node, the ESSL LL^T factorization time is 1.07 s whereas the ESSL LDL^T factorization time is 1.27 s.

Moreover, we can notice that PSPASES is much more memory consuming than PaStiX. This is probably inherent to the multifrontal approach used by PSPASES with regards to the supernodal approach used by PaStiX.

Results show that PaStiX compares favorably to PSPASES and achieves better solving times in almost all cases up to 32 processors. Performances are quite equivalent on 64 processors when scalability limit is reached. Globally, these experimental results show that the two softwares have comparable time performances.

Table 4
Factorization performance results (**time** in seconds and Gigaflops) on the IBM SP2

Name	Number of processors					
	2	4	8	16	32	64
GRID1023	–	–	**14.57** (1.43)	**8.61** (2.42)	**5.56** (3.74)	**3.83** (5.44)
CUBE39	**43.58** (0.51)	**24.20** (0.93)	**13.20** (1.70)	**8.88** (2.52)	**6.32** (3.55)	**5.03** (4.46)
CUBE47	–	**73.31** (0.95)	**38.23** (1.82)	**24.08** (2.89)	**17.15** (4.06)	**12.59** (5.53)
BCSSTK32	**4.03** (0.29)	**2.46** (0.47)	**1.41** (0.82)	**1.08** (1.08)	**0.78** (1.49)	**0.71** (1.63)
BBMAT	**28.17** (0.44)	**14.73** (0.85)	**8.55** (1.46)	**5.24** (2.39)	**3.39** (3.69)	**2.72** (4.60)
TOOTH	**20.01** (0.31)	**10.70** (0.59)	**6.60** (0.95)	**4.01** (1.56)	**2.77** (2.26)	**2.49** (2.52)
OCEAN	**48.37** (0.27)	**18.39** (0.71)	**9.78** (1.33)	**6.26** (2.08)	**4.12** (3.16)	**4.17** (3.12)
M14B	–	–	**46.95** (1.30)	**25.84** (2.37)	**14.60** (4.19)	**9.21** (6.64)
OILPAN	**7.28** (0.41)	**3.81** (0.78)	**2.23** (1.34)	**1.51** (1.98)	**1.03** (2.90)	**0.92** (3.24)
QUER	–	**4.46** (0.74)	**2.77** (1.18)	**1.90** (1.73)	**1.35** (2.43)	**1.04** (3.16)
INVEXTR1	**11.59** (0.32)	**6.31** (0.60)	**4.03** (0.93)	**2.52** (1.49)	**1.76** (2.14)	**1.57** (2.40)
SMDOOR	**29.54** (0.52)	**15.52** (0.99)	**9.10** (1.68)	**5.25** (2.91)	**4.07** (3.76)	**2.91** (5.25)
SHIP001	**20.98** (0.43)	**10.91** (0.83)	**6.13** (1.47)	**3.78** (2.39)	**2.34** (3.86)	**1.96** (4.60)
X104	**31.53** (0.54)	**19.69** (0.87)	**11.02** (1.55)	**6.32** (2.71)	**5.30** (3.23)	**3.85** (4.44)
MT1	**37.92** (0.56)	**20.35** (1.04)	**12.91** (1.63)	**6.96** (3.03)	**4.33** (4.87)	**3.51** (6.01)
BMW3_2	–	**34.02** (0.88)	**20.78** (1.45)	**11.52** (2.61)	**7.90** (3.81)	**6.67** (4.51)
MIXTANK	**17.32** (0.42)	**10.02** (0.73)	**6.33** (1.16)	**3.80** (1.93)	**3.32** (2.20)	**3.10** (2.36)
BMWCRA_1	–	–	**30.97** (1.84)	**18.75** (3.04)	**9.55** (5.97)	**5.94** (9.60)
CRANKSG1	–	**33.54** (0.90)	**19.20** (1.57)	**10.54** (2.85)	**6.82** (4.41)	**5.34** (5.63)
SHIPSEC8	–	**43.40** (0.85)	**27.71** (1.33)	**15.40** (2.39)	**11.93** (3.09)	**9.23** (3.99)
CRANKSG2	–	**47.99** (0.96)	**25.33** (1.82)	**14.39** (3.20)	**8.45** (5.45)	**6.04** (7.62)
SHIPSEC5	–	**79.12** (0.88)	**40.36** (1.72)	**21.46** (3.24)	**15.36** (4.53)	**11.76** (5.91)
SHIP003	–	–	**45.83** (1.75)	**26.86** (2.98)	**16.69** (4.80)	**12.03** (6.66)
THREAD	**78.04** (0.50)	**41.14** (0.94)	**22.85** (1.70)	**13.45** (2.89)	**11.79** (3.29)	**6.70** (5.80)
COLOGB30	**9.88** (0.44)	**5.25** (0.83)	**3.23** (1.35)	**2.06** (2.12)	**1.69** (2.58)	**1.34** (3.25)
COLOGB75	–	**18.72** (0.82)	**8.99** (1.70)	**5.69** (2.69)	**3.89** (3.94)	**3.14** (4.88)
COUP1500T	–	–	–	–	–	**33.01** (11.51)
COUP2000T	–	–	–	–	–	**43.12** (11.62)

3.6. Influence of the partial aggregation strategy

Fig. 7 presents, for one test, the time penalty on the parallel factorization induced by the partial aggregation method. The results are performed on different numbers of processors and different percentages of memory reduction.

Firstly, it is important to remark that the total extra-memory due to the AUBs dramatically increases when the number of processors grows. In this case, the ratio in percentage, between the memory needed to store the AUBs and the memory needed to store the distributed matrix coefficients, represents 11% on eight processors, 22% on 16 processors, 49% on 32 processors and 88% on 64 processors.

In Fig. 7, the x-axis is the percentage of memory reduction with respect to the total extra-memory due to the AUBs on a processor. The y-axis is the percentage of

Table 5
Description of our test problems

Name	Columns	NNZ_A	$NNZ_{L(SCOTCH)}$	$OPC_{L(SCOTCH)}$	$NNZ_{L(METIS)}$	$OPC_{(METIS)}$
OILPAN	73 752	1 761 718	8.912e + 06	2.985e + 09	9.065e + 06	2.751e + 09
QUER	59 122	1 403 689	9.119e + 06	3.281e + 09	9.586e + 06	3.448e + 09
SMDOOR	162 610	3 873 534	2.542e + 07	1.531e + 10	2.404e + 07	1.237e + 10
SHIP001	34 920	2 304 655	1.428e + 07	9.034e + 09	1.481e + 07	9.462e + 09
X104	108 384	5 029 620	2.634e + 07	1.713e + 10	2.728e + 07	1.412e + 10
MT1	97 578	4 827 996	3.115e + 07	2.109e + 10	3.455e + 07	2.269e + 10
SHIPSEC5	179 860	4 966 618	5.650e + 07	6.952e + 10	5.256e + 07	5.509e + 10
SHIP003	121 728	3 982 153	5.873e + 07	8.008e + 10	5.910e + 07	7.587e + 10
THREAD	29 736	2 220 156	2.404e + 07	3.884e + 10	2.430e + 07	3.583e + 10

NNZ_A is the number of off-diagonal terms in the triangular part of matrix A, NNZ_L is the number of off-diagonal terms in the factorized matrix L and OPC is the number of operations required for the factorization.

Table 6
Factorization performance results (**time** in seconds and Gigaflops) on the IBM SP2

Name	Number of processors					
	2	4	8	16	32	64
OILPAN	**7.28** (0.41)	**3.81** (0.78)	**2.23** (1.34)	**1.51** (1.98)	**1.03** (2.90)	**0.92** (3.24)
	–	–	**2.79** (0.99)	**1.73** (1.59)	**1.25** (2.20)	**0.93** (2.96)
QUER	–	**4.46** (0.74)	**2.77** (1.18)	**1.90** (1.73)	**1.35** (2.43)	**1.04** (3.16)
	–	–	–	**2.01** (1.72)	**1.30** (2.65)	**0.96** (3.59)
SMDOOR	**29.54** (0.52)	**15.52** (0.99)	**9.10** (1.68)	**5.25** (2.91)	**4.07** (3.76)	**2.91** (5.25)
	–	–	–	**7.32** (1.69)	**3.91** (3.16)	**2.58** (4.79)
SHIP001	**20.98** (0.43)	**10.91** (0.83)	**6.13** (1.47)	**3.78** (2.39)	**2.34** (3.86)	**1.96** (4.60)
	–	–	**8.11** (1.17)	**4.48** (2.11)	**2.98** (3.17)	**2.14** (4.42)
X104	**31.53** (0.54)	**19.69** (0.87)	**11.02** (1.55)	**6.32** (2.71)	**5.30** (3.23)	**3.85** (4.44)
	–	–	–	–	**4.92** (2.87)	**3.18** (4.44)
MT1	**37.92** (0.56)	**20.35** (1.04)	**12.91** (1.63)	**6.96** (3.03)	**4.33** (4.87)	**3.51** (6.01)
	–	–	–	–	**5.70** (3.98)	**3.59** (6.32)
SHIPSEC5	–	**79.12** (0.88)	**40.36** (1.72)	**21.46** (3.24)	**15.36** (4.53)	**11.76** (5.91)
	–	–	–	–	–	**11.81** (4.66)
SHIP003	–	–	**45.83** (1.75)	**26.86** (2.98)	**16.69** (4.80)	**12.03** (6.66)
	–	–	–	–	–	**14.08** (5.39)
THREAD	**78.04** (0.50)	**41.14** (0.94)	**22.85** (1.70)	**13.45** (2.89)	**11.79** (3.29)	**6.70** (5.80)
	–	–	–	–	**11.24** (3.19)	**6.41** (5.59)

For each matrix, the first line gives the PASTIX results and the second line the PSPASES ones.

time penalty with respect to the factorization time given in Table 4. These results show that the memory reduction is acceptable, in terms of time penalty, up to

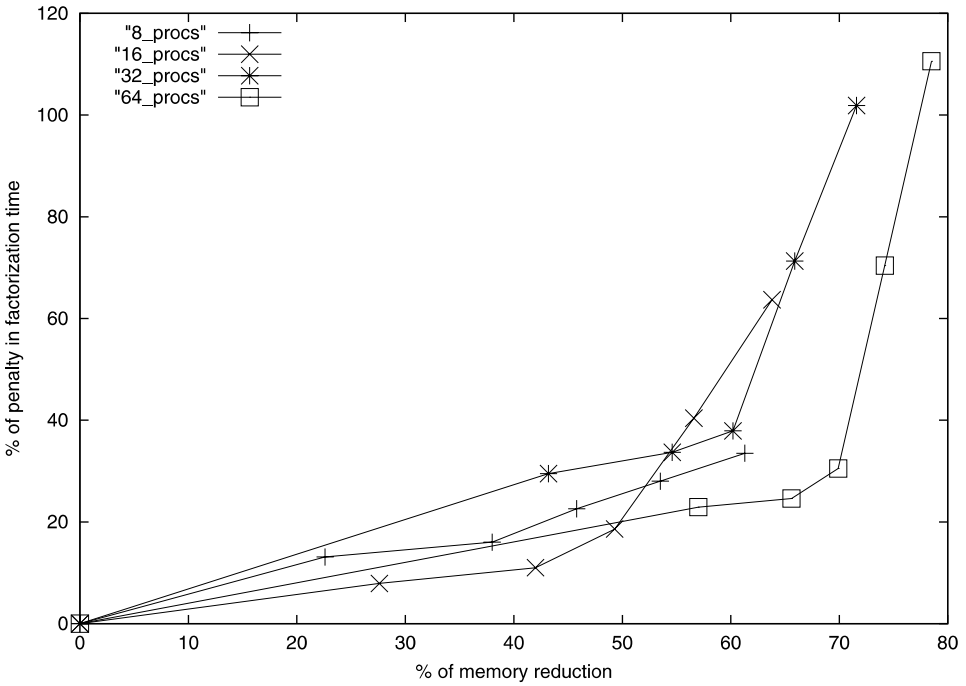

Fig. 7. Influence of memory reduction (with a partial aggregation strategy) on the factorization time for BMWCRA_1 matrix.

50% of extra-memory reduction. In that context, our extra-memory management leads to a good memory scalability.

4. Conclusion and perspectives

An efficient mixed 1D/2D distribution scheme and the induced static scheduling of the block computations for a parallel sparse direct solver has been presented. This work is still in progress. Currently work on a finer criterion to switch between the 1D and 2D distributions to still improve scalability is pursued.

A modified version of our strategy to take into account architectures based on SMP nodes is also under development. Some preliminary experimental studies on IBM SP3 are promising. For instance, on 16 WinterHawk + nodes (64 processors 375 MHz Power3), the factorization of the COUP1500T matrix is performed in 11.8 s (32.2 gigaflops), and the COUP2000T matrix in 15.1 s (33.3 gigaflops). The achieved performances are rather good and confirm that the proposed scheduling strategy is suitable for heterogeneous SMP architectures.

Finally, comparisons between PaStiX and MUMPS softwares for parallel time performances and for memory aspects will be reported in a forthcoming paper.

Acknowledgements

This work is supported by the French Commissariat à l'Énergie Atomique CEA/CESTA under contract No. 7V1555AC, and by the GDR ARP (iHPerf group) of the CNRS.

References

[1] P.R. Amestoy, T. Davis, I.S. Duff, An approximate minimum degree ordering algorithm, SIAM J. Matrix Anal. Appl. 17 (1996) 886–905.
[2] P.R. Amestoy, I.S. Duff, Memory management issues in sparse multifrontal methods on multiprocessors, Int. J. Supercomput. Appl. 7 (1993) 64–82.
[3] P.R. Amestoy, I.S. Duff, J.Y. L'Excellent, Multifrontal parallel distributed symmetric and unsymmetric solvers, Comput. Meth. Appl. Mech. Eng. 184 (2000) 501–520.
[4] P.R. Amestoy, I.S. Duff, J.-Y. L'Excellent, X.S. Li, Analysis, tuning and comparison of two general sparse solvers for distributed memory computers, Technical report RT/APO/00/2, ENSEEIHT-IRIT, June 2000. France–Berkeley project report, also Lawrence Berkeley National Laboratory report LBNL-45992.
[5] C. Ashcraft, The fan-both family of column-based distributed Cholesky factorization algorithms, in: Graph Theory and Sparse Matrix Computation, IMA, vol. 56, Springer, Berlin, 1993, pp. 159–190.
[6] C. Ashcraft, S.C. Eisenstat, J.W.-H. Liu, A fan-in algorithm for distributed sparse numerical factorization, SIAM J. Sci. Stat. Comput. 11 (3) (1990) 593–599.
[7] C. Ashcraft, S.C. Eisenstat, J.W.-H. Liu, B.W. Peyton, A. Sherman, A compute-ahead fan-in scheme for parallel sparse matrix factorization, in: D. Pelletier (Ed.), Fourth Canadian Supercomputing Symposium, 1990, pp. 351–361.
[8] C. Ashcraft, S.C. Eisenstat, J.W.-H. Liu, A. Sherman, A comparison of three column based distributed sparse factorization schemes, in: Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing, 1991.
[9] J. Bunch, L. Kaufmann, Some stable methods for calculating inertia and solving symmetric linear systems, Math. Comput. 31 (1977) 162–179.
[10] P. Charrier, J. Roman, Algorithmique et calculs de complexité pour un solveur de type dissections emboîtées, Numer. Math. 55 (1989) 463–476.
[11] J.M. Conroy, S.G. Kratzer, R.F. Lucas, Multifrontal sparse solvers in message passing and data parallel environments – a comparitive study, in: Proceedings of PARCO, 1993.
[12] I.S. Duff, Sparse numerical linear algebra: direct methods and preconditioning, Technical report TR/PA/96/22, CERFACS, 1996.
[13] I.S. Duff, J.K. Reid, The multifrontal solution of indefinite sparse symmetric linear equations, ACM Trans. Math. Software 9 (1983) 302–325.
[14] I.S. Duff, J.K. Reid, The multifrontal solution of unsymmetric sets of linear equations, SIAM J. Sci. Stat. Comput. 5 (3) (1984) 633–641.
[15] G.A. Geist, E. Ng, Task scheduling for parallel sparse Cholesky factorization, Int. J. Parallel Program. 18 (4) (1989) 291–314.
[16] A. George, J.W.-H. Liu, Computer Solution of Large Sparse Positive Definite Systems, Prentice Hall, Englewood Cliffs, NJ, 1981.
[17] D. Goudin, Mise en œuvre d'une Bibliothèque d'Outils pour la Résolution Parallèle Hautes Performances par Méthode Directe de Grands Systèmes Linéaires Creux et application à un Code de Mécanique des Structures, Ph.D. thesis, Université Bordeaux 1, France, 2000.
[18] A. Gupta, G. Karypis, V. Kumar, Highly scalable parallel algorithms for sparse matrix factorization, IEEE Trans. Parallel Distrib. Syst. 8 (5) (1997) 502–520.
[19] M.T. Heath, E.G.-Y. Ng, B.W. Peyton, Parallel algorithms for sparse linear systems, SIAM Rev. 33 (1991) 420–460.

[20] P. Hénon, P. Ramet, J. Roman, A mapping and scheduling algorithm for parallel sparse fan-in numerical factorization, in: Proceedings of EuroPAR'99, Lecture Notes in Computer Science, vol. 1685, Springer, Berlin, September 1999, pp. 1059–1067.
[21] P. Hénon, P. Ramet, J. Roman, PaStiX: A parallel sparse direct solver based on a static scheduling for mixed 1D/2D block distributions, in: Proceedings of Irregular'2000, Lecture Notes in Computer Science, vol. 1800, Springer, Berlin, May 2000, pp. 519–525.
[22] M. Joshi, G. Karypis, V. Kumar, A. Gupta, Gustavson F. PSPASES: scalable parallel direct solver library for sparse symmetric positive definite linear systems, Technical report, University of Minnesota and IBM Thomas J. Watson Research Center, May 1999.
[23] G. Karypis, 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.
[24] X.S. Li, Sparse gaussian elimination on high performance computers, Ph.D. thesis, University of California at Berkeley, 1996.
[25] X.S. Li, J.W. Demmel, A scalable sparse direct solver using static pivoting, in: Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing, San Antonio, TX, March 22–24, 1999.
[26] J.W.-H. Liu, Modification of the minimum-degree algorithm by multiple elimination, ACM Trans. Math. Software 11 (2) (1985) 141–153.
[27] F. Pellegrini, J. Roman, P. Amestoy, Sparse matrix ordering with scotch, in: Proceedings of HPCN'97, Vienna, Lecture Notes in Computer Science, vol. 1225, April 1997, pp. 370–378.
[28] F. Pellegrini, J. Roman, P. Amestoy, Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering, Concurrency: Practice Experience 12 (2000) 69–84, Preliminary version published in Proceedings of Irregular'99, Lecture Notes in Computer Science, vol. 1586, pp. 986–995.
[29] A. Pothen, C. Sun, A mapping algorithm for parallel sparse Cholesky factorization, SIAM J. Sci. Comput. 14 (5) (1993) 1253–1257.
[30] E. Rothberg, Performance of panel and block approaches to sparse Cholesky factorization on the iPSC/860 and Paragon multicomputers, SIAM J. Sci. Comput. 17 (3) (1996) 699–713.
[31] E. Rothberg, A. Gupta, An evaluation of left-looking, right-looking, and multifrontal approaches to sparse cholesky factorization on hierarchical-memory machines, Int. J. High Speed Comput. 5 (1993) 537–593.
[32] E. Rothberg, A. Gupta, An efficient block-oriented approach to parallel sparse Cholesky factorization, SIAM J. Sci. Comput. 15 (6) (1994) 1413–1439.
[33] E. Rothberg, R. Schreiber, Improved load distribution in parallel sparse Cholesky factorization, in: Proceedings of Supercomputing'94, IEEE, New York, 1994, pp. 783–792.
[34] O. Schenk, K. Gärtner, W. Fichtner, Efficient sparse LU factorization with left–right looking strategy on shared memory multiprocessors, BIT 40 (1) (2000) 158–176.