Baltzer Journals

Two-dimensional Block Partitionings for the Parallel Sparse Cholesky Factorization

B. Dumitrescu[1,*], M. Doreille[2†], J.-L. Roch[2†] and D. Trystram[2†]

[1] *Politehnica University of Bucharest, Department of Automatic Control and Computers, 313, Splaiul Independentei, 77206 Bucharest, Romania.* E-mail: bogdan@indinf.pub.ro

[2] *IMAG-LMC, 100 rue des Mathématiques, 38041 Grenoble cedex France.* E-mail: [*Mathias.Doreille, Jean-Louis.Roch, Denis.Trystram*]*@imag.fr*

This paper presents a discussion on 2D block mappings for the sparse Cholesky factorization on parallel MIMD architectures with distributed memory. It introduces the fan-in algorithm in a general manner and proposes several mapping strategies. The grid mapping with row balancing, inspired from Rothberg's work [20, 21], is proved to be more robust than the original fan-out algorithm. Even more efficient is the proportional mapping, as show the experiments on a 32 processors IBM SP1 and on a Cray T3D. Subforest-to-subcube mappings are also considered and give good results on the T3D.

Subject classification: AMS(MOS) 65F50, 65Y05

Keywords: sparse Cholesky factorization, parallel algorithms, fan-in communication, 2D block partitioning, proportional mapping.

Many problems in scientific and engineering computation request to solve a linear system $Ax = b$, where A is a sparse symmetric positive definite matrix. To solve the system, the Cholesky factorization $A = LL^T$ is the most time consuming step.

Although a classic problem, the factorization continues to request interest due to the effort to find algorithms well adapted to actual computer architectures. The class of parallel algorithms is especially targeted, since many approaches are possible and promising.

We will present a brief state of the art in the first section of this paper, together with a presentation of the basic tools needed for an efficient parallel sparse Cholesky factorization and the framework of our contributions: the class of two-dimensional block algorithms, using fan-in communication, on MIMD architectures with distributed memory. In section 2, the fan-in algorithm for 2D mapping is introduced in full detail, in a general manner. Section 3 contains the key to efficiency: specific mapping strategies, which combine known heuristics in a new way; we add to (block) column mapping algorithms – like proportional [17] and subforest-to-subcube [11] – efficient ideas of mapping blocks inside a column. Section 4 is devoted to experiments, which show the good behavior of the new fan-in algorithms. Finally, in section 5, we discuss some promising perspective issues.

[*] This work was performed while the author was visiting the Laboratoire de Modélisation et Calcul de Grenoble, supported by a TEMPRA grant.

[†] Research supported by the (CNRS – INRIA – INPG – UJF) joint project *Apache.*

1 Introduction

As widely known, sparse Cholesky factorization is performed in three distinct stages. First, the matrix A is permuted such that fill-in is reduced; second, the symbolic factorization of A is computed, i.e. the structure of the Cholesky factor L; third, the numerical factorization is performed. This third stage is the most time consuming stage, and we will deal with it in the rest of the paper.

1.1 Two-dimensional block partitionings

When classifying parallel algorithms for numerical factorization, a main criterion is the way the matrix is mapped to processors. The first algorithms were column oriented, i.e. a column of L was mapped to a single processor; the survey [12] presents several such algorithms. With the advent of new processor architectures, favoring block operations, the use of BLAS 3 [6] routines became crucial. Thus, supernodes – groups of consecutive columns with the same row structure – were used instead of columns. The columns of a supernode can be factored together as for a dense matrix, allowing block operations. This approach, named block column (or 1D), was used, to give only few examples, in [16] or [8], but goes back to [4] and even earlier.

Since the supernode structure is specific to each matrix, there are two ways to adapt supernode sizes. *Amalgamation* – as proposed by Ashcraft and Grimes [3] – allows the grouping of several supernodes into a greater supernode, with the sacrifice of treating some zeros as nonzeros, such that all columns of the new supernode have the same row structure. On the other hand, large supernodes can be split into *panels*, i.e. groups of consecutive columns which obviously have the same properties as supernodes; Rothberg's thesis [18] contains a description and the use of this technique.

Schreiber [22] was the first to explain that one-dimensional mappings have poor scalability, and thus two-dimensional (2D) mappings are required for parallel efficiency. At the present time, two classes of algorithms seem to take great advantage of the 2D mapping: the block fan-out algorithm of Rothberg and Gupta [20], based on a classic right-looking Gaussian elimination, and the multifrontal parallel algorithm proposed by Gupta, Karypis and Kumar [11]. The recent survey of Duff [7] offers some other bibliographical references.

Since our study is in the same framework, let us start describing the basic lines of Rothberg's approach. After splitting the matrix vertically into N block columns (panels), as described above, the same split is applied by rows and thus a 2D partitioning is obtained. A block sparse matrix results, i.e. sparsity is now thought in terms of blocks L_{IJ} (we will use capital letters for block indices). However, nonzero blocks are not necessarily full. All we can say is that if a row of a block is nonzero, then all its elements are nonzero. Only diagonal blocks are surely full lower triangular. We present in figure 1 the 2D block structure of a sparse matrix after amalgamation. Filled circles are genuine nonzeros, while the other two circles are zeros considered as nonzeros, in order to increase supernodes size. There are 3 block columns (rows); block L_{21} is zero; blocks L_{31}, L_{32} have only one nonzero row.

The sequential Gaussian elimination algorithm at block level is similar to the element level one. We present in figure 2 the kij, or "right-looking" version. This algorithm should be viewed only as a general framework. We denote by $col(K)$ the set of row indices of subdiagonal blocks of column K, i.e. $col(K) = \{I \mid L_{IK} \neq 0,\ I > K\}$. The Cholesky

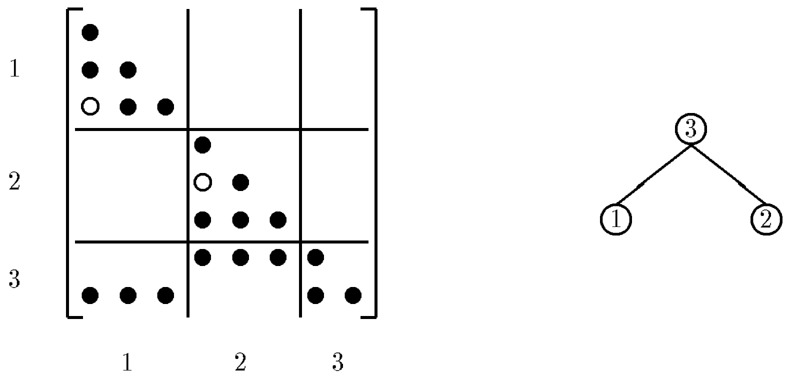

Figure 1: Block sparse Cholesky factor and its elimination tree.

```
    BLOCK GAUSSIAN ELIMINATION ($L$ is initialized with $A$)
1.  for $K = 1 : N$
2.      compute in place the Cholesky factor of $L_{KK}$
3.      for $I \in col(K)$
4.          $L_{IK} \leftarrow L_{IK} L_{KK}^{-T}$
5.      for $J \in col(K)$
6.          for $I \in col(K)$ and $I \geq J$
7.              $L_{IJ} \leftarrow L_{IJ} - L_{IK} L_{JK}^{T}$
```

Figure 2: 2D block-level Gaussian elimination.

factorization of a diagonal block may be performed with the DPOTRF routine from LAPACK. BLAS 3 routines may be used for the other operations; in statement 4, DTRSM is used directly for the computation of L_{IK}, since a block contains full rows; the matrix multiplication $L_{IK}L_{JK}^T$ is performed with DGEMM; the result of this multiplication usually has not the same structure as L_{IJ}, neither on rows or columns; an insert-add operation completes the update 7 of L_{IJ}. The techniques outlined here are detailed in [20]. Adapting the traditional notation, we will denote by $bdiv(K, I)$ and $bmod(K, I, J)$ the operations 4 and 7, respectively (standing for "block division" and "block modification").

An important tool for both sequential and parallel algorithms is the elimination tree associated with the Cholesky factor. In this tree, each node represents a column; the father of node j is node i, where l_{ij} is the first subdiagonal nonzero on column j (i.e. the smallest $i > j$ such that $l_{ij} \neq 0$). For many algorithms, including some of this paper, it is important that the elimination tree be numbered in postorder, i.e. a father be numbered immediately after its sons. For an extensive study of elimination trees, see Liu [15]. After a 2D block partitioning, we use a block column elimination tree, defined similarly to the column elimination tree, as seen in figure 1.

The elimination tree illustrates the intrinsic parallelism of the sparse Cholesky factor-

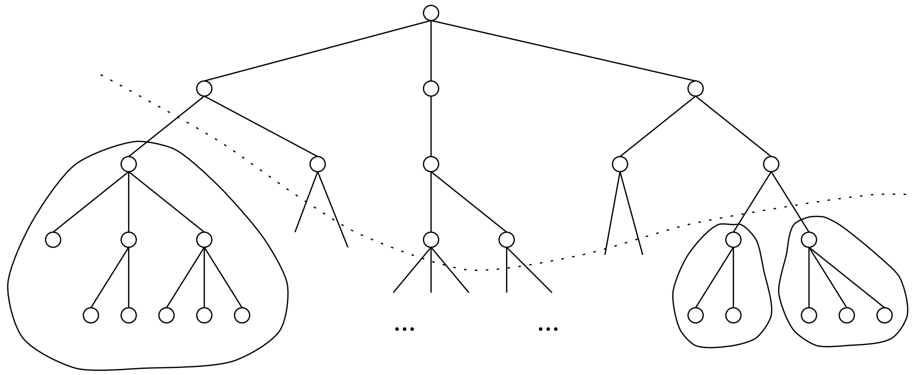

Figure 3: An elimination tree: local subtrees and distributed panels.

ization; the computation of a panel J of L may be done after all columns in the subtree rooted in J are factorized. Intuitively, an elimination tree with small height and large width is more appropriate to parallel computation than a high thin tree. The elimination tree offers a column view on parallelism, but we must stress that parallelism exists also for operations within a column.

1.2 Fan-out, fan-in algorithms

We describe now in general terms the possible organization of parallel algorithms based on the 2D block structure of a sparse matrix.

The 2D partitioning allows the mapping of a block column K to several processors. The set of processors sharing K will be called $group(K)$, or simply *group*, when the context is clear. We will denote by $owner(I, K)$ the processor holding the block L_{IK}. From now on we will use "panel" instead of "block column".

Although the 2D mapping was proved to be superior to 1D mapping, there is still interest in allocating all the blocks of some panels to the same processor. If $|group(K)| > 1$, communication is needed; for example, $owner(K, K)$ must send L_{KK} to all other processors in $group(K)$ in order that *bdiv* operations are possible. The benefit is greater for *bmod*. Let us assume that a whole subtree of the elimination tree is mapped to a processor, as in figure 3. Then, at least for panels K and J belonging to this subtree, where K is an descendant of J, the operation $bmod(K, I, J)$ is local for any I. The greater (higher) the local subtrees, the smaller the communication volume.

Certainly, there must be a tradeoff between the size of local subtrees and balancing processor load. Deferring the details to section 3, we assume now that some algorithm has been used and the elimination tree is separated into several local subtrees and panels that are distributed among processors. In figure 3, a dotted curve separates the local subtrees and the distributed panels. We note that several subtrees may be mapped to the same processor.

Let us further study how a parallel algorithm could be described, given the above distribution of blocks. Two strategies may be used, well known as fan-out and fan-in,

		Distributed step	
		Fan-out	Fan-in
Local	Fan-out	2D block mapping: costly Column mapping: [10]	
step	Fan-in	2D block mapping: [20] Column mapping: [13]	2D block mapping: this paper Column mapping: [2]

Figure 4: Communication strategies and possible combinations.

depending on the processor computing $bmod(K, I, J)$.

In the fan-out strategy, $bmod(K, I, J)$ is computed by $owner(I, J)$ ("at destination", as sometimes said). To this purpose, L_{IK} and L_{JK} must be sent to $owner(I, J)$.

In the fan-in strategy, $bmod(K, I, J)$ is computed either by $owner(I, K)$ or, "at source", by $owner(J, K)$. We assume that $owner(I, K)$ performs the required update, recalling that $I \geq J$. Obviously, the block L_{JK} must be sent to $owner(I, K)$. Each processor involved in the computation of L_{IJ} holds its own contribution in a local block; only $owner(I, J)$ initializes this block with A_{IJ}, while the other processors initialize with zero. When a processor has finished all updates of its contribution, it sends it to $owner(I, J)$ which adds it to L_{IJ}. In fact, this is actually an insert-add operation, since the contributions structure may differ of that of L_{IJ}.

We must distinguish now between the local computation step and the distributed one. Part of the local computation potentially affects panels that are distributed; to be more precise, let assume that K is a local panel and L_{JK}, L_{IK} are subdiagonal blocks; if panel J is also local (and necessarily mapped to the same processor), then $bmod(K, I, J)$ may be performed locally, as we already remarked; if panel J is distributed, then some blocks must be sent to $owner(I, J)$, depending on the strategy. No matter the strategy, the communication could be postponed until all local subtrees are computed. A 2D block parallel algorithm for the Cholesky factorization will thus have three steps :

1. Factorization of panels into local subtrees (only local computation).

2. Communication of blocks (or updates) computed at step 1 and affecting distributed panels.

3. Factorization of distributed panels (involving computation and communication).

It is not necessary that the same communication strategy (fan-out or fan-in) be used in the local and distributed steps. Figure 4 presents the four possible combinations. Three of them have "ancestors", i.e. column oriented algorithms; the fourth, which is fairly unnatural, was never tried; despite the age of about a decade of these algorithms, we never saw such a classification.

For 2D block mappings, the only existent algorithm belongs to Rothberg and Gupta [20] and is of "fan-in fan-out" type. The "fan-out fan-out" algorithm is presumably less efficient due to large communication costs, as for the column case; we implemented this algorithm and its efficiency is indeed lower; however, less communication is required than for a 1D based algorithm, since only few blocks of local panels are sent. The "fan-in fan-in" algorithm is much more appealing; we will present it in the next sections. For the sake of brevity, Rothberg's algorithm will be called "fan-out", while our algorithm "fan-in".

We present now an outline of the fan-out algorithm distributed step. The underlying architecture is supposed to be a $p_r \times p_c$ grid; a cyclic mapping is used, i.e. a block L_{IJ} is mapped to processor $(I \bmod p_r, J \bmod p_c)$. Thus, the communication is reduced: a computed block L_{IK} must be sent only to processors owning row I or column I of L, to participate if necessary to updates $bmod(K, I, J)$ or $bmod(K, J, I)$. The algorithm is data driven; after receiving a block, a processor looks at what updates this block can participate, perform these updates and, if it is the case, diagonal block factorizations and *bdiv* operations on local blocks; finally, it sends any newly factorized block to appropriate processors. A very clear description of the algorithm is presented in [20]. Since the cyclic mapping is rigid and may cause load imbalancing, Rothberg and Schreiber [21] proposed an improved mapping scheme; matrix block rows and columns are still mapped to processor grid rows and columns, but not cyclically; instead, a balancing of grid rows (columns) load is searched. (The problem reduces to bin packing.) For rows, the idea was indeed effective; for columns, it seems that usually the cyclic mapping is fairly satisfactory.

2 The fan-in algorithm: general presentation

We present in this section the details of the fan-in algorithm, excepting matrix blocks mapping, which is the key of efficiency, but is not affecting the validity of the algorithm; we use only the notion of group of processors owning blocks in a certain panel K ($group(K)$). Only the distributed step is detailed, since the others are straightforward. The main structure of the algorithm is presented in figure 5, and two important functions are listed in figures 6 and 7.

The general communication pattern was described in the previous section. We recall that once a block L_{IK} of the Cholesky factor is computed, it is broadcast by its owner to all processors in $group(K)$. By the other hand, each processor accumulates all its updates for non local blocks and sends them only once to the owner. Similarly to Rothberg's fan-out algorithm, the number of updates a block must suffer is computed initially; we denote it by $nmod(I, J)$; this computation may take place in the symbolic factorization stage. During the factorization, $nmod(I, J)$ is decremented at each update.

Let us explain the data structures appearing in the algorithm; $wait(K)$ is a list containing row indices of local blocks from panel K which need only the *bdiv* operation to be factorized (i.e. the availability of block L_{KK}); $ready(K)$ is a list of local or received blocks of panel K of the Cholesky factor (i.e. already computed); $diag(K)$ is a flag indicating if the block L_{KK} was factorized and is present in the local memory; $queue$ is a list of local blocks that can be factorized with local information; their factorization was postponed for a simpler organization of the algorithm. All these variables have an initial value set to zero or empty.

An important question is when a processor sends to $owner(I, J)$ its contribution to L_{IJ}, since this communication must be performed only after all possible local updates were aggregated. The solution we adopted is to count, initially, the number of updates a processor must perform on each block; the cost of this operation similar to a partial symbolic factorization, is negligible, moreover when block size is large; we use *nmod*(I, J) to keep the number of updates; note that *nmod* has different significations for local and non local blocks.

We will denote by *bmod_send*(K, I, J) (see figure 6) the operation *bmod*(K, I, J), followed, if L_{IJ} is not local, by a check of the number of updates and by a send, if all updates were performed; note that the number of updates must be transmitted to *owner*(I, J), in order to adjust *nmod*(I, J) in line 15 of the main algorithm; if L_{IJ} is local, and is completely updated, then it is added to the list *wait*(J) or to the *queue*.

Although the algorithm is detailed, there are several aspects to be made more clear. Statements 24 in the main routine, 8 and 13 in *last_block_op* are not quite correct; when scanning *ready*(K) for indices J, there are three situations when the current processor performs *bmod_send*(K, I, J): if L_{IK} and L_{JK} are both local; if L_{IK} is local, L_{JK} is a received block (in the sense that *owner*$(J, K) \neq$ me), and $I > J$; if L_{IK} is a received block, L_{JK} is local and $I < J$. In a real program, when calling *bmod_send*(K, I, J), second argument value must be greater than (or equal to) third argument value; our description doesn't actually respect this rule.

The condition in statement 8 of the main routine is implemented by initially counting the number of blocks of the Cholesky factor to be received by a processor and updating this number after each receive of such a block; another counter with the number of local blocks is decremented when a block is factorized.

Another small remark is that when *last_block_op* is called for a block L_{IK} extracted from the *queue*, this block is always factorized, i.e. one of the following conditions is true: $I = K$ or *diag*$(K) = 1$.

3 Mappings for the fan-in algorithm

We present now several methods to map the blocks of a matrix such that the fan-in algorithm is efficient. There are two objectives (recall figure 3): to find large enough local subtrees; to find a reasonable processors load imbalance, and to map the blocks of the distributed panels. These objectives may be attained by the same, or by distinct algorithms.

Such algorithms are based on costs associated with the computation performed on blocks, which can be computed with low overhead in the stage of symbolic factorization; the cost of a block operation may be modeled as the sum of the number of floating point operations and a constant representing other operations (function calls, block address calculation, etc.) [21]; the costs of block L_{KK} Cholesky factorization and of *bdiv*(I, K) are associated naturally with blocks L_{KK} and L_{IK}, respectively; the cost of *bmod*(K, I, J) is associated either with L_{IK} or L_{IJ}, for the fan-in or fan-out methods, respectively. The cost associated with a panel is the sum of its blocks costs; the cost associated with a subtree is the sum of its panels costs.

It is not very easy to evaluate a mapping; there are two criteria, usually in conflict: communication volume and load balance. For evaluation, the relative importance of these criteria depends on the properties of the parallel computer, especially on the ratio communication vs. computation speed. In this section we will thus present only qualitative

```
(me is the id number of the current processor)
// local step and involved communication
for $K = 1 : N$
    if $K$ is a local panel
        factorize all blocks in $col(K)$
        for all $I, J \in col(K)$ with $I \geq J$,  $bmod\_send(K, I, J)$
    else if me $\in group(K)$
        performed all possible computation on local data
        initialize lists $wait(K)$ and $ready(K)$
// distributed step
while there are blocks to be received and local blocks to compute
    while $queue$ is not empty
        take block $L_{IK}$ from $queue$
        $last\_block\_op(I, K)$
    if a block $L_{IJ}$ was received
        if $L_{IJ}$ is an update block
            insert-add it to local block $L_{IJ}$
            subtract from $nmod(I, J)$ the number of updates of
                the received block
            if $nmod(I, J) = 0$
                $last\_block\_op(I, J)$
        else ($L_{IJ}$ is a block of the Cholesky factor)
            if $I = J$
                $diag(J) \leftarrow 1$
                $bdiv(I', J)$ for all $I' \in wait(J)$
            else
                put $I$ in list $ready(J)$
                perform all my $bmod\_send(J, I, I')$, for $I' \in ready(J)$
```

Figure 5: The fan-in 2D block algorithm.

```
function $bmod\_send(K, I, J)$
    $bmod(K, I, J)$
    $nmod(I, J) \leftarrow nmod(I, J) - 1$
    if $nmod(I, J) = 0$
        if me $= owner(I, J)$
            if $I = J$ or $diag(J) = 1$
                put $L_{IJ}$ in the $queue$
            else put $I$ in list $wait(J)$
        else
            send $L_{IJ}$ to $owner(I, J)$
```

Figure 6: Function *bmod_send*.

```
function $last\_block\_op(I, K)$
    if $I = K$
        compute in place the Cholesky factor of $L_{KK}$
        $diag(K) \leftarrow 1$
        broadcast $L_{KK}$ to processors in $group(K)$
        for all $I' \in wait(K)$
            $bdiv(I', K)$
            put $I'$ in list $ready(K)$
            perform all my $bmod\_send(K, I', J)$, for $J \in ready(K)$
    else if $diag(K) = 1$
        $bdiv(I, K)$
        broadcast $L_{IK}$ to processors in $group(K)$
        put $I$ in list $ready(K)$
        perform all my $bmod\_send(K, I, J)$, for $J \in ready(K)$
    else
        put $I$ in list $wait(K)$
```

Figure 7: Function *last_block_op*.

comparisons between the mappings presented below, with respect to the two criteria. In the next section, the experiments will give a more precise insight.

3.1 Grid mappings

The Geist and Ng [9] algorithm is used to map local subtrees. They proposed to keep a list of subtrees, initialized with the root, and to try a mapping based on a bin-packing algorithm (i.e. giving subtrees in decreasing cost order to the currently least loaded processor). If the workload imbalance is unacceptable, the heaviest subtree is deleted from the list, and its sons are added to the list.

Distributed panels blocks are mapped following an idea of Rothberg and Gupta [20], which they used for the fan-out algorithm. The p processors are assumed to be connected in a $p_r \times p_c$ grid, and a block row (column) of the matrix is mapped to a row (column) of the grid. The algorithm from [21] is used to balance grid rows work. This mapping algorithm is adapted to the fan-in strategy by only changing costs associated with blocks, as *bmod* operations are performed at source and not at destination.

This grid mapping is attractive for its simplity and for limiting communication: since $group(K)$ is a grid column, a processor will broadcast a factorized block only to $p_r - 1$ processors. On the other hand, $owner(I, J)$ will receive updates only from processors on its row because $bmod(K, I, J)$ is always performed by $owner(I, K)$ (recall that $I \geq J$). Generally, the fan-in strategy implies a smaller number of communicated blocks compared to fan-out. It is difficult to make a similar assertion about communication volume; supernodes tend to became larger for greater indices, and thus destination blocks (L_{IJ}) usually have greater size than source blocks. However, our experiments showed a smaller communication volume of fan-in methods. For further reference, we will call *fan-in on grid* (FI_GRID) the fan-in algorithm with grid mapping with row and column locality and row balancing.

3.2 Proportional mappings

The grid mapping is somehow rigid; that is, always allocating a grid column to a panel is a restrictive scheme. As noticed in the previous section, we are free to choose processors in $group(K)$ as desire without affecting the correctness of the fan-in algorithm. Let us suppose that the cost of a communication between any two processors is the same, no matter the physical connectivity of the architecture (in fact this is a fair assumption for many actual parallel distributed-memory computers).

Looking again at figure 3, let imagine that a distributed panel J have two local subtrees as descendants, like in the right side of the figure. Let suppose that two different processors are in charge with the local computation of the two subtrees. Since all updates $bmod(K, I, J)$ are performed by the two processors, it is natural to map panel J to them; all updates targeted to panel J will be communicated only between these processors. Moreover, the factorization of panel J will also imply the same pattern. Finally, global load balance seems to be preserved, with two conditions. First, that work on local subtrees has been balanced, which is supposed to be true, and second, that the work on panel J is evenly distributed between the two processors forming $group(J)$. This mapping idea can be immediately generalized for the whole elimination tree. More precisely

$$group(J) = \bigcup_{K \in sons(J)} group(K),$$

i.e. a panel is mapped to all processors owning its sons.

The Geist and Ng [9] mapping of local subtrees can be used, but it has the drawback of mapping several subtrees to the same processor. Presumably, more communication will be necessary than in the case of one subtree per processor, because large groups of processors tend to appear (every processor may be allocated to any subtree rooted in a distributed panel).

The proportional mapping of Pothen and Sun [17], meant by the authors for the multifrontal method, seems more promising. Although the one local subtree per processor condition is not guaranteed, this mapping is very effective in providing disjoint groups. We present the basic lines of a slightly modified version of the algorithm in figure 8. The principle is simple: each son K (in the elimination tree) of a panel J is mapped to a subset of $group(J)$ of size proportional to the cost $w(J)$ of the subtree rooted in K. The algorithm is recursively applied; the recursion is stopped when a group has size 1, i.e. a local subtree was obtained. The calling arguments for *prop_map* are the root of the elimination tree (argument J), a list of all processors ($g \equiv group(J)$) and a list of length p initialized with zeros, standing for processor total work (pw).

Since $|group(K)|$ is an integer, there are problems caused by rounding, e.g. sons for which less than i processors can be allocated; the original paper [17] shows how to avoid all these problems. We did not address the problem of how the i processors are selected in line 6; it seems difficult to propose a heuristic; so that we simply took the first i processors from the list g.

The blocks of a panel J are mapped after the mapping of all subtrees rooted in sons of J, i.e. a bottom-top approach is preferred. The explanation is natural; the mapping in line 15 may be based on the most recent processor workload information, since the Gaussian

```
         function $prop\_map(J, g, pw)$
1.           $sons(J)$ is the list of panel $J$ sons, ordered upon decreasing cost
2.           $m \leftarrow |g|$, i.e. the number of processors in $group(J)$
3.           $w \leftarrow \sum_{K \in sons(J)} w(K)$
             // allocate processors in $g$ to sons of $J$
4.           for $K \in sons(J)$ (in decreasing cost order)
5.               $i \leftarrow \lfloor w(K)/w * m + 0.5 \rfloor$
6.               $group(K) \leftarrow$ a set of $i$ processors in $g$
7.               $g \leftarrow g \setminus group(K)$
             // recursive proportional mapping
8.           for $K \in sons(J)$
9.               if $|group(K)| > 1$
10.                  $prop\_map(K, group(K), pw)$
11.              else
12.                  map the subtree rooted in $K$ to $P$ $(group(K) = \{P\})$
13.                  $pw(P) \leftarrow pw(P) + w(K)$
             // map current panel blocks to $group(K)$
14.          for $I \in col(J) \cup \{J\}$
15.              map $L_{IJ}$ to a processor $P \equiv owner(I, J) \in group(J)$
16.              add to $pw(P)$ the cost associated with $L_{IJ}$
```

Figure 8: An outline of the proportional mapping algorithm.

elimination evolution is from bottom to top in the elimination tree. We intentionally left unaddressed the details of how blocks in a panel are mapped. Any processor in $group(K)$ may be a candidate to own any block. We will distinguish two classes of mappings, favorizing load balancing and communication volume, respectively.

Greedy mapping. The simpler idea is to give, in line 15 of *prop_map*, the current block to the least loaded processor of the group. The loop 14 goes in decreasing order of row indices, usually close to a decreasing block cost order. A good load balancing is assured. A possible drawback is that there is no attempt to limit communication (if we don't count the limitations provided by the proportional mapping itself); a factorized block will be broadcast to all processors in the group, while updates may be received from all processors in the group.

Subgrid mappings. Let imagine the case of a supernode divided in several panels; in the elimination tree, these panels form a chain; if the supernode is fundamental – and, if not, in most cases – all the panels will be mapped to the same group of processors. Structuring the group becomes interesting, as a way to further reduce communication; if the group is split in subgroups and each panel is mapped to a subgroup, broadcasts will occur only inside subgroups.

The most natural (to the block Cholesky factorization) is the (sub)grid structure. A group of m processors may be thought as a $m_r \times m_c$ grid. A panel will be mapped to a column of the subgrid. Of course, there are some precautions to observe. Subgrid dimensions m_r and m_c may be chosen among the divisors of m; we take m_r and m_c

such that the subgrid be as close of a square as possible. Since the proportional mapping algorithm doesn't offer a control of group sizes, m may have few divisors. We adopted the following heuristic necessary conditions to use subgrid mapping instead of the greedy one: $m_r \leq m_c$, $m_c/m_r \leq \alpha$, $m_r \neq 1$; the constant α is used to limit the "distance" to a square grid; we used $\alpha = 3$, but this choice is purely intuitive.

The problem is now how to map panels to columns of subgrids. We identified two appealing techniques:

- mapping a panel to the currently least loaded subgrid column;
- wrapping the panels of a supernode on subgrid columns (starting with the least loaded column).

Further on panel blocks must be mapped; again, we can favorize load balancing or communication reduction. Two strategies result, respectively

- mapping a block to the least loaded processor on current subgrid column;
- wrap mapping nonzero blocks of the panel on the subgrid column.

The subgrid column and row strategies are independent so that four combinations result from the above discussion. (We should say however that other possibilities exist, for example the trivial cyclic mapping. We chose those techniques that seemed offering more robustness.)

For further reference, we will use the following notations for the variants of the fan-in algorithm with proportional mapping: FI_PROP_G for the greedy mapping and FI_PROP_SG_xx for the subgrid mappings, where the first x is for the column strategy and the second for the row strategy; we use the letter L for the "least loaded" approach and W for the wrap mapping.

3.3 Forest-to-subcube mappings

As we mentioned, a difficulty of the proportional mapping is the lack of control on group sizes. Gupta, Karypis and Kumar [11] proposed the *subforest-to-subcube* mapping, which ensures that groups are always subcubes of a hypercube (the underlying architecture being a hypercube). The basic idea is to try to split a list of unassigned subtrees, initialized with the root, into two parts of roughly equal costs, and to map each part to a half of the current group (initially, the whole hypercube); the algorithm is recursively applied for the resulted sublists and subgroups (which are always subcubes of the hypercube); if, when splitting the list, the imbalance is unacceptable, then the root of the heaviest subtree is mapped to the whole group, the root is deleted from the list, its child subtrees are added to the list and the algorithm is applied to the new list and the same group.

We can introduce in this algorithm, which was originally panel oriented, the same techniques for mapping the blocks of a panel as in the previous subsection. Let us name the algorithm FI_CUBE (suffixes may be added like for FI_PROP). Since group sizes are a power of 2, the advantage over the proportional mapping is that subgrids are always a square or a rectangle with $m_c = 2m_r$. A possible drawback is that there may be more local subtrees per processor; moreover, like Geist and Ng's mapping, a bound of the accepted imbalance must be input to the algorithm; the proportional mapping has no parameters. We do not insist anymore because the experiments showed that usually the proportional mapping is better.

3.4 Heuristic comparisons

When evaluating algorithms, we insisted above on communication volume; it is presumable that the proportional mapping is the best for this criterion. Load balancing is harder to evaluate in general, but we can appreciate that FI_PROP and FI_CUBE are better than FI_GRID, due to their flexibility. Another feature of interest is the intrinsic parallelism offered by the algorithm, besides load balancing; even if globally the processors have the same amount of work, idle times may occur for some processors due to the lack of received blocks. The proportional mapping seems more subject to idle times; a processor has assigned blocks in panels that form a chain in the elimination tree, i.e. has no other work alternative if idle. For the other mappings, a processor may have assigned blocks in panels of the whole elimination tree; a temporary lack of work on one subtree may be compensated by available work on another.

4 Experiments

We implemented in C all methods described in the previous sections, using MPI [23] for communication. BLAS 3 routines were used whenever possible. Portable and (hopefully) efficient programs resulted.

We tested our programs on several matrices of the Harwell-Boeing collection presented in table 1. For our use, we abbreviated their names; B15 stands for BCSSTK15, etc. The last matrix, G150, is a 5-point grid discretization. We have added a suffix indicating the method used for ordering; K stands for the method used in MeTiS [14], using nested dissection principle (K is from Karypis, the first author); A stands for the approximate (external) minimum degree of Amestoy, Davis and Duff [1] (which is faster than exact minimum degree and gave similar results for our matrices). Matrix G150 is ordered with optimal nested dissection.

At least for these matrices, minimum degree methods seem less suited for parallel methods; we will report thus mainly for MeTiS orderings; let us mention that for B17 and B25, the matrices where minimum degree offered better sequential performance, MeTiS ordering allowed better parallel times for $p \geq 8$ and $p \geq 16$, respectively.

As we described in section 1, supernodes were identified, amalgamated and split into panels. To give an idea, let say that even after amalgamation supernode size is rather small, about 10 in average; however, there are few large supernodes (among them, the last is the largest) with sizes in the hundreds. Amalgamation implies a greater number of operations for factorization; table 1 gives the flop count without amalgamation; however we will report our Mflops performances to this flop count; as a consequence, the actual figures are 5-10% better.

Panel size is an important issue. Intuitively, large panels favorize BLAS 3 routines, while small panels offer a greater intrinsic parallelism. Small panels slow down not only computation, but also communication, due to the increased effect of latency. A compro-

Matrix name	Size	Nonzeros in A	Nonzeros in L	Mflop for L
B15_K	3,948	117,816	574,104	122.48
B15_A			627,763	155.45
B16_K	4,884	290,378	754,734	150.53
B16_A			812,183	186.42
B17_K	10,974	428,650	1,188,305	207.13
B17_A			1,055,927	162.99
B18_K	11,948	149,090	673,070	111.58
B18_A			645,717	131.67
B25_K	15,439	252,241	1,842,860	491.96
B25_A			1,479,108	316.32
G150	22,500	111,900	721,862	62.51

Table 1: Test matrices.

mise is necessary between these opposing tendencies. In our experiments, we varied panel size in order to find the best choice. We will present some recommendations which cannot be generalized without care. When no explicit mention, we report results for the best panel size.

We run our experimented on two parallel computers. The first is a 32 processors IBM SP1, located at LMC–IMAG. Its peak performance for processors for BLAS 3 routines is about 100 Mflops. The communication rate can go to 30 Mbytes/s, while the start-up time for a message is about 60 μs. Each node has 64 Mbytes of local memory. For sparse matrix computations, we cannot hope at maximal computation and communication speed. For the block sizes resulted for the test matrices, fair figures are 60 Mflops and 10 Mbytes/s. This means a ratio of about 50 flop for one transmitted double precision floating point, which is rather high; i.e. communication is slow with respect to computation. More than that, decreasing panel size implies a greater degradation of this ratio.

The second is a Cray T3D located at CEA Grenoble. This peak performance for processors for BLAS 3 is about 110 Mflops, i.e. similar to the SP1. The communication rate with MPI is at most 35 Mbytes/s. However, compared to the SP1, communication is roughly three times faster for current block sizes.

In the SP1, any pair of processors can be physically connected by the means of a switch. In the T3D, the communication network is a three-dimensional torus. On both computers, communication speed is the same between any two processors if we neglect possible delays due to conflicts on reserving switch channels on the SP1 or communication paths on the T3D. Since programs execution times are affected by paging effects and possibly by communication contention, we always report the best of four successive executions. Usually, there were not significant variations.

4.1 Sequential performance

The sequential version of our programs attained usually 50-60 Mflops on the SP1, which may be considered satisfactory for test matrix sizes, but only 25-30 Mflops on the T3D. We must stress that the performance varies enough function of panel size. An example is

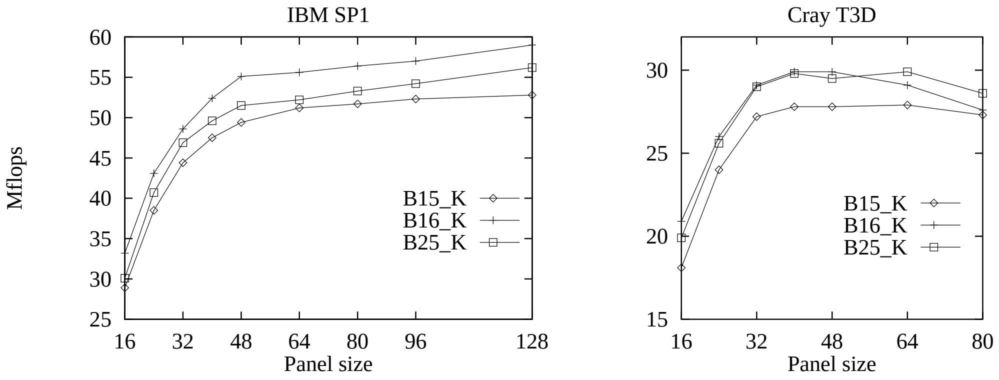

Figure 9: Sequential performance (Mflops) for variable panel size.

given in figure 9, for three of our matrices. On the SP1, it can be seen that for a block size of 16, the computation speed is little more than one half of the maximal speed (notice that for panels larger than 128, performance is still slowly growing); this fact is clearly limiting parallel speed-up, since small panels enhance problem parallelism. On the T3D, the higher performance is attained for smaller blocks than on SP1; more than that, for very large panels, performance becomes worse. These remarks, together with the better communication vs. computation speed ratio, allow us to anticipate better speed-ups on the T3D.

4.2 Parallel performance

We present here some significant results obtained from many timings for the different methods, matrices, panel sizes and computers. There are still parameters we did not vary; the most important is the imbalance bound in Geist and Ng's algorithm, used for local subtrees mapping in FO_GRID and FI_GRID; we used the value 1.4 (the ratio between the largest load of a processor and the average load), which gave good results for $p = 16$ processors; for $p = 2$ we used the bound 1.2; it is clear that for some given matrix, p and panel size a better value can be found, but we appreciate that the improvement is minor.

Since FO_GRID was implemented by Rothberg on a iPSC/860 [19] having a ratio of communication vs. computation speed of the same order as the IBM-SP1, we compared his and our speed-ups. They are similar, but we must remark that Rothberg used only minimum degree orderings, to which the comparison was thus limited.

Grid mappings. Let us first compare the two grid algorithms. For a small number of processors (i.e. $p \leq 8$), FO_GRID and FI_GRID have very similar performance. On the IBM-SP1, for $p = 16$, there is a small advantage for FI_GRID; for $p = 32$, FI_GRID is always better. On the Cray-T3D, it is difficult to say which method to choose, although FI_GRID seems slightly better. See table 2. The general performance of FI_GRID on the test matrices is presented in figure 10.

Although FI_GRID brings an improvement over FO_GRID, the scalability of the algorithm seems poor. In fact, the matrices used for experiments require a small amount

	p	B15_K	B16_K	B17_K	B18_K	B25_K	G150	B17_A	B25_A
on SP1	16	0.7%	4.9%	0.4%	0.9%	0%	10.8%	0.7%	0%
	32	10.2%	9.9%	4.3%	15.5%	6.5%	18.0%	7.5%	7.1%
on T3D	16	0.6%	3.6%	-2.6%	6.7%	2.0%	7.9%	0%	1.1%
	32	-0.2%	5.2%	1.8%	4.3%	3.5%	7.2%	-4.2%	-1.9%

Table 2: Performance improvement: FI_GRID with respect to FO_GRID.

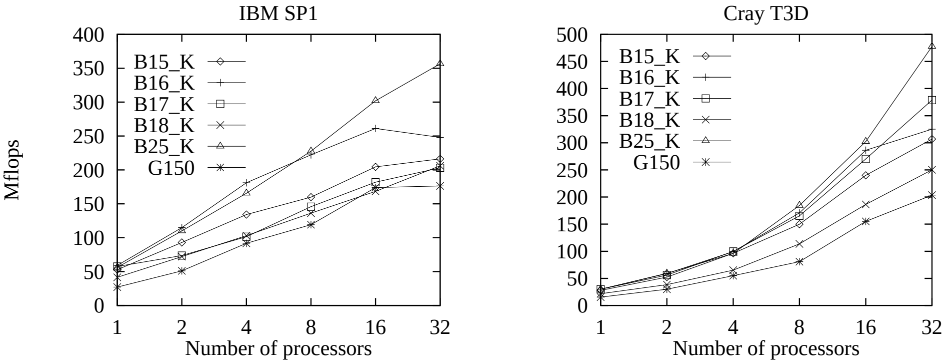

Figure 10: Performance (Mflops) for the fan-in on grid (FI_GRID) algorithm.

of work, compared to processor power. Even the largest, B25_K, is factorized in little more than one second on 32 processors. We can thus expect much better results for larger matrices. It is also to say that the test matrices are also irregular, excepting G150.

Proportional mappings. The proportional mapping reveals its superiority over the grid mappings, for nested dissection orderings. Again, for $p \leq 8$, it is difficult to distinguish a better method; for the proportional mapping it is intuitive that a small number of processors is not favorable, because rounding errors in group allocation may be important (line 5 in figure 8). For $p = 16$, $p = 32$, FI_PROP_G is largely superior, as seen in table 3; compare also to table 2. However, let remark that the improvement on the T3D is smaller than on the SP1.

The performance of FI_PROP_G is presented in figure 11, which show a better scalability of this algorithm, compared to grid ones. The best result is again for B25_K, letting us estimate that larger matrices will furnish even better performance.

However, minimum degree orderings give advantage to grid algorithms. The explanation is simple; the elimination tree is high and thin, at least in its upper part; the proportional mapping produces groups with slowly decreasing size; a greater communication volume results. In this case, grid algorithms are more robust, with their general and simple communication pattern.

		B15_K	B16_K	B17_K	B18_K	B25_K	G150
on SP1	$p = 16$	1.9%	21.8%	4.2%	14.4%	14.0%	16.3%
	$p = 32$	18.6%	43.1%	18.0%	27.5%	27.0%	40.2%
on T3D	$p = 16$	-6.4%	3.6%	-5.6%	8.1%	6.8%	6.3%
	$p = 32$	13.7%	32.7%	8.4%	28.4%	9.1%	13.8%

Table 3: Performance improvement: FI_PROP_G with respect to FO_GRID.

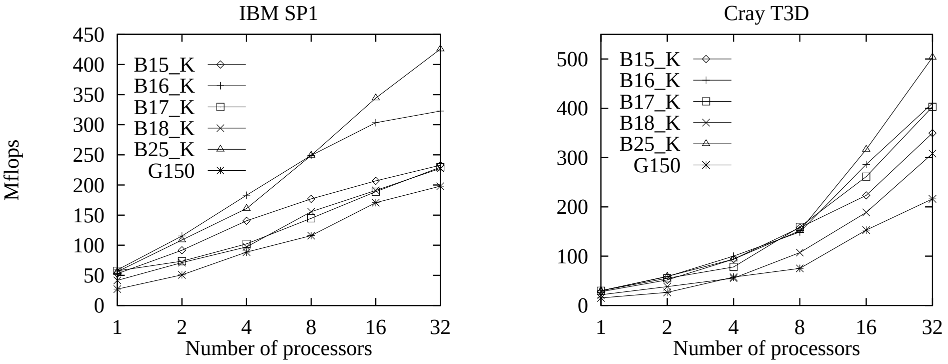

Figure 11: Performance (Mflops) for the fan-in proportional (FI_PROP_G) algorithm.

Experiments with FI_PROP_SG_xx algorithms did not show an improvement over the greedy variant; among them, the "wrap" scheme (FI_PROP_SG_WW) seems the most promising; on the SP1, it gives better results than FI_PROP_G on G150, but slightly lower on the other matrices; on the T3D, it is on the same level as FI_PROP_G. However, further investigation is needed, since there are several parameters to be tuned before stating a firm conclusion.

Subforest-to-subcube mappings gave poor results on the SP1; the cause is the great communication volume; we will exemplify later this issue. On the T3D, these mappings were clearly the best for a small number of processors and left the advantage to FI_PROP_G only for $p = 32$. FI_CUBE_SG_WW is the best in the family; we give in figure 12 a comparison between FI_PROP_G and FI_CUBE_SG_WW on the T3D.

4.3 The effect of panel size

We present here some results and recommendations concerning panel size. In our experiments we varied panel size from 16 to 128 on the SP1 and to 80 on the T3D.

A simple and general rule is obvious: as the number of processors increases, the best panel size is decreasing. On the SP1, a panel of 128 is still the best for $p = 2$

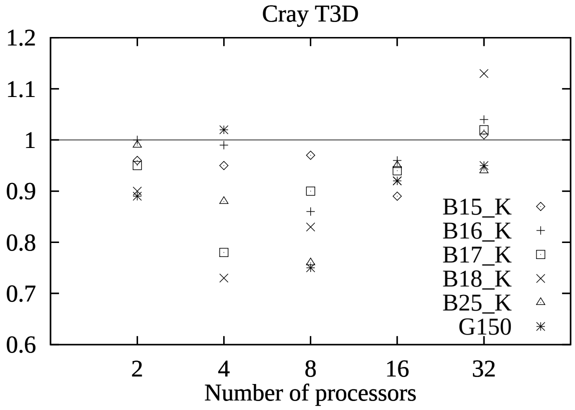

Figure 12: Relative performance: FI_PROP_G vs. FI_CUBE_SG_WW on the T3D.

and $p = 4$, for all algorithms, while for $p = 32$ the best size is smaller and depends on the method. It is interesting to remark that the fan-out algorithm requires a smaller panel size to reach the optimum, than fan-in methods (going down to 16, for certain matrices); the same situation occurs when fan-in on grid is compared to proportional mapping. Figure 13 presents the execution times of four algorithms for B25_K, $p = 32$; we may affirm that curves shape is representative. FO_GRID and FI_GRID have similar behavior for small panels, but FI_GRID is less affected by large panel size. FI_PROP_G has also a good behavior for large panel size (due probably to the greedy strategy). On the contrary, FI_PROP_SG_WW is not so much affected by small panels, when the wrap mapping preserves load balancing; for large panels, the same mapping may cause "accidents" (however, the degradation is not usually as important as in figure 13).

On the T3D, the variation of optimal panel size is not so large, when the number of processors is increased. This is natural, if we remember the sequential performance curve from figure 9. For small number of processors, the best panel size is from 40 to 64, depending on the matrix; for $p \geq 16$, the best panel size is 32 or 40. For a fixed p, FI_CUBE_G and FI_CUBE_SG_WW have similar behaviors to FI_PROP_G and FI_PROP_SG_WW, respectively.

4.4 Communication volume

We argumented some of our strategies with intuitive evaluations of communication volume. Using the same example – B25_K and $p = 32$ – we present in figure 14 the number of messages and the communication volume of the six discussed algorithms.

It results – and the fact is true for the other matrices and other number of processors – that fan-in generally implies less communication than fan-out; the exception is FI_CUBE_G. Other remarks are that proportional mapping reduces communication with respect to grid mapping and that the subgrid wrapping schemes reduce communication with respect to greedy mappings. Moreover, FI_GRID, FI_PROP_G and FI_CUBE_G manifest a decrease of communication when panel size is increased, which partially moti-

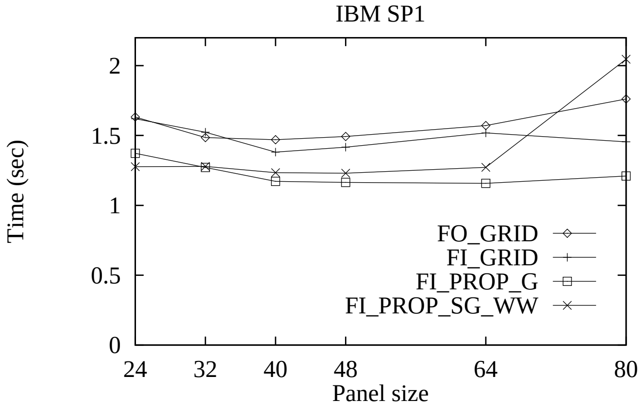

Figure 13: Execution times for B25_K, $p = 32$ and variable panel size, on the SP1.

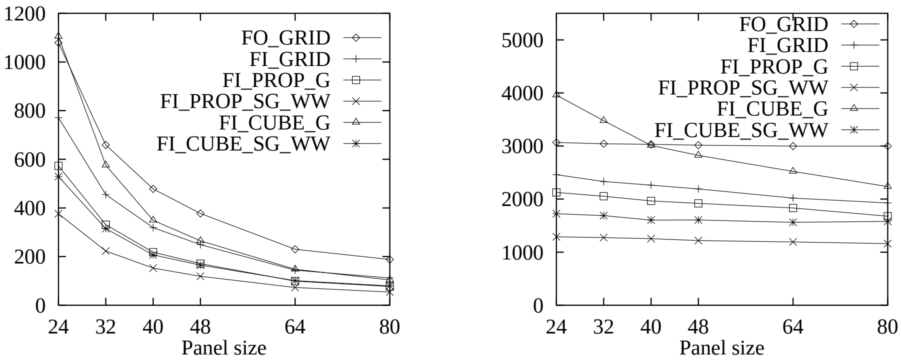

Figure 14: Average number of messages and communication volume (in Kbytes) per processor for B25_K, $p = 32$ and variable panel size.

vates the good behavior of these methods for large panels.

5 Conclusions and future work

We presented a fan-in algorithm for computing the sparse Cholesky factorization, using several 2D block mappings. The algorithm FI_GRID, using a grid mapping, is usually better than Rothberg's fan-out algorithm. Even better is the FI_PROP_G, based on proportional mapping of columns and a greedy mapping inside a column.

Experiments on a IBM SP1 showed that mapping design aiming communication volume reduction is successful, since proportional mapping was proved to be the best. On a Cray T3D, where communication is cheaper, the differences between the proposed algorithms are attenuated; while the fan-in principle remain better than fan-out, the subforest-to-subcube mappings become an alternative for the proportional mapping, especially for a small number of processors.

Although our algorithms proved their efficiency, there is still place to improvements and comparisons. We will also continue to investigate the possibility of finding better mappings and to test those already proposed on larger matrices and on a greater number of processors.

We are also working on the implementation of the sparse Cholesky factorization with the ATHAPASCAN 1 programming interface [5], currently under development in the Apache project. This programming interface makes a clear distinction between the application description and the run time scheduling policy. Additionally, it is possible to choose the better policy for a specific application and a target machine architecture. In the ATHAPASCAN C++ library, we intend to integrate some of the various mapping techniques presented in this paper for the sparse Cholesky bi-dimensional factorization algorithm in order to compare the MPI programming model used in this paper to the alternative ATHAPASCAN approach.

References

[1] P.R. Amestoy, T.A. Davis, and I.S. Duff. An Approximate Minimum Degree Ordering Algorithm. *SIAM J.Matrix Anal.Appl.*, 17(4):886–905, October 1996.

[2] C. Ashcraft, S.C. Eisenstat, and J.W.H. Liu. A Fan-in Algorithm for Distributed Sparse Numerical Factorization. *SIAM J.Sci.Stat.Comput.*, 11(3):593–599, May 1990.

[3] C. Ashcraft and R. Grimes. The Influence of Relaxed Supernode Partitions on the Multifrontal Method. *ACM Trans.Math.Soft.*, 15(4):291–309, December 1989.

[4] C. Ashcraft, R. Grimes, J. Lewis, B. Peyton, and H. Simon. Progress in Sparse Matrix Methods for Large Linear Systems on Vector Supercomputers. *Internat.J.Supercomput.Appl.*, 1:10–29, 1987.

[5] Gerson Cavalheiro and Mathias Doreille. ATHAPASCAN: A C++ library for parallel programming. In *Stratagem'96*, page 75, Sophia Antipolis, France, July 1996. INRIA.

[6] J.J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A Set of Level-3 Basic Linear Algebra Subprograms. *ACM Trans.Math.Software*, 16:1–17,18–28, 1990.

[7] I.S. Duff. Sparse numerical linear algebra: direct methods and preconditioning. Technical Report TR/PA/96/22, CERFACS, 1996.

[8] L. Facq and J. Roman. Distribution par bloc pour une factorisation parallèle de Cholesky. In Authié, G. and al., editor, *Parallèlisme et applications irrégulières*, pages 135–147. Hermès, 1995.

[9] G.A. Geist and E. Ng. Task Scheduling for Parallel Sparse Cholesky Factorization. *Internat. J. Parallel Programming*, 18:291–314, 1989.

[10] A. George, M.T. Heath, J. Liu, and E. Ng. Sparse Cholesky Factorization on a Local-Memory Multiprocessor. *SIAM J.Sci.Stat.Comput.*, 9(2):327–340, March 1988.

[11] A. Gupta, G. Karypis, and V. Kumar. Highly Scalable Parallel Algorithms for Sparse Matrix Factorization. Technical Report 94-63, Department of Computer Science, University of Minnesota, Minneapolis, 1994.

[12] M.T. Heath, E. Ng, and B.W. Peyton. Parallel Algorithms for Sparse Linear Systems. *SIAM Review*, 33(3):420–460, September 1991.

[13] L. Hulbert and E. Zmijewski. Limiting Communication in Parallel Sparse Cholesky Factorization. *SIAM J.Sci.Stat.Comput.*, 12(5):1184–1197, September 1991.

[14] G. Karypis and V. Kumar. METIS – Unstructured Graph Partitioning and Sparse Matrix

Ordering System, version 2.0. Technical report, Department of Computer Science, University of Minnesota, Minneapolis, 1995.

[15] J.W.H. Liu. The Role of Elimination Trees in Sparse Factorization. *SIAM J.Matrix Anal. Appl.*, 11(1):134–172, January 1990.

[16] E. Ng and B.W. Peyton. A Supernodal Cholesky Factorization Algorithm for Shared-memory Multiprocessors. *SIAM J.Sci.Comput.*, 14(4):761–769, July 1993.

[17] A. Pothen and C. Sun. A Mapping Algorithm for Parallel Sparse Cholesky Factorization. *SIAM J.Sci.Comput.*, 14(5):1253–1257, September 1993.

[18] E. Rothberg. *Exploiting the Memory Hierarchy in Sequential and Parallel Sparse Cholesky Factorization.* PhD thesis, Stanford University, January 1993.

[19] 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):699–713, May 1996.

[20] E. Rothberg and A. Gupta. An Efficient Block-oriented Approach to Parallel Sparse Cholesky Factorization. *SIAM J.Sci.Comput.*, 15(6):1413–1439, November 1994.

[21] E. Rothberg and R. Schreiber. Improved Load Distribution in Parallel Sparse Cholesky Factorization. In *Supercomputing '94*, pages 783–792, 1994.

[22] R. Schreiber. Scalability of Sparse Direct Solvers. In A. George, J.R. Gilbert, and J.W.H. Liu, editors, *Graph Theory and Sparse Matrix Compution*, pages 191–209. The IMA Volumes in Mathematics and its Applications, Volume 56, 1993.

[23] Marc Snir, Steve W. Otto, S. Hess-Lederman, David Walker, and Jack J. Dongarra. *MPI: The Complete Reference.* MIT Press, Cambridge, Mass., 1996. Available electronically; see http://www.netlib.org/utk/papers/mpi-book.html.