Available online at www.sciencedirect.com
ScienceDirect
Parallel Computing 34 (2008) 345–362
PARALLEL COMPUTING
www.elsevier.com/locate/parco

On finding approximate supernodes for an efficient block-ILU(k) factorization

Pascal Hénon *, Pierre Ramet, Jean Roman

ScAlApplix Project, INRIA Futurs and LaBRI UMR 5800, Université Bordeaux 1, 33405 Talence Cedex, France

Received 16 January 2007; received in revised form 5 December 2007; accepted 12 December 2007
Available online 1 January 2008

Abstract

Among existing preconditioners, the level-of-fill ILU has been quite popular as a general-purpose technique. Experimental observations have shown that, when coupled with block techniques, these methods can be quite effective in solving realistic problems arising from various applications. In this work, we consider an extension of this kind of method which is suitable for parallel environments. Our method is developed from the framework of high performance sparse direct solvers. The main idea we propose is to define an adaptive blockwise incomplete factorization that is much more accurate (and numerically more robust) than the scalar incomplete factorizations commonly used to precondition iterative solvers. These requirements lead to a robust class of parallel preconditioners based on generalized versions of block ILU techniques.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Sparse linear algebra; Incomplete ILU factorization; High performance computing

1. Introduction

Solving large sparse linear systems by iterative methods [27] has often been unsatisfactory when dealing with practical "industrial" problems. The main difficulty encountered by such methods is their lack of robustness and, generally, the unpredictability and inconsistency of their performance when they are used over a wide range of different problems. Some methods can work quite well for certain types of problems but can fail on others. This has delayed their acceptance as "general-purpose" solvers in a number of important applications.

Meanwhile, significant progress has been made in developing parallel direct methods for solving sparse linear systems, due in particular to advances made in both the combinatorial analysis of Gaussian elimination process and on the design of parallel block solvers optimized for high performance computers. For example, it is now possible to solve real-life three-dimensional problems with several to a few tens of millions of

* Corresponding author. Tel.: +33 5 40 00 38 21.
E-mail addresses: henon@labri.fr (P. Hénon), ramet@labri.fr (P. Ramet), roman@labri.fr (J. Roman).

0167-8191/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2007.12.003

unknowns, very effectively, with sparse direct solvers. This is achievable by a combination of state-of-the-art algorithms along with careful implementations which exploit superscalar effects of the processors and other features of modern architectures [6,12–14,16]. However, direct methods will still fail to solve very large three-dimensional problems, due to the potentially huge memory requirements for these cases. On the other hand, the iterative methods using a generic preconditioner like an ILU(k) factorization [27] require less memory, but they are often unsatisfactory when the simulation needs a solution with a good precision or when the systems are ill-conditioned. The incomplete factorization technique usually relies on a scalar implementation and thus does not benefit from the superscalar effects provided by the modern high performance architectures. Furthermore, these methods are difficult to parallelize efficiently, more particularly for high values of level-of-fill. Some improvements to the classical scalar incomplete factorization have been studied to reduce the gap between the two classes of methods. In the context of domain decomposition, some algorithms that can be parallelized in an efficient way have been investigated in [21]. In [25], the authors proposed to couple incomplete factorization with a selective inversion to replace the triangular solutions (that are not as scalable as the factorization) by scalable matrix-vector multiplications. The multifrontal method has also been adapted for incomplete factorization with a threshold dropping in [15] or with a fill level dropping that measures the importance of an entry in terms of its updates [9]. In [10], the authors proposed a block ILU factorization technique for block tridiagonal matrices.

The approach investigated in this paper consists in exploiting the parallel blockwise algorithmic approach used in the framework of high performance sparse direct solvers to develop robust parallel incomplete factorization based preconditioners [27] for iterative solvers. The idea is then to define an adaptive blockwise incomplete factorization that is much more efficient than the scalar incomplete factorizations commonly used to precondition iterative solvers. Indeed, by using the same ingredients which make direct solvers effective, these incomplete factorizations exploit the latest advances in sparse direct methods, and can be very competitive in terms of CPU time due to the effective usage of CPU power. At the same time, this approach can be far more economical in terms of memory usage than in terms of direct solvers. Therefore, this should allow to solve the system of much larger dimensions than the one that is solved by the direct solvers. This paper is organized as follows: Section 2 recalls the principal key concepts on which direct solvers are based, Section 3 gives the principles of the block ILU factorization based on the level-of-fill, Section 4 presents our algorithms to obtain the approximate supernode partition which aims at creating the sparse block structure of the incomplete factors and finally, in Section 5, we present some experiments obtained with our method.

2. Background on sparse direct solvers

This section provides a brief background on sparse Gaussian elimination techniques, including their graph models. Details can be found in [3,4] among others. Consider the linear system

$$Ax = b, \tag{1}$$

where A is a symmetric definite positive matrix of size $n \times n$ or an unsymmetric matrix with a symmetric nonzero pattern. In this context, sparse matrix techniques often utilize the non-oriented adjacency graph $G = (V, E)$ of the matrix A, a graph whose n vertices represent the n unknowns, and whose edges (i, j) represent the couplings between unknowns i and j. The Gaussian elimination ($L.L^t$ Cholesky factorization or $L.U$ factorization) process introduces "fill-ins", i.e., new edges in the graph. The quality of the direct solver depends critically on the ordering of the unknowns as this has a major impact on the number of fill-ins generated.

Sparse direct solvers often utilize what is known as the "filled graph" of A, which is the graph of the (complete) Cholesky factor L, or rather of $L + U$. This is the original graph augmented with all the fill-in generated during the elimination process. We will denote by $G^* = (V, E^*)$ this graph.

Two well-known and useful concepts will be needed in the later sections. The first is that of *fill-paths*. This is a path between two nodes i and j (with $i \neq j$) in the original graph $G = (V, E)$, whose intermediate nodes are all numbered lower than both i and j. A theorem by Rose and Tarjan [19] states that there is a fill-path between i and j if and only if (i, j) is in E^*, i.e., there will be a fill-in in position (i, j).

The second important concept is that of *elimination tree* [17], which is useful among other things, for scheduling the tasks of the solver in a parallel environment [1,13]. The elimination tree captures the dependency between columns in the factorization. It is defined from the filled graph using the parent relationship:

$$\text{parent}(j) = \min\{i \mid i > j \text{ and } (i,j) \in E^*\}.$$

Two broad classes of reorderings for sparse Gaussian elimination have been widely utilized. The first, which tends to be excellent at reducing fill-in, is the *minimal degree* ordering. This method, which has many variants, is a local heuristics based on a greedy algorithm [5]. The class of *nested dissection* algorithms [4], considered in this paper, is common in the "static" variants of Gaussian elimination which preorder the matrix and define the tasks to be executed in parallel at the outset. The nested dissection ordering [4] recursively utilizes a sequence of separators. A separator C is a set of nodes which splits the graph into two subgraphs G_1 and G_2 such that there is no edge linking a vertex of G_1 to a vertex of G_2. This is then done recursively on the subgraphs G_1 and G_2. The left side of Fig. 1 shows an example of a physical domain (e.g., a finite element mesh) partitioned recursively in this manner into a total of 8 subgraphs. The labeling used by nested dissection can be defined recursively as follows: label the nodes of the separator last after (recursively) labeling the nodes of the children. This naturally defines a tree structure as shown on the right side of Fig. 1. The remaining subgraphs are then ordered by a minimal degree algorithm under constraint [24].

An ideal separator is such that G_1 and G_2 are of about the same size while the set C is small. A number of efficient graph partitioners have been developed in recent years which attempt to reach a compromise between these two requirements, see, e.g., [2,22,23] among others.

After a good ordering is found, a typical solver performs a *symbolic factorization* which essentially determines the pattern of the factors. This phase can be elegantly expressed in terms of the elimination tree. We will denote by $[i]$ the sparse pattern of a column i which is the list of rows indices in the increasing order corresponding to non-zeros terms.

Algorithm 1. Sequential symbolic factorization algorithm

1 Build $[i]$ for each column i in the original graph
2 **for** $i = 1, \ldots, n-1$ **do**
3 $\quad$ [parent(i)] = `merge` ([i], [parent(i)])
4 **end**

where `merge`($[i], [j]$) is a function which merges the patterns of columns i and j in the lower triangular factor. In a parallel implementation, this is better done with the help of a post-order traversal of the tree: the pattern of a given node only depends on the patterns of the children and can be obtained once these are computed. The symbolic factorization is a fairly inexpensive process since it utilizes two nested loops instead of the three loops normally required by Gaussian elimination. Note that all computations are symbolic, the main kernel being the `merge` of two column patterns.

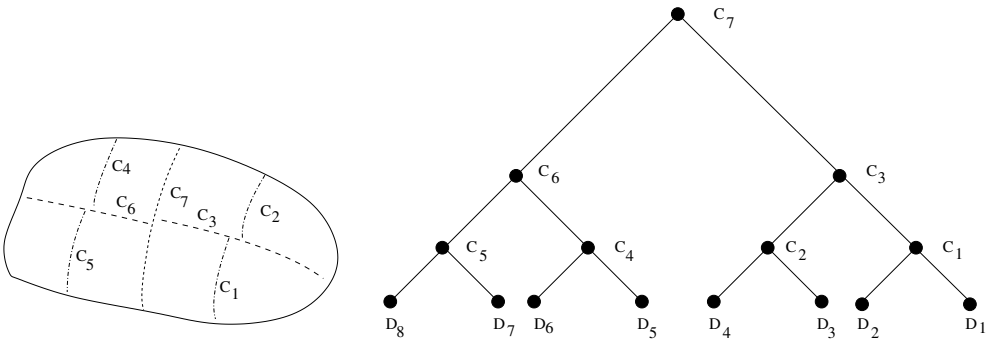

Fig. 1. The nested dissection of a physical mesh and corresponding tree.

Most sparse direct solvers take advantage of dense computations by exhibiting a *dense block structure in the matrix L*. This dense block structure is directly linked to the ordering techniques based on the nested dissection algorithm (ex: METIS [2] or SCOTCH [22]). Indeed, the columns of L can be grouped in sets such that all columns of a same set have a similar non-zero pattern. These sets of columns, called *supernodes*, are then used to prune the block structure of L. The supernodes obtained with such orderings mostly correspond to the separators found in the nested dissection process of the adjacency graph G of matrix A.

An important result used in direct factorization is that the partition $\mathscr{P}$ of the unknowns induced by the supernodes can be found without knowing the non-zero pattern of L [20]. The partition $\mathscr{P}$ of the unknowns is then used to compute the block structure of the factorized matrix L during the so-called *block symbolic factorization*. The block symbolic factorization exploits the fact that when $\mathscr{P}$ is the partition of separators obtained by nested dissection, then we have

$$Q(G,\mathscr{P})^* = Q(G^*,\mathscr{P}),$$

where $Q(G,\mathscr{P})$ is the quotient graph of G with regards to partition $\mathscr{P}$. Then, we can deduce the *block elimination tree* which is the elimination tree associated with $Q(G,\mathscr{P})^*$ and which is well suited for parallelism [13]. Fig. 2 shows the block structure obtained for a 7×7 grid ordered by nested dissection. 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. In this block data structure, an off-diagonal block is represented by a pair of integers (α,β) corresponding to the first and the last rows of this block in the column-block. We will denote by $[i]$ the block sparse pattern of a supernode i which is the list in the increasing order of such intervals.

As a result, the block symbolic factorization will now need to merge two sets of intervals.

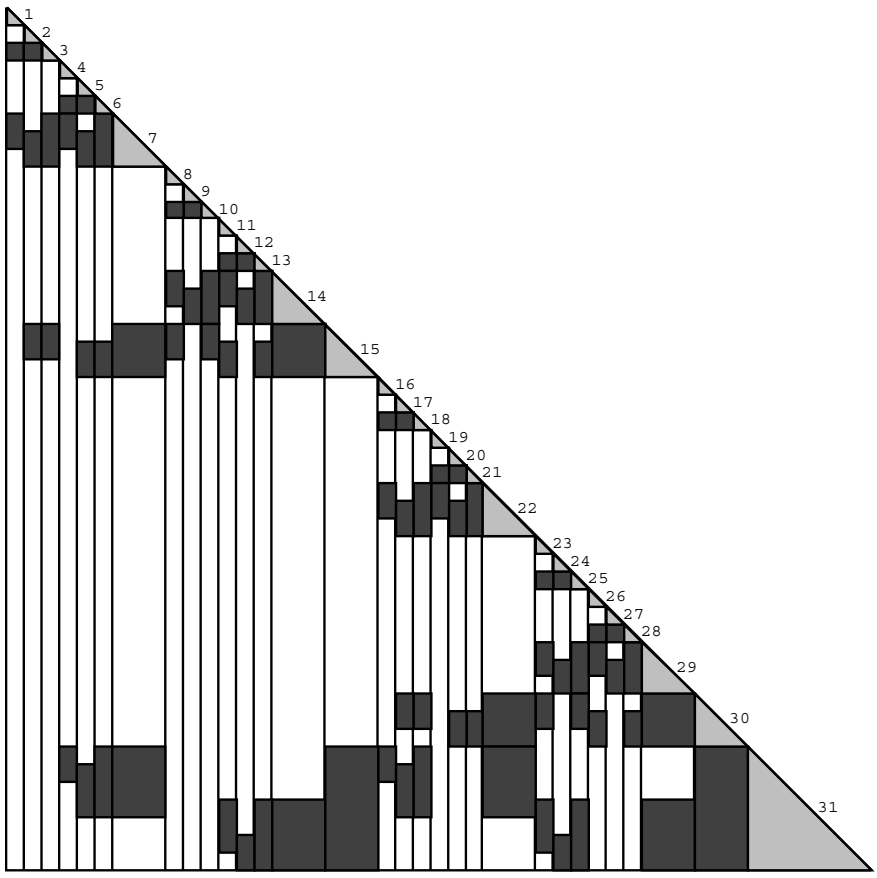

Fig. 2. Block data structure of a factorized matrix. $N = 31$ column-blocks. Diagonal blocks are drawn in light grey and off-diagonal blocks in dark grey.

Algorithm 2. Block symbolic factorization

```
1 Build [i] for each supernode i in the original graph
2 for i = 1, ..., N − 1 do
3    [parent(i)] = Bmerge ([i], [parent(i)])
4 end
```

where `Bmerge` will merge two sorted lists of intervals and where parent( ) defines the father relationship in the block elimination tree. It was shown by Charrier and Roman [11] that the cost of computing the block symbolic factorization when a nested dissection algorithm is used is $O(N)$ for most graphs. In other words, the total number of blocks to be handled is $O(N)$ which means that a total of $O(N)$ pointers and operations are required to perform the block symbolic factorization.

A sparse block factorization algorithm can be obtained by restricting the standard dense block algorithm to the sparsity pattern which has been computed by the symbolic factorization phase.

Algorithm 3. Sparse block Cholesky factorization

```
1 for k = 1, ..., N do
2    Factor A_{k,k} into A_{k,k} = L_k D_k L_k^T
3    for j ∈ [k] do
4       Compute A_{jk}^T = D_k^{-1} L_k^{-1} A_{jk}^T
5    end
6    for j ∈ [k] do
7       for i ∈ [k], i > j do
8          A_{ij} := A_{ij} − A_{ik} D_k A_{jk}^T
9       end
10   end
11 end
```

These algorithms are parallelized and implemented in our supernodal direct solver PaStiX [13,14]. The following presents an approach that uses some of the key concepts of this direct solver to develop a parallel incomplete block factorization. Since PaStiX deals with matrices that have a symmetric pattern, the algorithms presented in the remaining part of this article are based on the assumption that the adjacency graph is symmetric. The algorithms and adaptations that are discussed in Sections 3 and 4 have been implemented in PaStiX; the experiments in Section 5 show the results obtained with this implementation.

3. Methodology

Preconditioned Krylov subspace methods utilize an accelerator and a preconditioner [7]. The goal of the ILU-based preconditioners is to find approximate LU factorizations of the coefficient matrix which are then used to facilitate the iterative process.

Incomplete LU (ILU) or incomplete Cholesky (IC) factorization is based on the premise that most of the fill-in entries generated during a sparse direct factorization will tend to be small. Therefore, a fairly accurate factorization of A can be obtained by dropping most of the entries during the factorization. There are essentially two classical ways of developing incomplete LU factorizations.

The first (historically) consists of dropping terms according to a recursive definition of a level: a fill-in, which is itself generated from another fill-in, will tend to be smaller and smaller as this chain of creation continues. The notion of level-of-fill, first suggested by engineers is described next as it is the stepping stone into the approach described in this paper.

The second is based on the use of thresholds during the factorization. Thus, ILUT [26] is commonly implemented as an upward-looking row-oriented algorithm which computes the ILU factorization row by row. It

utilizes two parameters, the first τ being used as tolerance for dropping small terms relative to the norm of the row under consideration, and the second p determines the maximum number of non-zero elements to keep in each row.

3.1. ILU(k) preconditioners

The incomplete LU factorization ILU(k) implements dropping with the help of a *level-of-fill* associated with each fill-in introduced during the factorization. Initially, each non-zero element has a level-of-fill of zero, while each zero element has (nominally) an infinite level-of-fill. Thereafter, the level-of-fill of l_{ij} is defined from the formula:

$$\mathrm{levf}(l_{ij}) = \min\{\mathrm{levf}(l_{ij}); \mathrm{levf}(l_{ki}) + \mathrm{levf}(l_{jk}) + 1\}. \tag{2}$$

This definition and its justification were originally given by Watts for problems arising in petroleum engineering [28]. Later, Forsyth and Tang provided a graph-based definition, which was then reinterpreted by Hysom and Pothen [18] within the framework of the fill-path theorem [19]. The interpretation of the level-of-fill is that it is equal to $len(i,j) - 1$, where *len* is the length of the shortest fill-path between i and$_j$. It can be easily seen that the path-lengths follow the simpler update rule: $len(i,j) = \min\{len(i,j); len(i,k) + len(k,j)\}$. During Gaussian elimination, we eliminate nodes k, labeled before a certain pair of nodes (i,j). Then, it can be proved that the shortest fill-path from i to j is the shortest of all shortest fill-paths from i to some k plus the shortest fill-path from k to j. This is illustrated in Fig. 3.

The incomplete symbolic ILU(k) factorization has a theoretical complexity similar to the numerical factorization, but based on the graph interpretation of the level-of-fill, an efficient algorithm that leads to a practical implementation, which can be easily parallelized, have been proposed in [18]. Thus, the symbolic factorization for ILU(k) method, though more costly than in the case of exact factorizations, is not a limitation in our approach.

3.2. Block-ILU(k)

The idea of level-of-fill has been generalized to blocks in the case of block matrices with dense blocks of equal dimensions, see, e.g., [8]. Such matrices arise from discretized problems when there are m degrees of freedom per mesh-point, such as fluid velocities and pressure. This is common in particular in Computational Fluid Dynamics.

It is clear that for such matrices, it is preferable to work with the quotient graph, since this reduces the dimension of the problem. It is clear that the factorization ILU(k) obtained using the quotient graph and the original is then identical. In other words, we consider the partition $\mathscr{P}_0$ constructed by grouping set of unknowns that have the same row and column pattern in A; these set of unknowns are the cliques of G. In this case, if we denote by G^k the adjacency graph of the elimination graph for the ILU(k) factorization, then we have

$$Q(G^k, \mathscr{P}_0) = Q(G, \mathscr{P}_0)^k.$$

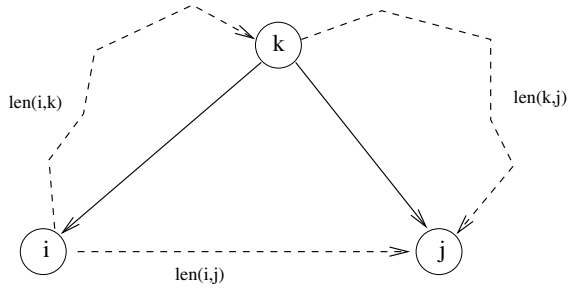

Fig. 3. Fill-paths from i to j.

For coarser partition than $\mathscr{P}_0$, these properties are not true anymore in the general case. Therefore, the ILU(k) symbolic factorization can be performed with a significant lower complexity than the numerical factorization algorithm. In addition, by using the algorithm presented in [18] that is easily parallelizable, the symbolic bloc incomplete factorization is not a bottleneck in our approach.

For direct factorization, the supernode partition usually produces blocks that have a sufficient size to obtain a good superscalar effect using the BLAS-3 subroutines. The exact supernodes that are exhibited from the incomplete factor non-zero pattern are usually very small. Consequently, a blockwise implementation of the ILU(k) preconditioner based on the exact supernode partition would not be very efficient and can even be worse than a classical columnwise implementation due to the overcost of calling the BLAS subroutines. A remedy to this problem is to merge supernodes that have *nearly the same structure*. This process induces some *extra fill-in* compared to the exact ILU(k) factors but the increase of the number of operations is largely compensated by the gain in time achieved thanks to BLAS subroutines. The convergence of our Block-ILU(k) preconditioner is at least the one obtained by scalar ILU(k). Furthermore, it can also improve the convergence of the solver since the extra-fill admitted in the factors mostly correspond to numerically non-null entries that may improve the accuracy of the preconditioner.

The principle of our heuristics to compute the new supernode partition is to iteratively merge supernodes for which non-zero patterns are the most similar until we reach a desired extra fill-in tolerance. To summarize, our incomplete block factorization consists in the following steps:

(1) find the partition $\mathscr{P}_0$ induced by the supernodes of A;
(2) compute the block symbolic incomplete factorization $Q(G,\mathscr{P}_0)^k$;
(3) find the exact supernode partition in $Q(G,\mathscr{P}_0)^k$;
(4) given an *extra fill-in* tolerance α, construct an approximated supernode partition $\mathscr{P}_\alpha$ to improve the block structure of the incomplete factors (detailed in next section);
(5) apply a block incomplete factorization using the parallelization techniques developed for our direct solver PaStiX.

The incomplete factorization can then be used to precondition a Krylov method to solve the system. The next section will focus on step 4; it gives the details of an amalgamation algorithm that is used to find an approximate supernode partition.

4. Amalgamation algorithm

A blockwise implementation of the ILU(k) factorization is directly obtained using the direct blockwise Algorithm 3. In fact, the exact supernodes that can be exhibited from the symbolic ILU(k) factor are usually too small to allow a good BLAS efficiency in the numerical factorization and in the triangular solves. To address this problem, we propose an amalgamation algorithm which aims at grouping some exact supernodes that have almost similar non-zero pattern to get bigger supernodes. By construction, the exact supernode partition found in an ILU(k) factor is always a sub-partition of the *direct supernode partition* (i.e., corresponding to the direct factorization) since G^k can be obtained by removing some edges from G^*.

As mentioned before, the amalgamation problem consists in merging as many supernodes as possible while adding the fewer extra non-zeros in the sparse block pattern of the incomplete factors.

We propose a heuristics based on a greedy strategy. Here are some extra notations used in Algorithm 4:

- $[k]$, parent(k) and `Bmerge` are defined in the same way as in Section 2;
- nnz_A is the number of non-zeros in A;
- merge_cost (k) is the cost of merging the supernode k with its father in terms of extra-fill;
- son(k) is the set of supernode indices corresponding to the sons of supernode k in the block elimination tree;
- by convention, if k is the root of the block elimination tree, then parent(k) $= k$ and merge_cost(k) $= \infty$.

Algorithm 4. Amalgamation algorithm

1 $nnz = 0$
2 Compute $Q(G, \mathscr{P}_0)^k$ and find S the set of all exact supernodes in G^k
3 **While** $nnz < \alpha * nnz_A$ *and* $S \neq \emptyset$ **do**
4 Choose k/merge_cost$(k) = \min_{i \in S}\{\text{merge_cost}(i)\}$
5 $[k] :=$ `Bmerge`$([k], [\text{parent}(k)])$
6 $S = S - \{\text{parent}(k)\}$
7 $\text{son}(k) := \text{son}(k) \bigcup \text{son}(\text{parent}(k))$
8 $\text{parent}(k) := \text{parent}(\text{parent}(k))$
9 $nnz := nnz + \text{merge\_cost}(k)$
10 Recompute merge_cost(k)
11 **for** $i \in \text{son}(k)$ **do**
12 Recompute merge_cost(i)
13 **end**
14 **end**

Given the set of all exact supernodes, it consists in iteratively merging the couple of supernodes $(i, \text{parent}(i))$ which creates as few as possible additional fill-in the factor (see Fig. 4) while the extra fill tolerance α is respected. Each time a couple of supernodes is merged into a single one, the total amount of extra-fill is increased: the same operation is repeated until the amount of additional fill entries reaches the tolerance α (given as a percentage of the number of non-zero elements found by the ILU(k) symbolic factorization). We denote by merge_cost(i) the number of extra fill created when the supernodes $(i, \text{parent}(i))$ are merged into a single one. Thus, the algorithm consists in choosing at each step the supernode k such that merge_cost(k) is minimum (line 4) and to merge its father with it (line 5). The supernode k is then replaced by the new merged supernode and the supernode parent(k) is deleted from S (lines 5 and 6). The increasing of the global number of extra non-zeros is given by merge_cost(k) (line 9). Since the sparse block structure of k has changed, its merge_cost has to be recomputed (line 10) and the merge_cost of its sons too (lines 11–13).

4.1. Complexity of the amalgamation algorithm

We denote by $\mathscr{P}_e$ the exact supernode partition of G^k (line 2), N_e the cardinal of $\mathscr{P}_e$ and d the maximum degree of a vertex in $Q(G^k, \mathscr{P}_e)$. The amalgamation algorithm requires the set S to be sorted with respect to the

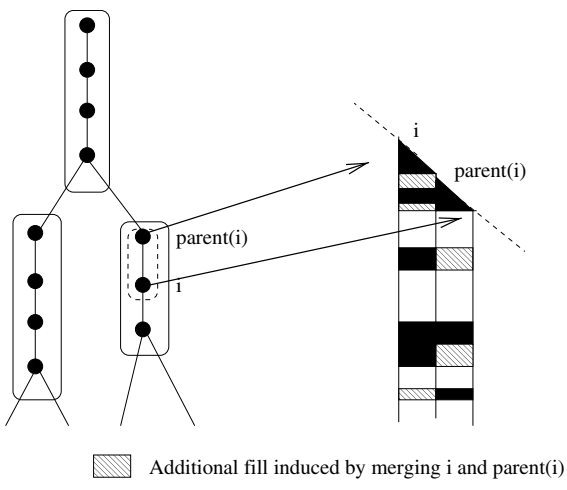

Fig. 4. Additional fill created when merging the supernodes i and parent(i).

merge_cost metric and we have to keep S sorted each time a merge operation is done. One can use a heap to implement the set S; therefore, the cost to add or update an element in S will be at most in $O(\log(N_e))$ and the cost to get the lowest element is constant.

Computing merge_cost( ) or `Bmerge`( ) requires to merge two sorted lists of at most d intervals (a block is represented by an integer interval) and then this operation is bound by $O(d)$. In line 2, all the exact supernodes have to be sorted by the increasing order of merge_cost in S. This operation is thus in $O(N_e \cdot (\log(N_e) + d))$.

Inside the `while` loop, each time a supernode is merged with its father there is a `Bmerge` operation (complexity in $O(d)$) and if we make the assumption that the cardinal of son(k) is bounded by a constant (for a separator tree obtained by nested dissection, the constant will be 2 in most of the cases), then the cost of recomputing the merge_cost of the supernode and its sons is also in $O(d)$ and the cost to update the heap S is in $O(\log(N_e))$. The global cost of an iteration of the `while` loop is then in $O(d + \log(N_e))$. Since in the worst case (where all the exact supernodes would be merged in a single one, leading to a dense matrix) only $N_e - 1$ iterations can be done, a complexity bound of the amalgamation algorithm is in $O(N_e \cdot (\log(N_e) + d))$.

4.2. A variant of the amalgamation algorithm

The amalgamation Algorithm 4 aims at minimizing the number of supernodes according to an extra fill tolerance α. The assumption made here is that the triangular solution and the incomplete block factorization will be all the more efficient that the supernode partition is coarser. A variant to the amalgamation objective is to merge some supernodes to minimize as far as possible the CPU time to apply the triangular solve or the incomplete factorization. Usually in an iterative method, the total time spent in the iterations is more important than the time to compute the preconditioner. So we will focus on reducing the time of the triangular solves. To estimate the time spent in the triangular solve, we use a time model of the BLAS-2 routines used in the blockwise triangular solve algorithm. The time model of a BLAS routine consists in a polynomial that interpolates the curve of the CPU times spent in the routine in function of the block dimensions. These polynomials are obtained once for all on a given architecture (such models are already used in PaStiX to balance the workload and schedule the computation tasks before the parallel factorization). For example, the time to compute a dense matrix-vector product $M.v$ (GEMV BLAS subroutine) mostly depends on the dimensions (x, y) of M. Since the complexity of this operation is in $O(x.y)$, we use a polynomial model $a \cdot x \cdot y + b \cdot x + c \cdot y + d$. Thanks to experimental measures, a multi-variable regression is used to set the coefficients of the polynomial. Fig. 5 illustrates the model obtained on the IBM Power5 architecture and the experimental measures.

Thanks to this polynomial BLAS time model, we can estimate the CPU time W_i that corresponds to the time spent in the supernode i in the triangular solve. What we seek, in this variant, is to merge the couple of supernodes $(k, \text{parent}(k))$ such that the gain of *CPU time per additional non-zero* allowed in the sparse block structure is the best. So, we evaluate the gain of merging k with parent(k) by the function:

$$\text{merge\_gain}(k) = \frac{W_k + W_{\text{parent}(k)} - W_{\texttt{Bmerge}([k],[\text{parent}(k)])}}{\text{merge\_cost}(k)}.$$

Thus, if merge_gain$(k) > 0$, it means that merging k and parent(k) will lower the CPU time in the triangular solves; the higher this value is the better the trade-off (CPU time)/(extra non-zeros stored) is. On the contrary, if merge_gain$(k) < 0$, then it indicates that one should not merge k and its father because the number of additional floating operations induced by the extra non-zeros in the block structure is too important to be balanced by the superscalar effects in the BLAS routines.

Algorithm 5 gives the amalgamation algorithm that aims at reducing the CPU time of the triangular solve for a given additional fill tolerance. The difference with Algorithm 4 is that the couple of supernodes to be merged is chosen as the one that has the higher merge_gain (line 4). Another difference is that any couple of supernodes that would increase the CPU time if they were merged are removed from the possible choices (lines 2, 12 and 15) to decrease the number of operations. This means we make the approximation that a supernode will never get a positive gain even if its father becomes bigger thanks to amalgamation. In practice, this approximation is verified most of the time accordingly to the BLAS model we obtained.

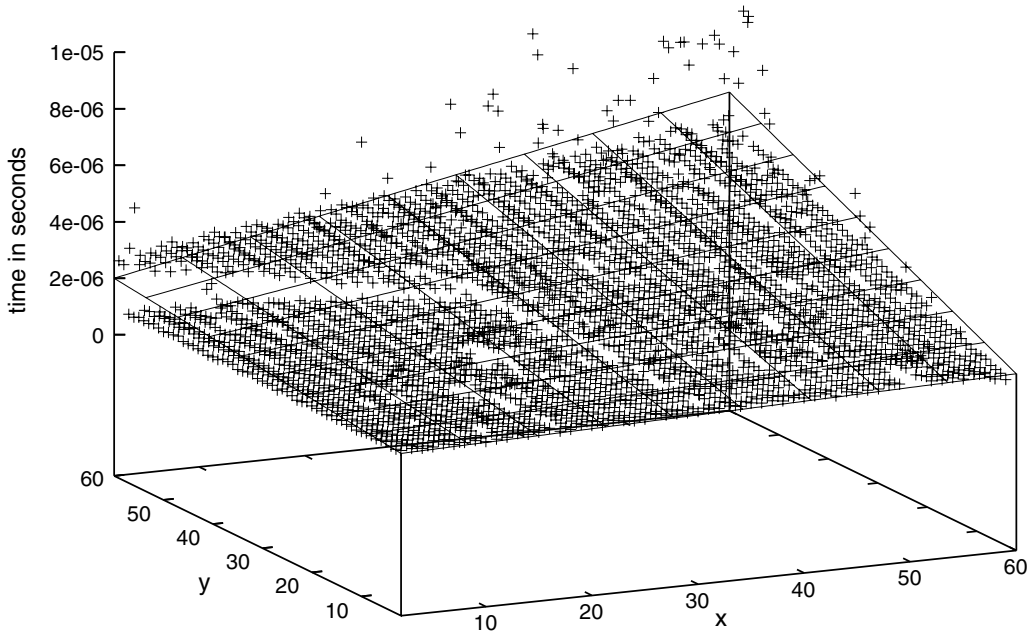

Fig. 5. Polynomial model (surface) and experimental measures (cross) for GEMV obtained on IBM Power5.

Algorithm 5. Amalgamation algorithm variant

1 $nnz = 0$
2 Compute $Q(G, \mathscr{P}_0)^k$ and find S the set of all exact supernodes in G^k that have a merge_gain > 0
3 **while** $nnz < \alpha * nnz_A$ *and* $S \neq \emptyset$ **do**
4 Choose k/merge_gain$(k) = \max_{i \in S}\{\text{merge_gain}(i)\}$
5 $[k] := \texttt{Bmerge}([k], [\text{parent}(k)])$
6 $S = S - \{\text{parent}(k)\}$
7 $\text{son}(k) := \text{son}(k) \bigcup \text{son}(\text{parent}(k))$
8 $\text{parent}(k) := \text{parent}(\text{parent}(k))$
9 Compute merge_cost(k)
10 $nnz := nnz$+merge_cost(k)
11 Recompute merge_gain(k)
12 **if** merge_gain$(k) \leqslant 0$ **then** $S = S - \{k\}$
13 **for** $i \in \text{son}(k)$ **do**
14 Recompute merge_gain(i)
15 **if** merge_gain$(i) \leqslant 0$ **then** $S = S - \{i\}$
16 **end**
17 **end**

In the case of Algorithm 4, if we set $\alpha = \infty$, then it would merge all the supernodes and the sparse matrix L (resp. U) would be considered as a dense matrix. An interesting property of Algorithm 5 is that if we set $\alpha = \infty$, then it stops as soon as it cannot find any amalgamation of supernodes such that the global CPU time decreases (test $S = \emptyset$ in line 3) or as soon as it reaches the extra fill tolerance given by α. Thus, it provides a convenient way to use the amalgamation algorithm without having to choose an arbitrary α (that corresponds to $\alpha = \infty$) parameter.

Algorithm 5 requires to keep S sorted by increasing merge_gain; though the merge_gain is more costly than the merge_cost, it also has a complexity in $O(d)$; therefore, Algorithm 5 has the same asymptotical complexity than Algorithm 4.

5. Experimental results

We consider three test cases (see Table 1) for the numerical validation of our block preconditioner. nnz_A is the number of off-diagonal terms in the triangular part of the original matrix, nnz_L is the number of off-diagonal terms in the complete factor and OPC is the number of operations required for the complete factorization. We consider the ordering provided by the SCOTCH software [22] used in our parallel direct solver PASTIX [13]. AUDI and INLINE test cases (structural mechanic problems from PARASOL collection) are both symmetric problems whereas MHD1 (Magneto-Hydro-Dynamic 3D problem) is an unsymmetric problem.

Numerical experiments were performed on an IBM Power5 SMP node (16 processors per node) at the computing center of Bordeaux 1 University, France. We used a GMRES version without "restart". The stopping iteration criterion used in GMRES is the right-hand side relative residual norm and is set to 10^{-7}.

As some matrices are symmetric definite positive, one could use a preconditioned conjugate gradient method; but at this time we only have implemented the GMRES method to treat unsymmetric matrices as well. The choice of the iterative accelerator is not in the scope of this study.

5.1. Study of the amalgamation algorithm behaviour

Tables 2–4 are related to the influence of the amalgamation parameter α (one for each test case). We consider the amalgamation Algorithm 4 of Section 4 for different values of the extra fill tolerance ($0\% \leqslant \alpha \leqslant 120\%$). The particular value ∞ corresponds to the amalgamation Algorithm 5 when we set $\alpha = \infty$ (referred as "auto" in the following figures).

For the three problems the tables report:

- the level-of-fill (k),
- the percentage of amalgamation ratio (α),
- the number of supernodes,
- the number of blocks,
- the number of non-zeros in the incomplete factors divided by the number of non-zeros in the initial matrix (`Fill-in`),
- the time of amalgamation in seconds (`Amalg.`),
- the time of sequential incomplete factorization in seconds (`Inc. Fact.`),
- the time of triangular solve (forward and backward) in seconds (`Triang. Solve`),
- the number of iterations,
- the total time in seconds.

The best total time is written in **bold face** for each value of the level-of-fill. "–" indicates that GMRES did not converge in less than 200 iterations and "+" indicates that there is not enough memory to allocate the block data structure of the incomplete factors.

We can see in Tables 2–4 that our amalgamation algorithm allows to significantly reduce the number of supernodes and the number of blocks in the dense block pattern of the matrix.

As a consequence, the superscalar effects are greatly improved as the amalgamation parameter grows: this is particularly true for the incomplete factorization which exploits BLAS-3 subroutines (matrix by matrix operations). The superscalar effects are less important on the triangular solve that requires much less floating point operations and use only BLAS-2 subroutines (matrix by vector operations). We can also verify that the time to

Table 1
Description of our test problems

Name	Columns	nnz_A	nnz_L	OPC
AUDI	943,695	39,297,771	1.214519e+09	5.376212e+12
INLINE	503,712	18,660,027	1.762790e+08	1.358921e+11
MHD1	485,597	24,233,141	1.629822e+09	1.671053e+13

Table 2
Effect of amalgamation ratio α for AUDI problem

k	α	# Supernodes	# Blocks	Fill-in	Amalg.	Inc. Fact.	Triang. Solve	Iterations	Total
1	∞	53,287	1,461,344	4.93	15.23	53.7	2.91	118	397
1	0	299,919	11,868,384	2.89	4.13	285	12.3	147	2093
1	10	177,565	6,227,707	3.18	6.06	149	7.06	139	1130
1	20	131,929	4,391,998	3.47	6.97	107	5.41	134	831
1	40	82,438	2,543,946	4.05	8.05	71.9	3.80	126	550
1	60	56,510	1,676,279	4.64	8.72	58.7	3.08	120	428
1	80	41,350	1,204,084	5.23	9.18	53.1	2.73	116	369
1	100	31,596	911,653	5.82	9.43	51.7	2.52	112	333
1	120	24,922	718,089	6.42	9.67	52.0	2.47	110	**323**
3	∞	38,139	1,767,148	9.47	25.18	167	4.12	69	451
3	0	291,143	26,811,673	6.44	7.50	1740	24.7	85	2101
3	10	130,503	11,221,779	7.09	10.80	728	11.7	79	1652
3	20	84,577	6,142,749	7.73	12.13	414	7.64	76	994
3	40	41,286	2,234,392	9.03	13.61	195	4.47	70	507
3	60	23,918	1,077,860	10.32	14.25	144	3.69	66	387
3	80	15,901	645,155	11.62	14.53	136	3.57	63	**360**
3	100	11,523	450,356	12.92	14.72	143	3.72	61	369
3	120	8695	335,906	14.24	14.82	156	3.89	59	385
5	∞	35,905	2,247,942	11.93	29.56	304	5.15	56	592
5	0	274,852	34,966,449	8.65	9.60	+	+	+	+
5	10	100,372	12,849,540	9.52	13.55	1380	13.6	65	2264
5	20	62,317	6,416,944	10.39	14.89	710	8.34	61	1218
5	40	28,548	1,908,658	12.12	16.20	293	4.84	54	554
5	60	16,299	831,195	13.86	16.76	225	4.29	50	439
5	80	10,744	494,044	15.62	17.06	222	4.39	48	**432**
5	100	7559	330,553	17.39	17.16	240	4.63	47	457
5	120	5583	234,931	19.18	17.27	269	4.98	46	498

compute the amalgamation is negligible in comparison to the numerical incomplete factorization time. As expected, the number of iterations decreases with the amalgamation fill parameter: this indicates that some of the extra-fill terms allowed by the amalgamation corresponds to numerical non-zeros in the factors; this is to say that they correspond to terms that have a fill greater than the level-of-fill k parameter selected.

Additional graphical representations are provided at Figs. 6–9 for the AUDI problem. We give accordingly to the fill-in ratio:

- the number of iterations,
- the time of sequential incomplete factorization in seconds,
- the time of sequential triangular solve in seconds,
- the total sequential time in seconds,

for both scalar (with level-of-fill $k = 1, 2, \ldots, 7$) and block implementations (with level-of-fill $k = 1, 2, 3$). For the block implementation, at each level-of-fill value, the amalgamation ratio varies between 10% and 120% and for the scalar implementation the level-of-fill k varies between 1 and 7. We also add on these graphics the values obtained by our automatic criteria (large dots) based on Algorithm 5 ($\alpha = \infty$).

The scalar implementation has better total time, for each level-of-fill value, when the amalgamation ratio α is set to 0% or 10%, but is not competitive for higher values. This is also verified for the incomplete factorization time. We can see the real improvement provided by the amalgamation: for instance, when $k = 5$, by allowing some extra fill-in, the time can be divided by almost 4.

A great difference is observed in the incomplete factorization between the scalar implementation and the blockwise implementation. The BLAS-3 subroutines offer a great improvement over the scalar implementation

Table 3
Effect of amalgamation ratio α for INLINE problem

k	α	# Supernodes	# Blocks	Fill-in	Amalg.	Inc. Fact.	Triang. Solve	Iterations	Total
1	∞	20,197	317,596	4.03	6.28	11.2	0.95	–	–
1	0	151,981	4,286,156	2.39	1.82	75.9	4.38	–	–
1	10	85,642	1,999,163	2.63	2.66	34.3	2.42	–	–
1	20	60,585	1,288,038	2.87	3.09	23.3	1.79	–	–
1	40	34,349	639,754	3.35	3.56	14.7	1.21	–	–
1	60	21,701	371,140	3.85	3.85	11.9	0.98	–	–
1	80	14,894	239,567	4.34	4.00	11.3	0.89	–	–
1	100	10,924	168,783	4.85	4.10	11.6	0.87	–	–
1	120	8436	126,540	5.36	4.15	12.5	0.87	–	–
3	∞	16,175	288,642	5.74	8.02	18.0	1.09	159	191
3	0	139,072	7,349,868	4.31	2.64	274	6.93	–	–
3	10	51,923	1,861,082	4.74	4.04	66.7	2.41	–	–
3	20	28,211	727,479	5.18	4.52	30.1	1.44	179	287
3	40	11,702	194,858	6.05	4.93	17.3	1.03	151	**172**
3	60	7092	104,856	6.95	5.05	18.1	1.02	154	175
3	80	4950	69,801	7.88	5.09	21.0	1.08	160	193
3	100	3700	49,851	8.82	5.12	24.9	1.15	150	197
3	120	2905	37,821	9.79	5.14	29.4	1.23	137	197
5	∞	15,648	312,182	6.37	8.21	22.5	1.17	145	192
5	0	1,25,576	8,062,805	5.17	3.05	407	7.50	185	1794
5	10	34,220	1,327,668	5.69	4.57	65.5	1.99	161	385
5	20	17,070	396,987	6.21	4.99	26.8	1.23	163	227
5	40	7816	120,491	7.27	5.23	20.5	1.07	163	194
5	60	4956	70,480	8.39	5.30	23.7	1.13	148	**190**
5	80	3489	46,752	9.52	5.35	28.5	1.21	143	201
5	100	2640	33,830	10.73	5.35	34.3	1.33	123	197
5	120	2108	25,907	11.92	5.38	40.5	1.47	111	203

especially for the higher level-of-fill values that provide the bigger dense blocks and the number of floating point operations in the factorization. For the triangular solves, the results are less favorable since an amalgamation ratio greater than 40% is needed to improve the time of the scalar implementation. This is certainly due to the fact that the size of the blocks must be sufficient for BLAS-2 efficiency.

Our automatic criteria (based on time optimization) for amalgamation are a good compromise between minimizing the fill-in and optimizing the total time. For a given value of k, the variation between the time achieved by this automatic method and the best observed time for $0\% \leqslant \alpha \leqslant 120\%$ is always lower than 25%. We will use these automatic criteria for the next experiments concerning the parallel implementation.

Another interesting remark is that for small values of α ($\leqslant$40%), the number of iterations for the blockwise implementation follows the curve of the scalar implementation. But, for higher values, one should prefer to increase the level-of-fill value to improve the convergence with a same fill-in ratio.

5.2. Parallel experiments

Table 5 shows the results for the three problems both in sequential and in parallel for different levels-of-fill. The amalgamation parameter is now set to ∞: this means that the amalgamation is performed while the time of the solve decreases accordingly to the BLAS time model.

As we can see, the results are quite good since the speed-up is about 10 in most cases on 16 processors, more particularly for higher values of level-of-fill. It achieves a rather good scalability for both incomplete factorization and solve steps. This is particularly good considering the small amount of floating point operations required in the triangular solves.We show the results only until 16 processors because these test cases do

Table 4
Effect of amalgamation ratio α for MHD problem

k	α	# Supernodes	# Blocks	Fill-in	Amalg.	Inc. Fact.	Triang. Solve	Iterations	Total
1	∞	40,215	528,640	4.04	4.36	12.3	1.05	153	172
1	0	132,615	1,585,901	1.77	1.51	16.0	2.04	172	366
1	10	103,119	1,199,872	1.96	1.83	14.3	1.67	164	288
1	20	91,975	1,086,335	2.16	1.92	13.7	1.55	164	267
1	40	71,767	903,500	2.57	2.15	12.7	1.36	162	233
1	60	56,402	753,112	2.98	2.33	12.1	1.22	158	204
1	80	43,912	651,356	3.41	2.48	12.2	1.12	156	186
1	100	33,590	561,447	3.81	2.62	12.4	1.05	153	173
1	120	26,375	479,080	4.19	2.76	12.8	0.97	149	**157**
3	∞	50,361	803,279	6.34	6.32	37.4	1.44	88	164
3	0	132,485	3,202,800	3.69	2.20	85.6	3.52	100	437
3	10	93,524	2,296,760	4.10	2.67	66.7	2.66	98	327
3	20	76,199	1,862,555	4.51	2.93	57.8	2.30	97	280
3	40	53,717	1,390,499	5.34	3.26	50.1	1.91	94	229
3	60	38,617	1,091,834	6.15	3.53	46.5	1.67	92	200
3	80	27,862	853,927	6.96	3.75	44.6	1.49	89	177
3	100	20,806	658,369	7.74	3.92	43.8	1.35	86	159
3	120	16,239	521,220	8.57	4.05	44.2	1.27	83	**149**
5	∞	47,646	932,096	8.66	7.84	73.6	1.71	67	188
5	0	131,806	4,633,544	5.42	2.76	217	4.81	79	596
5	10	83467	3,215,398	6.03	3.42	164	3.56	78	441
5	20	64,718	2,692,706	6.65	3.69	145	3.12	77	385
5	40	41,257	1,811,205	7.82	4.13	115	2.38	73	288
5	60	27,553	1,181,086	8.97	4.45	94.3	1.91	69	226
5	80	19,373	833,068	10.15	4.68	85.5	1.65	66	194
5	100	14,174	608,861	11.35	4.84	80.4	1.53	64	178
5	120	10,875	455,535	12.51	4.96	79.6	1.48	61	**169**

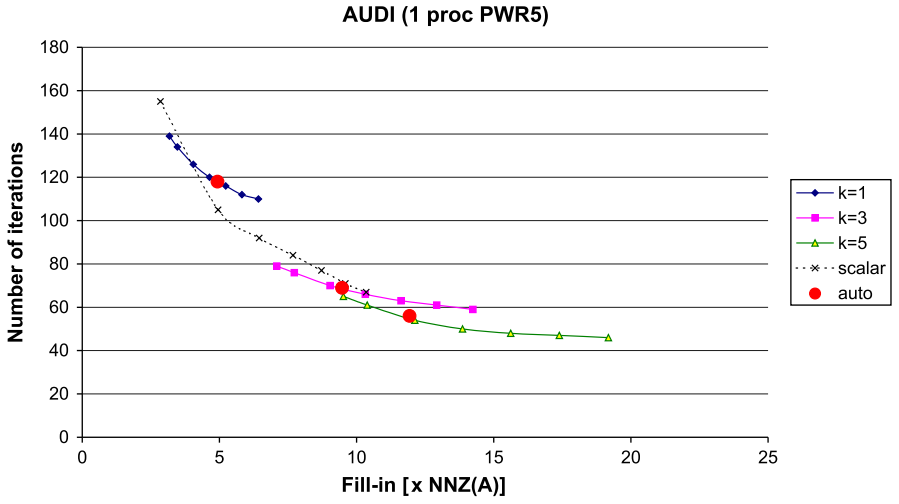

Fig. 6. Number of iterations for AUDI problem.

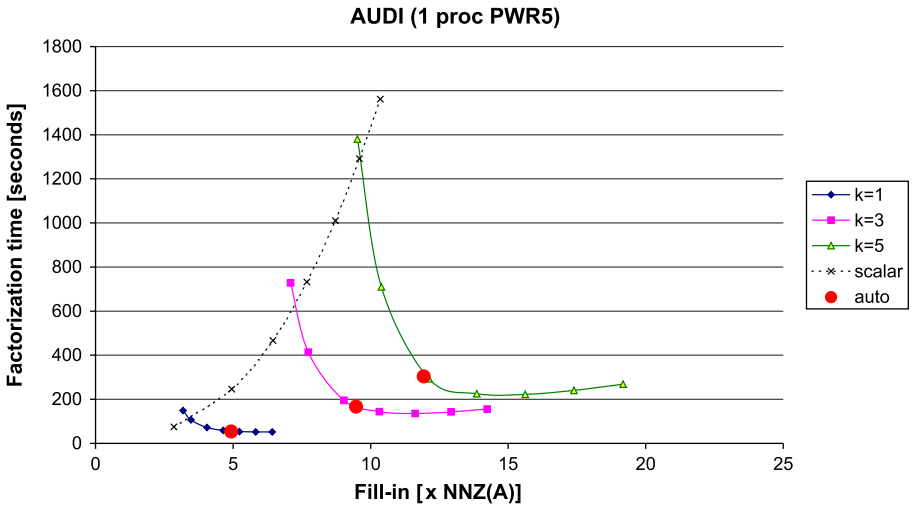

Fig. 7. Time of sequential incomplete factorization for AUDI problem.

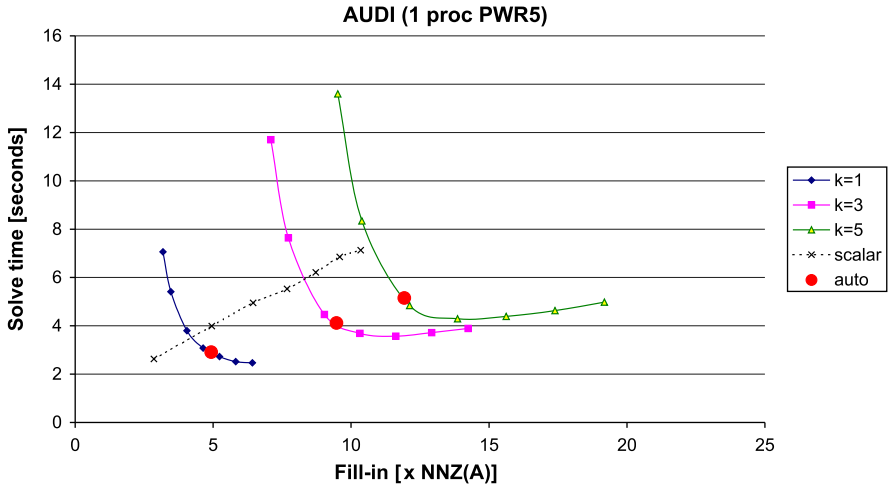

Fig. 8. Time of sequential triangular solve for AUDI problem.

not provide enough computations in our iterative solver to be parallelized efficiently on 32 processors (almost no gain in time).

For a comparison, we present in Table 6 the fill-in and the time needed to solve the three problems with our direct solver PaStiX on 16 processors. Let us remind that the precision of the solution with a direct method is about 10^{-15} whereas the precision is only set to 10^{-7} for our incomplete factorization.

As expected, our preconditioned iterative solver allows to significantly reduce the memory needs particularly for the MHD and AUDI test cases (compare with Tables 2–4). The MHD and AUDI test cases correspond to 3D problem discretizations (fill-in ratio of the factors is important in this case); as expected, the total time to solve the system is much higher in the direct solver than in the iterative solver. On the contrary, for the INLINE problem that corresponds to a 2D problem, the total memory needed for the direct solver amounts to about 2 or 3 times more than that needed for our iterative solver but the direct solver is about 4 or 5 times faster (with a significantly better precision).

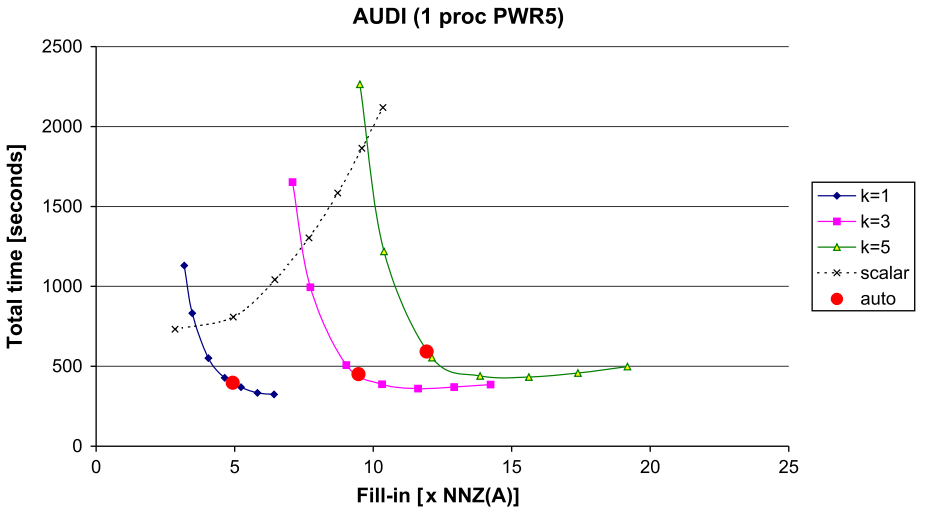

Fig. 9. Total sequential time for AUDI problem.

Table 5
Performances on 1, 4, 8 and 16 processors PWR5 for three test cases

k	1 Processor			4 Processors		
	Inc. Fact.	Triang. Solve	Total	Inc. Fact.	Triang. Solve	Total
AUDI						
1	53.7	2.91	397	14.2	0.84	113
3	167	4.12	451	43.1	1.21	126
5	304	5.15	592	78.2	1.55	165
	8 Processor			16 Processors		
1	7.56	0.51	68.4	6.34	0.39	52.2
3	22.4	0.74	73.8	12.3	0.52	48.4
5	40.8	0.91	91.7	22.1	0.76	64.9
INLINE						
	1 Processor			4 Processors		
1	11.2	0.95	–	3.18	0.29	–
3	18.0	1.09	191	4.83	0.34	59.2
5	22.5	1.17	192	6.18	0.37	61.6
	8 Processor			16 Processors		
1	1.63	0.15	–	1.44	0.11	–
3	2.45	0.18	32.3	1.52	0.11	20.1
5	3.14	0.21	33.7	1.82	0.12	19.5
MHD						
	1 Processor			4 Processors		
1	12.3	1.05	172	3.25	0.29	48.2
3	37.4	1.44	164	9.83	0.41	46.5
5	73.6	1.71	188	19.6	0.50	53.3
	8 Processor			16 Processors		
1	1.94	0.18	29.6	2.08	0.17	27.9
3	5.25	0.27	28.9	4.17	0.25	26.1
5	10.2	0.32	31.8	6.56	0.29	26.2

Table 6
Direct factorization on 16 processors

Name	Columns	nnz_A	Fill-in	Num. Fact.	Triang. Solve
AUDI	943,695	39,297,771	30.1	91.4	1.21
INLINE	503,712	18,660,027	9.03	4.02	1.19
MHD	485,597	24,233,141	45.7	139	0.56

6. Conclusions

The main aims of this work have been reached. The blockwise adaptation of the ILU(k) factorization presented in this work allows to significantly reduce the time to solve linear systems compared to the classical columnwise algorithm. It also benefits from the parallelization techniques developed for direct solvers (in our case PaStiX). We think that this approach, could be adapted to other direct solvers (in particular, other supernodal solvers) and thus provides a generic way to build efficient and parallel iterative solvers.

References

[1] P.R. Amestoy, A. Guermouche, J.-Y. L'Excellent, S. Pralet, Hybrid scheduling for the parallel solution of linear systems, Parallel Computing 32 (2) (2006) 136–156.
[2] G. Karypis, V. Kumar, A fast and high-quality multi-level scheme for partitioning irregular graphs, SIAM Journal on Scientific Computing 20 (1998) 359–392.
[3] I.S. Duff, A.M. Erisman, J.K. Reid, Direct Methods for Sparse Matrices, Clarendon Press, Oxford, 1986.
[4] J.A. George, J.W.-H. Liu, Computer Solution of Large Sparse Positive Definite Systems, Prentice-Hall, Englewood Cliffs, NJ, 1981.
[5] P.R. Amestoy, T.A. Davis, I.S. Duff, An approximate minimum degree ordering algorithm, SIAM Journal on Matrix Analysis and Applications 17 (1996) 886–905.
[6] P.R. Amestoy, I.S. Duff, S. Pralet, C. Vömel, Adapting a parallel sparse direct solver to architectures with clusters of SMPs, Parallel Computing 29 (11-12) (2003) 1645–1668.
[7] Y. Saad, Iterative Methods for Sparse Linear Systems, second ed., SIAM, Philadelphia, PA, 2003.
[8] A. Chapman, Y. Saad, Deflated and augmented Krylov subspace techniques, Numerical Linear Algebra with Applications 4 (1997) 43–66.
[9] Y. Campbell, T.A. Davis, Incomplete LU factorization: a multifrontal approach. <http://www.cise.ufl.edu/d~avis/techreports.html>.
[10] T.F. Chang, P.S. Vassilevski, A framework for block ILU factorizations using block-size reduction, Mathematics of Computation 64 (1995).
[11] P. Charrier, J. Roman, Algorithmique et calculs de complexité pour un solveur de type dissections emboî tées, Numerische Mathematik 55 (1989) 463–476.
[12] Anshul Gupta, Recent progress in general sparse direct solvers, Lecture Notes in Computer Science 2073 (2001) 823–840.
[13] P. Hénon, P. Ramet, J. Roman, PaStiX: a high-performance parallel direct solver for sparse symmetric definite systems, Parallel Computing 28 (2) (2002) 301–321.
[14] P. Hénon, P. Ramet, J. Roman, Efficient algorithms for direct resolution of large sparse system on clusters of SMP nodes, in: SIAM Conference on Applied Linear Algebra, Williamsburg, Virginie, USA, July 2003.
[15] G. Karypis, V. Kumar, Parallel threshold-based ILU factorization, in: Proceedings of the IEEE/ACM SC97 Conference, 1997.
[16] 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, Texas, March 22–24, 1999.
[17] J.W.H. Liu, The role of elimination trees in sparse factorization, SIAM Journal on Matrix Analysis and Applications 11 (1990) 134–172.
[18] D. Hysom, A. Pothen, A scalable parallel algorithm for incomplete factor preconditioning, SIAM Journal on Scientific Computing 22 (6) (2001) 2194–2215.
[19] D.J. Rose, R.E. Tarjan, Algorithmic aspects of vertex elimination on directed graphs, SIAM Journal on Applied Mathematics 34 (1978) 176–197.
[20] Joseph W.H. Liu, Esmond G. Ng, Barry W. Peyton, On finding supernodes for sparse matrix computations, SIAM Journal on Matrix Analysis and Applications 14 (1) (1993) 242–252.
[21] M. Magolu monga Made, A. Van der Vorst, A generalized domain decomposition paradigm for parallel incomplete LU factorization preconditionings, Future Generation Computer Systems 17 (8) (2001) 925–932.
[22] F. Pellegrini, Scotch 4.0 User's guide, Technical Report, INRIA Futurs, April 2005. Available from: <http://www.labri.fr/p~elegrin/papers/scotch_user4.0.ps.gz>.
[23] B. Hendrickson, R. Leland, An improved spectral graph partitioning algorithm for mapping parallel computations, Technical Report SAND92-1460, UC-405, Sandia National Laboratories, Albuquerque, NM, 1992.

[24] F. Pellegrini, J. Roman, P. Amestoy, Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering, Concurrency: Practice and Experience 12 (2000) 69–84.
[25] P. Raghavan, K. Teranishi, E.G. Ng, A latency tolerant hybrid sparse solver using incomplete Cholesky factorization, Numerical Linear Algebra (2003).
[26] Y. Saad, ILUT: a dual threshold incomplete ILU factorization, Numerical Linear Algebra with Applications 1 (1994) 387–402.
[27] Y. Saad, Iterative Methods for Sparse Linear Systems, second ed., SIAM, 2003.
[28] J.W. Watts Jr., A conjugate gradient truncated direct method for the iterative solution of the reservoir simulation pressure equation, Society of Petroleum Engineers Journal 21 (1981) 345–353.