Applying parallel direct solver techniques to build robust high performance preconditioners

Pascal Hénon[1], François Pellegrini[1], Pierre Ramet[1], Jean Roman[1] and Yousef Saad[2]

[1] ScAlApplix Project, INRIA Futurs and LaBRI UMR 5800
Université Bordeaux 1, 33405 Talence Cedex, France
{henon, pelegrin, ramet, roman}@labri.fr
[2] Dept of computer Science & Eng.
University of Minnesota, 200 Union st. SE.
Minneapolis, MN 55155, USA
saad@cs.umn.edu

Abstract. The purpose of our work is to provide a method which exploits the parallel blockwise algorithmic approach used in the framework of high performance sparse direct solvers in order to develop robust preconditioners based on a parallel incomplete factorization. The idea is then 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.

1 Introduction

Solving large sparse linear systems by iterative methods [18] has often been unsatisfactory when dealing with pratical "industrial" problems. The main difficulty encountered by such methods is their lack of robustness and, generally, the unpredictability and unconsistency of their performance when they are used over a wide range of different problems; some methods work quite well for certain types of problems but can fail completely on others.

Over the past few years, direct methods have made significant progress thanks to research studies on both the combinatorial analysis of Gaussian elimination process and on the design of parallel block solvers optimized for high-performance computers. It is now possible to solve real-life three-dimensional problems having in the order of several millions equations, in a very effective way with direct solvers. These is achievable by exploiting superscalar effects of modern processors and taking advantage of computer architectures based on networks of SMP nodes (IBM SP, DEC-Compaq, for example) [1, 7, 9, 10, 12]. However, direct methods will still fail to solve very large three-dimensional problems, due to the large amount of memory needed for these cases.

Some improvments 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 [14]. In [16], 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 [11] or with a fill level dropping that measures the importance of an entry in terme of its updates [2]. In [3], the authors proposed a block ILU factorization technique for block tridiagonal matrices.

Our goal is to provide a method which exploits the parallel blockwise algorithmic approach used in the framework of high performance sparse direct solvers in order to develop robust parallel incomplete factorization based preconditioners [18] for iterative solvers.

The originality of our work is to use a supernodal algorithm what allows us to drop some blocks during the elimination process on the quotient graph. Unlike multifrontal approaches for which parallel threshold-based ILU factorization has been studied, the supernodal method permits to build at low cost a dense block structure for an ILU(k) factorization with an important fill-in. Indeed, using dense block formulation is crutial to achieved high performance computations.

The idea is then 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. Such incomplete factorizations can take advantage of the latest breakthroughts in sparse direct methods and can therefore be very competitive in terms of CPU time due to the effective usage of CPU power. At the same time this approach does not suffer from the memory limitation encountered by direct methods. Therefore, we can expect to be able to solve systems in the order of hundred millions of unknowns on current platforms. Another goal of this paper is to analyze the chosen parameters that can be used to define the block sparse pattern in our incomplete factorization.

The remainder of the paper is organized as follows: the section 2 describes the main features of our method. We provide some experiments in section 3. At last we give some conclusions in section 4.

2 Methodology

The driving rationale for this study is that it is easier to incorporate incomplete factorization methods into direct solution software than it is to develop new incomplete factorizations. As a starting point, we can take advantage of the algorithms and the software components of PaStiX [10] which is a parallel high performance supernodal direct solver for sparse symmetric positive definite systems and for sparse unsymmetric systems with a symmetric pattern. These different components are (see Fig. 1):

1. the ordering phase, which computes a symmetric permutation of the initial matrix such that factorization process will exhibit as much concurrency as possible while incurring low fill-in. In this work, we use a tight coupling of the Nested Dissection and Approximate Minimum Degree algorithms [15];

2. the block symbolic factorization phase, which determines the block data structure of the factorized matrix associated with the partition resulting from the ordering phase. This structure consists of several column-blocks, each of them containing a dense diagonal block and a set of dense rectangular off-diagonal blocks [5];

3. the block repartitioning and scheduling phase, which refines the previous partition by splitting large supernodes in order to exploit concurrency within dense block computations in addition to the parallelism provided by the block elimination tree, both induced by the block computations in the supernodal solver. In this phase, we compute a mapping of the matrix blocks (that can be made by column block (1D) or block (2D)) and a static optimized scheduling of the computational and communication tasks according to BLAS and communication time models calibrated on the target machine. This static scheduling will drive the parallel factorization and the backward and forward substitutions [8, 10].

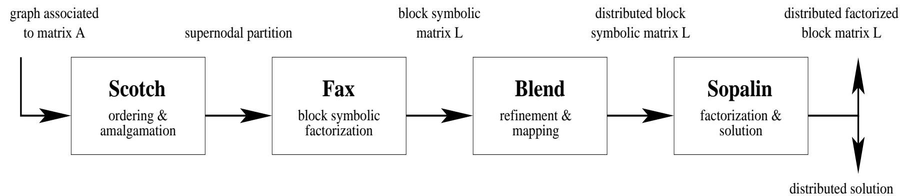

Fig. 1. Phases for the parallel complete block factorization.

Our approach consists of *computing symbolically the block structure of the factors that would have been obtained with a complete factorization, and then deciding to drop off some blocks of this structure according to relevant criteria.* This incomplete factorization induced by the new sparse pattern is then used in a preconditioned GMRES or Conjugate Gradient solver [18]. Our main goal at this point is to achieve a significant reduction of the memory needed to store the incomplete factors while keeping enough fill-in to make the use of BLAS3 primitives cost-effective. Naturally, we must still have an ordering phase and a mapping and scheduling phase (this phase is modified to suit the incomplete factorization block computation) to ensure an efficient parallel implementation of the block preconditioner computation and of the forward and backward substitutions in the iterations. Then, we have the new processing chain given at Fig. 2.

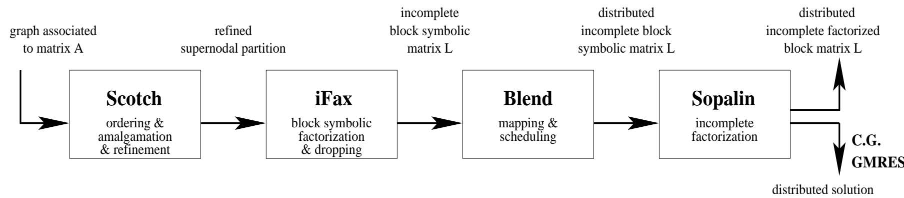

Fig. 2. Phases for the parallel incomplete block factorization.

The first crucial point is then to find a good initial partition that can be used in the dropping step after the block symbolic factorization. It cannot be the initial supernodal partition computed by the ordering phase (phase 1) because it would be too costly to consider the diagonal blocks as dense blocks like in a complete factorization. Therefore, we resort to a refined partition of this supernodal partition which will then define the elementary column blocks of the factors. We obtain the refined partition by splitting the column blocks issued from the supernodal partition according to a maximum blocksize parameter. This allows more options for dropping some blocks in the preconditioner. This blocksize parameter plays a key role in finding a good trade-off as described above. An important result from theoretical analysis is that if we consider nested dissection ordering based on separator theorems [13], and if we introduce some asymptotically refined partitions, one can show that the total number of blocks computed by the block symbolic factorization and the time to compute these blocks are quasi-linear (these quantities are linear for the supernodal partition).

The second crucial point concerns the various criteria that are used to drop some blocks from the blockwise symbolic structure induced by the refined partition. The dropping criterion we use is based on a generalization of the level-of-fill [18] metric that has been adapted to the elimination process on the quotient graph induced by the refined partition.

One of the most common ways to define a preconditioning matrix M is through Incomplete LU (ILU) factorizations. ILU factorizations are obtained from an approximate Gaussian elimination. When Gaussian elimination is applied to a sparse matrix A, a large number of nonzero elements may appear in locations originally occupied by zero elements. These fill-ins are often small elements and may be dropped to obtain approximate LU factorizations.

The simplest of these procedures, ILU(0) is obtained by performing the standard LU factorization of A and dropping all fill-in elements that are generated during the process. In other words, the L and U factors have the same pattern as the lower and upper triangular parts of A (respectively). More accurate factorizations denoted by ILU(k) and IC (k) have been defined which drop fill-ins according to their “levels”. Level-1 fill-ins for example are generated from level-zero fill-ins (at most). So, for example, ILU(1) consists of keeping all fill-ins that

have level zero or one and dropping any fill-in whose level is higher. We now provide a few details.

In level-based ILUs, originally introduced by Watts III [19], a *level-of-fill* is associated with every entry in the working factors L and U. Conceptually these factors together are the L and U parts of a certain working matrix A which initially contains the original matrix. At the start of the procedure, every zero entry has a level-of-fill of infinity and every nonzero entry has a level-of-fill of zero. Whenever an entry is modified by the standard Gaussian Elimination update

$$a_{ij} := a_{ij} - a_{ik} * a_{kj}/a_{kk}$$

its level-of-fill is updated by the formula

$$lev_{ij} = \min\{lev_{ij}, lev_{ik} + lev_{kj} + 1\}.$$

In practice, these levels are computed in a symbolic phase first and used to define the patterns of L and U a priori. As the level-of-fill increases, the factorization becomes more expensive but more accurate. In general, the robustness of the iterative solver will typically improve as the level-of-fill increases. It is common practice in standard preconditioned Krylov methods to use a very low level-of-fill , typically no greater than 2. Now it is easy to see what will be the main limitation of this approach: the level-of-fill is based entirely on the adjacency graph of the original matrix, and its definition implicitly assumes some form of diagonal dominance, in order for the dropped entries to be indeed small. It cannot work for matrices that are highly indefinite for example. There are two typical remedies, each with its own limitations. The first is to replace the level-of-fill strategy by one that is based on numerical values. This yields the class of ILUT algorithms [17, 18]. Another is to resort to a block algorithm, i.e., one that is based on a block form of Gaussian elimination. Block ILU methods have worked quite well for harder problems, see for example [4, 6].

In the standard block-ILU methods, the blocking of the matrix simply represents a grouping of sets of unknowns into one single unit. The simplest situation is when A has a natural block structure inherited from the blocking of unknowns by degrees of freedom at each mesh-point. Extending block ILU to these standard cases is fairly easy. It suffices to consider the quotient graph obtained from the blocking.

The situation with which we must deal is more complex because the block partitioning of rows varies with the block columns. An illustration is shown in Figure 3. The main difference between the scalar and the block formulation of the level-of-fill algorithm is that, in general, a block contribution may update only a part of the block (see figure 3(a)). As a consequence, a block is split according to its level-of-fill modification. Nevertheless, we only allow the splitting along the row dimension in order to preserve an acceptable blocksize (see figure 3(b)).

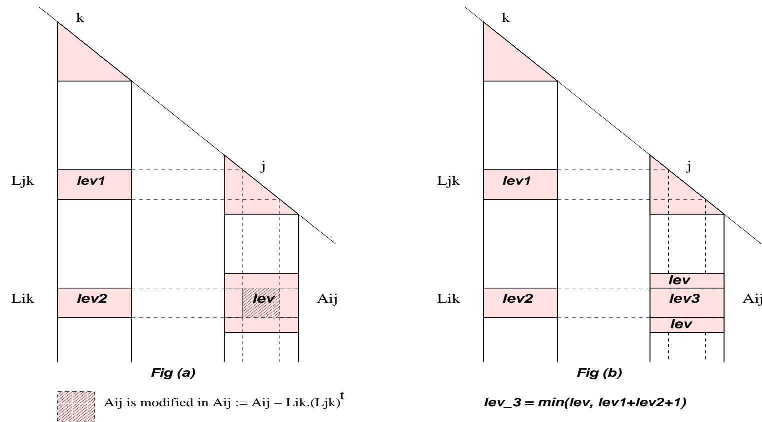

Fig. 3. Computation of the block level of fill-in the block elimination process.

As shown in table 1, two consecutive levels-of-fill can produce a large increase of the fill ratio in the factors (NNZ_A is the number of non-zeros in A, NNZ_L is the number of non-zeros that would have beeen obtained with a direct factorization, and OPC_L is the number of operations required for the direct factorization).

Table 1. Fill rate for a 47x47x47 3D mesh (finite element, 27 connectivity)

level of fill	% NNZ_L	$\times NNZ_A$	% OPC_L
≤ 0	9.28	3.53	0.29
≤ 1	23.7	9.02	2.33
≤ 2	38.4	14.6	7.61
≤ 3	54.9	20.9	19.0
≤ 4	66.1	25.2	31.5
≤ 5	75.4	28.7	45.1
≤ 6	83.2	31.6	58.5
...	...	...	...
≤ 15	100	38.1	100

In order to choose intermediate ratios of fill between two levels, we have introduced a second criterion to drop some blocks inside a level-of-fill. We allow the possibility to choose a fill ratio according to the formula:

$$NNZ_{\text{prec}} = (1 - \alpha).NNZ_A + \alpha.NNZ_L$$

where $\alpha \in [0, 1]$ and NNZ_{prec} is the number of non-zeros in the factors for the incomplete factorization. To reach the exact number of non-zeros corresponding to a selected α, we consider the blocks allowed in the fill-in pattern in two steps. If we denote by NNZ_k the number of non-zeros obtained by keeping

all the blocks with levels-of-fill $\leq k$, then we find the first value λ such that $NNZ_\lambda \geq NNZ_{\text{prec}}$. In a second step, until we reach NNZ_{prec}, we drop the blocks, among those having a level-of-fill λ, which undergo the fewest updates by previous eliminations.

3 Tests

In this section, we give an analysis of some first convergence and scalability results for practical large systems. We consider two difficult problems for direct solvers (see table 2). The AUDI matrix (symmetric) corresponds to an automotive crankshaft model and the MHD1 is a magnetohydrodynamic problem (unsymmetric). The ratio, between the number of non-zeros in the complete factor and the number of non-zeros in the initial matrix A is about 31 for the AUDI test case and about 67 for the MHD1 one.

Table 2. Description of our test problems.

Name	Columns	NNZ_A	NNZ_L	OPC_L
AUDI	943695	39297771	1.21e+09	5.3e+12
MHD1	485597	24233141	1.62e+09	1.6e+13

Numerical experiments were run on a 28 NH2 IBM nodes (16 Power3+, 375Mhz, 1.5 Gflops, 16GB) located at CINES (Montpellier, France) with a network based on a Colony switch. All computations are performed in double precision and all time results are given in seconds. We use the PaStiX software with recent improvements such as an efficient MPI/Thread implementation to compute the preconditioner. The stopping criterion for GMRES iterations uses the relative residual norm and is set to 10^{-7}.

Table 3 presents results for different values of the fill rate parameter α for the AUDI problem. The blocksize (for the refined partition) is set to 8 and the results are performed on 16 processors.

Table 3. AUDI problem with blocksize=8

$\alpha = 0.1$			$\alpha = 0.2$			$\alpha = 0.3$		
inc.fact.	*nb.iter.*	*time/iter.*	*inc.fact.*	*nb.iter.*	*time/iter.*	*inc.fact.*	*nb.iter.*	*time/iter.*
24	429	0.71	39	293	1.30	55	279	1.64
$\times NNZ_A$	$\%\ OPC_L$	*tot.time*	$\times NNZ_A$	$\%\ OPC_L$	*tot.time*	$\times NNZ_A$	$\%\ OPC_L$	*tot.time*
3.77	0.51	**328.6**	6.75	2.91	**419.9**	9.75	5.97	**512.5**

$\alpha = 0.4$			$\alpha = 0.5$			$\alpha = 0.6$		
inc.fact.	*nb.iter.*	*time/iter.*	*inc.fact.*	*nb.iter.*	*time/iter.*	*inc.fact.*	*nb.iter.*	*time/iter.*
80	144	2.12	135	49	3.13	195	36	3.70
$\times NNZ_A$	$\%\ OPC_L$	*tot.time*	$\times NNZ_A$	$\%\ OPC_L$	*tot.time*	$\times NNZ_A$	$\%\ OPC_L$	*tot.time*
12.84	8.32	**385.3**	15.92	15.63	**288.4**	18.99	24.10	**328.2**

For each run we give:

- in the first line, the time to compute the incomplete factorisation (*inc.fact.*), the number of iterations (*nb.iter.*) and the time to perform one iteration (*time/iter.*);
- in the second line, the ratio between the number of non-zeros in the incomplete factors and the number of non-zeros in the matrix A ($\times NNZ_A$), the percentage of the number of operations to compute the incomplete factorization compared with the number of operations required for the direct factorization ($\%\ OPC_L$), and the total time (*tot.time*).

The same results for a blocksize set to 16 can be found in table 4. Theses results can be compared with the time required by the direct solver: on 16 processors, PaStiX needs about 482s to solve the problem and the solution has an accuracy (relative residual norm) about 10^{-15}.

Table 4. AUDI problem with blocksize=16

$\alpha = 0.1$			$\alpha = 0.2$			$\alpha = 0.3$		
inc.fact.	*nb.iter.*	*time/iter.*	*inc.fact.*	*nb.iter.*	*time/iter.*	*inc.fact.*	*nb.iter.*	*time/iter.*
11	214	0.84	19	196	1.20	40	177	1.87
$\times NNZ_A$	$\%\ OPC_L$	*tot.time*	$\times NNZ_A$	$\%\ OPC_L$	*tot.time*	$\times NNZ_A$	$\%\ OPC_L$	*tot.time*
5.29	0.97	**190.8**	6.50	2.16	**254.2**	9.57	6.82	**371.0**

$\alpha = 0.4$			$\alpha = 0.5$			$\alpha = 0.6$		
inc.fact.	*nb.iter.*	*time/iter.*	*inc.fact.*	*nb.iter.*	*time/iter.*	*inc.fact.*	*nb.iter.*	*time/iter.*
62	186	2.13	77	140	2.41	130	42	3.45
$\times NNZ_A$	$\%\ OPC_L$	*tot.time*	$\times NNZ_A$	$\%\ OPC_L$	*tot.time*	$\times NNZ_A$	$\%\ OPC_L$	*tot.time*
12.76	11.93	**458.2**	15.55	15.57	**414.4**	19.01	24.39	**274.9**

As expected, the time for the incomplete factorization and for the iterations increases with the fill rate parameter whereas the number of iterations decreases. We can see that the best result is obtained with α set to 0.1 and a blocksize set to 16. Thus, we can solve the problem 2.5 faster than with our direct solver and with only 17.2% of NNZ_L (about $5.3 \times NNZ_A$). We have also report results with higher values for the blocksize (32): block computations are more efficient but these blocksizes do not allow to drop enough entries to be competitive.

For next results the blocksize is set to 16 and α to 0.1. With such a fill rate parameter, the number of iterations for the MHD1 problem is small (5 iterations to reduce the residual norm by 10^{-7}). This problem is easier than the AUDI problem, and in that case, the advantage of our approach will be less important compared with traditional iterative solvers.

Table 5 shows that the run-time scalability is quite good for up to 64 processors for both the incomplete factorization and the iteration phase. We remind that the number of iterations is *independent* of the number of processors in our approach.

Table 5. Scalability results with $\alpha = 0.1$ and blocksize=16

Name	Number of processors						
	1	2	4	8	16	32	64
AUDI *inc.fact.*	90.9	52.5	29.2	15.7	10.6	5.9	3.3
AUDI *time/iter.*	10.5	5.56	3.06	1.49	0.84	0.45	0.33
MHD1 *inc.fact.*	48.1	26.2	15.6	9.1	5.1	3.0	2.2
MHD1 *time/iter.*	3.82	1.97	1.06	0.61	0.40	0.32	0.25

So on 64 processors, for a relative precision set to 10^{-7}, the total time is 74s what is twice faster than the 152s needed by the direct solver.

4 Conclusion

In conclusion, we have shown a methodology for bridging the gap between direct and iterative solvers by blending the best features of both types of techniques: low memory costs from iterative solvers and effective reordering and blocking from direct methods. The approach taken is aimed at producing robust parallel iterative solvers that can handle very large 3-D problems which arise from realistic applications. The preliminary numerical examples shown indicate that the algorithm performs quite well for such problems. Robustness relative to standard (non-block) preconditioners is achieved by extracting more accurate factorizations via higher amounts of fill-in. The effective use of hardware is enabled by a careful block-wise processing of the factorization, which yields fast factorization and preconditioning operations. We plan on performing a wide range of experiments on a variety of problems, in order to validate our approach and to understand its limitations. We also plan on studying better criteria for dropping certain blocks from the blockwise symbolic structure.

Acknowledgment. The work of the authors was supported from NSF grants INT-0003274. The work of the last author was also supported from NSF grants ACI-0305120 and by the Minnesota Supercomputing Institute.

References

1. P. R. Amestoy, I. S. Duff, S. Pralet, and C. Vömel. Adapting a parallel sparse direct solver to architectures with clusters of SMPs. *Parallel Computing*, 29(11-12):1645–1668, 2003.
2. Y. Campbell and T.A. Davis. Incomplete LU factorization: A multifrontal approach. *http://www.cise.ufl.edu/~davis/techreports.html*
3. T.F. Chang and P.S. Vassilevski. A framework for block ILU factorizations using block-size reduction. *Math. Comput.*, 64, 1995.
4. A. Chapman, Y. Saad, and L. Wigton. High-order ILU preconditioners for CFD problems. *Int. J. Numer. Meth. Fluids*, 33:767–788, 2000.
5. P. Charrier and J. Roman. Algorithmique et calculs de complexité pour un solveur de type dissections emboîtées. *Numerische Mathematik*, 55:463–476, 1989.

6. E. Chow and M. A. Heroux. An object-oriented framework for block preconditioning. Technical Report umsi-95-216, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN, 1995.
7. Anshul Gupta. Recent progress in general sparse direct solvers. *Lecture Notes in Computer Science*, 2073:823–840, 2001.
8. P. Hénon. *Distribution des Données et Régulation Statique des Calculs et des Communications pour la Résolution de Grands Systèmes Linéaires Creux par Méthode Directe*. PhD thesis, LaBRI, Université Bordeaux I, France, November 2001.
9. P. Hénon, P. Ramet, and J. Roman. PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. *Parallel Computing*, 28(2):301–321, January 2002.
10. P. Hénon, P. Ramet, and 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.
11. G. Karypis and V. Kumar. Parallel Threshold-based ILU Factorization. *Proceedings of the IEEE/ACM SC97 Conference*, 1997.
12. X. S. Li and 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.
13. R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM Journal of Numerical Analysis*, 16(2):346–358, April 1979.
14. M. Magolu monga Made and A. Van der Vorst. A generalized domain decomposition paradigm for parallel incomplete LU factorization preconditionings. *Future Generation Computer Systems*, Vol. 17(8):925–932, 2001.
15. F. Pellegrini, J. Roman, and P. Amestoy. Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering. *Concurrency: Practice and Experience*, 12:69–84, 2000.
16. P. Raghavan, K. Teranishi, and E.G. Ng. A latency tolerant hybrid sparse solver using incomplete Cholesky factorization. *Numer. Linear Algebra*, 2003.
17. Y. Saad. ILUT: a dual threshold incomplete ILU factorization. *Numerical Linear Algebra with Applications*, 1:387–402, 1994.
18. Y. Saad. *Iterative Methods for Sparse Linear Systems, Second Edition*. SIAM, 2003.
19. J. W. Watts III. A conjugate gradient truncated direct method for the iterative solution of the reservoir simulation pressure equation. *Society of Petroleum Engineers Journal*, 21:345–353, 1981.