# Partitioning and Blocking Issues for a Parallel Incomplete Factorization

Pascal Hénon, Pierre Ramet, and Jean Roman

ScAlApplix Project, INRIA Futurs and LaBRI UMR 5800
Université Bordeaux 1, 33405 Talence Cedex, France
{henon, ramet, roman}@labri.fr

**Abstract.** The purpose of this work is to provide a method which exploits the parallel block-wise algorithmic approach used in the framework of high performance sparse direct solvers in order to develop robust and efficient preconditioners based on a parallel incomplete factorization.

## 1 Introduction

Over the past few years, parallel sparse direct solver have made significant progress. They are now able to solve efficiently real-life three-dimensional problems having in the order of several millions of equations (see for example [1, 5, 6]).

Nevertheless, the need of a large amount of memory is often a bottleneck in these methods. On the other hand, the iterative methods using a generic preconditioner like an ILU(k) factorization [19] 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. Futhermore, these methods are difficult to parallelize efficiently, more particulary for high values of level-of-fill.

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 [12]. In [17], 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 [8] or with a fill level dropping that measures the importance of an entry in terms 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 block-wise algorithmic approach used in the framework of high performance sparse direct solvers in order to develop robust parallel incomplete factorization based preconditioners [19] for iterative solvers.

For direct methods, in order to achieve an efficient parallel factorization, solvers usually implement the following processing chain:

– the *ordering* phase, which computes a symmetric permutation of the initial matrix $A$ such that factorization 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];
– the *block symbolic factorization* phase, which determines the block data structure of the factored matrix $L$ associated with the partition resulting from the ordering phase;
– the *block repartitioning and scheduling* phase, which refines the partition, by splitting large supernodes in order to exploit concurrency within dense block computations, and maps it onto the processors;
– the *parallel numerical factorization* and the *forward/backward elimination* phases, which are driven by the distribution and the scheduling of the previous step.

In our case, we propose to extend our direct solver PaStiX [6] to compute an incomplete *block factorization* that can be used as a preconditioner in a krylov method. The main work will consist in replacing the *block symbolic factorization* step by some algorithms able to build a dense block structure in the incomplete factors. We keep the ordering computed by the direct factorization to exhibit parallelism. *Reverse Cuthill and McKee* techniques are known to be efficient for small values of level-of-fill (0 or 1), but, to obtain robust preconditioners, we have to considere higher values of level-of-fill. In addition, the *Reverse Cuthill and McKee* leads to an ordering that does permit independent computation in the factorization and thus it is not adapted for parallelization. The extensions that are described have also to preserve the dependences in the elimination tree on which relies all the direct solver algorithms.

## 2 Methodology

In the direct methods relying on a Cholesky factorization ($A = L.L^t$), the way to exhibit a dense block structure in the matrix $L$ is directly linked to the ordering techniques based on the nested dissection algorithm (ex: MeTiS [13] or Scotch [14]). Indeed the columns of $L$ can be grouped in sets such that all columns of a same set have a similar non zero pattern. Those 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 $G(A)$ of matrix $A$. Another essential property of this kind of ordering is that it provides a block elimination tree that is well suited for parallelism [6].

An important result used in direct factorization is that the partition $\mathcal{P}$ of the unknowns induced by the supernodes can be found without knowning the non zero pattern of $L$. The partition $\mathcal{P}$ of the unknowns is then used to compute the block structure of the factorized matrix $L$ during the so-called *block symbolic*

*factorization*. This block symbolic factorization for direct method is a very low time and memory consuming step since it can be done on the quotient graph $Q(G(A), \mathcal{P})$ with a complexity that is quasi-linear in respect to the number of edges in the quotient graph. We exploit the fact that:

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

where the exponent $*$ means "elimination graph". It is important to keep in mind that this property can be used to prune the block structure of the factor $L$ because one can find the supernode partition from $G(A)$ [11]. For an incomplete ILU(k) factorization, those properties are not true anymore in the general case. The incomplete symbolic ILU(k) factorization has a theorical complexity similar to the numerical factorization, but an efficient algorithm that leads to a practical implementation have been proposed [16]. The idea of this algorithm is to use searches of elimination paths of length $k+1$ in $G(A)$ in order to compute $G^k(A)$ which is the adjacency graph of the factor in ILU(k) factorization.

Another remark to reduce the cost of this step is that any set of unknowns in $A$ that have the same row structure and column structure in the lower triangular part of $A$ can be compressed as a single node in $G(A)$ in order to compute the symbolic ILU(k) factorization. Indeed the corresponding set of nodes in $G(A)$ will have the same set of neighbors and consequently the elimination paths of length $k+1$ will be the same for all the unknowns of such a set. In other words, if we consider the partition $\mathcal{P}_0$ constructed by grouping set of unknowns that have the same row and column pattern in $A$ then we have:

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

This optimization is often payfull for matrices that come from finite element discretization since a node in the mesh graph represents a set of several unknowns (the degrees of freedoms) that forms a clique. Then the ILU(k) symbolic factorization can be devised with a significant lower complexity than the numerical factorization algorithm.

Once the elimination graph $G^k$ is computed, the problem is to find a block structure of the incomplete factors. For direct factorization, the supernode partition usually produces some blocks that have a sufficient size to obtain a good superscalar effect using the BLAS 3 subroutines. The supenodes that are exhibited from the incomplete factor non zero pattern are usually very small. 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 principle of our heuristic to compute the new supernode partition is to iteratively merge supernode 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:

– find the partition $\mathcal{P}_0$ induced by the supernodes of $A$;

- compute the block symbolic incomplete factorization $Q(G(A, \mathcal{P}_0))^k$
- given a *extra fill-in* tolerance $\alpha$ , construct an aproximated supernode partition $\mathcal{P}_\alpha$ to improve the block structure of the factors.
- Apply a block incomplete factorization using the parallelization techniques implemented in our direct solver PaStiX [6, 7].

The incomplete factorization is then used in a GMRES to solve the system.

## 3 Amalgamation algorithm

The previous section shows that the symbolic factorization of ILU(k) method, though more costly than in the case of exact factorizations, is not a limitation in our approach. What remains critical is to obtain dense blocks with a sufficient size in the factor in order to take advantage of the superscalar effects provided by the BLAS subroutines. 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 supernodes that have almost similar non-zero pattern in order to get bigger supernodes. By construction, the exact supernode partition found in any ILU(k) factor is always a sub-partition of the *direct supernode partition* (i.e. corresponding to the direct factorization). We impose the amalgamation algorithm to merge only ILU(k) supernodes that belong to the same direct supernode. That is to say that we want this *approximated supernode partition* to remain a sub-partition of the direct supernode partition. The rational is that when this rule is respected, the additional fill entries induced by the approximated supernodes can correspond to fill-paths in the elimination graph $G^*(A)$ whereas merging supernodes from different supernodes will result in "useless" extra-fill (zero terms that does not correspond to any fill-path in $G^*(A)$). Thus, the extra-fill created when respecting this rule has a better chance to improve the convergence rate. Some futur works will investigate a generalized algorithm that releases this constraint.

As mentioned before, the amalgamation problem consists to merge as many supernodes as possible while adding the fewer extra-fill. We propose a heuristic based on a greedy algorithm: given the set of all supernodes, it consists to iteratively merge the couple of succesive supernodes $(i, i+1)$ which creates the lesser extra-fill in the factor (see figure 1) until a tolerance $\alpha$ is reached. Each time a couple of supernode 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 $\alpha$ (given as a percentage of the number of non-zero elements found by the ILU(k) symbolic factorization). This algorithm requires to know at each step which couple of supernodes will add the lesser fill-in in the factors when they are merged. This is achieved by maintaining a heap that contains all the couple of supernodes sorted by their cost (in terms of new entries) to merge them. As said before, we only consider couple of ILU(k) supernodes that belong to the same *direct supernode*). This means that, each time two supernodes are merged, the cost to merge the new supernode with its father and

its son (it can only has one inside a direct supernode) has to be updated in the heap.

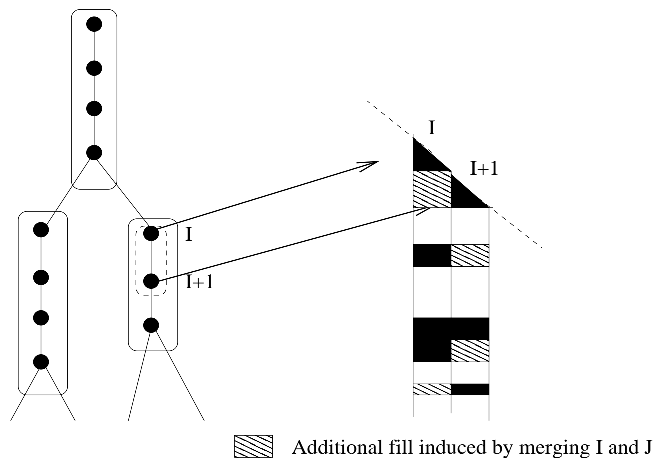


**Fig. 1.** Additional fill created when merging two supernodes I and J.

The next section gives some results on the effect of the $\alpha$ parameter and a comparison to the classic scalar ILU(k) preconditioner.

## 4 Results

In this section, we considere 3 test cases from the PARASOL collection (see table 1). $\mathrm{NNZ_A}$ is the number of off-diagonal terms in the triangular part of matrix $A$, $\mathrm{NNZ_L}$ is the number of off-diagonal terms in the factorized matrix $L$ (for direct method) and $OPC$ is the number of operations required for the factorization (for direct method).

**Table 1.** Description of our test problems.

| Name | Columns | $\mathrm{NNZ_A}$ | $\mathrm{NNZ_L}$ | OPC |
|---|---|---|---|---|
| **SHIPSEC5** | 179860 | 4966618 | 5.649801e+07 | 6.952086e+10 |
| **SHIP003** | 121728 | 3982153 | 5.872912e+07 | 8.008089e+10 |
| **AUDI** | 943695 | 39297771 | 1.214519e+09 | 5.376212e+12 |

Numerical experiments were performed on an IBM Power5 SMP node (16 processors per node) at M3PEC (Bordeaux, FRANCE). The stopping iteration criterion used in GMRES is the relative residual norm and is set to $10^{-7}$. We used a GMRES version without "restart".

The table 2 gives the influence of the amalgamation parameter $\alpha$ that is the percentage of extra entries in the factors allowed during the amalgation algorithm in comparison to the ones created by the exact ILU(k) factorization. The table reports the number of supernodes, the number of blocks, the time of sequential factorization, the time of a triangular solve (forward and backward) and the number of iterations for the AUDI problem for several levels of fill ($k$).

We can see in table 2 that our amalgamation algorithm allows to reduce significantly 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 particulary true for the factorization which exploits BLAS-3 subroutines (matrix by matrix operations). The superscalar effects are less important on the triangular solves that require much less floating point operations and use only BLAS-2 subroutines (matrix by vector operations). We can also verify that the time to compute the amalgamation is negligible in comparison to the numerical factorization time. As expected the number of iterations decreases with the amalgamation fill parameter: this indicates that the extra-fill allowed by the amalgamation corresponds to numerical non-zeros in the factors and are usefull in the preconditioner.

**Table 2.** Effect of amalgamation ratio $\alpha$ for AUDI problem

| $k$ | $\alpha$ | # Supernodes | # Blocks | Fill-in | Amalg. | Num. Fact. | Triang. Solve | Iterations |
|---|---|---|---|---|---|---|---|---|
| 1 | 10% | 198541 | 6332079 | 3.17 | 1.92 | 173 | 7.18 | 138 |
| 1 | 20% | 163286 | 4672908 | 3.46 | 4.24 | 77 | 4.92 | 133 |
| 1 | 40% | 127963 | 3146162 | 4.03 | 4.92 | 62 | 4.86 | 126 |
| 3 | 10% | 172026 | 11110270 | 7.07 | 6.24 | 463 | 8.99 | 78 |
| 3 | 20% | 136958 | 6450289 | 7.70 | 7.31 | 287 | 8.22 | 74 |
| 3 | 40% | 108238 | 3371645 | 8.97 | 8.06 | 177 | 7.53 | 68 |
| 5 | 10% | 153979 | 12568698 | 9.50 | 7.72 | 908 | 10.70 | 63 |
| 5 | 20% | 125188 | 6725165 | 10.35 | 8.84 | 483 | 9.44 | 59 |
| 5 | 40% | 102740 | 3254063 | 12.08 | 9.87 | 276 | 8.65 | 52 |

The table 3 shows the results on the 3 problems both in sequential and in parallel for different levels-of-fill and different amalgamation fill parameter values. The "–" indicates that the GMRES did not converged in less than 200 iterations. As we can see the parallelization is quiet good since the speed-up is about 10 in most cases. This is particulary good considering the small amount of floating point operations required in the triangular solves; indeed the parallel overcost due to the communications is usually all the more difficult to overlap by computation in this kind of situation.

The performances of a sequential scalar implementation of the column-wise ILU(k) algorithm are reported in table 4. The "–" corresponds to cases where the GMRES did not converged in less than 200 iterations. When compared to tables 3 what can be noticed is that the scalar implementation is often better for a level-of-fill of 1 but is not competitive for higher level-of-fill values. The scalar implementation of the triangular solves is always the best compared to the block wise implementation: we explain that by the fact that the block-wise

**Table 3.** Performances on 1 and on 16 processors PWR5 for 3 test cases

| AUDI | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | 1 processor | | | 16 processors | | |
| $k$ | $\alpha$ | Iter. | Num. Fact. | Triang. Solve | Total | Num. Fact. | Triang. Solve | Total |
| 1 | 10% | 138 | 173 | 7.18 | 1163.84 | 26 | 0.65 | 115.70 |
| 1 | 20% | 133 | 77 | 4.92 | 731.36 | 18 | 0.55 | 91.15 |
| 1 | 40% | 126 | 62 | 4.86 | 674.36 | 11 | 0.47 | 70.22 |
| 3 | 10% | 78 | 463 | 8.99 | 1164.22 | 58 | 1.25 | 155.50 |
| 3 | 20% | 74 | 287 | 8.22 | 895.28 | 33 | 0.97 | 104.78 |
| 3 | 40% | 68 | 177 | 7.53 | 689.04 | 17 | 0.70 | 64.60 |
| 5 | 10% | 63 | 908 | 10.70 | 1582.10 | 89 | 1.59 | 189.17 |
| 5 | 20% | 59 | 483 | 9.44 | 1039.96 | 47 | 1.26 | 121.34 |
| 5 | 40% | 51 | 276 | 8.65 | 725.80 | 23 | 0.82 | 65.64 |

| SHIP003 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | 1 processor | | | 16 processors | | |
| $k$ | $\alpha$ | Iter. | Num. Fact. | Triang. Solve | Total | Num. Fact. | Triang. Solve | Total |
| 1 | 10% | – | 1.41 | 0.28 | – | 0.32 | 0.05 | – |
| 1 | 20% | – | 1.41 | 0.28 | – | 0.28 | 0.05 | – |
| 1 | 40% | – | 1.58 | 0.29 | – | 0.28 | 0.04 | – |
| 3 | 10% | 76 | 4.14 | 0.45 | 38.14 | 0.69 | 0.07 | 6.01 |
| 3 | 20% | 75 | 4.05 | 0.45 | 37.80 | 0.62 | 0.05 | 4.37 |
| 3 | 40% | 64 | 4.43 | 0.42 | 31.31 | 0.60 | 0.04 | 3.16 |
| 5 | 10% | 49 | 7.81 | 0.55 | 34.76 | 1.13 | 0.07 | 4.56 |
| 5 | 20% | 35 | 6.98 | 0.55 | 26.23 | 0.90 | 0.06 | 3.0 |
| 5 | 40% | 34 | 7.24 | 0.49 | 23.9 | 0.98 | 0.06 | 3.02 |

| SHIPSEC5 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | 1 processor | | | 16 processors | | |
| $k$ | $\alpha$ | Iter. | Num. Fact. | Triang. Solve | Total | Num. Fact. | Triang. Solve | Total |
| 1 | 10% | 121 | 1.28 | 0.32 | 40.0 | 0.28 | 0.03 | 3.91 |
| 1 | 20% | 117 | 1.26 | 0.32 | 38.7 | 0.25 | 0.03 | 3.76 |
| 1 | 40% | 111 | 1.44 | 0.33 | 38.07 | 0.24 | 0.03 | 3.57 |
| 3 | 10% | 70 | 2.29 | 0.44 | 33.09 | 0.41 | 0.04 | 3.21 |
| 3 | 20% | 66 | 2.29 | 0.43 | 30.67 | 0.38 | 0.04 | 3.02 |
| 3 | 40% | 62 | 2.83 | 0.42 | 28.87 | 0.43 | 0.04 | 2.91 |
| 5 | 10% | 54 | 3.32 | 0.51 | 30.86 | 0.54 | 0.05 | 3.24 |
| 5 | 20% | 51 | 3.40 | 0.49 | 28.39 | 0.50 | 0.05 | 3.05 |
| 5 | 40% | 47 | 4.11 | 0.47 | 26.2 | 0.59 | 0.05 | 2.94 |

implementation of the triangular solves suffers of the overcost paid to call the BLAS subroutines. It seems that this overcost is not compensated by the acceleration provided by BLAS-2 subroutines compared to the scalar implementation. This is certainly du to the size of the block not sufficient for BLAS-2. In the contrary, a great difference is observed in the incomplete factorization between the scalar implementation and the block-wise implementation. In this case, the BLAS-3 subroutines offer a great improvment over the scalar implementation especially for the higher level-of-fill values that provide the bigger dense blocks and number of floating point operations in the factorization.

## 5 Conclusions

The main aims of this work have been reached. The block-wize algorithms presented in this work allow to significantly reduce the complete time to solve linear systems with incomplete factorization technique. High values of level-of-fill are manageable even in a parallel framework. Some futur works will investigate a

**Table 4.** Performances of a scalar implementation of the column-wise ILU(k) algorithm

| AUDI | | | | | |
|---|---|---|---|---|---|
| $k$ | Fill-in | Num. Fact. | Triang. Solve | Total | Iterations |
| 1 | 2.85 | 75.0 | 2.63 | 482.65 | 155 |
| 3 | 6.45 | 466.9 | 4.95 | 922.3 | 92 |
| 5 | 8.72 | 1010.4 | 6.21 | 1488.57 | 77 |
| SHIP03 | | | | | |
| $k$ | Fill-in | Num. Fact. | Triang. Solve | Total | Iterations |
| 1 | 1.99 | 3.32 | 0.16 | – | – |
| 3 | 4.15 | 15.69 | 0.29 | 39.47 | 82 |
| 5 | 5.93 | 33.74 | 0.37 | 58.53 | 67 |
| SHIPSEC5 | | | | | |
| $k$ | Fill-in | Num. Fact. | Triang. Solve | Total | Iterations |
| 1 | 1.79 | 3.38 | 0.22 | 33.08 | 135 |
| 3 | 2.76 | 10.83 | 0.33 | 38.55 | 84 |
| 5 | 3.46 | 19.56 | 0.38 | 46.16 | 70 |

generalized algorithm that releases the constraint that we impose the amalgamation algorithm to merge only ILU(k) supernodes that belong to the same direct supernode.

## 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. Anshul Gupta. Recent progress in general sparse direct solvers. *Lecture Notes in Computer Science*, 2073:823–840, 2001.
6. 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.
7. 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.
8. G. Karypis and V. Kumar. Parallel Threshold-based ILU Factorization. *Proceedings of the IEEE/ACM SC97 Conference*, 1997.
9. 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.
10. R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM Journal of Numerical Analysis*, 16(2):346–358, April 1979.
11. Joseph W. H. Liu, Esmond G. Ng, and Barry W. Peyton. On finding supernodes for sparse matrix computations. *SIAM J. Matrix Anal. Appl.*, 14(1):242–252, 1993.

12. 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.
13. G. Karypis and V. Kumar. A fast and high-quality multi-level scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20:359–392, 1998.
14. F. Pellegrini. SCOTCH 4.0 User's guide. Technical Report, INRIA Futurs, April 2005. Available at URL `http://www.labri.fr/~pelegrin/papers/scotch_user4.0.ps.gz`.
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. D. Hysom and A. Pothen. Level-based Incomplete LU factorization: Graph Model and Algorithms. (pdf file) Tech Report UCRL-JC-150789, Lawrence Livermore National Labs, 19 pp., Nov 2002.
17. P. Raghavan, K. Teranishi, and E.G. Ng. A latency tolerant hybrid sparse solver using incomplete Cholesky factorization. *Numer. Linear Algebra*, 2003.
18. Y. Saad. ILUT: a dual threshold incomplete ILU factorization. *Numerical Linear Algebra with Applications*, 1:387–402, 1994.
19. Y. Saad. *Iterative Methods for Sparse Linear Systems, Second Edition*. SIAM, 2003.
20. 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.