Towards an implementation of a multilevel ILU preconditioner on shared-memory computers

Meijster, Arnold; Wubs, Fred

Published in:
HIGH PERFORMANCE COMPUTING AND NETWORKING, PROCEEDINGS

Towards an Implementation of a Multilevel ILU Preconditioner on Shared-Memory Computers*

Arnold Meijster[1] and Fred Wubs[2]

[1] Computing Centre of the University of Groningen
A.Meijster@rc.rug.nl, http://www.rug.nl/hpc/people/arnold
[2] Research Institute of Mathematics and Computer Science
P.O. Box 800, 9700 AV GRONINGEN, The Netherlands
F.W.Wubs@math.rug.nl, http://www.math.rug.nl/~wubs

Abstract. Recently, substantial progress has been made in the development of multilevel ILU-factorizations. These methods are attractive for very large problems due to their good convergence properties. We consider the parallelization of the instance MRILU, where we restrict to a version intended for scalar problems. The most time consuming parts in using MRILU are repeated multiplication of two sparse matrices in the construction phase and the multiplication of a sparse matrix and a full vector in the solution phase. Algorithms for these operations, as well as matrix transposition, are presented and have been tested on a Cray J90.

1 Introduction

Many physical phenomena can be described by partial differential equations (PDEs). Irrespective whether one likes to compute eigenvalues, to use implicit time-integration methods or to do continuation on a steady solution one ends up with a linear system to be solved, which is often very large and sparse. In almost all cases, the solution thereof forms the bottleneck with respect to computation time. For such problems, users like to have the availability of a black-box linear-system solver, which can handle complicated systems of PDEs reasonably efficient. Developing a special purpose solver may take too much time and a direct solver is far too expensive.

An important class of iteration methods for linear systems form preconditioned CG methods and among the preconditioners Incomplete LU factorizations play a dominant role. If we consider the problem $Ax = b$ then for the ILU-factorization holds $A = LU + R$ and the CG-type method is applied to the preconditioned system

$$L^{-1}AU^{-1}\tilde{x} = L^{-1}b, \qquad \tilde{x} = Ux$$

For the classical ILU and modified ILU factorization using the same fill as the original matrix, the number of flops needed to gain a fixed amount of digits increases with the grid size which is unfavorable for very large problems.

* This research has been supported by the Stichting Nationale Computerfaciliteiten (National Computing Facilities Foundation, NCF).

M. Bubak et al. (Eds.): HPCN 2000, LNCS 1823, pp. 109–118, 2000.

The ideal case would be if the preconditioned matrix $L^{-1}AU^{-1}$ is close to identity, which is obviously the case if R is small or in other words if the factorization is nearly exact. This is accomplished in the multi-level ILU factorization. As accelerator, the classical CG method can be used when the preconditioned matrix is symmetric positive definite and e.g. Bi-CGSTAB or GMRES when it is not. However, it is our experience, that the choice of accelerator is far less critical than the preconditioning.

Our multilevel ILU method MRILU (for details see [3]) has already successfully been applied to the incompressible Navier-Stokes equations (in some cases extended with a k-ε turbulence model), to Rayleigh-Bénard flow, and to convection-diffusion problems in two and three dimensions. For a comparison of a variety of solvers including MRILU on Laplace-like equations see [2]. Similar methods like MRILU are described in [1,8,10]. Furthermore, there is also a link to algebraic multigrid, e.g. [7].

As was recognized also by others, e.g. [8], multilevel ILU methods can be parallelized. The basic steps in the factorization phase form the search of an independent set, multiplication of two sparse matrices, transposition, and for the solution phase the multiplication of a sparse matrix with a full vector. It appeared that the construction of the independent set is least critical, hence we confined our attention to the other parts. We parallelized already existing code, in which sparse matrices are stored in CSR format (section 3), and did not want to rewrite the whole code. We thus decided to stick to the CSR format[1].

The experiments in this paper have all been carried out on a Cray J90. Loops which can be parallelized are coded such that the compiler automatically generates parallel tasks for these loops. During run time execution the creation of these tasks is handled by the operating system and does involve some overhead. At the end of the loops all tasks have to synchronize (i.e. wait until all tasks have finished). This introduces some overhead as well.

In the remainder of the paper we will describe MRILU and indicate the parts to be parallelized (section 2). In section 3 we introduce data structures for storing sparse matrices, and an extension of this format for concurrent computation. Section 4 discusses parallel multiplications using these formats. In section 5 a parallel transposition of a matrix is presented. The latter two sections contain practical results obtained on a Cray J 90. Conclusions are drawn in section 6.

2 MRILU

In this section the multi-level ILU method MRILU (MR stands for matrix renumbering) will be described in short and we will comment on the parallelization aspects. A more detailed description can be found in [3]. In Algorithm 1 the basic steps in the factorization process are given. For a sparse matrix the partitioning of step 1 can always be made by extracting a set of unknowns that

[1] Though the matrix-vector multiplication using CSR format is becoming part of sparse BLAS, it is currently not implemented for the Cray J90.

Set $A^{(0)} = A$
for i=1..M
1. Reorder and partition $A^{(i-1)}$, to obtain $\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$, such that the matrix A_{11} is sufficiently diagonal dominant.
2. Approximate A_{11} by a diagonal matrix $\tilde{A}_{11}$.
3. Drop small elements in A_{12} and A_{21}.
4. Make an incomplete LU factorization, $\begin{bmatrix} I & 0 \\ \tilde{A}_{21}\tilde{A}_{11}^{-1} & I \end{bmatrix} \begin{bmatrix} \tilde{A}_{11} & \tilde{A}_{12} \\ 0 & A^{(i)} \end{bmatrix}$, where $A^{(i)} = A_{22} - \tilde{A}_{21}\tilde{A}_{11}^{-1}\tilde{A}_{12}$ (Schur complement of $\tilde{A}_{11}$).

endfor

Make an exact (or accurate incomplete) factorization of $A^{(M)}$.

Algorithm 1. The basic steps of MRILU.

are not directly connected, the so-called independent set. By allowing also weak connections, which are deleted in step 2, this set can be enlarged.

The dropping strategy used in steps 2 and 3 is based on the ratio of the element at hand and the diagonal element, and on the amount dropped so far in the corresponding row and column.

Since $\tilde{A}_{11}$ is diagonal, also its inverse and consequently the new Schur complement constructed in step 4 will be sparse, which makes it possible to repeat the process. However, the fill slowly increases in subsequent steps.

Let us discuss briefly the parallelization aspects of the respective steps. The independent set needed to create the ordering in step 1 is not unique, and finding the largest one is even an NP-complete problem. Hence, one usually uses some greedy algorithm to find an independent set. This process is sequential in nature and hence hard to parallelize, however some attempts have been made [5,6] but we expect that this leads to smaller independent sets. Since in our case the independent set selection is not the most critical part we did not parallelize it.

The parallelization of the dropping in steps 2 and 3 is straight-forward. The construction of the Schur-Complement is the more difficult one, especially the multiplication of the two sparse matrices $\tilde{A}_{21}$ and $\tilde{A}_{11}^{-1}\tilde{A}_{12}$ (the multiplication with the diagonal matrix is easy). So in general we are interested in speeding up the multiplication of two sparse matrices.

In the solution phase the L and U factor have diagonal blocks. So solving a system with these matrices amounts to multiplication of a sparse matrix and a full vector (see also the level-scheduling idea in [9]). We studied this step already for scalar equations in [4] but we will reconsider it here.

3 Data Structure

We have chosen to adopt two storage schemes. The *CSR format* (Compressed Sparse Row), which stores matrices row-wise, and the *CSC format* (Compressed

Sparse Column), which stores matrices column-wise. The reason for using two formats is the difficulty one will encounter in creating the L matrix in CSR format. In this matrix one wants to store A_{21} in each reduction step. If L were to be in CSR format then in each step A_{21} has to be merged with the hitherto formed L, which can be avoided by transposing A_{21} into CSC format and storing it thus in L. Efficient transposition is discussed in section 5.

3.1 CSR/CSC Format

The CSR data structure consists of a 5-tuple (nr, nc, cf, col, beg), where nr and nc are integers representing the number of rows and columns, respectively. The elements cf, col, and beg are arrays. The array cf contains the non-zero entries of the matrix, stored row-wise. These values are floating-point numbers. The array col is of the same length as cf, and beg has length $nr + 1$. Both arrays are integer valued. For entry $cf(i)$, its corresponding column number is found in $col(i)$. Since entries of the same row are stored consecutively in cf, we only need to know the index of the first entry of this row, and the index of the last entry. Therefore, rows are not stored explicitly (as in the case of columns), but only the index of the first element of each row is stored in the array beg. The index of the first element of row i is stored in $beg(i)$, while the index of its last element is $beg(i + 1) - 1$. The array beg has length $nr + 1$, since we need to know the index of the last element of row nr. In the next section this format is extended for use on shared memory architectures. An example of the format is given at the end of that section. The CSC format is a sort of natural dual of the CSR format. Instead of storing entries row-wise, they are stored column-wise.

3.2 Extension for Parallelism

If we try to implement algorithms on these data structures using multiple processors (cpu's), it is natural to split the data representation in as many (preferably) equal sized chunks as there are cpu's. Each cpu is assigned a chunk, from now on called its *private chunk*. On a shared memory computer, each cpu can access each memory location. In view of this machine model, it might appear a bit strange to split these data structures in chunks, since each processor can access the entire structure. Still it is useful to do this, for two reasons. The first is to reduce the amount of synchronization as much as possible. If each cpu may only modify a 'private' part of a shared resource, there is no need to protect this shared resource against simultaneous updates of this resource by more than one cpu. Such a protection is always expensive, regardless whether we program these protections explicitly ourselves (using semaphores), or let a compiler generate a data-parallel executable by using loop-parallelism. A second reason is that this data-layout mimics to some extent the distributed memory model, and thus the resulting algorithms are, with some effort, likely to be portable to distributed memory machines using message passing.

Assuming that the most frequently used operations are multiplications of matrices with vectors, it is natural to distribute the CSR data structure row-wise.

We assume that the distribution of non-zero entries in the matrix is reasonably uniform, i.e. the number of non-zero elements per row does not vary very much. Thus, if we assign to each processor (almost) the same number of rows in its chunk, we probably get a reasonable load-balance.

Let us assume that we deal with a matrix consisting of nr rows, and have the availability of P cpu's, numbered $p_1, \ldots, p_P$. Then, we distribute the matrix as follows. We compute $c = \lfloor nr/P \rfloor$. If P is a divisor of nr each processor is assigned a chunk with exactly c rows. If P is not a divisor of nr, we compute the remainder $r = nr - c * P$. It is possible to assign a single processor to deal with these r extra rows, but this might introduce significant load-imbalance (worst case $r = P - 1$). Therefore, we decide that each cpu which has a processor identification number less or equal to r is assigned $c + 1$ rows, resulting in an imbalance of only a single row. This results in the following simple algorithm to compute the lower bound (lwb) and the upper bound (upb) of the chunk assigned to processor p if the number of rows is n:

```
procedure chunk (n, p, nprocs : integer; var lwb, upb : integer);
  c := n / nprocs; r := n - c * nprocs;
  if p ≤ r
  then lwb := (p − 1) * c + p; upb := lwb + c
  else lwb := (p − 1) * c + r + 1; upb := lwb + c - 1
end
```

Thus, we augment the CSR format with an integer array par which has length $P + 1$, which has basically the same structure as the array beg. For processor p, the starting row of its chunk can be found in $par[p]$, and the last row of its chunk can be found in $par[p + 1] - 1$. The augmented CSR structure will from now on be called the *PCSR* (parallel CSR) structure.

As an example, using $P = 3$, we would find for the 5×5 matrix A the following PCSR representation[2].

$$A = \begin{pmatrix} a & 0 & b & 0 & 0 \\ c & d & e & f & 0 \\ g & 0 & h & 0 & 0 \\ i & 0 & 0 & j & k \\ l & 0 & 0 & m & n \end{pmatrix}$$

nr	5													
nc	5													
par	1	3	5	6										
beg	1	3	7	9	12	15								
cf	a	b	c	d	e	f	g	h	i	j	k	l	m	n
col	1	3	1	2	3	4	1	3	1	4	5	1	4	5

For the CSC format, the same distribution technique is used on the columns of the matrix. This format will from now on be called the *PCSC* format.

4 Matrix Multiplication Algorithms

One of the most common operations on (sparse) matrices is multiplication. Two variants are needed. Let A, B, and C be sparse matrices, and x, y be full vectors. We have to deal with the following two cases.

[2] The entries are deliberately chosen to be symbolic, instead of actual numbers, in order to avoid confusion between entries and indices.

- $y := A \times x$, i.e. a sparse matrix times a full vector, and the result is stored in a full vector.
- $C := A \times B$, i.e. a sparse matrix times a sparse matrix, and the result is stored in a new sparse matrix.

4.1 Multiplication of a Sparse Matrix with a Full Vector

We have to perform nr inner products between the rows of A, and the vector x. We choose therefore to represent A using the PCSR format. In this format the entries are stored consecutively, and the first and last element of a row is directly accessible, which makes the algorithm for performing this multiplication relatively simple. The sequential algorithm is given below(left). A direct parallelization of this algorithm is to distribute the outer loop using the *par* field of the PCSR structure, and use private variables for r, and c resulting for cpu p in the algorithm on the right.

```
for r := 1 to nr →
  y(r) := 0
  for c := beg(r) to beg(r + 1) − 1 →
    y(r) := y(r) + cf(c) ∗ x(col(c))
```

```
for r := par(p) to par(p + 1) − 1 →
  y(r) := 0
  for c := beg(r) to beg(r + 1) − 1 →
    y(r) := y(r) + cf(c) ∗ x(col(c))
```

Assuming reasonable load-balancing, we expect a speedup linear in the number of processors on shared memory machines. If the architecture employs vector-pipes to speed up vector operations the algorithm ought to be modified to gain performance from parallelization as well as vectorization. The inner loop is vectorizable, however the length of this loop is of size $beg(r + 1) - beg(r)$, which is in practical cases smaller than the length of the vector pipes. Therefore we try to lengthen the inner loop as much as possible. This is unfortunately only partly realizable, since row-wise addition is unavoidable. We introduce on each cpu a private auxiliary array *tmp*, and initialize it such that $tmp(i) = x(col(i))$, and distribute its initialization out of the inner loop, such that we can unroll it completely. When this array has been constructed an element-wise product $tmp := cf * tmp$ can be computed in a fully vectorizable loop. From this new result, it is easy to compute the final result using simple summation, in smaller vectorizable loops. This leads to the following algorithm for processor p:

```
lwb := beg(par(p));
upb := beg(par(p + 1));
for r := lwb to upb − 1 →
  tmp(r) := x(col(r));
for r := lwb to upb − 1 →
  tmp(r) := tmp(r) ∗ cf(r);
for r := par(p) to par(p + 1) − 1 →
  y(r) := 0;
  for c := beg(r) to beg(r + 1) − 1 →
    y(r) := y(r) + tmp(c)
```

Although the length of this algorithm is longer than the trivial parallel solution, the amount of computational work is the same. The compiler, however, can vectorize the last two loops, and thus performance increase might be expected. The amount of extra memory needed is linear in the total number of non-zero entries, i.e. the size of the array cf.

We performed a performance test, the results of which are shown in the following table. Since absolute timings give only a measure of the performance

of the cpu's, and not of the algorithm itself, we only present speed-up factors (i.e. T_1/T_p, where T_p is the absolute time measured using p cpu's). We computed $y = Ax$, where A is a $10^4 \times 10^4$ matrix, with on average 10 non-zeroes per row.

NCPUS	1	2	4	6	8	10	12	14	16
Speed-up	1.0	1.9	3.7	5.0	6.1	7.0	7.7	8.2	8.8

On a small number of cpu's the algorithm scales almost linearly in the number of processors. If we use more than 4 cpu's some degradation is observed, which can mainly be accounted for by the fact that the data structure uses several indirections, which easily results in slightly different running times per cpu. The total execution time however, is equal to the running time of the task which ran longest. Also the creation of processes results in some operating system overhead. However, even on 16 cpu's an efficiency of more than 50% is achieved.

4.2 Multiplication of Two Sparse Matrices

We consider the 'assignment' $C := A \times B$, where A and B are sparse matrices. We start with explaining how the parallelization can be done by means of a multiplication of two full matrices in order to not obscure the approach by details on the handling of the sparsity. The sequential algorithm for multiplying an $M \times N$ matrix A, with an $N \times R$ matrix B consists of three nested loops.

```
for r := 1 to M → (* rows of A *)
 for c := 1 to R → (* columns of B *)
   c(r,c) := 0
   for i := 1 to N → (* dot product *)
     c(r,c) := c(r,c) + a(r,i) * b(i,c)
```

Clearly, each iteration of the outer loop can be performed independent of all other iterations, which allows a direct parallelization. We simply use the partitioning of the data using the PCSR format. It is useless to parallelize the second loop as well, since it would only interfere with the parallelization of the outer loop.

```
lwb := parA(p);
upb := parA(p+1);
for r := lwb to upb - 1 →
  for c := 1 to ncB → (* initialize row r of C *)
    c(r,c) := 0
  for i := 1 to ncA → (* adapt row r of C *)
    for c := 1 to ncB → (* all columns of B *)
      c(r,c) := c(r,c) + a(r,i) * b(i,c)
```

In the actual implementation we have to deal with the sparsity. On each processor we introduce a temporary full array d in which we will store all the intermediate contributions. Using reference arrays we keep precisely track of where elements in d are stored. To limit the length of the presentation we have not included these operations in the following code.

```
lwb := parA(p); upb := parA(p + 1);
for c := 1 to ncB → (∗ initialize temporary full row of C ∗)
    d(c) := 0;
initialize reference arrays for d
for r := lwb to upb − 1 →
  for i := begA(r) to begA(r + 1) − 1 → (∗ adapt row r of C ∗)
    for c := begB(colA(i)) to begB(colA(i) + 1) − 1 →
      d(colB(c)) := d(colB(c)) + cfA(i) ∗ cfB(c);
      adapt referencing d;
  copy d to begC(p), colC(p), cfC(p) using the reference arrays;
  reset d and reference arrays;
```

The resetting in the last step can be done using the same reference arrays in a time proportional to the fill. In the actual implementation this is performed together with the copying as shown in the previous program line. Indeed in this line we see that every processor has its own $begC$, $colC$ and cfC array. This is necessary since we do not know in advance the number of nonzero entries of C on a certain processor. Hence, afterwards we assemble the parts of the sparse matrix into the global $begC$, $colC$ and cfC array, which can be executed in parallel apart from a loop with length of the number of tasks.

We note that due to the fact that the reference arrays are constructed in order of occurrence of a new fill in d that the column numbers on a row are not necessarily ordered, which does not affect the remainder of the program.

We studied the speed-up of this algorithm on the product of two sparse matrices of order $(10^3 \times 10^3)$ with an average fill of 10 per row.

NCPUS	1	2	4	6	8	10	12	14	16
Speed-up	1.0	1.6	3.3	4.8	6.3	8.2	8.8	9.9	11

The speed-up is reasonable. This time, performance degradation is less severe on a larger number of cpu's. This can be explained by the fact that the amount of computation per task is a lot larger. The waiting time at the synchronization point at the end of the tasks is about the same as in the previous case (matrix times vector), but it has become relatively small compared to computation time.

5 Transpose of a PCSR Matrix

In this section we consider the transposition of a sparse matrix. The input and the output of the algorithm is a matrix stored in PCSR format. The process of transposing a PCSR matrix consists of three stages. The first, and the last being highly parallel, while the middle is purely sequential. The sequential part, however, is negligible in computation time.

Let us assume we deal with an $m \times n$ matrix (m rows) A and we want to implement $B := A^T$ using P processors. A problem is to determine the beg and par array corresponding with the destination array B. The computation of these arrays is done in two stages. In the first stage, each processor computes a private histogram of the number of elements per column in its private part of the source

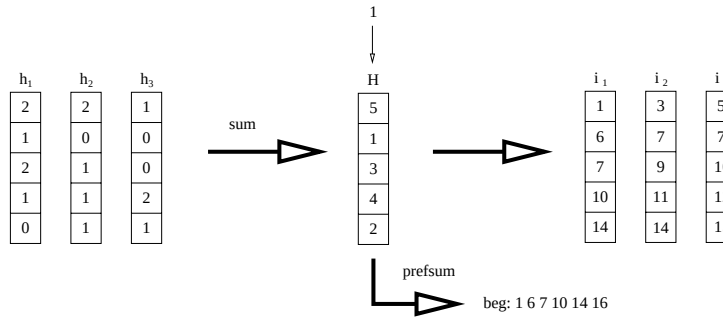

Fig. 1. Parallel transpose of a matrix using PCSR format.

matrix. This is a trivial linear time loop. The size of a histogram is n integers, and thus the total amount of extra memory is $P \times n$. Clearly, each processor can compute its private histogram, without any communication with any other process. Since the amount of work per processor is almost the same, we expect each processor to finish the first stage at approximately the same time. In Fig. 1 at the left side three histograms ($h_1,\ h_2,\ h_3$) are given. In the second stage, these histograms are summed resulting in the histogram H which shows the number of elements per column for the whole source matrix, and therefore the number of elements per row of the destination matrix. The summing of these histograms can only be performed when all 'private' histograms have been computed, i.e. each cpu has finished the first stage. This summing is performed in the second stage, on only one cpu. From the summed histogram H the *beg* array for B is easily obtained as follows. Shift the histogram H by one index, and insert 1 at H(1). Then we have $beg(i+1) = \sum_{k=1}^{i} H(i)$ (a so-called prefix-sum), and $beg(1) = 1$. This *beg* array gives also the begin locations of the columns to be built by the matrix part on the first processor (i_1). For the second processor we simply have to add i_1 and h_1 to find the begin locations for the columns of the matrix on that processor. Similarly i_3 is the sum of i_2 and h_2. In the last stage, all the elements of the source matrix are visited and copied into the destination matrix using the data structure obtained in the previous stage. This stage, just like the first one, scales linearly in the number of cpu's.

We computed $B = A^T$, where A is a $10^4 \times 10^4$ matrix, with on average 10 non-zeroes per row. The speed-up results are in the table below.

NCPUS	1	2	4	6	8	10	12	14	16
Speed-up	1.0	1.7	3.6	3.9	4.4	5.1	5.1	4.9	4.6

Two stages are highly parallel, while the middle stage is sequential. Hence, the processes have to wait for each-other when all processes have computed their private histograms and when the global histogram H has been computed. The merging of the private histograms into the global histogram is performed by process 1, which also creates the other processes at startup. The speed-ups are

similar to the expected ones considering Amdahl's law. However, they are slightly decreasing for a large number of cpu's, due to process creation overhead. Besides the amount of computation is far less than in the previous algorithms, making the middle section relatively even more a bottleneck.

6 Conclusions

In this paper the parallelization of three essential parts of MRILU is studied: the product of a sparse matrix with a full vector, the product of two sparse matrices and the transposition of a sparse matrix. The matrices are all in CSR-format which we extended to a parallel format PCSR by giving each parallel task a number of rows of the matrix. We found that the smaller the parallel tasks the sooner the speed-up drops as the number of processors grows. For the transposition the speed-up drops if more than 4 processors are used and even the runtime becomes constant due to a small sequential part. A maximum speed-up of about 5 was observed. The matrix-vector product scales linearly for up to 4 processors and after that the speed-up increases more slowly. On 16 processors a speed-up of about 9 was observed. The matrix-matrix multiplication has the largest parallel task and there a speed-up of 11 on 16 processors is observed. Thus, on average we found a speed-up of about an order of magnitude, which will, once implemented seriously improve the performance of MRILU.

References

1. R.E. Bank and C. Wagner. Multilevel ILU decomposition. *Numer. Math.*, 82:543–576, 1999.
2. E.F.F. Botta, K. Dekker, Y. Notay, A. van der Ploeg, C. Vuik, F.W. Wubs, and P.M. de Zeeuw. How fast the Laplace equation was solved in 1995. *Appl. Numer. Math.*, 24:439–455, 1997.
3. E.F.F. Botta and F.W. Wubs. Matrix Renumbering ILU: An effective algebraic multilevel ILU-preconditioner for sparse matrices. *SIAM J. Matrix Anal. Appl.*, 20(4):1007–1026, 1999.
4. E.F.F. Botta, F.W. Wubs, and A. van der Ploeg. A fast linear-system solver for large unstructured problems on a shared-memory computer. In O. Axelsson and B. Polman, editors, *Proceedings of the Conference on Algebraic Multilevel Iteration Methods with Applications*, pages 105–116, Nijmegen, The Netherlands, 1996. University of Nijmegen.
5. M.T. Jones and P.E. Plassman. A parallel coloring heuristic. *SIAM J. Sci. Comput.*, 14(3):654–669, 1993.
6. M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 4:1036–1053, 1986.
7. A. Reusken. On a robust multigrid solver. *Computing*, 56(3):303, 1996.
8. Y. Saad. ILUM: A multi-elimination ILU preconditioner for general sparse matrices. *SIAM J. Sci. Comput.*, 17(4):830–847, 1996.
9. Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS, 1996.
10. Y. Saad and J. Zhang. BILUM: Block versions of multi-elimination and multilevel ILU preconditioner for general sparse linear systems. Technical Report UMSI 97-126, University of Minnesota, Minneapolis, 1997.