Universiteit Utrecht

Department
of
Mathematics

A Parallel linear system solver for

circuit simulation problems

by

Wim Bomhof

and

Henk A. van der Vorst

Preprint

nr. 1123

December 1999

A Parallel linear system solver for circuit simulation problems

Wim Bomhof[*] Henk A. van der Vorst[†]

December 3, 1999

Abstract

This paper presents a parallel mixed direct/iterative method for solving linear systems $Ax = b$ arising from circuit simulation. The systems are solved by a block LU factorization with an iterative method for the Schur complement. The Schur complement is a small and rather dense matrix. Direct LU decomposition of the Schur complement takes too much time in order to achieve reasonable speedup results. Our iterative method for the Schur complement is often much faster than the direct LU approach. Moreover, the iterative method is better parallelizable. This results in a fast sequential and well parallelizable method.

Keywords: Preconditioner, Parallel iterative method, mixed direct/iterative method, Sparse LU factorization, Circuit simulation, Iterative solution methods, Schur complement, GMRES.

AMS subject classifications: 65F10, 65F05, 65F50, 65Y05, 94C05

1 Introduction

In circuit simulation often a series of linear systems has to be solved. For example, in transient analysis, a DAE leads in each time-step to a system of nonlinear equations, usually to be solved with the Newton method. For a single Newton step a linear system

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

has to be solved. Most circuit simulators handle this problem by LU factorization of A.

Iterative methods for linear systems have been less effective in circuit simulation. However, recently Lengowski [16] has successfully used CGS with an incomplete LU drop tolerance preconditioner. In this approach, very small drop tolerances and only a few CGS steps are used. Compared to the LU approach this saves up to about 50 percent in the number of

[*]Mathematical Institute, University of Utrecht, Budapestlaan 6, Utrecht, The Netherlands and Philips Research Laboratories, WAY41, Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands, e-mail: bomhof@math.uu.nl

[†]Mathematical Institute, University of Utrecht, Budapestlaan 6, Utrecht, The Netherlands, e-mail: vorst@math.uu.nl

flops. Nguyen[1] has used a block Gauss-Seidel method with a dynamic partitioning. This works well, but only for some types of circuit simulation problems. Unpreconditioned Krylov methods converge too slowly in practice.

Existing parallel LU codes do not perform well for circuit simulation matrices. Our experiments with the SuperLU code by Demmel *et al* [3] showed no speedup for relevant circuit simulation test problems (the matrices of Section 2). SuperLU uses the column elimination tree of A for coarse grain parallelization. The column elimination tree of A is the elimination tree of A^TA, and for circuit matrices A^TA is often a nearly dense matrix. In that case, the column elimination tree is almost a single chain and then parallelization on a coarse grain level is not successful. On a fine grain level, SuperLU parallelizes the computations by pipelining the computations of the dependent columns. This introduces some parallel overhead, and the poor speedup indicates that this overhead is too large.

The undocumented sparse LU code PSLDU of Rothberg [22], only available on SGI platforms, is also not very suitable for circuit simulation matrices. This code allows no pivoting (SuperLU has partial pivoting), in order to make parallelization easier. However, this is not really effective, the speedup results were unsatisfactory for our test matrices. Also, for our problems, some form of pivoting is necessary for numerical stability. In the experiments with PSLDU, we preordered A with a suitable row permutation in order to avoid too small pivots on the diagonal. This is not very realistic, because in practice a suitable permutation can only be determined during the LU factorization. However, we were mainly interested in the maximum attainable speedup for PSLDU.

In this paper, we will propose a block LU factorization with an iterative method for the Schur complement to solve linear systems $Ax = b$. An outline of the method is in Section 3. The preconditioner for the Schur complement is described in Section 4. In the sequential case there is some freedom in choosing the number of unknowns for the iterative part of the method. With a suitable choice it is possible to optimize the method see Section 5. Section 6 describes an algorithm to find a suitable parallel block partition of the matrix. Possible pivot problems are discussed in section 7. The last section of this paper describes the numerical experiments.

2 Properties of circuit matrices

Matrices from circuit simulation are very sparse, e.g. the average number of nonzeros per row is usually smaller than 10. The nonzero pattern is often nearly symmetric. Table 1 shows some characteristics of our test matrices, arising from actual circuit simulation. The matrices are taken at some Newton step during a transient simulation of a circuit. Modified Nodal Analysis [21] was used to model the circuits. The Sparse Tableau Analysis method [21] for modeling circuits might lead to matrices that are less suitable for our method.

We will assume that the matrix A already has a fill reducing ordering. The minimum degree ordering [19] of $A + A^T$ is a good ordering for circuit matrices that do not already have a fill reducing ordering. The minimum degree ordering Q should be applied symmetrically to A,

[1]Personal communication with T.P Nguyen (Philips Electronics, Eindhoven)

that is, the permuted matrix is $Q^T AQ$.

The diagonal pivot is not always a suitable pivot in the LU factorization, and partial pivoting is necessary for numerical stability. In practice threshold pivoting [5] with very small thresholds, say 0.001, works fine (see also Section 8 and [14]).

problem	n	nnz(A) $\times 10^3$	h	flops $\times 10^6$	nnz($L+U$) $\times 10^3$
circuit_1	2624	36	131	0.86	41
circuit_2	4510	21	95	0.51	32
circuit_3	12127	48	85	0.54	68
circuit_4	80209	308	308	15.28	461

Table 1: Characteristics of test matrices. The dimension of the problem is n. nnz(A) is the number of nonzeros of A. flops is the number of MATLAB flops for $[L, U] = \texttt{lu}(A, 0.001)$ and $x = U\backslash(L\backslash b)$. h is the height of the elimination tree of $A + A^T$ (assume no exact numerical cancellation). The matrices are available from: `http://www.math.uu.nl/people/bomhof/`.

Our test problems are rather small. Solving the largest problem takes only a few seconds on an average workstation. Nevertheless, even for these problems a parallel solver is useful because one may have to solve thousands of these systems for one circuit simulation.

The amount of fill-in in LU factorization is usually very small for circuit simulation problems. For our test problems $\text{nnz}(L+U) < 1.6\text{nnz}(A)$, where nnz($A$) is the number of nonzeros of A. Furthermore, the number of floating point operations per nonzero entry is small (between 11 and 50). This implies that an iterative method is only cheaper than a direct method, when the number of iterations is small. Note that one matrix vector product costs $2\text{nnz}(A)$ flops, so the third and the fifth column indicate how much room there is for an iterative method. The height h of the elimination tree gives a rough indication of the possibilities for parallel elimination of unknowns, see section 6.

3 Outline of the parallel solver

The parallel algorithm is based on a doubly bordered block diagonal matrix partition:

$$\hat{A} = P^T AP = \begin{bmatrix} A_{00} & 0 & \dots & 0 & A_{0m} \\ 0 & A_{11} & \ddots & \vdots & A_{1m} \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \dots & 0 & A_{m-1m-1} & \vdots \\ A_{m0} & A_{m1} & \dots & \dots & A_{mm} \end{bmatrix}. \tag{2}$$

P is a permutation matrix. The vectors x and b, of $Ax = b$, are permuted in the same way to $\hat{x}$ and $\hat{b}$. For circuit matrices often $n_S := \text{size}(A_{mm}) \leq n/20$. The matrix $\hat{A}$ is suitable for our parallel block LU linear solver with m processors:

Algorithm 1: The parallel linear solver

parallel_for $i = 0 : m-1$,
  Decompose A_{ii}: $L_{ii}U_{ii} = P_{ii}A_{ii}$
  $L_{mi} = A_{mi}{U_{ii}}^{-1}$
  $U_{im} = {L_{ii}}^{-1}P_{ii}A_{im}$
  $y_i = {L_{ii}}^{-1}P_{ii}b_i$
  $S^{(i)} = L_{mi}U_{im}$
  $z^{(i)} = L_{mi}y_i$
end
$S = A_{mm} - \sum_{i=0}^{m-1} S^{(i)}$
$y_m = b_m - \sum_{i=0}^{m-1} z^{(i)}$
Solve parallel: $Sx_m = y_m$
parallel_for $i = 0 : m-1$,
  $x_i = {U_{ii}}^{-1}(y_i - U_{im}x_m)$
end

This is a coarse grained parallel algorithm, which means that the negative influence of parallel overhead on the speedup will be relatively small. As a part of the system solve one has to solve the a reduced system

$$Sx_m = y_m \ ,$$

with the Schur complement S. This has to be solved as fast as possible in order to achieve a reasonable speedup. A dense direct method may of interest, because S is rather dense and the Mflop rates for dense methods are much higher than for sparse direct methods. However, often S will be too small (usually size(S) < 500) for efficient parallelization of the direct solver, especially in a distributed memory environment. Solving $Sx_m = y_m$ directly may cost up to 80 percent of the total flops for solving $Ax = b$. So, the Schur complement forms a bottleneck for parallel computation. In the next section, we will present a preconditioner which makes it very attractive to use an iterative method for the Schur complement. The most expensive part of the iterative method, the matrix-vector product, is better parallelizable than a direct solution method.

A nice property of the above algorithm is that in exact arithmetic the residual of $Ax = b$ is equal to the residual of $Sx_m = y_m$, if $Sx_m = y_m$ is solved iteratively and $x_i = {U_{ii}}^{-1}(y_i - U_{im}x_m)$. This is easy to show.

In Section 6 we describe how to identify a permutation matrix P, which permutes the matrix A into the block form (2). Note that a circuit simulator can use the same permutation for P each Newton step because the sparsity pattern of A does not change. So, the costs of constructing a suitable P can be amortized over the Newton steps.

4 The preconditioner for the Schur complement

Unpreconditioned iterative methods for solving the reduced system $Sx_m = y_m$ converge very slowly for circuit simulation problems. Preconditioning is a prerequisite for such problems, but it is difficult to identify effective preconditioners for the Schur complement S. The diagonal elements of S are often relatively large, but zeros on the diagonal may occur, so that simple diagonal scaling is not robust. The ratio of the smallest and the largest eigenvalues of S may be very large, 10^6 or more, and S is very likely to be indefinite, that is, it has both eigenvalues with positive and negative real part. This makes the problem very difficult for iterative solution methods.

Our preconditioner C is based on discarding small elements of S. The elements that are larger than a relative threshold define the preconditioner:

$$c_{ij} = \begin{cases} s_{ij} & \text{if } |s_{ij}| > t|s_{ii}| \text{ or } |s_{ij}| > t|s_{jj}| \\ 0 & \text{elsewhere,} \end{cases}$$

t is a parameter: $0 \leq t < 1$. $t = 0.02$ is often a good choice. Note that the preconditioner C will be symmetric if S is symmetric. The number of nonzeros of C can be much smaller than the number of nonzeros of S, see Figures 1 and 2. Construction of the preconditioner C costs only $3\mathrm{nnz}(S)$ flops and is easy to parallelize.

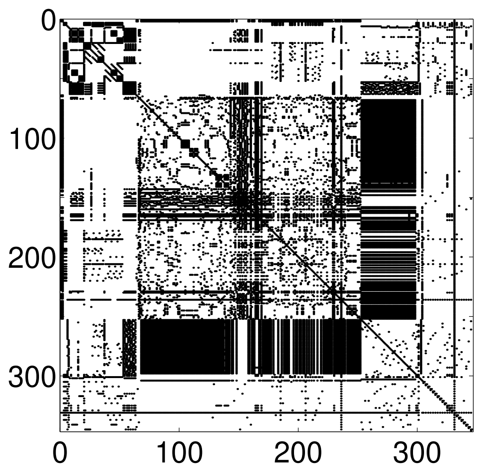

Figure 1: Nonzero pattern of a Schur complement S of problem 'circuit_4', $\mathrm{nnz}(S) = 26675$, $\mathrm{size}(S) = 346$.

For the preconditioner action, it is necessary to solve systems of the form $Cz = r$. These systems can be solved very efficiently by sparse direct techniques. A sparse LU decomposition of C will be too expensive, because of a large amount of fill-in. However, the amount of fill-in will be very small after reordering the matrix with the minimum degree algorithm. For example, the matrix of Figure 2 has 1181 nonzeros and the sparse LU factorization of the reordered matrix has 1206 nonzeros, a fill-in of only 25 elements.

The minimum degree algorithm defines a permutation matrix P, so that we actually deal with D:

$$D = P^T C P \ .$$

The sparse LU factorization is performed with partial pivoting. This leads again to a permutation matrix, Q:

$$LU = QD = QP^T CP \quad \text{or} \quad C = PQ^T LUP^T \ .$$

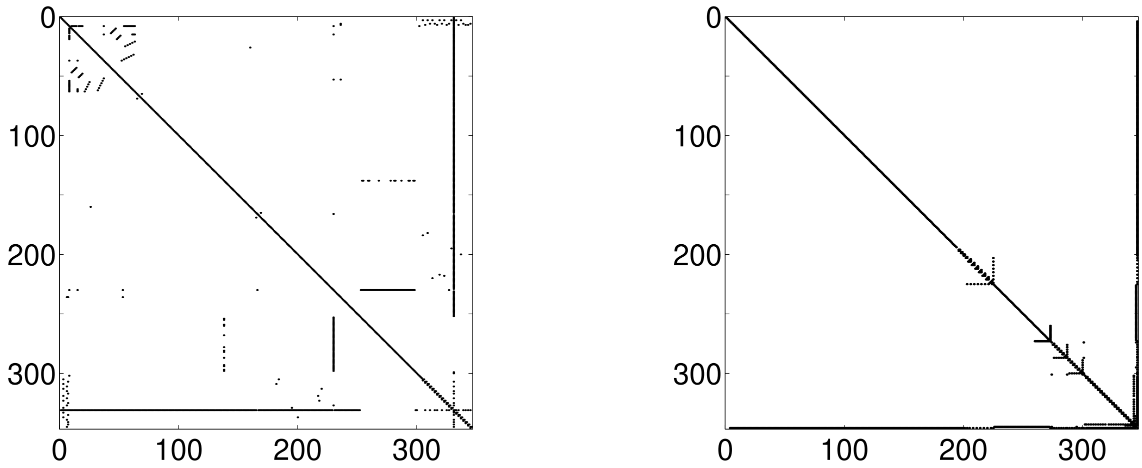

Figure 2: Left: nonzero pattern of the preconditioner C of the S from Figure 1, with $t = 0.02$, $\mathrm{nnz}(C) = 1181$. Right: Matrix D, this is C after the minimum degree ordering.

Solving the system $Cz = r$ is straight forward.

Other orderings P for the preconditioner C are possible as well. For example 's `colperm` permutation is a very effective ordering for circuit simulation problems. In this ordering the number of nonzeros per column of D is nondecreasing. This ordering is very cheap to generate, because we only have to sort the number of nonzeros per column. This can be done in $\mathcal{O}(n_S)$ operations which is important because the ordering algorithm is part of the sequential bottleneck of the parallel program. This `colperm` ordering sometimes gives very poor results for non-circuit simulation problems. For instance, for discretized PDE problems the reverse Chuthill Mckee ordering [10] turns out to be a successful alternative ordering for C.

In Section 8, we show that this preconditioner is very effective. The convergence of GMRES [23] is not very sensitive to the threshold parameter t. The matrix vector product Sv is the most expensive part of the GMRES process. For sufficiently large S, it makes sense to parallelize this matrix vector product in order to increase the scalability of the parallel solver.

For the comparison of the direct and the iterative approach for solving $Sx_m = y_m$, we will treat S as a dense matrix. Solving the system with a dense direct LU solver costs $\approx 2/3{n_S}^3$ flops. Now, assume that after k_G GMRES steps the residual is sufficiently small and assume that the matrix vector product flops of the GMRES process are the CPU-determining costs. Then the GMRES process costs $k_G \cdot 2{n_S}^2$ flops. Under these assumptions, the iterative approach is cheaper from about $n_S > 3k_G$.

5 Switching from direct to iterative

In this section we will propose a criterion for the switch from the direct to the iterative part. With this criterion we try to minimize the computational costs of the method.

Solving $Sx_m = y_m$ with a preconditioned iterative method is much more efficient than with a direct method if S is relatively small (but not too small) and dense. The iterative method performs very badly, compared with the sparse direct method, if S is large and sparse. Therefore, at some stage of the sparse LU factorization of A, it is more efficient to switch to the iterative approach. We will try to find a criterion for this.

Now, suppose that there are $m+1$ unknowns left at a stage of the Gaussian elimination (GE) process, so the right lower submatrix in the GE process is an $(m+1) \times (m+1)$ matrix T:

$$T = \left[\begin{array}{cc} c & d^T \\ e & V \end{array} \right] ,$$

with V an $m \times m$ matrix. We will assume that d and e have the same sparsity pattern, because the sparsity pattern of A is nearly symmetric. So this is a reasonable approximation. T is a Schur complement and we can take it for system $Sx_m = y_m$, to be solved with an iterative solver. The costs of k_G matrix vector products Sw are approximately

$$k_G \; (\; 4 \, \mathrm{nnz}(e) + 2 \, \mathrm{nnz}(V) \;) \text{ flops.} \tag{3}$$

We can also decide to perform one more Gaussian elimination step. Then the Schur complement becomes

$$S = V - (1/c) \, ed^T \ ,$$

and k_G matrix vector products plus the Gaussian elimination cost approximately

$$2 \, k_G \, \mathrm{nnz}(V) + 2 \, \mathrm{nnz}(e)^2 \text{ flops} \ \ , \tag{4}$$

if we assume that bc^T introduces no extra fill-in in V. The amount in (3) is smaller than in (4) if

$$\mathrm{nnz}(e) > 2 \, k_G \ . \tag{5}$$

This is a motivation to use the number of nonzeros as a criterion for the choice between the direct and the iterative approach. However, we do not know k_G in advance. Moreover, we have neglected some numbers of flops, that are different for the m and $m+1$ case. For example, the flops for constructing the preconditioner, the flops for the $\alpha x + y$ and $x^T y$ operations in GMRES and the flops for the forward and the backward solve of the preconditioner. Note also that the number of GMRES steps k_G, required to reach a sufficiently small residual, increases slowly with increasing m. We conclude that criterion (5) is of limited use in practice. We use the criterion $\mathrm{nnz}(e) > q$, with a guessed parameter q instead of $2k_G$. For the circuit simulation test problems $q = 40$ turns out to work well. Both CPU-time and the number of flops to solve the system $Ax = b$ are small for this q, see section 8. The results are not sensitive to q.

The direct/iterative partition of the unknowns can be made before the actual block LU factorization takes place. It is based on the symbolic Cholesky factor L_C of $A + A^T$. In the algorithm c_i is used for the number the number of nonzeros in column i of L_C:

$$c_i = \mathrm{nnz}(L_C(:, i)) \ . \tag{6}$$

The nonzero structures of the L and U factors of $LU = A$ (without pivoting) are related to the nonzero structure of L_C: $\mathrm{struct}(L) \subseteq \mathrm{struct}(L_C)$ and $\mathrm{struct}(U) \subseteq \mathrm{struct}({L_C}^T)$. The equal sign holds if $\text{struct}(A) = \text{struct}(A^T)$. The nonzero pattern of A is nearly symmetric for circuit simulation matrices.

The partitioning algorithm is:

Algorithm 2: Direct/iterative partitioning

input: matrix A, parameter q
output: boolean vector *direct*

```
[L_C, parent] = symbolic_Cholesky(A + A^T)
direct(1 : n) = true
direct(root) = false
for i = 1 : n
    if (c_i ≥ q)
        j = i
        while (direct(j) = true)
            direct(j) = false
            j = parent(j)
        end
    end
end
```

The vector *parent* is a side product of the symbolic Cholesky algorithm. $parent(i)$ is the parent of i in the elimination tree of $A + A^T$. $direct(i) = false$ means that unknown i will be solved iteratively. Once the algorithm marks an unknown i 'iterative', all its ancestors are marked 'iterative'.

In the parallel case, one has to choose S at least so large that a load balanced distribution of the parallel tasks is possible, see next section.

6 Finding a suitable block partition

We use the elimination tree of $A + A^T$ to determine a permutation matrix P, which permutes the matrix A into form (2). In our algorithm we use the direct/iterative partition proposed in Section 5. The elimination tree breaks up into many (one or more) subtrees if we remove the unknowns of the Schur complement in the elimination tree. These subtrees are independent of each other in the Gaussian elimination process, see [18]. Grouping the subtrees in p groups (p is the number of processors), with approximately equal weights, gives the block matrix partition (2). A load balanced partition will not be possible if there are too large subtrees. In that case, one can easily split these subtrees into smaller ones, by moving more unknowns to the Schur complement. For some problems this results in a large Schur complement which leads to modest parallel speed ups. With a small height h of the elimination tree it is usually possible to have both a load balanced partition and a small Schur complement. Circuit simulation matrices often have a small h relative to the number of unknowns n.

We will describe the partition algorithm in more detail in another paper [1]. Other implementation issues will also be discussed in that paper.

Note that we have found the permutation matrix P in an indirect way. We start with a fill reducing ordering, then we use the elimination tree to find the independent blocks. Better parallel orderings might be possible by directly partitioning the graph of the matrix A into p independent subgraphs instead of partitioning the elimination tree of A. There exist packages such as METIS [13] and Chaco [11] for this task. It is not clear whether these packages are suitable for circuit simulation problems or not. The aim of METIS is to find a load balanced distribution with a Schur complement which is as small as possible. This does not have to be optimal for our mixed direct/iterative method where we like to have dense rows and columns in the Schur complement. METIS can assign weights to vertices and edges of the graph. However, in our case the weights depend strongly on the elimination ordering which is not known in advance. For a large number of processors it is desirable to have the Schur complement as small as possible because the direct part of the method (Algorithm 1) parallelizes much better than the iterative part. In that case packages like METIS and Chaco are useful.

7 Pivoting

Threshold pivoting [5] with a small threshold t is used in the local LU decomposition of

$$\begin{bmatrix} A_{ii} \\ A_{mi} \end{bmatrix} .$$

So, the diagonal entry is chosen as pivot if $a_{ii}^{(k)} \geq t\, a_{ji}^{(k)}$ for $i < j \leq n$ The diagonal block A_{ii} almost always contains a suitable pivot for circuit simulation problems. However, in very exceptional cases there might be a suitable pivot only in the A_{mi} block. Then A_{ii} is nearly singular. This is not allowed, because exchanging rows of $\hat{A}$ destroys the structure of the block matrix (2). This problem is solved by moving the trouble causing unknown to the iterative part of the solver. So, the size of A_{ii} is decreased by one and the size of S is increased by one. This will not happen very often for circuit simulation problems. In fact we implemented the method without this pivot strategy, but that did not cause any stability problems for the circuit simulation test problems. This pivot strategy is comparable to the delayed pivot strategy which Duff and Koster [6] use in the multifrontal code.

This strategy can be refined by using two pivot threshold parameters. One for pivoting inside the diagonal blocks and one much smaller threshold to detect if it is necessary to move an unknown to the Schur complement. This improves the stability of the local LU decomposition of the diagonal blocks.

Note that in the sequential case pivoting is not restricted to the diagonal block A_{00}, partial pivoting without any restrictions is possible.

Another possibility is to replace small diagonal pivots a_{ii} by a suitably large entry if there is no pivot inside the diagonal block. This is equivalent by making a rank one correction $\beta e_i e_i^T$ to the original matrix A. Afterwards, the local L and U factors together with the preconditioner C can be used as a preconditioner for the system $Ax = b$. A drawback of this approach is that the iterative process for $Ax = b$ is much more expensive than for $Sx_m = y_m$. This is due to the longer vector lengths ($n \gg n_S$) and due to the more expensive matrix vector product. In [17] Li and Demmel do not use partial pivoting during the LU decomposition at all. Instead, they permute large entries to the diagonal before the actual LU factorization. During the LU factorization they replace small pivots by larger ones. Iterative refinement is applied afterwards. This results in a scalable parallel LU code.

In subsequent Newton steps of a circuit simulation it is often possible to use the same pivot sequence during a number of Newton steps, as long as the pivots satisfy the threshold pivot criterion. The local LU factorization can take much advantage of an unchanged pivot sequence, because of reduction of symbolic overhead which leads to a faster factorization. Note that the matrices in the local LU factorization are extremely sparse which implies that the symbolic part of the LU factorization is relatively expensive.

8 Numerical experiments

8.1 The preconditioner

In this subsection we report on the preconditioner results for the problems of Section 2. We enlarged this set of problems with a number of other circuit simulation and non-circuit simulation problems. The full set of test problems is given in Table 2.

problem	n	nnz(A) $\times 10^3$	h	flops $\times 10^6$	nnz($L+U$) $\times 10^3$
circuit_1	2624	36	131	0.86	41
circuit_2	4510	21	95	0.51	32
circuit_3	12127	48	85	0.54	68
circuit_4	80209	308	308	15.28	461
memplus	17758	99	137	2.27	122
TIa	3432	25	249	8.07	100
TIb	18510	145	579	45.30	458
TIc	1588	13	65	0.18	18
TId	6136	53	306	11.63	155
lap_md128	16129	80	533	57.24	705
lap_nd128	16129	80	368	67.86	902
lap_nd256	65025	324	750	589.61	4563
orsirr_1	1030	7	195	2.65	51
watt__1	1856	11	355	11.47	126
sherman3	5005	20	482	22.28	220
heat	6972	28	696	116.11	692

Table 2: Characteristics of test matrices, see Tabel 1 for explanation of the symbols.

The matrices 'memplus', 'orsirr_1', 'watt__1' and 'sherman3' are available from the Matrix Market [20]. Matrix 'memplus' is a circuit simulation matrix. The 'TIx' matrices were kindly provided by Kai Shen. He used these matrices to test a parallel distributed memory sparse LU code [12]. The 'TIx' matrices are circuit simulation matrices resulting from a transient simulation. All the matrices from the Matrix Market and the 'TIx' matrices were permuted symmetrically with MATLAB's minimum degree ordering `symmmd` [10] of $A + A^T$. The original ordering of 'circuit_x' is already a fine ordering. The matrices lap_... are discretized Laplacian operators on the unit square. Matrix 'lap_nd128' is on a 128×128 grid and ordered with nested dissection, 'lap_nd256' is on a finer 256×256 grid and 'lap_md128' is ordered with minimum degree. For some problems there was no right hand side b available. In that case we used a vector of all ones for b.

Note that the number of flops for an LU factorization of a circuit simulation matrix is rather sensitive to the ordering that is used. For example Larimore [15] reports for 'memplus' $5597.6 \cdot 10^6$, $698.5 \cdot 10^6$, and $30.4 \cdot 10^6$ flops, for an LU factorization of A with the orderings colamd, colmmd, and amdbar. With MATLAB's `symmmd` only $2.0 \cdot 10^6$ flops are needed.

Two parameters are fixed for each different problem; the pivot threshold for the local LU is 0.001 and the GMRES tolerance is 10^{-7}. For circuit simulation problems we use a preconditioner threshold of $t = 0.02$, a value of $q = 40$ for the direct/iterative parameter, and the `colperm` fill reducing ordering for C. This ordering is nearly as good as the minimum degree ordering, but much faster to compute. For non-circuit simulation problems we use $t = 0.005$ and the minimum degree ordering on C. The parameter q is chosen problem dependent: $q = \max(60, 0.5 \max_j(\text{nnz}(L(:,j)))$, with L the symbolic Cholesky factorization of $A + A^T$. It turns out that these q, t and fill reducing orderings lead to small CPU-times and a nearly optimal (with respect to different parameter choices) number of flops. The parameter q is for circuit simulation problems smaller than for other problems. This can be explained partly by the slower GMRES convergence of non-circuit simulation problems. Note that the results are not very sensitive to the value of the parameters q and t. The results are shown in Table 3.

Although we developed the method for circuit simulation problems, the methods also works reasonably well for some other problems. For problems 'circuit_3' and 'TIc' parameter q is too large to do anything iteratively. For all the other problems the GMRES convergence is very fast which leads to a significant reduction of flops compared with the direct method. For 'circuit_4' and 'memplus' the number of nonzeros of the preconditioner C is more than 20 times smaller than the number of nonzeros of S. Nevertheless, the iterative method converges very well. For 'circuit_4', 'TIa', 'TIb' and 'TId' solving the Schur complement directly costs more than 80 percent of the flops for solving $Ax = b$ directly. This part can be done much faster with our preconditioned iterative method. For these problems a fast Schur complement solver is a prerequisite to have reasonable speedup results. Note that we have $n_S > 3k_G$ for each problem , which indicates that it is attractive to use the iterative approach for the Schur complement, see section 4.

For problem 'TIb' the LU factorization of the permuted preconditioner D is rather expensive. Considerable savings (in the number of flops) are possible by discarding small elements during the LU factorization of D. This can be done as follows: Start with the LU factorization of D and stop when the columns of L become too dense. Take the Schur complement, discard small entries as described in Section 4, and reorder this approximate Schur complement. Proceed with the LU decomposition until completion. Now we have a sort of incomplete LU factorization of D which can be used instead of the exact L and U factors of D. Problem

	parameters							flops	
problem	q	t	Ord	n_S	nnz(S) $\times 10^3$	nnz(C) $\times 10^3$	k_G	direct $\times 10^6$	hybrid $\times 10^6$
circuit_1	40	0.020	col	127	7.5	.7	6	0.86	0.47
circuit_2	40	0.020	col	57	2.5	.3	5	0.51	0.47
circuit_3	40	0.020	col	0	0	0	0	0.54	0.54
circuit_4	40	0.020	col	344	26.8	1.1	6	15.28	3.15
memplus	40	0.020	col	166	10.8	.4	5	2.27	0.94
TIa	40	0.020	col	348	22.2	2.4	5	8.07	0.93
TIb	40	0.020	col	1265	63.0	10.4	12	45.30	7.38
TIc	40	0.020	col	0	0	0	0	0.18	0.18
TId	40	0.020	col	434	28.2	2.6	7	11.63	1.58
lap_md128	112	0.005	mmd	1011	138.5	15.8	19	57.24	27.65
lap_nd128	95	0.005	mmd	1305	140.3	16.0	17	67.86	30.44
lap_nd256	191	0.005	mmd	2649	591.3	32.2	24	589.61	252.81
orsirr_1	60	0.005	mmd	245	19.4	1.4	14	2.65	1.43
watt__1	88	0.005	mmd	432	51.4	9.1	8	11.47	5.26
sherman3	95	0.005	mmd	627	83.9	13.6	11	22.28	15.64
heat	193	0.005	mmd	894	222.6	26.4	18	116.11	56.04

Table 3: Solving the Schur complement iteratively. Ord is the symmetric ordering applied to C. Minimum degree (MATLAB's `symmmd`) is mmd and `colperm` is col. n_S is the dimension of the Schur complement S. The number of GMRES steps is k_G. The (MATLAB) flop counts are the number of flops to solve $Ax = b$ by a direct LU method and by our hybrid (mixed direct/iterative) method.

'TIb' can be solved with only $4.02 \cdot 10^6$ flops by using this approach. This is a reduction of 46 percent compared to our original mixed direct/iterative method. Most of the test problems reported here are too small to benefit from this approach.

Our experiments showed that the time step, that the circuit simulator selects in the circuit simulation process, has almost no influence on the GMRES convergence for $Sx_m = y_m$.

8.2 Sequential and parallel experiments

The parallel method was implemented with multiprocessing C compiler directives for SGI shared memory systems. The matrix vector product of the GMRES method was parallelized. The other parts of the GMRES method are not parallelized. We will discuss implementation issues in another paper [1]. Note that a distributed memory implementation is possible as well because we have a coarse grained algorithm that does not need complicated communication.

The direct solver we used (for the direct part of our method) was our own implementation of GP-Mod. This is the sparse LU method of Gilbert and Peierls [9] extended with the symmetric reductions of Eisenstat and Liu [8], [7], see also [4]. Any sparse LU method for the Schur complement is allowed as a direct solver, including, for example, multifrontal methods. We choosed GP-Mod because it is relatively easy to implement. Moreover, in [4] it is reported that, for circuit simulation problem 'memplus', GP-Mod is faster than SuperLU and also faster than the multi frontal code UMFPACK [2].

The same parameters are used as in section 8.1. But for the non-circuit simulation matrices

Liu's minimum degree ordering [19] is applied to C instead of MATLAB's `symmmd`. The results on the SGI Power Challenge are showed in Table 4. Table 5 shows the parallel results for the SGI Origin 200. With the Power Challenge we could measure the wall clock time because we were allowed to run our processes with the highest possible priority. For the Origin 200 we do not report the wall clock time because there were other minor processes on the system, which made accurate wall clock timing unreliable. Therefore, we measured the CPU-time, which is close to the wall clock time if the system load is modest. Note that processes which are waiting for synchronization are still consuming full CPU-time, so it is fair to measure CPU-time.

problem	Time (sec.)			Speedup				
	SuperLU	direct	hybrid	2	4	6	8	10
circuit_1	0.085	0.052	0.051	1.66	2.25	3.83	3.76	5.20
circuit_2	0.082	0.035	0.040	1.88	2.58	3.58	3.29	3.88
circuit_3	0.334	0.084	0.085	1.99	3.94	5.45	5.91	6.98
circuit_4	2.367	1.130	0.709	1.99	3.60	5.94	7.14	9.27
memplus	0.440	0.191	0.168	1.92	3.68	4.69	6.18	8.00
lap_md128	1.470	2.096	1.544	1.88	3.34	4.55	4.13	5.65
lap_nd128	1.740	2.534	1.643	1.79	2.97	3.95	5.31	4.56
lap_nd256	11.364	28.401	11.387	1.49	2.56	3.18	4.72	5.34
orsirr_1	0.076	0.083	0.075	1.42	1.59	2.36	2.27	2.50
watt__1	0.241	0.331	0.354	1.76	1.44	1.75	2.04	1.74
sherman3	0.440	0.662	0.559	1.61	1.85	2.43	2.08	2.10
heat	2.639	5.758	2.671	1.64	1.86	1.65	1.31	1.00

Table 4: Parallel results on a SGI Power Challenge with 12 R10000 processors at 195 MHz. The time to solve the system by SuperLU [4], by our own direct solver and by our hybrid solver is in columns 2, 3 and 4. The speedup results in the next columns are relative to the results in the hybrid column.

From the results in section 8.1 one might expect a large reduction in CPU-time for some circuit simulation problems. But, for example, for 'circuit_4' the gain is less than a factor of two. This is easy to explain by an example: Suppose we solve a problem directly and 80 percent of the flops is in the Schur complement part of the computations. The other 20 percent are very sparse flops which are much slower. So, suppose that the 80 percent takes 0.5 seconds and that the 20 percent also takes 0.5 seconds. Now, suppose we can speedup the Schur complement part by a factor of 10 by using an interative method. Then the number of flops reduces from 100 percent to $20 + 80/10 = 28$ percent. The CPU-time reduces only from 1 second to $0.5 + 0.5/10 = 0.55$ seconds. Moreover, the direct Schur complement flops are faster than the iterative ones and the iterative method introduces some extra overhead due to datastructures etc. In the worst case ('circuit_2') there is even a small loss in CPU-time.

Problems 'circuit_3' and 'TIc' have no iterative part in the sequential case. So they do not benefit from the mixed direct/iterative approach, as already noticed in the section 8.1. In the parallel case both problems have a small Schur complement, so the work involved with the Schur complement is only a small fraction of the overall amount of work, and the method is still parallelizable in this case. Problem 'circuit_3' has good speed-up results, 'TIc' is too small to have good parallel results.

problem	Time (sec.)			Speedup		
	SuperLu	direct	hybrid	2	3	4
circuit_1	0.080	0.051	0.048	1.57	2.00	2.17
circuit_2	0.074	0.035	0.037	1.80	2.22	2.37
circuit_3	0.285	0.074	0.074	1.86	2.75	3.62
circuit_4	1.781	0.882	0.558	1.89	2.81	3.50
memplus	0.358	0.169	0.143	1.83	2.49	3.19
TIa	0.430	0.254	0.090	1.54	1.57	1.63
TIb	3.198	1.509	0.538	1.53	2.30	2.39
TIc	0.037	0.017	0.017	1.74	2.34	2.45
TId	0.564	0.376	0.151	1.60	1.69	1.87
lap_md128	1.289	1.783	1.298	1.77	2.28	2.68
lap_nd128	1.557	2.191	1.416	1.67	2.15	2.82
orsirr_1	0.074	0.087	0.074	1.35	1.58	1.81
watt__1	0.209	0.331	0.307	1.61	1.33	1.35
sherman3	0.402	0.641	0.489	1.55	1.72	1.74
heat	2.227	3.905	2.202	1.48	1.73	1.83

Table 5: Parallel results on a SGI Origin 200 with 4 R10000 processors at 180 MHz. See also Table 4 for explanation.

For reference the results of the SuperLU code [4] are also in the tables. For circuit simulation problems our direct method is faster than SuperLU. This is not remarkable, because of the sparsity of circuit simulation problems, see [4]. The difference in time between our direct implementation and the SuperLU code shows that our direct implementation is sufficiently efficient although the Mflop rate is low. For 'circuit_4' we have only 15.39/1.1297 = 13.6 Mflops which is much lower than the peak performance of 390Mflops of the SGI R10000 processor. For the mixed direct/iterative solver the Mflops rate is even worse. This is normal for circuit simulation problems because of, among other things, the poor cache reuse and the large symbolic overhead for these extreme sparse problems.

For the non-circuit simulation problems SuperLU is always faster than our direct sparse LU implementation. However, the speed of our hybrid method is often comparable to the SuperLU method for these problems. This is remarkable because it was not the intention to solve these kind of problems with our method. Note that SuperLU uses dense supernodes in order to speed up the computations when the L and U factors become dense. In contrast, our method tries to keep everything sparse. The fast results of our method were obtained with a q parameter such that a relatively large part of the work is in the iterative part of the method. As a consequence the parallel speedup results are not exceptional, because the iterative part is less well parallelizable than the direct part of the method (only the matrix vector product of the iterative method has been parallelized). For non-circuit simulation problems, there is often a significant amount of work in the other parts of the iterative method. However, for our test problems these parts are too small for parallelization.

For circuit simulation problems 'TIa', 'TIb' and 'TId' something similar occurs. The sequential timings for these problems are up to three times faster than the timings for the direct method and the parallel speedup results are a bit disappointing. This is due to the expensive Schur complement system solve. We may conclude that a good sequential performance often leads to less well parallel results. The best parallel results are obtained with the three circuit simulation matrices: 'circuit_3', 'circuit_4' and 'memplus'. These problems have a heigth h of the elimination tree which is small relative to the number of unknowns n, this is good for parallelization, see Section 6. The problems 'circuit_1', 'circuit_2' are too small for reasonable speedups.

The overhead for determining the block partition (see Section 6) and for memory allocation is not included in the timings because we assumed that we are in a Newton process. So these actions have to be done only once and can be reused in the Newton process.

For all the problems reported here it was possible to find a suitable pivot inside the diagonal block. However, there exist problems for which this is not possible (for example, Matrix Market [20] problems 'lnsp3937' and 'e20r3000'). A solution to this problem is described in section 7. We have not implemented this in the code. Therefore, we had to choose a smaller LU pivot threshold in some cases. For problems 'circuit_1' and 'circuit_3' we used a pivot threshold of 0.0001 if there were 5 or more processors. For 'TIb' we always used a value of 0.0001. These small pivot thresholds do not lead to stability problems for these problems.

For the CPU-time measurements of the sequential method we used the same parameters as in the previous subsection. The iterative part of the method is less well parallelizable than the direct part of the method. Therefore, we increased parameter q for some non-circuit simulation problems and for the problems 'TIx' if there were 3 or more processors. This results in more direct work and in less iterative work. Moreover we increased the preconditioner threshold t in order to reduce the costs of the not parallelized preconditioner action and to increase the number of parallel matrix vector products. Globally this leads to slightly faster parallel results.

Demmel, Gilbert and Li also report on parallel results for problems 'memplus' and 'sherman3' with their shared memory SuperLU code [3]. These results have been copied in Table 6. The speed-up results for 'memplus' are much worse than our ones. For the non-circuit simulation problem 'sherman3' our results are worse.

	Speedup	
problem	4 proc.	8 proc.
memplus	1.73 (3.68)	1.73 (6.18)
sherman3	2.36 (1.85)	2.78 (2.08)

Table 6: Parallel speed-ups of SuperLU on a SGI Power Challenge with 16 R8000 processors at 90 MHz, from [3]. Our results (from Table 4) are between parentheses.

Jiang, Richman, Shen, and Yang report 12.81 seconds for 'TIb' on 8 processors of a 450 MHz Cray T3E [12] which is not very fast. This is caused by the unlucky combination of the minimum degree ordering and the 'S^+' factorisation method for circuit simulation problems, although a better ordering has not been identified yet. For a number of non-circuit simulation problems they have very nice results.

9 Conclusions

In this paper we have proposed a preconditioned iterative method for the solution of a Schur complement system for circuit simulation problems. This leads to an efficient sequential method which is sometimes much faster than direct sparse LU factorization. Moreover, the method is often well parallelizable which is supported by the parallel experiments. Note that a good sequential performance does not automatically lead to good parallel results. The method is not restricted to circuit simulation problems, although the results for these problems are better than for most other problems.

References

[1] W. Bomhof, *Implementation of a parallel mixed direct/iterative linear system solver.* To appear.

[2] T.A. Davis, I.S. Duff, *An unsymmetric-pattern multifrontal method for sparse LU factorization.* SIAM J. Matrix Anal. Appl., 18(1) (1997), pp. 140-158.

[3] J.W. Demmel, J.R. Gilbert. X.S. Li, *An asynchronous parallel supernodal algorithm for sparse Gaussian elimination.* SIAM J. Matrix Anal. Appl., 20(4) (1999), pp. 915-952.

[4] J.W. Demmel, S.C. Eisenstat, J.R. Gilbert, X.S. Li, J.W.H. Liu, *A supernodal approach to sparse partial pivoting.* SIAM J. Matrix Anal. Appl., 20(3) (1999), pp. 720-755.

[5] I.S. Duff, A.M. Erisman, J.K. Reid, *Direct methods for sparse matrices.* Oxford University Press, Oxford, 1986.

[6] I.S. Duff, J. Koster, *The design and use of algorithms for permuting large entries to the diagonal of sparse matrices.* SIAM J. Matrix Anal. Appl., 20(4) (1999), pp. 889-901.

[7] S.C. Eisenstat, J.W.H. Liu, *Exploiting structural symmetry in a sparse partial pivoting code.* SIAM J. Sci. Comput., 14(1) (1993), pp. 253-257.

[8] S.C. Eisenstat, J.W.H. Liu, *Exploiting structural symmetry in unsymmetric sparse symbolic factorization.* SIAM J. Matrix. Anal. Appl., 13(1) (1992), pp. 202-211.

[9] J.R. Gilbert, T. Peierls, *Sparse partial pivoting in time proportional to arithmetic operations.* SIAM J. Sci. Comput., 9(5) (1988), pp. 862-874.

[10] J.R.Gilbert, C. Moler, R Schreiber, *Sparse matrices in Matlab: design and implementation.* SIAM J. Matrix Anal. Appl., 13(1) (1992), pp. 333-356.

[11] B. Hendrickson, R. Leland, *An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations.* SIAM J. Sci. Stat. Comput., 16(2) (1995), pp. 452-469.

[12] B. Jiang, S. Richman, K. Shen, and T. Yang, *Efficient Sparse LU Factorization with Lazy Space Allocation.* In SIAM 1999 Parallel Processing Conference on Scientific Computing.

[13] G. Karypis, V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs.* SIAM J. Sci. Comput., 20 (1999), pp. 359-392.

[14] K. Kundert, *Sparse matrix techniques.* In: Circuit Analysis, Simulation and Design, Albert Ruehli (Ed.), North-Holland, 1986.

[15] S. I. Larimore, *An approximate minimum degree column ordering algorithm.* MS Thesis, CISE Tech Report TR-98-016, University of Florida, 1998. Available at: `ftp://ftp.cise.ufl.edu/cis/tech-reports/tr98/tr98-016.ps`

[16] L. Lengowski, *CGS preconditioned with ILUT as a solver for circuit simulation.* Nat. Lab Unclassified Report 828/98, Philips Electronics N.V., 1998.

[17] X.S. Li, J.W. Demmel, *Making Sparse Gaussian elimination scalable by static pivoting* Proceedings of Supercomputing 98 conference, November 7-13, 1998, in Orlando.

[18] J.W.H. Liu, *The role of elimination trees in sparse factorization.* SIAM J. Matrix Anal. Appl., 11(1) (1990), pp. 134-172.

[19] J.W.H. Liu, *Modification of the minimum degree algorithm by multiple elimination.* ACM Trans. Math. Software, 11 (1985), pp. 141-153.

[20] Matrix Market, Collection of test matrices, available at: `http://math.nist.gov/MatrixMarket/index.html`.

[21] W.J. McCalla, *Fundamentals of computer-aided circuit simulation.* Kluwer Acad. Publ. Group, Dordrecht, the Netherlands, 1988.

[22] E. Rothberg, Silicon Graphics, Inc., man-page for PSLDU.

[23] Y. Saad, M. Schultz, *A generalized minimal residual algorithm for solving nonsymmetric linear systems.* SIAM J. Sci. Statist. Comput., 7 (1986), pp. 856-869.