Recent advances in sparse linear solver technology for semiconductor device simulation matrices

(Invited Paper)

Olaf Schenk and Michael Hagemann
Department of Computer Science
University of Basel
Basel, Switzerland
Email: {olaf.schenk|michael.hagemann}@unibas.ch

Stefan Röllin
Integrated Systems Laboratory
Swiss Federal Institute of Technology
Zurich, Switzerland
Email: roellin@iis.ee.ethz.ch

Abstract— **This paper discusses recent advances in the development of robust direct and iterative sparse linear solvers for general unsymmetric linear systems of equations. The primary focus is on robust methods for semiconductor device simulations matrices, but all methods presented are solely based on the structure of the matrices and can be applied to other application areas e.g. circuit simulation. Reliability, a low memory-footprint, and a short solution time are important demands for the linear solver. Currently, no black-box solver exists that can satisfy all criteria. The linear systems from semiconductor device simulations can be highly ill-conditioned and therefore quite challenging for direct and preconditioned iterative solver. In this paper, it is shown that nonsymmetric permutations and scalings aimed at placing large entries on the diagonal greatly enhance the reliability of direct and iterative methods. The numerical experiments indicate that the overall solution strategy is both reliable and very cost effective. The paper also compares the performance of some common software packages for solving general sparse systems.**

I. Introduction

In semiconductor device simulation a system of three coupled nonlinear partial differential equations has to be solved. The discretization and linearization of these equations leads to large general sparse unsymmetric linear systems, which are highly ill-conditioned and thus quite challenging to solve. There are two main approaches to solving these unsymmetric sparse linear systems. The first approach is more conservative and uses sparse direct solver technology. In the last few years algorithmic improvements [1], [2], [3] alone have reduced the time for the direct solution of unsymmetric sparse systems of linear equations by almost one or two orders of magnitude. Remarkable progress has been made in the increase of reliability and parallelization, and consistent high performance is now achieved for a wide range of computing architectures. As a result, a number of sparse direct solver packages for solving such systems are available [2], [4], [5], [6] and it is now common to solve these unsymmetric sparse linear systems of equations which might have been considered impractically large to solve with a direct method until recently.

This work was supported by the Swiss Commission of Technology and Innovation under contract number 5648.1, and the Strategic Excellence Projects (SEP) on Computational Science and Engineering of the Swiss Federal Institute of Technology Zurich.

Nevertheless, in large three-dimensional simulations with more than 100K grid nodes, the memory requirements of direct methods as well as the time for the factorization may be too high. Therefore, a second approach, namely, preconditioned Krylov subspace methods, is often employed to solve these systems. This iterative approach has smaller memory requirements and often smaller CPU time requirements than a direct method. However, an iterative method may not converge to the solution in some cases where a direct method is capable of finding the solution.

A good preconditioner is mandatory to achieve satisfactory convergence rates with Krylov subspace methods. It is well known that iterative methods combined with simple preconditioning techniques like $\mathrm{ILU}(0)$ work well when the coefficient matrix is, at least to some degree, diagonally dominant or well conditioned. In contrast, when the coefficient matrix has zeros on the diagonal or is highly unsymmetric, the preconditioners are often unstable and iterative methods may fail. Furthermore, preconditioned Krylov subspace methods rarely converge well when the off-diagonal values of the coefficient matrix are an order of magnitude larger than the diagonal entries. This is often the case in semiconductor device simulation simulations.

In [3] Duff and Koster introduce new permutation and scaling strategies for Gaussian elimination to avoid extensive pivoting strategies. The goal is to transform the coefficient matrix A with diagonal scaling matrices D_r and D_c and a permutation matrix P_r in order to obtain an equivalent system with a matrix $P_rD_rAD_c$ that is better scaled and more diagonally dominant. This preprocessing has a beneficial impact for both direct and iterative methods. The accuracy of direct solvers is improved and the need for partial pivoting is reduced, thereby speeding up the factorization process. Since the diagonal elements become large relative to the off-diagonal elements, it is intuitively clear, that diagonal preconditioners or incomplete LU-factorizations will benefit from these nonsymmetric permutations and scalings and thus are able to accelerate preconditioned iterative methods.

The paper is organized as follows. In Section 2, an overview of nonsymmetric matrix permutations and scalings is given. Section 3 briefly reviews current sparse direct solvers and iterative Krylov subspace methods that are routinely used in large semiconductor device simulations. In Section 4, the matrices for the numerical experiments are described. A comparison of different direct methods and the influence of the nonsymmetric orderings and scalings applied to iterative methods are presented in Section 5. Finally, in Section 6, the conclusions are presented.

II. Nonsymmetric Permutations and Scalings

This section gives an introduction to the known techniques to find nonsymmetric permutations, which try to maximize the elements on the diagonal of the matrix. For a deeper understanding, we refer the reader to the original paper of Duff and Koster [3].

Matrices with zeros on the diagonal can cause problems for both direct and iterative methods (for the latter the creation of the preconditioner can fail). In some fields like chemical engineering or circuit simulation a lot of zeros happen to be on the diagonal. The matrices originating from device simulation usually, but not always, have zero free diagonals. A remedy is to permute the rows of the matrix, such that only nonzero elements remain on the diagonal. By finding a perfect matching in a bipartite graph corresponding to the matrix structure, a permutation with the desired properties is defined.

A drawback of the above strategy is, that only the structure of the matrix is taken into account, which can lead to small values on the diagonal. Therefore, other approaches try to maximize the diagonal values in some sense. In the bottleneck transversal, the goal is to permute the rows of the given matrix, such that the smallest element of the diagonal is maximal. This nonsymmetric permutation can be found in two different ways. Both methods do not generate unique permutations and are sensitive to a prior scaling of the matrix. A major drawback is that only the smallest values on the diagonal are regarded, which was already reported in [7].

Instead of maximizing only the smallest value on the diagonal, the sum of all diagonal elements can be maximized. In other words, we look for a permutation σ, which maximizes the sum

$$\sum_{i=1}^{n} |a_{\sigma(i)i}|. \tag{1}$$

This problem is known as (linear sum) assignment problem or bipartite weighted matching problem in combinatorial optimization. The problem is solved by a sparse variant of the Kuhn-Munkres algorithm. The complexity is $O(n^3)$ for full $n \times n$ matrices and $O(n\tau \log n)$ for sparse matrices with τ entries. For matrices, whose associated graphs fulfill special requirements, this bound can be reduced further to $O(n^\alpha(\tau + n \log n)$ with $\alpha < 1$. All graphs arising from finite-difference or finite element discretizations meet these conditions.

A further strategy for maximizing the diagonal elements is to look for a permutation σ, which maximizes

$$\prod_{i=1}^{n} |a_{\sigma(i)i}|. \tag{2}$$

Here, the product of the diagonal elements is maximized. This problem is solved by reformulating the product into a sum and then to use the same algorithms as mentioned above. For this approach, it is possible to define two diagonal matrices D_r and D_c together with a permutation matrix P_r, such that the scaled and permuted matrix $A_1 = P_r D_r A D_c$ is an I-matrix, for which holds:

$$|a_{ii}^1| = 1, \tag{3}$$

$$|a_{ij}^1| \leq 1. \tag{4}$$

Olschowka and Neumaier [8] introduced these scalings and permutations in order to reduce pivoting in Gaussian elimination of full matrices. We use the abbreviation MPS for these scalings and the permutation, which stands for “maximize product on diagonal with scalings”. In the section presenting the numerical results, we only list the results without a nonsymmetric permutation and with MPS, since this variant delivers the best results.

III. Solvers for Sparse Linear Systems of Equations

In this section the algorithms and strategies that are used in the direct and preconditioned iterative linear solvers in the numerical experiments are discussed.

A. Sparse direct solver technology

Figure 1 outlines the PARDISO approach [2] to solve an unsymmetric sparse linear system of equations. According to [5], it is very beneficial to precede the ordering by performing a nonsymmetric permutation to place large entries on the diagonal and then to scale the matrix so that the diagonal entries are equal to one. Therefore, in step (1) the row permutation matrix, P_r, is chosen so as to maximize the absolute value of the product of the diagonal entries in $P_r A$. The code used to perform the permutations is taken from MC64, a set of Fortran routines that are included in HSL (formerly known as Harwell Subroutine Library). The diagonal scaling matrices, D_r and D_c, are selected so that the diagonal entries of $A_1 = P_r D_r A D_c$ are 1 in absolute value and its off-diagonal entries are less than or equal to 1 in absolute value.

In step (2) any symmetric fill-reducing ordering can be computed based on the structure of $P_r A + A^T P_r^T$, e.g. minimum degree or nested dissection. All experiments reported in this paper with PARDISO were conducted with the Metis nested dissection algorithm [9].

Like other modern sparse factorization codes, PARDISO heavily relies on supernodes to efficiently utilize the memory hierarchies in the hardware. An interchange among the rows and columns of a supernode, referred to as complete block diagonal supernode pivoting, has no effect on the overall fill-in and this is the mechanism for finding a suitable pivot in PARDISO. However, there is no guarantee that the numerical factorization algorithm will always succeed in finding a suitable pivot within the supernode block. When the algorithm reaches a point where it cannot factor the supernode based

```
(1)  Row/column equilibration $A_1 \leftarrow P_r \cdot D_r \cdot A \cdot D_c$,
     where $D_r$ and $D_c$ are diagonal matrices and $P_r$ is
     a row permutation that maximizes the magnitude of
     the diagonal entries.
(2)  Find a symmetric permutation $P_{fill}$ to preserve
     sparsity: $A_2 \leftarrow P_{fill} \cdot A_1 \cdot P_{fill}^T$ and based on
     $\hat{A} = A_1 + A_1^T$.
(3)  Level-3 BLAS factorization $A_2 = Q_r LU Q_c$ with
     diagonal block supernode pivoting permutations $Q_r$
     and $Q_c$. The growth of diagonal elements is controlled
     with:
         if ($|l_{ii}| < \epsilon \cdot \|A_2\|_\infty$) then
             set $l_{ii} = sign(l_{ii}) \cdot \epsilon \cdot \|A_2\|_\infty$
         endif
(4)  Solve $Ax = b$ using the block $L$ and $U$ factors, the
     permutation matrices $P_{fill}, P_r, D_r, D_c, Q_r$ and $Q_c$
     and iterative refinement.
```

Fig. 1. Pseudo-code of the complete block diagonal supernode pivoting algorithm for general unsymmetric sparse matrices.

on the previously described supernode pivoting, it uses a pivot perturbation strategy similar to [5]. The magnitude of the potential pivot is tested against a constant threshold of $\alpha = \epsilon \cdot \|A_2\|_\infty$, where ϵ is the machine precision and $\|A_2\|_\infty$ is the ∞-norm of the scaled and permuted matrix A_2.

Therefore, in step (3), any tiny pivots encountered during elimination are set to $\text{sign}(l_{ii}) \cdot \epsilon \cdot \|A_2\|_\infty$ — this trades off some numerical stability for the ability to keep pivots from getting too small. Although many failures could render the factorization well-defined but essentially useless, in practice it is observed that the diagonal elements are rarely modified for the large class of matrices that has been used in the numerical experiments. The result of this pivoting approach is that the factorization is, in general, not exact and iterative refinement may be needed in step (4).

B. *Incomplete LU factorizations*

Iterative methods are usually combined with preconditioners to improve the convergence rates. Especially for ill-conditioned matrices, iterative methods fail without the application of a preconditioner. We briefly discuss some of the most common preconditioners. For a deeper understanding we refer the reader to [10].

A frequently used class of preconditioners are incomplete LU-factorizations. In contrary to full Gaussian elimination, the factors L and U are not computed exactly, but some elements are disregarded during the elimination, which makes it more economical to compute, store and solve with. Several strategies have been proposed in the literature to determine which elements are kept and which are dropped. One of the simplest ideas is to keep those elements in L and U, whose corresponding values in the given matrix are nonzero. This version is called ILU(0).

The generalization of ILU(0) is the incomplete LU-factorization ILU(p), which uses the concept of "level-of-fill". In this method, each element of L and U has an associated level during the elimination. Here, an element is dropped during the factorization, if its level becomes larger than a given threshold p. The complexity of this preconditioner is higher than for ILU(0). In addition, the memory demand is not known until the computation is completed. Since it is solely based on the structure of the matrix and the numerical values of the matrix are not taken into account, the resulting preconditioning can be poor.

In the ILUT(ε, q) factorization, the dropping is based on the numerical values rather than the positions. Most incomplete factorizations are either row (or column) oriented. After a row has been computed in the ILUT(ε, q) factorization, all elements in this row of L and U smaller than the given tolerance ε are disregarded. In order to limit the size of the factors, only the q largest elements in each row are kept. For non-diagonally dominant and indefinite matrices, this preconditioner usually gives better results than ILU(p).

The combination of ILU(p) and ILUT(ε, q) leads to a factorization, which is not often mentioned in the literature. We call this method ILUPT(p, ε) in the remainder of this document. An element is always kept in this factorization, if its level is zero. If the level is positive, the element is dropped if either the value is smaller than the given tolerance ε or its level exceeds p.

IV. Description of the test problems

This section gives an overview of the matrices that are used for the numerical experiments. Some general informations about the matrices are given in Table I. They are extracted from different semiconductor device simulations with different simulators. A part stems from simulations with FIELDAY [11] from the IBM Thomas Watson Research Center. Others originate from the semiconductor device simulator DESSIS$_{ISE}$ [12], which is a product of ISE Integrated Systems Engineering Inc.

As stated earlier, most of the matrices are very ill-conditioned. The influence of MPS on the condition numbers is significant for most of the matrices. The spectrum benefits from nonsymmetric matrix scalings and permutations and the

TABLE I
General informations and statistics of the matrices used in the numerical experiments.

Name	Unknowns	Elements	Dimension	Simulation
2D_bjtcai	27'628	442'898	2D	Fielday
2D_highK	54'019	996'414	2D	Fielday
3D_Tetra	28'984	599'170	3D	Fielday
field_3D	51'448	1'056'610	3D	Fielday
barrier2-9	115'625	3'897'557	3D	Dessis
ibm_2	51'448	1'056'610	3D	Fielday
igbt3	10'938	234'006	2D	Dessis
matrix_3	125'329	2'678'750	3D	Fielday
matrix_9	103'430	2'121'550	3D	Fielday
nmos3	18'588	386'594	2D	Dessis
para-4	153'226	5'326'228	3D	Dessis

TABLE II

LU FACTORIZATION TIMES AND PEAK MEMORY USAGE (MEM) FOR DIFFERENT SOLVERS ON THE IBM POWER4. THE BEST TIMES ARE SHOWN IN BOLDFACE, THE SMALLEST MEMORY USAGE IS UNDERLINED. THE MEMORY USAGE WAS MEASURED WITH MEMPROF ON LINUX.

Matrices	SuperLU$_{dist}$ 2.0		UMFPACK 4.1		WSMP 1.9.7		PARDISO 3.0	
	time (sec.)	mem (MByte)	time (sec.)	mem (MByte)	time (sec.)	mem (MByte)	time (sec.)	mem (MByte)
2D_bjtcai	0.6	68	0.7	51	**0.4**	198	0.5	51
2D_highK	1.7	156	1.9	128	**0.9**	441	1.2	117
3D_Tetra	5.5	158	43.0	553	**3.4**	263	4.3	129
field_3D	20.4	370	18.6	516	12.8	465	**12.0**	276
barrier2-9	302.4	1863	‡	‡	**94.6**	1357	107.5	1119
ibm_2	24.8	370	18.5	516	13.0	465	**11.8**	276
igbt3	0.2	29	0.3	22	**0.1**	42	0.1	24
matrix_3	177.3	1408	149.9	2065	**85.3**	1283	95.3	1005
matrix_9	204.0	1268	150.0	1859	115.5	1347	**79.8**	847
nmos3	0.5	53	0.5	42	**0.2**	170	0.5	42
para-4	‡	‡	‡	‡	‡	‡	**163.5**	1487
Thresh	static		0.1		0.01		supernodal	

ill-conditioning is greatly reduced for most of the matrices. The number of diagonal dominant rows and columns is also affected by the permutation and scaling and increases after the preprocessing with MPS. The condition numbers and the number of diagonal dominant rows and columns can be found in [13].

V. NUMERICAL RESULTS

The numerical experiments were performed on one node on a Regatta pSeries 690 Model 681 SMP with Power4 processors running at 1.3 GHz. All algorithms were implemented in Fortran 77 and C. The codes were compiled by *xlf* and *xlc* with the -O3 optimization option in 32-bit mode and are linked with the IBM Engineering and Scientific Subroutine Library.

A. Direct Solvers

1) Serial performance of some general sparse solvers: Tables II and III list the time for the factorization in seconds and the peak memory usage in MByte of some common packages for solving large sparse systems of linear equations on a single IBM Power 4 processor. This table is shown to contrast the performance of PARDISO 3.0 [2] with other well-known software packages. The sparse solvers compared in this section are SuperLU$_{dist}$ [5], UMFPACK 4.1 [6], WSMP [4] and PARDISO. The package SuperLU$_{dist}$ is designed for distributed memory computers using MPI, whereas the target architecture for WSMP and PARDISO is a shared memory system using Pthreads or OpenMP, respectively, and UMFPACK is a sequential code.

A memory failure in the computation of the 32-bit factorization is marked with "‡" and indicates that the solver needs more than 2GB of memory for the factorization. For all solvers the recommended default options were used.

The smallest pivoting threshold for UMFPACK and WSMP has been chosen so that all completed factorizations yielded a backward error that is close to machine precision. As a result, the threshold value is 0.01 for WSMP, and 0.1 for UMFPACK. SuperLU$_{dist}$ does not have an option for partial pivoting since it is significantly more complex to implement numerical pivoting on distributed memory architectures, hence the threshold is indicated as static pivoting. PARDISO uses complete block diagonal pivoting; hence the threshold is indicated as supernodal pivoting.

In addition to the pivoting approach used, there are other significant differences between the solvers. By default, PARDISO and SuperLU$_{dist}$ use the maximal matching algorithm [3] to maximize the product of the magnitudes of the diagonal entries for all matrices. WSMP uses a similar preprocessing only on matrices if the structural symmetry is less than 80%, and UMFPACK 4.1 does not use it at all. Secondly, by default, SuperLU$_{dist}$, WSMP, and PARDISO use a symmetric permutation computed on the structure of $A+A^T$. SuperLU$_{dist}$ uses the multiple minimum degree, WSMP and PARDISO use a nested dissection ordering, and UMFPACK uses a column approximate minimum degree algorithm to compute a fill-in reducing reordering. The third difference is that WSMP is the only solver that reduces the coefficient matrix into a block triangular form.

From the numerical experiments summarized in Table II it appears that PARDISO and WSMP have the smallest overall factorization time with default solver options, and PARDISO in general requires significantly less memory compared to the other solvers and thus it is able to solve all matrices in 32-bit mode.

2) Parallel performance: For the parallel performance and scalability, the LU factorization of PARDISO is compared with that of WSMP in Table III. WSMP uses the Pthreads library and PARDISO uses the OpenMP parallel directives. In contrast to SuperLU$_{dist}$ and UMFPACK, WSMP and PARDISO are designed for a shared-address-space paradigm so no additional constraints or overhead may occur in the comparison of these two similar solvers. The solver WSMP was run in parallel with the environment variables SPINLOOPTIME and YIELDLOOPTIME set to their recommended values.

The observation that can be drawn from the table is that the factorization times are affected by the preprocessing and

TABLE III
LU FACTORIZATION TIME IN SECONDS, AND SPEEDUP (S) OF WSMP AND PARDISO ON ONE (T_1) AND EIGHT (T_8) 1.3 GHZ IBM POWER 4 PROCESSORS. THE BEST TIME AND SPEEDUP WITH EIGHT PROCESSORS IS SHOWN IN BOLDFACE, THE BEST TIME WITH ONE PROCESSOR IS UNDERLINED.

	WSMP			PARDISO		
Matrices	T_1 (s)	T_8 (s)	S $\frac{T_1}{T_8}$	T_1 (s)	T_8 (s)	S $\frac{T_1}{T_8}$
2D_bjtcai	0.42	0.20	2.10	0.46	**0.12**	**3.75**
2D_highK	0.94	0.31	3.02	1.20	**0.25**	**4.81**
3D_Tetra	3.41	1.58	2,15	4.27	**0.76**	**5.16**
field_3D	12.8	4.10	3.12	12.0	**1.90**	**6.31**
barrier2-9	94.6	31.7	2.98	107.	**18.5**	**5.78**
ibm_2	13.0	4.07	3.19	11.8	**1.98**	**5.91**
igbt3	0.11	0.14	0.78	0.14	**0.04**	**3.51**
matrix_3	85.3	23.0	3.70	95.3	**16.1**	**5.91**
matrix_9	115.	33.0	3.48	79.8	**13.1**	**6.01**
nmos3	0.21	0.13	1.61	0.35	**0.10**	**3.50**
para-4	‡	‡	‡	163.	**23.2**	**7.01**

WSMP is, in most cases, faster on a single Power 4 processor. The reason is that WSMP is the only sparse direct solver which reduces the coefficient matrix into a block triangular form. As a result WSMP in general needs less operations for the sequential factorization and is therefore faster than PARDISO on one processor. However, the two-level scheduling [2] employed in PARDISO results in better scalability and a significantly faster factorization with eight processors.

B. Iterative Solvers

In this section, we also present the numerical results to see how nonsymmetric permutations influence the iterative solution of the linear systems in semiconductor device simulation. We used BiCGStab [14] preconditioned with three different incomplete LU-factorizations to carry out the numerical experiments. The preconditioner was applied from the left. Two different stopping criteria have been used: the iteration is stopped, if either the preconditioned residual is reduced by a factor of 10^{-8} or 200 iterations are reached. Like in direct methods, the linear systems are ordered before the preconditioner is computed. Thus the iterative solution consists of four steps:

1) Determination of a nonsymmetric matrix permutation and scaling (MPS)
2) A symmetric permutation with RCM is computed
3) Creation of a preconditioner.
4) Call of an iterative method (BiCGStab)

The first two steps are optional. For the second step, we use RCM, since this is often the best choice for incomplete factorizations. A real right hand side b was used in the numerical experiments for the FIELDAY and DESSIS$_{ISE}$ matrices. The initial guess x_0 was always zero for the preconditioned iterative methods.

In Table IV we have listed a number of iterations counts of BiCGStab for different incomplete LU-factorizations with and without a nonsymmetric permutation and scaling. The cases in which the iterative method did not converge are labeled with "‡".

For ILU(0), the influence of MPS on the number of iterations is not as high as one would expect. For some matrices the number of iterations is even worse with MPS than without. However, in our experience ILU(0) with the nonsymmetric permutation and scaling is slightly more stable. If the iterative method failed with MPS, then it also did not succeed without it. But in some situations, the linear systems could only be solved with MPS. Our results coincide with observations from others [15], that is, if a system can be solved by ILU(0) without MPS, then its influence is not significant or even disadvantageous. We have also tested other nonsymmetric permutations, but the results are worse than with MPS.

The behavior of the ILUPT(5,0.01) factorization is different than for ILU(0). A lot of systems can be solved in combination with or without nonsymmetric permutation. Despite one linear system could not be solved with MPS, on the average, the number of iterations is lower with nonsymmetric permutation and scaling. Other nonsymmetric permutations not listed in Table IV give comparable results.

In our ILUT(ε, q) implementation we do not limit the number of entries in each row, i.e. we set $q = \infty$. The impact of nonsymmetric permutations on ILUT(0.01,∞) is quite high. A lot of systems can only be solved with MPS and the number of iterations is significantly reduced. In addition, there are often fewer nonzeros in the incomplete factors for this ordering than without a nonsymmetric permutation and therefore one iteration step requires fewer floating point operations. Numerical experiments with other nonsymmetric permutations showed, that they can not compete with MPS.

The number of iterations gives a good indication of the reliability of preconditioner. However, the total time to perform the four steps for the iterative solution is more important because it directly influences the time for a semiconductor device simulation. In Table V, the total time for different preconditioners without a nonsymmetric permutation and with

TABLE IV
NUMBER OF ITERATIONS OF BiCGStab PRECONDITIONED WITH THREE DIFFERENT INCOMPLETE LU-FACTORIZATIONS. THE LOWEST ITERATION COUNTS ARE SHOWN IN BOLDFACE.

	ILU(0)		ILUPT(5,0.01)		ILUT(0.01,∞)	
Matrix	None	MPS	None	MPS	None	MPS
2D_bjtcai	‡	177	155	**76**	‡	125
2D_highK	156	169	96	**83**	‡	89
3D_Tetra	‡	‡	194	86	‡	**79**
field_3D	66	64	59	37	135	**32**
barrier2-9	‡	128	**72**	‡	‡	171
ibm_2	88	93	64	39	136	**34**
igbt3	106	107	**36**	44	‡	54
matrix_3	125	143	89	48	88	**47**
matrix_9	68	66	50	31	77	**30**
nmos3	‡	‡	51	30	‡	**22**
para-4	‡	168	**51**	53	77	53

TABLE V

TOTAL TIME IN SECONDS FOR COMPLETE ITERATIVE SOLUTION FOR THREE DIFFERENT INCOMPLETE LU-FACTORIZATIONS. THE BEST TIME IS SHOWN IN BOLDFACE.

	ILU(0)		ILUPT(5,0.01)		ILUT(0.01,∞)	
Matrix	None	MPS	None	MPS	None	MPS
2D_bjtcai	‡	2.53	2.13	**1.19**	‡	1.26
2D_highK	4.45	5.11	3.15	2.95	‡	**2.08**
3D_Tetra	‡	‡	3.26	1.81	‡	**1.02**
field_3D	2.56	2.53	2.51	2.29	8.54	**1.18**
barrier2-9	‡	22.2	15.0	‡	‡	**10.9**
ibm_2	2.96	3.34	2.70	2.36	8.54	**1.25**
igbt3	0.68	0.75	0.36	0.40	‡	**0.31**
matrix_3	10.4	11.2	7.93	6.39	23.3	**3.35**
matrix_9	4.86	4.80	4.73	5.05	18.4	**3.08**
nmos3	‡	‡	0.77	0.54	‡	**0.33**
para-4	‡	35.8	16.9	13.2	6.63	**6.31**

MPS are given. ILU(0) without a nonsymmetric permutation is often faster than with MPS. However, the latter succeeds for more systems. ILUT(0.01,∞) together with MPS is for all but one example the fastest combination. It is between two and three times faster and significantly more stable than ILU(0). The performance of ILUPT(5,0.01) is slightly better than ILU(0) but does not reach the one for ILUT. A comparison of the times to perform the factorization shows, that ILUT is much cheaper to compute than ILUPT. For the largest matrices, the former is about ten times faster. The large difference stems from the fact, that ILUPT contains at least the nonzero structure of the original matrix. This has a significant influence on the number of elements in the factors. As an example, for matrix "para-4" more than four times as many elements appear in the factors of ILUPT. As a result, this makes both the factorization and each iteration step more expensive. It is interesting to note, that MPS reduces the factorization time of ILUPT in about half of the systems, but the number of nonzeros remains the same.

VI. CONCLUSION

We presented an overview of the performance of current direct solvers for linear systems that emerge in semiconductor device simulation. The results show a variance of a factor of up to three in both the solution times as well as the memory usage. The unsymmetric diagonal maximization reorderings significantly reduce the demand for pivoting in direct solution methods and thus enabled algorithmic improvements in the factorization, that enhance the robustness as well as the scalability of recent solvers [2].

A recently discovered feature of these reorderings is their beneficial effect on matrices from semiconductor device simulation. From our experiments, we see that, for the preconditioned iterative Krylov subspace solvers, nonsymmetric permutations combined with scalings give the best results in terms of the number of required iterations and the time to compute the solution. Especially for the ILUT preconditioner, where the dropping is based on the numerical values, the nonsymmetric permutation has a significant impact. The influence is smaller for the incomplete factorizations ILU(0) and ILUPT, where the positions of the values determine the dropping. The robustness of those preconditioners is nevertheless improved with the use of the nonsymmetric permutations. On an average, the most efficient preconditioner for our matrices was the ILUT factorization with nonsymmetric scaling and permutation (ILUT+MPS), which outperformed the others in a number of cases. The method ILUT+MPS is both robust and cost effective and it is the only algorithm that could solve almost all our test matrices from semiconductor device simulations.

In direct comparison, the iterative methods can be up to 25 times faster than the direct ones, especially for large matrices. Although they generally bear the risk of being unstable for certain matrices, the presented results show that the unsymmetric reorderings enable the efficient and robust solution of systems, that were as yet not solvable with standard Krylov subspace methods.

REFERENCES

[1] A. Gupta, "Improved symbolic and numerical factorization algorithms for unsymmetric sparse matrices," *SIAM Journal on Matrix Analysis and Applications*, vol. 24, no. 2, pp. 529–552, 2002.

[2] O. Schenk and K. Gärtner, "Solving unsymmetric sparse systems of linear equations with PARDISO," *to appear in Future Generation Computer Systems*, 2003.

[3] I. S. Duff and J. Koster, "On algorithms for permuting large entries to the diagonal of a sparse matrix," *SIAM J. Matrix Analysis and Applications*, vol. 22, no. 4, pp. 973–996, 2001.

[4] A. Gupta, "WSMP: Watson sparse matrix package (Part-II: direct solution of general sparse systems," IBM T. J. Watson Research Center, Yorktown Heights, NY, Tech. Rep. RC 21888 (98472), November 20, 2000.

[5] X. Li and J. Demmel, "A scalable sparse direct solver using static pivoting," in *Proceeding of the 9th SIAM conference on Parallel Processing for Scientic Computing*, San Antonio, Texas, March 22-34,1999.

[6] T. A. Davis, "Algorithm 8xx: UMFPACK V4.1, an unsymmetric-pattern multifrontal method with a column pre-ordering strategy," University of Florida, Tech. Rep. TR-03-007, 2003, submitted to ACM Trans. Math. Software, http://www.cise.ufl.edu/research/sparse/umfpack.

[7] I. S. Duff and J. Koster, "The design and use of algorithms for permuting large entries to the diagonal of sparse matrices," *SIAM J. Matrix Analysis and Applications*, vol. 20, no. 4, pp. 889–901, 1999.

[8] M. Olschowka and A. Neumaier, "A new pivoting strategy for gaussian elimination," *Linear Algebra and its Applications*, vol. 240, pp. 131–151, 1996.

[9] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[10] Y. Saad, *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, 1996.

[11] IBM Thomas Watson Research Center, *Fielday Reference Manual*, IBM Thomas Watson Research Center (http://www.research.ibm.com), 2003.

[12] Integrated Systems Engineering AG, *DESSIS*$_{-\mathrm{ISE}}$ *Reference Manual*, ISE Integrated Systems Engineering AG (http://www.ise.com), 2003.

[13] O. Schenk, S. Röllin, and A. Gupta, "The effects of nonsymmetric matrix permutations and scalings in semiconductor device and circuit simulation," Integrated Systems Laboratory, Swiss Fed. Inst. of Technology (ETH), Zurich, Switzerland, Tech. Rep. 2003/9, 2003, submitted to IEEE Transactions on Computer-Aided Design.

[14] H. van der Vorst, "Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of non-symmetric linear systems," *SIAM Journal on Scientific and Statistical Computing*, vol. 13, no. 2, pp. 631–644, 1992.

[15] M. Benzi, J. C. Haws, and M. Tuma, "Preconditioning highly indefinite and nonsymmetric matrices," *SIAM J. Scientific Computing*, vol. 22, no. 4, pp. 1333–1353, 2000.