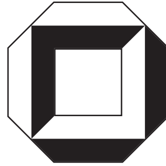


UNIVERSITÄT KARLSRUHE

A Numerical Evaluation of Preprocessing and ILU-type Preconditioners for the Solution of Unsymmetric Sparse Linear Systems Using Iterative Methods

Jan Mayer

Preprint Nr. 07/05

Institut für Wissenschaftliches Rechnen
und Mathematische Modellbildung

76128 Karlsruhe

Anschrift des Verfassers:

Dr. Jan Mayer
Institut für Angewandte und Numerische Mathematik
Universität Karlsruhe
D-76128 Karlsruhe

A Numerical Evaluation of Preprocessing and ILU-type Preconditioners for the Solution of Unsymmetric Sparse Linear Systems Using Iterative Methods

Jan Mayer
Institut für Angewandte und Numerische Mathematik
Universität Karlsruhe

Abstract

Recent advances in multilevel LU factorizations and novel preprocessing techniques have led to an extremely large number of possibilities for preconditioning sparse, unsymmetric linear systems for solving with iterative methods. However, not all combinations work well for all systems, so making the right choices is essential for obtaining an efficient solver. The numerical results for 256 matrices presented in this article give an indication of which approaches are suitable for which matrices (based on different criteria, such as total computation time or fill-in) and of the differences between the methods.

Key words: preconditioning, incomplete LU factorization, iterative methods, sparse linear systems.

AMS subject classification: 65F10, 65F50.

1 Introduction

Recently, significant advances have been made in preconditioning iterative methods for solving large, sparse, unsymmetric linear systems using incomplete LU factorizations. Most notably, the multilevel approach (see e.g. [1], [2], [4], [24]), new factorizations (see e.g. [5], [16], [18]), new dropping rules to preserve sparsity (see e.g. [4], [5], [19]) as well as new preprocessing techniques to make the coefficient matrix more suitable for incomplete factorization (see e.g. [9], [10], [14], [21], [17], [24]) have resulted in the greatest improvements. Although the main drawback of preconditioned iterative methods, namely of choosing an appropriate preconditioner and of choosing good parameters in particular, continues to exist, iterative methods can be as good and often are better than direct methods for a large number of problems.

Most of the articles mentioned provide numerical examples indicating that the new methods presented result in improvements over older methods for a number of problems. Hence, these methods are certainly promising and many have been used successfully in practice, but often it is not clear which methods and parameters work best for a particular problem. Needless to say, many researchers working in this field have considerable experience in making appropriate choices, but nevertheless, these results or even a comparative study of the methods do not appear to have been published or made available. The last systematic investigation in this direction appears to have been [6], but these results do not take

more recent developments into account and only consider problems that are small by today's standards. Additionally, the homepages of the software packages ILUPACK [3] and ILU++ [20] contain extensive numerical results for individual matrices. However, these results do not necessarily encompass some of the most recent advances and do not include an extensive analysis based on application area. Hence this article attempts to close this gap by performing extensive tests on 256 matrices made available by the University of Florida Sparse Matrix Collection [7] using modern ILU preconditioned iterative solvers.

An extremely large number of possibilities exist for forming incomplete LU factorization preconditioners, so we must restrict our attention to a few strategies. First of all, we only consider those based on Crout's implementation of Gaussian elimination, because other factorizations do not admit some of the newer dropping and pivoting strategies. Furthermore, we use only default configurations as made available by ILUPACK [3] and ILU++ [20], as these are the approaches most likely to be used in practice. For many of the problems tested, using direct methods is an alternative. Usually, direct methods do not require the user to select parameters carefully, so these are particularly attractive whenever suitable parameters for a preconditioner are not known. Hence, we also include a single direct solver, PARDISO [27] in our results. We selected PARDISO because it is a state-of-the-art solver and at least for symmetric problems, it appears to be among the best methods currently available, see [12].

We begin by providing a summary on ILU-type preconditioners. This includes the necessary details needed to understand the choices that a user of these methods needs to make. In particular, we provide some of the details of the factorizations themselves, including pivoting and dropping rules, and on the multilevel framework, including different approaches for selecting levels. We continue by introducing some of the preprocessing techniques that are used to make a matrix more suitable for incomplete factorization. Next, we provide some information on the software available and tested. We continue with a description of the numerical tests we performed. Furthermore, we provide numerical results based on total computation time and fill-in and analyze the sensitivity of total solve time to variations of the threshold parameter. These results indicate that much can be gained by selecting the method and/or preconditioner appropriately and it seems that the application area and matrix density can be useful criteria. We conclude with some specific recommendations for choosing a solver for a sparse linear system. Needless to say, these may be seen as a starting point, if the user has no better starting point for selecting a method and/or preconditioner, but as little more. These suggestions cannot replace experience.

2 A Description of Preprocessing and ILU-type Preconditioners

The preconditioners considered in the tests discussed here are preprocessed multilevel ILU factorizations as introduced in [4] and [24]. They are based on Crout's implementation of Gaussian elimination. The two main differences between the various preconditioners considered will consist of differences in the incomplete factorizations on one hand and of the different choices of preprocessing techniques on the other hand. Furthermore, the choice of preprocessing results in different strategies for forming levels, i.e. for selecting the block structure of the coefficient matrices. The different block structures in turn entail further differences between the various preconditioners. On the other hand, differences between the various ILU factorizations result from choosing different approaches for permuting rows and columns and from choosing different dropping rules. Before going into the details of

the various approaches, we begin be summarizing the ideas behind multilevel factorizations as presented in [4] and [24], which form the common framework for all preconditioners.

For simplicity, we will only describe a *complete* multilevel LDU factorization. We split a given square matrix $A_1 = A$ of dimension n into a block matrix

$$A = \begin{pmatrix} B & F \\ E & C \end{pmatrix}$$

such that the diagonal blocks B and C are square matrices of dimension n_B and $n_C = n - n_B$ respectively. Next, we calculate an LDU factorization $B = L_B D_B U_B$ of B and obtain

$$\begin{pmatrix} B & F \\ E & C \end{pmatrix} = \begin{pmatrix} L_B & 0 \\ E_B & I \end{pmatrix} \begin{pmatrix} D_B & 0 \\ 0 & S \end{pmatrix} \begin{pmatrix} U_B & F_B \\ 0 & I \end{pmatrix}.$$

Thus, L_B is a unit lower triangular, U_B is a unit upper triangular and D_B is a diagonal matrix. The matrices E_B and F_B are formally given by $E_B = EU_B^{-1}D_B^{-1}$ and $F_B = D_B^{-1}L_B^{-1}F$, respectively, and $S = C - E_B D_B F_B$ denotes the Schur complement. However, in practice, these matrices are calculated by Gaussian elimination and not by the formulas above. If we let $A_2 = S$, we can proceed recursively as described above by using A_2 for A. After a certain number of steps (or levels), we finish by completely factoring the final Schur complement. Note that in practice, this block structure need not be determined in advance. It is possible to begin factorization, to terminate whenever this seems to be a good idea and to proceed in calculating the Schur complement. In other words, the block structure is determined during the course of factorization and not in advance. We will address strategies for this approach in the sequel. Finally, it is possible to combine this factorization with row and column permutations during factorization as well as different preprocessing techniques between levels. As the modifications in the equations are obvious and not particularly enlightening in this context, we will not go into further details here. For the sake of completeness, we mention that for an incomplete multilevel ILU factorization, we simply apply a dropping rule to keep all matrices sparse.

Next, we will go into the details of the factorization, preprocessing and level termination techniques as much as necessary to understand the various options available. These options are summarized in Table 1 which includes the abbreviations used in the sequel.

2.1 Incomplete LDU Factorizations

All factorizations are based on Crout's implementation of Gaussian elimination as described in [5]. In its original form, no rows or columns are permuted during factorization. However, it is possible to use diagonal pivoting, i.e. to permute rows and columns with the same permutation so that the diagonal is kept intact during factorization and to avoid small pivots, see [4] and the references there. Alternatively, it is possible to use "dual pivoting", i.e. to permute rows and columns independently. In this case, columns are interchanged to avoid small pivots and rows are reordered to reduce fill-in, see [16], [18]. Optionally, all of these approaches can be implemented with a pivoting tolerance so that pivoting is only performed if this increases the absolute value of the pivot by at least a prescribed factor, see [23]. Note that diagonal pivoting requires little additional memory for calculations, whereas dual pivoting requires all matrices to be stored twice during factorization.

Various dropping rules are available to preserve sparsity. All are based on the dual threshold approach as introduced in [22]. Let x be a column of L or a row of U calculated in

the kth step of elimination. For a given threshold $\tau \geq 0$, all elements x_i of x are set to zero satisfying

$$w \cdot |x_i| < \tau,$$

where w is an appropriate weight. Next, if for a given threshold $p > 0$, more than p non-zero elements remain in x, we retain the p largest by absolute value and set the remainder to zero. Several different possibilities for choosing the weight w have been considered:

- $w = \frac{1}{||x||}$ is the standard dual threshold strategy, see [22].
- w may be chosen to heuristically minimize the "inverse errors" in L^{-1} and U^{-1}, see [1], [5].
- w may be chosen to heuristically minimize the propagation of errors in L and U during factorization, see [19].

Note that in addition to the options for the weight w as mentioned above, it is possible to multiply the weight w by the number of non-zero elements $\text{nnz}(x)$ of x. This approach is called "aggressive dropping" when combined with a heuristic to reduce the inverse errors, see [3]. However, we do not consider this in the sequel. Finally, rather than working with τ as above, we use $t = -\log_{10}\tau$. This is slightly more natural as now small values of t indicate inexpensive, sparse preconditioners and larger values indicate more expensive, denser preconditioners which are hopefully of higher quality.

2.2 Preprocessing Techniques

Preprocessing refers to permuting rows and/or columns as well as scaling rows and/or columns prior to (incomplete) factorizations in order to reduce fill-in and/or avoid small pivots. Note that symmetric positive definite problems, symmetric indefinite problems and unsymmetric problems generally require different approaches for effective preprocessing. For symmetric positive definite problems, preserving symmetry is a primary concern. Hence, the permutations for rows and columns are usually the same. Furthermore, reducing fill-in is generally the primary goal of preprocessing and not avoiding small pivots, which are unlikely because the matrix is positive definite. Hence, permutations obtained by reverse Cuthill-McKee, approximate minimum degree or (multilevel) nested dissection algorithms are often suitable for these problems. For symmetric indefinite problems the aim of preprocessing is to preserve symmetry, reduce fill-in and avoid small pivots. However finding a single permutation for both rows and columns (which is needed to preserve symmetry) while addressing the often conflicting goals of reducing fill-in and avoiding small pivots makes the preprocessing of these matrices quite challenging. Not surprisingly, work in this area has just begun, see [11], [13], [26]. For unsymmetric problems, the situation is perhaps somewhat simpler as it is possible to use different permutations for rows and columns. Generally, these permutations aim both at reducing fill-in and avoiding small pivots. As the preconditioners which we will use require permutations for unsymmetric problems, we will restrict our attention to these. Often the same preprocessing routine is applied to each level and this is what we will consider primarily. However, some of the standard configurations of ILUPACK [3] switch preprocessing used between levels.

2.2.1 PQ-type Reorderings

PQ-type reorderings are a class of algorithms to improve sparsity and diagonal dominance in an initial block of the matrix. These are discussed extensively in [24]. The basic (greedy) algorithm (Algorithm 3.2 in [24]) is as follows: We assign a weight

$$w_i = \frac{1}{\mathrm{nnz}(a_i)} \cdot \frac{||a_i||_\infty}{||a_i||_1}$$

to each row a_i of the matrix A. Here, $\mathrm{nnz}(a_i)$ denotes the number of non-zero elements of a_i. Large values of w_i result if one element of the row a_i (the element whose absolute value equals $||a_i||_\infty$) dominates the others and/or if the a_i contains only a few non-zero elements. Hence, large values of w_i indicate good properties for elimination, so that the rows of A are reordered in order of decreasing values of w_i. Columns are permuted in such a manner that the dominant element is moved onto the diagonal. Whenever the dominant element of different rows lies in the same column, moving both elements onto the diagonal is not possible. In this case, the row with the smaller weight is moved to a high index arbitrarily. These rows constitute the block for which it was not possible to improve diagonal dominance and they are a natural candidate for the Schur complement in the multilevel factorization. As we will only use this algorithm in the tests, we will not discuss the others, but do wish to point out that there are a number of variations that actually do guarantee some sort of diagonal dominance for some blocks. In particular, it is also possible to use a threshold to guarantee that the diagonal element dominates the others by at least a prescribed factor. Usually, these permutations should be preceded by scaling. Of the large number of possibilities, we only considered first scaling the columns of A to have norm 1, then scaling the rows to have norm 1 and finally calculating the PQ-reordering.

2.2.2 I-Matrix Preprocessing

Unlike PQ-type reordering, I-matrix preprocessing attempts to improve the diagonal dominance of the entire matrix. Recall that an I-matrix is a matrix having elements of absolute value 1 on the diagonal and of at most absolute value 1 elsewhere. Any non-singular matrix A can be transformed into an I-matrix by row permutation and by scaling rows and columns. Heuristically, it is clear that an I-matrix is more suitable for incomplete LU factorizations than general matrices. See [21] for the theoretical background and implementation details for dense matrices and [9], [10] for implementation details for sparse matrices.

Note that the row permutation needed to make an I-matrix is the permutation maximizing the absolute value of the product of the elements on the diagonal. The basic idea for calculating the permutation is to translate this multiplicative maximization problem into an additive minimization problem by applying the negative logarithm to the absolute value of the matrix coefficients. The resulting problem can be interpreted as the problem of finding a minimal weighted matching in a bipartite graph for which efficient algorithms exist. Furthermore, the minimization problem can also be interpreted as a linear program. The corresponding dual variables are in fact needed to calculate the scaling factors, see [21] for details.

2.2.3 Further Preprocessing for I-Matrices

After obtaining an I-matrix, it is possible to apply a symmetric permutation (i.e. the same permutation to rows and columns) in order to further improve the properties of the

Preprocessing Techniques	
N	normalize columns and subsequently normalize rows
PQ	PQ reordering
I	I-matrix preprocessing
sf	"sparse first" reordering for columns (to be used prior to I)
dd	symmetric permutation for a diagonally dominant initial block (after I)
M	multilevel nested dissection

Pivoting	
n	no pivoting
d	diagonal pivoting
D	dual pivoting

Dropping Rules	
s	standard dual threshold dropping
i	inverse-based dropping
e	propagation of error-based dropping

Level Termination	
P	based on absolute value of pivot
PQ	based on PQ reordering
F	based on fill-in

Table 1: Summary of the options for making a multilevel ILU preconditioner and abbreviations

matrix. It is clear that symmetric permutations preserve I-matrices. A number of different possibilities for this approach are discussed in [17]. The most reliable approach overall is Algorithm 3, which produces a diagonally dominant initial block, so we will only consider this method for the numerical results. Furthermore, in this situation it is also possible to apply any of the reordering techniques designed for symmetric matrices mentioned previously, e.g. reverse Cuthill-McKee, approximate minimum degree or multilevel nested dissection, see [14].

Alternatively, it is possible to permute columns prior to making an I-matrix. From a theoretical point of view, this is equivalent to applying a symmetric permutation to an I-matrix. However, from a practical point of view, we have different options if we permute first. Here, we only consider the very cheap option of reordering columns by increasing number of elements. As I-matrix preprocessing preserves the number of non-zero elements in the columns, we expect to obtain a sparser initial block by this approach than by applying I-matrix preprocessing without prior reordering of the columns. A sparser initial block may have better diagonal dominance, hopefully resulting in a more accurate incomplete factorization and larger pivots.

2.3 Level Termination

A major difference between the various preconditioners also stems from the criteria for level termination. In other words, there are several options for deciding when a factorization should be terminated and the Schur complement should be calculated. First of all, the

option which is always available is to terminate a level based on the absolute value of the pivot. A level is terminated whenever this value falls below some threshold. If PQ-preprocessing is used, the preprocessed matrix has a block structure with an initial block having fairly good diagonal dominance, so this provides a point for level termination. Finally, if dual pivoting is used, the factorization keeps track of the fill-in which is likely to be produced by continuing with the factorization. So it is possible to terminate a level whenever fill-in is likely to exceed a certain threshold. See [16] for details.

2.4 Other Options for Making Multilevel ILU Preconditioners

For the sake of completeness, we also mention a few other options which are available and which have been implemented in various software packages, but which we do not consider in the sequel. First of all, the approximate Schur complement can be calculated by just using the incomplete factors L and U, which is the easiest approach. Alternatively, it is possible to obtain a more accurate approximate "Tismenetsky" Schur complement which takes the elements which were dropped into account, see [29]. Obviously, it is also possible to combine both methods. Furthermore, other options for implementing dual pivoting are possible based not only on the number of elements in the factors but also on the magnitude. This approach attempts to minimize the magnitude of the fill-in rather than the number of fill-in elements. Also, a large number of preprocessing techniques is available. The choice presented here is necessarily highly selective. Finally, it would be theoretically possible to use a different incomplete factorization not based on Crout's implementation of Gaussian elimination. For example, the delayed update implementation gives rise to the ILUT factorization, see [22], for which pivoting (which results in ILUTP) can be implemented more easily, see [23]. However, many of the other options discussed here, in particular some of the dropping rules and dual pivoting, cannot be combined with the delayed update version. Hence, the drawbacks of ILUT and ILUTP generally outweigh the advantages and we do not consider these in the sequel.

3 The Preconditioners and the Software

3.1 The Preconditioners

Clearly, the building blocks discussed in the previous section can be combined in a large number of ways to make many different preconditioners. Consequently, it is impossible to test all of these extensively, especially as a number of these (e.g. pivoting, PQ, level termination) also depend on a parameter. Furthermore, not all combinations appear to have been implemented in the available software and thus cannot be tested easily. Instead, we will select several of the most promising configurations made available by ILUPACK [3] and ILU++ [20]. This seems to be a reasonable approach for several reasons: First of all, these configurations appear to have been tested and seem to work well for a broad range of problems. This is not only true for the combination of preprocessing and factorization, but even more so for the choice of any parameters, Secondly, any user of these software packages is likely to use one of these configurations, so a comparative study of these will probably be of greatest interest to the general audience.

Generally, it seems that the choice of preprocessing and pivoting strategy has the largest effect on the resulting preconditioner. This is not surprising as both involve reordering rows and columns, which is, heuristically speaking, a highly discontinuous process, where

Preproc.	Pivoting	Dropping	Level Term.	Software
N+PQ	n	e	P	ILU++
I	n	e	P	ILU++
I+dd	n	e	P	ILU++
sf+I	n	e	P	ILU++
PQ	d	i	P	ILUPACK
I+M	d	i	P	ILUPACK
N	D	e	F	ILU++
I	D	e	F	ILU++
direct solver PARDISO				PARDISO

Table 2: Combinations of preprocessing and factorization tested

interchanging just two rows or columns can result in fundamentally different factorizations. Although dropping elements is also discontinuous (a particular element is either kept or dropped), changing the drop tolerance slightly or weighting elements differently is unlikely to result in differences between the factorizations comparable to those obtained by permuting rows or columns. Consequently, we will focus on comparing preprocessing and pivoting techniques and use the default dropping rule for each software package. Hence, we selected four preprocessing techniques which were likely to require no further pivoting during factorization and tested these without pivoting. Additionally, we chose two preconditioners using diagonal pivoting and selected preprocessing which seemed to be appropriate to be used with this factorization. Two preconditioners using dual pivoting were selected similarly. This selection may seem somewhat arbitrary, but in fact, these are the combinations which have worked quite well based on other tests, see [16], [17] and the numerical results in [3]. The actual configurations used for testing can be found in Table 2. In addition to the preconditioners tested, the linear systems were also solved using the direct solver PARDISO, see [25], [27], [28].

3.2 The Software Packages

Although it would be attractive to test all preconditioners under similar conditions, not all preconditioners are implemented in one software package and different software packages differ significantly in the implementation details. These differences will be discussed shortly. As these difference are present and cannot be eliminated easily, the preconditioners will be tested as they are implemented. In a certain sense, this is an appropriate comparison because this is also the form in which the software packages will be used in practice. On the downside, it will not be entirely clear if differences in performance will be due to the preconditioners used or to the implementation.

For ILUPACK, the default solver is GMRES(30) and the only alternative offered for unsymmetric systems is FGMRES. ILU++ offers GMRES, CGS and BiCGstab. It implements BiCGstab as the default method because it seems that the sparsest preconditioners sometimes result in successful solves for BiCGstab but not for the other iterative methods. The stopping criteria are also different. ILU++ terminates the iteration if for the preconditioned system both the final residual and the relative final residual (final residual divided by the initial residual) are less than a parameter ε. The default value is $\varepsilon = 10^{-8}$. ILUPACK, on the other hand, appears to terminate if the relative residual of the original system is less than ε, the default value being $\varepsilon = 10^{-12}$.

A major difference between the software packages also stems from the memory management. ILUPACK requires the user to provide an “elbow room” parameter which determines how much memory will be allocated in a single array to store the preconditioner. If the parameter is too large and the memory requested cannot be allocated or if the parameter is too small and the array overflows during the course of the computation, ILUPACK breaks down. Although this is usually not a problem when memory is plentiful, finding a suitable parameter can be difficult if memory is scarce. The memory management of ILU++ is more involved. Unless the user wishes to change the default configuration, no memory parameter is required. ILU++ allocates for each matrix composing the preconditioner (the matrices L and U for each level) a multiple of the memory required to store the coefficient matrix (default is 3.0). If the memory is insufficient for a particular matrix then ILU++ allocates memory for another, larger array, copies the data already calculated and frees the original memory. On the other hand, if after completing the computation of a particular matrix, some memory is unused, then it is freed. As all the arrays involved are fairly small, the additional memory for making intermediate copies is usually available. As a consequence, setup times are slightly longer than necessary, but this approach allows memory to be used quite efficiently allowing for somewhat larger problems to be solved. Furthermore, this approach is almost a necessity when using dual pivoting because dual pivoting requires a significant amount of memory for intermediate calculations.

Another difference between the software packages is the programming language used. The core of ILUPACK is programmed in Fortran and it appears that most of the computation is done by Fortran routines. However, these routines are wrapped in C code which is how the user interfaces. ILU++ is programmed entirely in C++. At least some differences in the the performance for the various preconditioners is likely due to the different programming languages and the fact that ILU++ makes extensive use of the object-oriented features that C++ provides.

Finally, we would like to mention that ILU++ uses its own routine to make an I-matrix, whereas ILUPACK offers a number of interfaces to other software packages to use these routines. We chose to use the routine implemented in PARDISO. Furthermore, the multilevel nested dissection used by ILUPACK comes from the software package METIS [15].

4 Numerical Results

4.1 Overview

In this section, we present extensive numerical results for 256 square, unsymmetric matrices from the University of Florida Sparse Matrix Collection [7] based on matrix density (defined as the number of nonzero elements of the matrix divided by its dimension) and application area. Perhaps the single most important criterion for judging the usefulness of a method is the number of successful solves. Implicitly, a successful solve implies that the solution was not unreasonably inaccurate and that the memory needed for computation was sufficient. Unfortunately, the latter is platform dependent. However, as a large number of matrices were tested of varying dimensions, any method requiring significantly more memory than another will likely result in more failures on any platform. Consequently, the absolute number of successful solves will vary on different platforms, but the differences between the various methods are likely to remain. Hence, using the number of successful solves as the first criterion for judging a method is reasonable. Assuming a successful solve, the second most important criterion is the total computation time required to solve a problem. Hence, we present results for the best total computation time that can be

Application Area	Number Matrices	Min. Dim.	Max. Dim.	Min. nnz	Max. nnz
Chemical Process Simulation	25	2 398	70 304	56 934	1 916 152
Circuit Simulation	40	2 904	682 862	21 199	3 871 773
Driven Cavity Problems	24	4 241	17 281	131 556	553 956
Economic Problems (scaled)	24	2 880	64 089	19 635	837 936
Economic Problems (unscaled)	24	2 880	64 089	19 635	837 936
Electromagnetics	11	1 700	33 861	10 114	1 350 309
Fluid Dynamics	40	1 409	85 623	19 996	3 833 077
Material Problems	9	7 500	157 464	28 462	3 866 688
Model Reduction	5	2 025	20 082	67 391	281 150
Semiconductor Devices	27	2 903	155 924	19 093	5 416 358
Thermal Problems	10	1 794	147 900	7 764	3 489 300
Misc. Problems with Geometry	17	2 339	259 156	25 220	8 516 500

Table 3: Application areas and information on matrices tested

obtained by varying the threshold parameter t. Here, we also present results on the fill-in required to obtain these optimal times. Although most of the test problems considered here are so small that neither fill-in nor memory is a real concern on a modern platform, in practice, memory will often be scarce. Hence, including results on fill-in is important.

Clearly, the minimal computation times can only be achieved in practice, if this value of t is known, but this is almost never the case. Next, we examine the sensitivity of the optimal time with regard to the threshold parameter. Finally, we will examine the variance of the optimal value of t for a particular preconditioner within each application area. If the variance is not too high, then the mean optimal t may give an indication for a good choice of t for similar problems.

We used the following procedure to test the various methods for a particular matrix. First, we created an artificial right hand side such that the exact solution was a vector of all ones, unless the collection provided a right hand side, in which case, it was used. For all preconditioners tested, we varied t using stepsize 0.2 in an interval appropriate for the matrix being tested. Recall that $t = -\log_{10}\tau$, τ being the drop parameter. No additional elements were dropped based on the second parameter p. Recall that in the general case, at most p elements were kept per column of L or row of U. The exact interval for t was determined on a case by case basis, but t always satisfied $t \in [-2, 12]$. Next, we used the various software packages in their default settings for solving the linear system and allowed a maximum of 500 iterations. If the solver reported a successful solve and if the absolute error of the solution in the maximum norm did not exceed 0.1, then we recorded a successful solve and denoted a failure otherwise. Even though an error of 0.1 is often still unacceptably large, we chose not to use a more restrictive criterion as it seems likely that further more iterations or iterative refinement would further reduce the error. This was, however, not tested. On the other hand, it would seem very inappropriate to report a successful solve if the solution has an even larger error. It seems that solutions with larger error are somewhat more prevalent in ILUPACK (and also PARDISO) rather than ILU++. This is likely due to the fact that a different stopping criterion is used for the iterative method.

The matrices tested all stem from the University of Florida Sparse Matrix Collection, [7]. Essentially, the goal was to test all real, unsymmetric matrices found in the collection by

Preprocessing Pivoting	N+PQ n	I n	I+dd n	sf+I n	PQ d	I+M d	N D	I D	PARD.
Low Density	57	74	76	71	80	80	61	81	62
Medium Density	55	64	75	65	72	72	52	69	54
High Density	67	61	80	79	67	67	71	81	71
Chemical Proc. Sim.	15	17	23	19	24	17	17	21	11
Circuit Simulation	38	34	37	38	40	40	36	37	38
Driven Cavity Probl.	21	24	23	24	24	24	24	24	24
Econ. Probl. (sc.)	14	23	22	17	24	24	14	24	6
Econ. Probl. (unsc.)	1	18	23	13	18	21	1	23	12
Electromagnetics	8	6	8	6	8	7	8	7	8
Fluid Dynamics	33	32	32	34	34	35	35	33	33
Material Probl.	8	5	8	8	8	9	7	7	8
Model Reduction	5	4	5	5	5	5	5	5	5
Semicond. Devices	11	11	24	25	11	11	11	24	19
Thermal Probl.	10	10	10	10	10	10	10	10	10
Misc. Probl. w/ Geom.	15	15	16	16	13	16	16	16	13
Total	179	199	231	215	219	219	184	231	187

Table 4: Number of successful solves out of 256 matrices for each method as listed in Table 2 based on matrix density and application area as listed in Table 3.

application. However, we excluded the smaller matrices, because the results for these are not particularly relevant nowadays. Furthermore, a large number of these smaller matrices would have required solve times which were so short that these could not have been measured accurately, making meaningful results impossible. Hence, the smaller matrices were excluded on a case by case basis, so that the total calculation time for each remaining matrix was at least 0.1 seconds. In terms of dimension, matrices of approximately dimension 2 000 and less were rejected. However, some denser matrices of smaller dimension were included because the total solve time for these was often longer. Additionally, a few of the largest matrices of the collection were not considered as it was clear that no method could be successful for these matrices due to memory restrictions on the platform used. Finally, as the intention was to test the matrices based on applications, all matrices were rejected coming from applications for which the collection contained only very few matrices. More detailed information of the 256 matrices tested can be found in Table 3.

For the classification based on matrix density, we divided the set of all matrices into three subsets such that these were approximately the same size. Hence, a matrix was classified of having low density if its density was less than 7, as having medium density if its density was between 7 and 27 and as having high density otherwise. Thus, 84 matrices constitute the low density set, 82 the medium density set and 90 the high density set.

4.2 Successful Solves

Before going into further details, we will first take a look at the total number of successful solves for each method. This is perhaps the most important indicator of a method's reliability. These results can be found in Table 4.

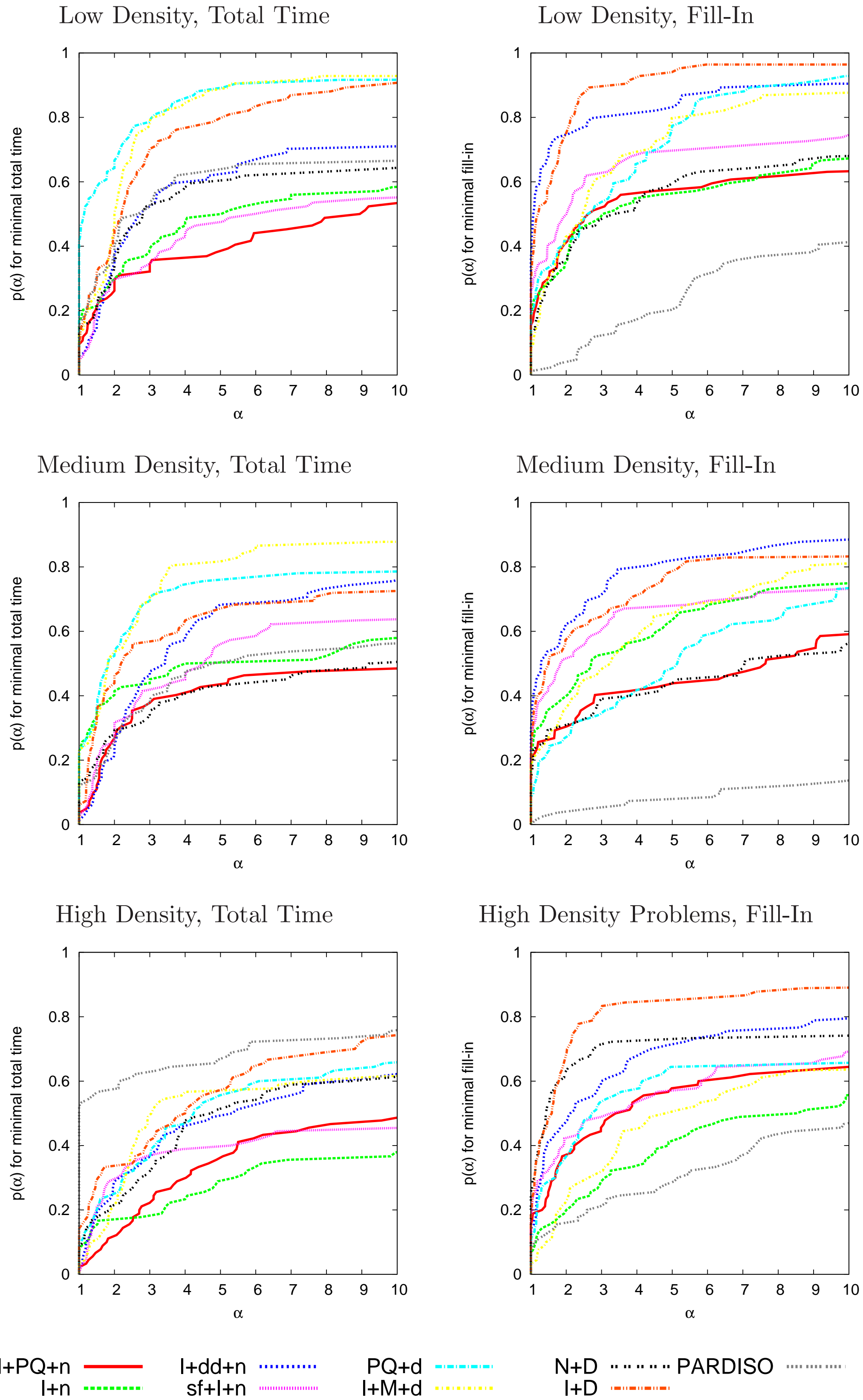


Table 5: Performance profiles for minimal total computation time and corresponding fill-in based on matrix density.

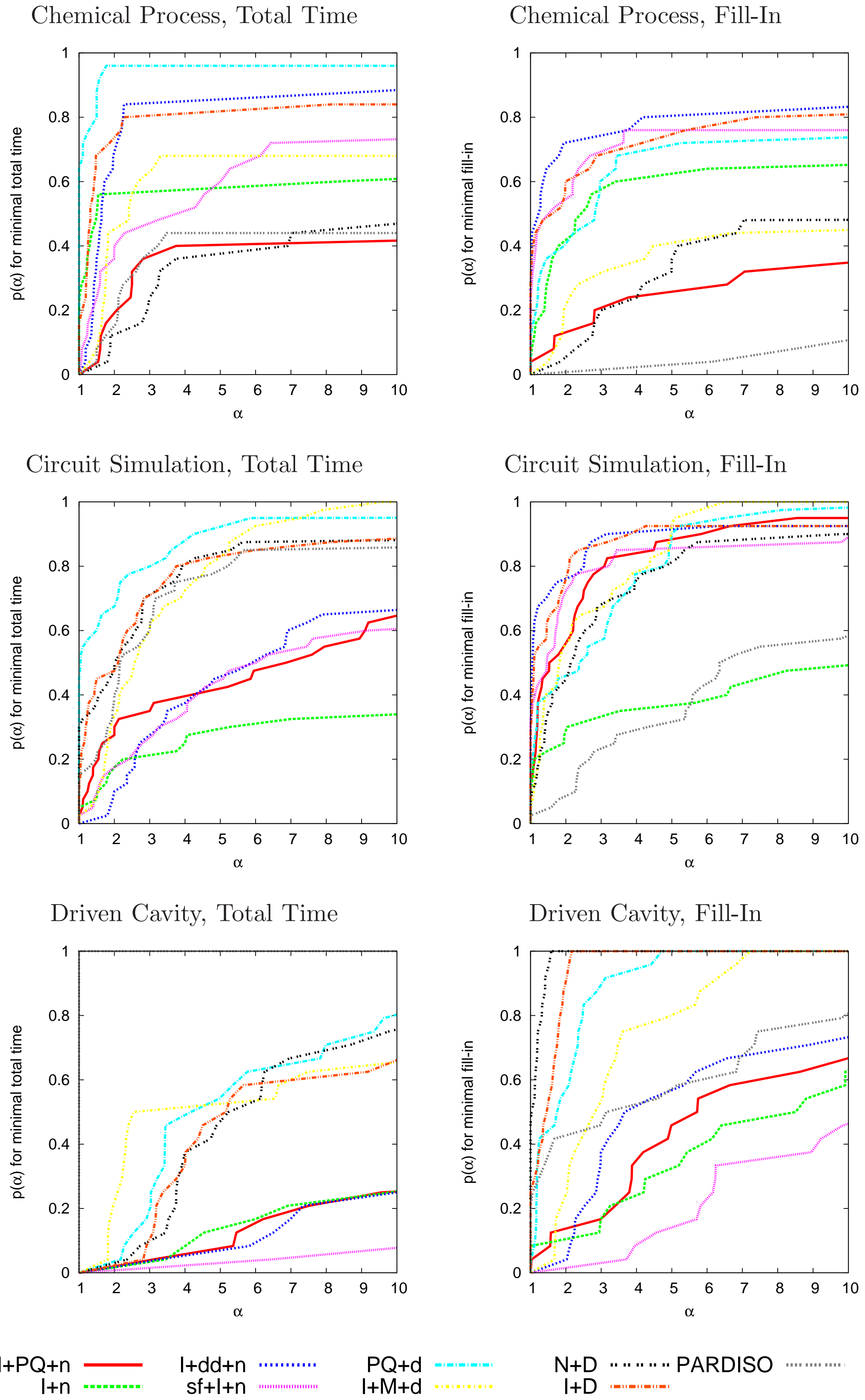


Table 6: Performance profiles for minimal total computation time and corresponding fill-in, part 1.

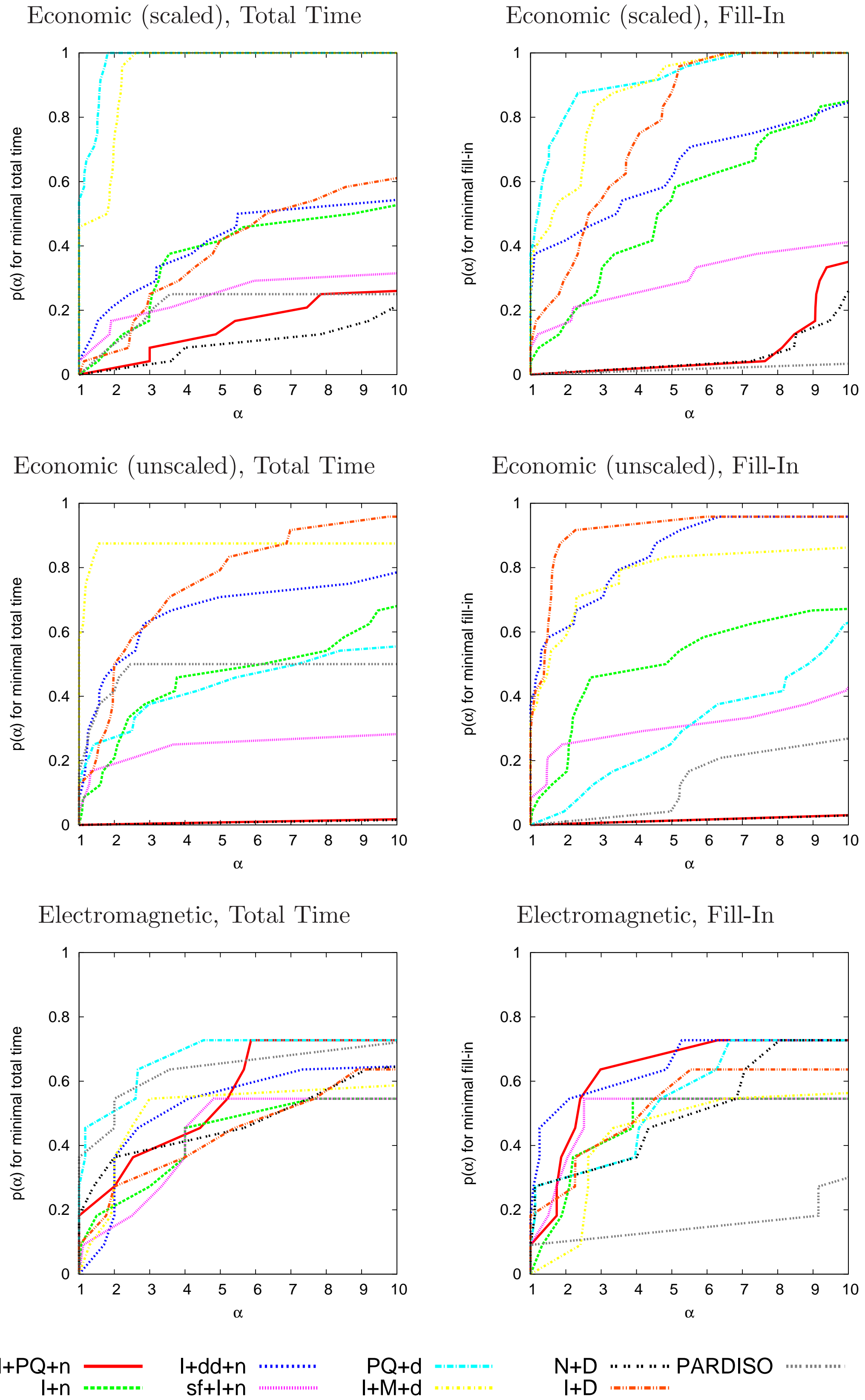


Table 7: Performance profiles for minimal total computation time and corresponding fill-in, part 2.

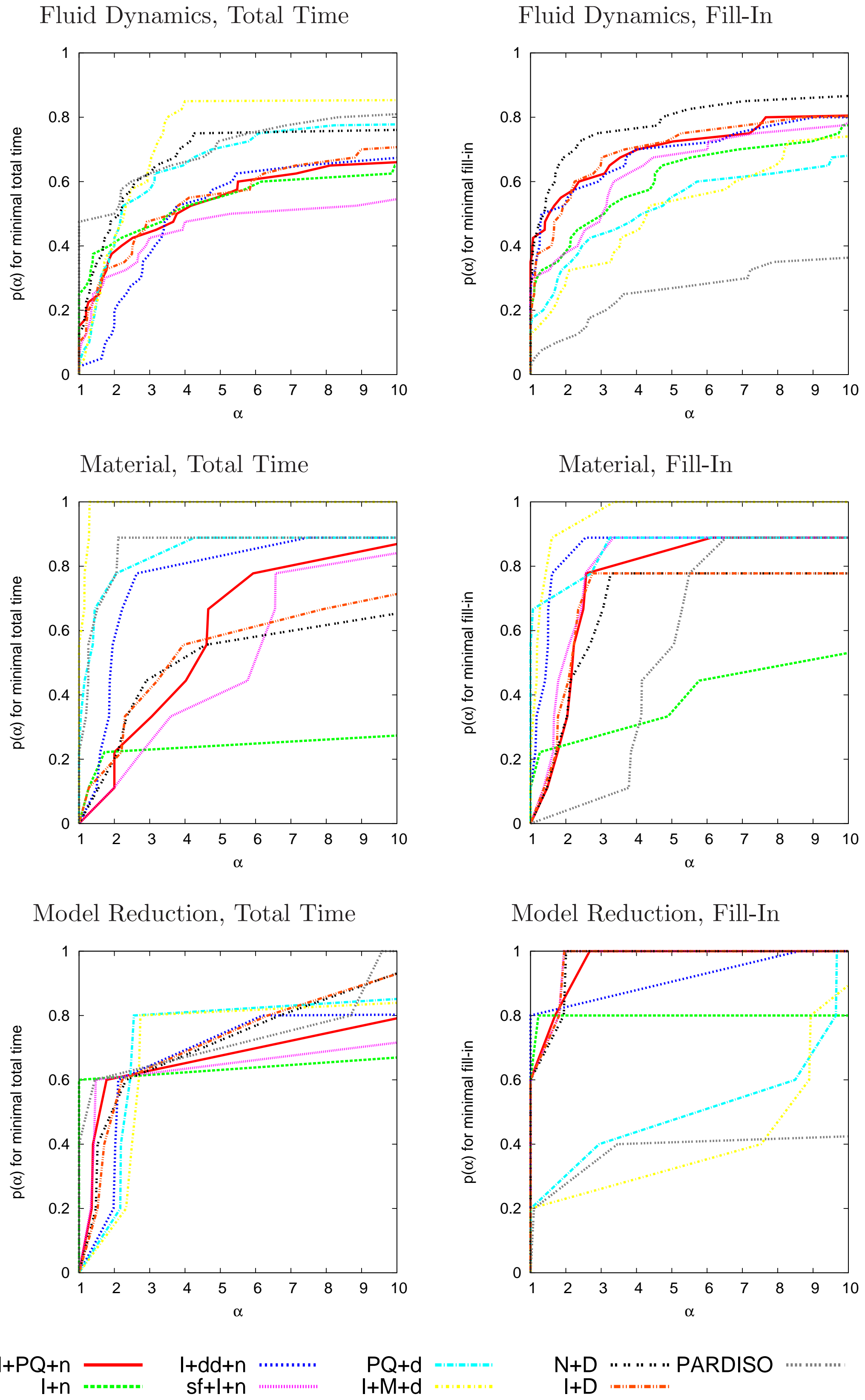


Table 8: Performance profiles for minimal total computation time and corresponding fill-in, part 3.

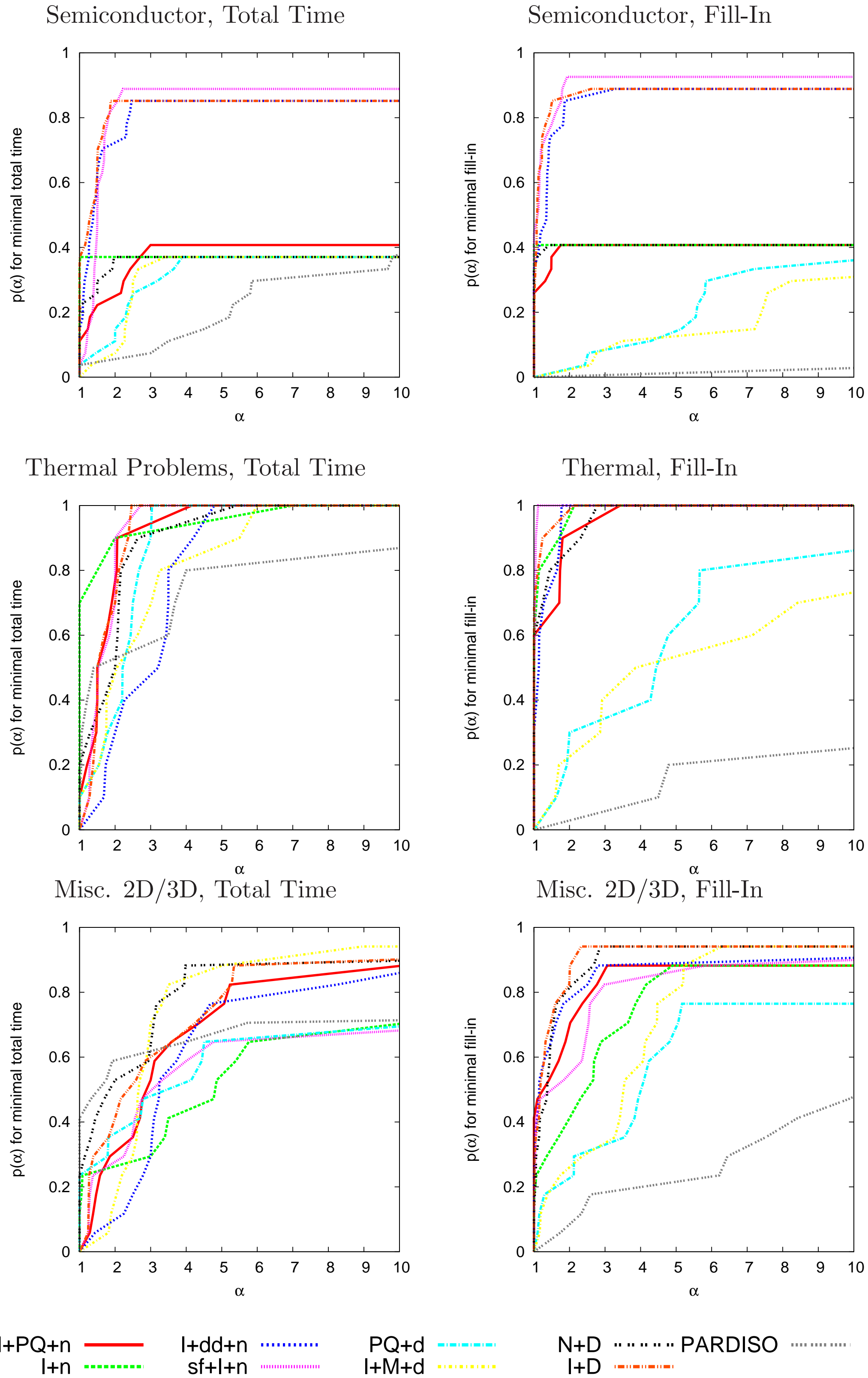


Table 9: Performance profiles for minimal total computation time and corresponding fill-in, part 4.

4.3 Performance Profiles

Performance profiles have become somewhat of a standard for comparing the results of different numerical methods for a particular number of test problems. They are introduced and studied extensively in [8] and have been used for comparing direct methods in [12]. We summarize briefly the approach along the same lines as in [16].

For a set T of problems (in our case linear systems with the matrices investigated) and a set of solvers S (in our case the preconditioners combined with the iterative solver as well as the direct method), we obtain a statistic $s_{ij} > 0$ for each $i \in S$ and $j \in T$. In our case, the statistic will be the minimal total computation time of a particular method and the corresponding fill-in. Generally, the statistic should be chosen in such a manner that small values indicate good properties. For each $j \in T$, we determine

$$s_j^* = \min\{s_{ij} \,|\, i \in S\}$$

which indicates the best possible performance for the problem j amongst all the solvers. For a particular $i \in S$, the list of quotients $\frac{s_{ij}}{s_j^*}$, $j \in T$ indicates how much worse solver i is than the best possible solver. Using these data, we can define the performance profile $p_i(\alpha)$, $\alpha \geq 1$ for every $i \in S$ so that $p_i(\alpha)$ indicates the fraction of problems for which solver i was within a factor of α of the best solver (according to the statistic chosen). For a more formal treatment, consult the references mentioned above. Also note that $\lim_{\alpha\to\infty} p_i(\alpha)$ is the fraction of problems for which solver i was successful. The performance profiles for the minimal total computation time (i.e. using the value of t for which total computation time was minimal) and the corresponding fill-in based on matrix density can be found in Table 5. The same results based on application area are in Tables 6, 7, 8 and 9. Although we will comment these results in greater detail later, it is already apparent that the different methods perform quite differently for different matrix densities and different problem types.

4.4 Sensitivity with Respect to the Threshold Parameter

On order to obtain good computation times for an iterative method in practice, not only do we need to choose the preconditioner wisely, but also the threshold parameter t. As the optimal value of t is generally not known, preconditioners which achieve good computation times for a larger range of t have a significant advantage over those that do not. As before, we define s_j^* to be the the best possible performance for the problem $j \in T$ amongst all the solvers. For each solver $i \in S$ and for each problem $j \in T$, we define $T_{ij}(\alpha)$ to be the set of all threshold parameters t such that the corresponding total computation times for t are within a factor of α of s_j^* and we let $w_{ij}(\alpha)$ be the width of the largest interval contained in $T_{ij}(\alpha)$. If $T_{ij}(\alpha)$ is empty, we set $w_{ij}(\alpha) = -1$. For PARDISO (or any method not depending on a threshold), we set $w_{ij}(\alpha) = \infty$ if the method was successful and -1 otherwise. Hence, $w_{ij}(\alpha)$ will be large if total computation times needed for a preconditioner are within a factor of α to the best possible result s_j^* for a large interval of t. In this case, choosing a good value of t should be easier. Hence, the total computation times need not only be fairly constant in order for $w_{ij}(\alpha)$ to be large, but they need to be sufficiently close to the optimal computation time as well. Let

$$W_i(\alpha) = (w_{ij}(\alpha))_{j\in T}$$

be the list of all interval lengths for a particular solver $i \in S$ sorted decreasingly and define $W_i(\alpha, n)$ as the nth element of this list. The plots of $W_i(\alpha, n)$ for different values

of α, which we will call the sensitivity profile, illustrates the sensitivity of a particular method with respect to the parameter t. The results for $\alpha = 4$ can be found in Tables 10 and 11. We did not plot negative values of $W_i(\alpha, n)$ so that the graph of $W_i(\alpha, n)$ terminates as soon as n is larger than the number of matrices for which successful solves were possible in computation time required. Note that choosing a smaller value α would not necessarily lead to more meaningful plots. In this case, a larger number of solvers would not have computed the solution in the required (more restrictive) total time for a larger number of problems, meaning that the plots of $W_i(\alpha, n)$ not have provided much information. Fortunately, solvers performing well based on $\alpha = 4$ usually performed well for smaller values as well. Hence, choosing $\alpha = 4$ for presenting results is a reasonable compromise between obtaining information on the solver and not being too permissive for the total solve time.

4.5 Variance in the Optimal Threshold Parameter

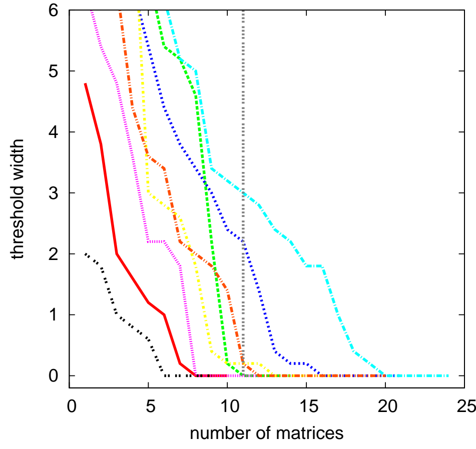
Often it is difficult to choose the dropping parameter t wisely. A common strategy is to determine a good value for t for a particular problem and to use this as the standard. In other words, the same value for t is used for similar problems and slightly smaller or slightly larger values are used for problems considered to be easier or harder, respectively. Such an approach makes sense if the problems being solved are sufficiently similar. The test problems in each of the application areas considered here are probably not similar enough for such an approach to work. Nevertheless, we record the mean values for the treshold parameter t and its standard deviation in each problem set in Table 12. Smaller standard deviations for one method than another should be interpreted to indicate that the strategy described for selecting t is more likely to work one method than the other, maybe not on the entire set constituting one problem area, but perhaps on a subset of more similar problems. However, we do not investigate this approach for choosing t in greater detail.

4.6 Interpretation of the Numerical Results

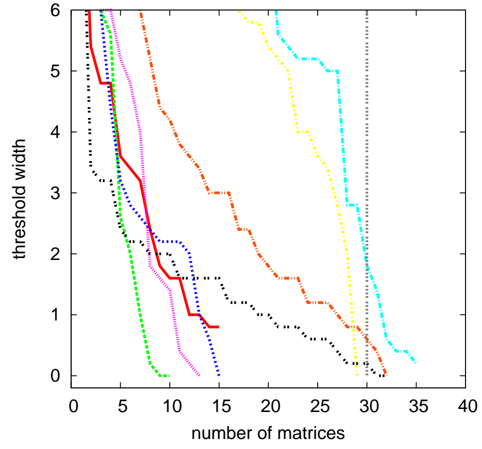
Even quick glance at the numerical results presented so far is sufficient to see that the performance of a particular method is highly dependent upon both the matrix density and the application area. Not surprisingly, the biggest difference lies between the preconditioned iterative methods and the direct solver PARDISO. Whenever no high quality incomplete factorization exists which is significantly sparser than the exact PARDISO factorization, then PARDISO is the best method. In this situation, the additional work that needs to be done to implement the iterative method and dropping during factorization seems to exceed the additional work that needs to be done to compute the exact factorization. This situation occurs in particular for most applications arising from partial differential equations where PARDISO is able to compute relatively sparse exact factorizations. For the other applications, such as the chemical process simulation problems, the economic problems, the semiconductor device problems and to a slightly lesser degree the circuit simulation problems, the iterative methods perform better.

This result is remarkable in the following sense: Until recently, iterative methods were used primarily for problems arising from partial differential equations because good preconditioners could be constructed for these problems and direct methods often required much fill-in. Consequently, iterative methods were often quite competitive when compared with direct solvers, both in terms of fill-in and total compution times. For other problems,

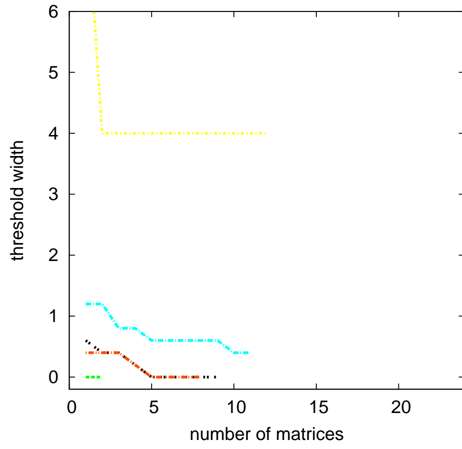
Chemical Process Simulation



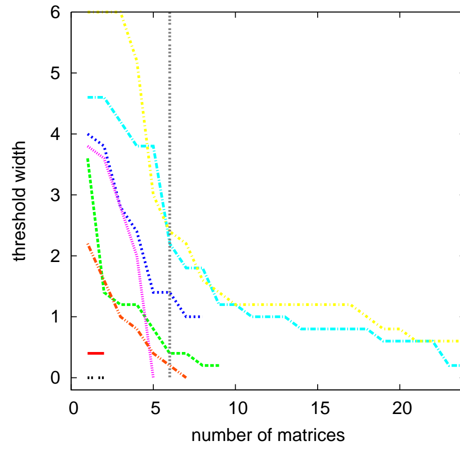
Circuit Simulation Problems



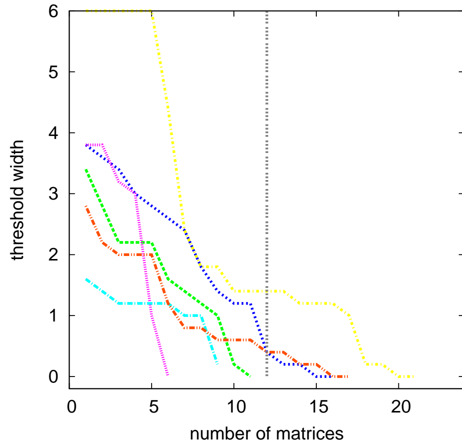
Driven Cavity Problems



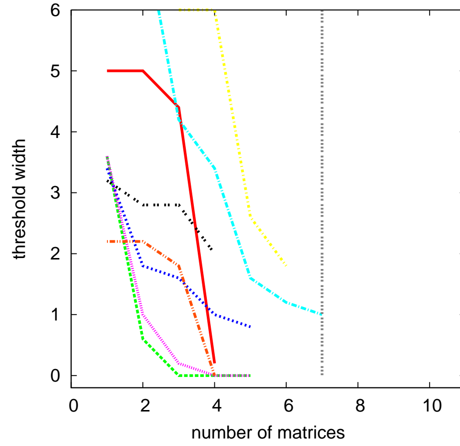
Economic Problems (scaled)



Economic Problems (unscaled)



Electromagnetic Problems



N+PQ+n I+dd+n PQ+d N+D PARDISO
I+n sf+I+n I+M+d I+D

Table 10: Sensitivity profiles for $\alpha = 4$ for minimal total computation time with respect to the threshold parameter, part 1.

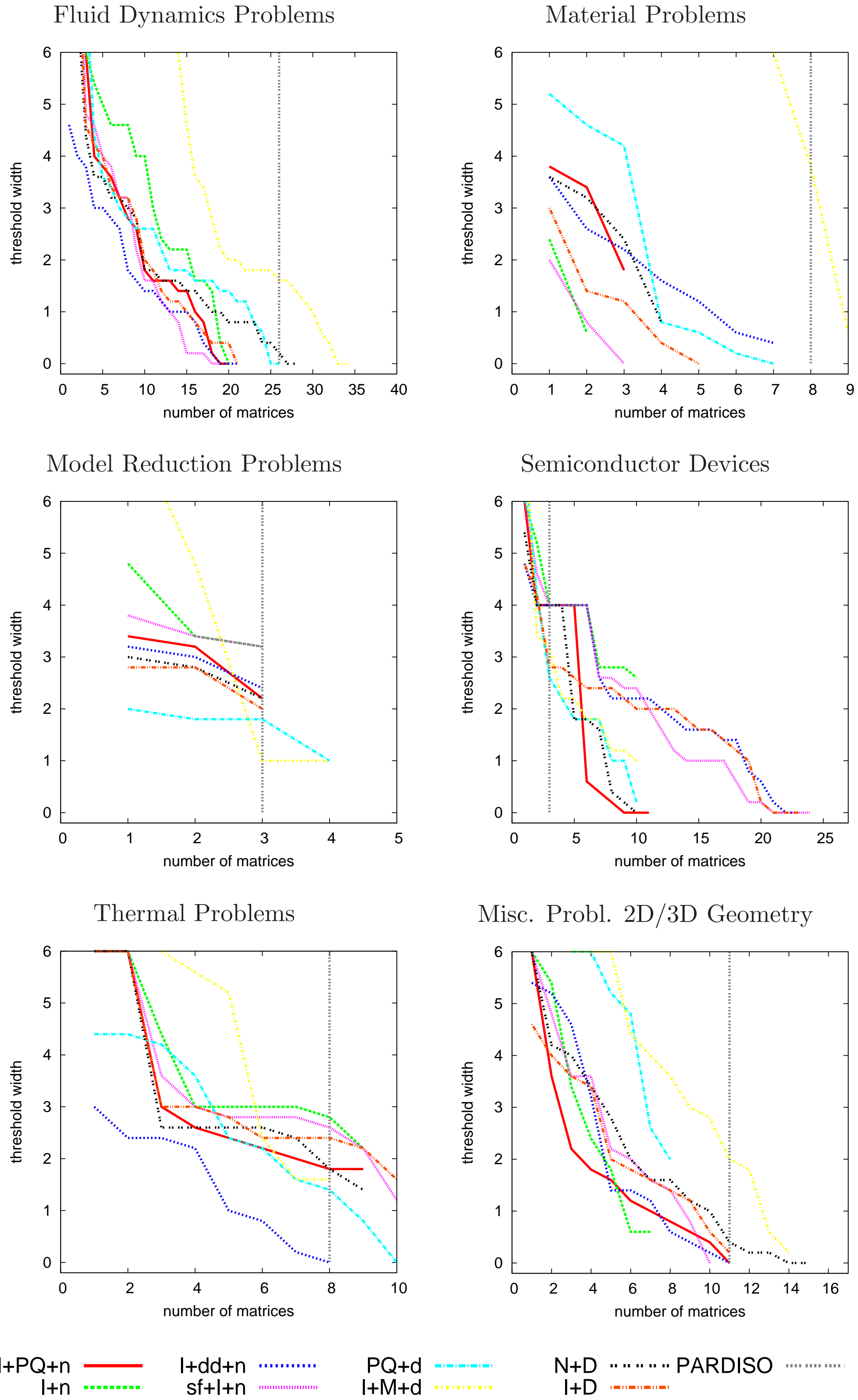


Table 11: Sensitivity profiles for $\alpha = 4$ for minimal total computation time with respect to the threshold parameter, part 2.

Mean of the optimal values of the threshold parameter t

Preprocessing Pivoting	N+PQ n	I n	I+dd n	sf+I n	PQ d	I+M d	N D	I D
Low Density	4.5	2.3	2.8	2.9	3.5	2.5	4.2	3.1
Medium Density	3.4	2.2	2.3	2.5	2.9	1.6	3.5	2.2
High Density	2.6	1.7	2.6	2.4	2.4	1.8	2.3	2.2
Chemical Proc. Sim.	6.3	2.1	4.2	2.6	4.2	3.8	6.3	3.5
Circuit Simulation	5.5	2.9	4.1	4.1	5.0	4.7	5.5	5.3
Driven Cavity Probl.	1.1	1.4	0.9	1.4	1.6	1.0	0.7	0.4
Econ. Probl. (sc.)	3.7	2.2	1.6	1.8	1.8	0.0	3.7	1.3
Econ. Probl. (unsc.)	4.0	1.8	1.7	1.8	4.5	0.4	4.0	1.4
Electromagnetics	1.8	1.4	1.4	1.0	1.9	0.8	1.8	1.7
Fluid Dynamics	2.4	2.1	1.7	2.2	2.3	2.1	2.2	1.6
Material Probl.	4.5	2.2	2.6	2.8	2.8	1.8	4.4	2.7
Model Reduction	2.4	2.5	2.2	2.3	2.4	1.6	2.4	2.2
Semicond. Devices	4.1	1.5	3.9	4.0	1.2	0.6	3.6	3.8
Thermal Probl.	2.3	2.4	2.1	2.1	1.8	2.3	2.1	2.5
Misc. Probl. w/ Geom.	2.2	1.6	1.8	2.0	2.3	1.1	1.9	1.6

Standard deviation of the optimal values of the threshold parameter t

Preprocessing Pivoting	N+PQ n	I n	I+dd n	sf+I n	PQ d	I+M d	N D	I D
Low Density	2.6	1.9	1.9	2.2	2.7	3.2	2.5	2.7
Medium Density	2.6	1.7	1.7	2.1	1.9	2.6	3.0	2.0
High Density	2.5	1.0	2.4	2.0	1.4	1.9	2.6	2.3
Chemical Proc. Sim.	3.0	0.6	3.0	1.6	1.6	3.2	3.2	2.5
Circuit Simulation	2.6	2.6	2.3	2.5	2.8	3.4	2.4	2.9
Driven Cavity Probl.	0.5	0.5	0.5	0.5	0.3	1.0	0.2	0.1
Econ. Probl. (sc.)	0.2	0.8	0.3	0.7	0.3	0.2	0.2	0.3
Econ. Probl. (unsc.)	0.0	0.5	0.6	0.7	1.2	1.0	0.0	0.4
Electromagnetics	0.3	0.5	0.3	0.2	0.9	1.3	0.8	0.7
Fluid Dynamics	2.6	2.4	1.7	2.4	2.2	2.6	3.0	2.0
Material Probl.	1.0	0.3	0.6	0.5	0.5	0.8	1.0	0.6
Model Reduction	1.3	1.0	1.0	1.1	1.5	1.7	1.7	1.5
Semicond. Devices	2.8	0.8	2.0	2.4	0.7	0.8	3.1	2.3
Thermal Probl.	1.5	1.5	1.1	1.8	1.3	2.2	1.5	1.4
Misc. Probl. w/ Geom.	1.0	0.4	0.8	1.0	0.8	1.7	1.0	0.8

Table 12: Mean and standard deviation of the optimal choice of threshold parameter t. Slightly lighter numbers indicate that the method solved between 50% and 80% of the number of problems which the best method was able to solve. Very light numbers indicate that less than 50% of these problems were solved successfully, see Table 4.

	reasonable t known	reasonable t unknown	memory restrictive
Low Density	PQ+d	no recomm.	I+D
Medium Density	I+M+d	no recomm.	I+dd+n, I+D
High Density	PARD.	PARD.	I+D
Chemical Proc. Sim.	PQ+d	PQ+d	I+dd+n
Circuit Simulation	PQ+d	PARD.	I+D, I+dd+n
Driven Cavity Probl.	PARD.	PARD.	N+D
Econ. Probl. (sc.)	PQ+d	PQ+d	PQ+d
Econ. Probl. (unsc.)	I+M+d	I+D, I+M+d	I+D
Electromagnetics	PARD.	PARD.	I+dd+n
Fluid Dynamics	PARD.	PARD.	I+D
Material Probl.	I+M+d, PARD.	PARD.	I+M+d
Model Reduction	PQ+d, I+M+d	PARD.	I+dd+n
Semicond. Devices	sf+I+n, I+D	I+dd+n, I+D	sf+I+n, I+D
Thermal Probl.	I+n	I+n	sf+I+n, I+n
Misc. Probl. w/ Geom.	N+D, I+M+d	I+M+d	I+D, N+D

Table 13: Recommendation for choosing a solver for a particular problem.

iterative methods were not considered suitable as it was difficult if not impossible to find suitable preconditioners. However, advances in both direct methods and preconditioning seems to have reversed the situation somewhat. The direct solver PARDISO is able to produce exact factorizations which are quite sparse for the problems arising from partial differential equations, so that PARDISO is the method of choice for these problems. On the other hand, many of the preconditioners employing recent developments tested here seem to perform quite well for the other application areas. They seem to produce accurate, sparse factorizations which result in quick convergence of the iterative methods, whereas PARDISO often requires significantly more fill-in leading to higher total solve times.

Commenting in greater detail on the numerical results for each and every method and application area is probably not necessary, as the various graphs speak for themselves, but a few comments do seem to be appropriate. The best preconditioners overall are the two ILUPACK preconditioners and the preconditioners I+dd+n and I+D as implemented in ILU++. In terms of reliability, there are no significant differences between all of them, except for the semiconductor problems, where the ILUPACK preconditioners fail significantly more often than the ILU++ preconditioners. These failures result from large errors in the solution. In terms of total solve times, the ILUPACK preconditioners are slightly better than the ILU++ preconditioners, a notable exception being again the semiconductor problems. A part of this advantage is likely due to the differences in implementations, in particularly the choice of programming language and memory management. The fill-in needed to obtain optimal computation time is best for the ILU++ preconditioners. Hence, the overall memory requirements are least for I+dd+n, because this preconditioner requires virtually no additional memory beyond what is needed to store the preconditioner itself. Although I+D requires additional memory during the course of the calculations, it seems that this need has not resulted in more failures. This is probably due to the lower fill-in and the efficient memory management of ILU++. Furthermore, the additional memory is needed only for each level and can be freed after that level has been calculated, so that the memory requirements for I+D may not be that formidable after all. the Tables 10 and 11 indicate that I+M+d is generally least sensitive to variations of the

optimal choice of t. However, the results in Table 12 indicate that the variance in the value for the optimal t is least for I+dd+n. Hence, if PARDISO is not fastest, then generally one of the ILUPACK preconditioners will be, except for semiconductor problems. On the other hand, the ILU++ preconditioners, particularly I+dd+n, are likely the better choice, whenever memory is scarce.

These observations allow for some concrete recommendations for solving large, sparse linear systems with one of the methods discussed in this article. It is clear that there is no universal best method, not even within a particular application area. Having the shortest possible solution times will often determine which method is optimal. However, in certain situations, memory restrictions may require a solver with very modest memory requirements, making sparsity the criterion for optimality. Similarly, if no reasonable estimate for a good value of t is available for a particular preconditioner, which would be expected to perform well, then it may be wise to use a direct solver, even if the computation time is likely to be longer than the computation time of an optimally preconditioned iterative solver. Taking these considerations into account, Table 13 makes a few specific recommendations. These are rules of thumb at best, or perhaps a good starting point for solving a linear system, if no better starting point is known, but they are certainly not golden rules. Certainly, they cannot replace expertise, but they may be a small aid to the novice.

5 Conclusion

The results presented in this article indicate that iterative methods preconditioned with incomplete multilevel LU factorizations have reached a degree of maturity to be able to complete with state-of-the-art direct methods for almost all unsymmetric problems. The underlying assumption is, however, that the “right” factorization and optimal parameter is chosen. In this case, the optimal preconditioner is more reliable and often faster than PARDISO (without iterative refinement). Although this was not tested, using iterative refinement with PARDISO is likely to improve its reliability, but not its speed. Consequently, for a significant number of problems, iterative methods are faster than PARDISO and usually require significantly less memory.

Achieving these optimal results in practice, however, may be quite difficult as the optimal choices for both the preconditioner and threshold parameter are not known. Furthermore, going to the effort of finding the appropriate choices may only be worthwhile if a significant number of similar problems needs to be solved. Nevertheless, whenever solvers using as little memory as possible are needed, often no alternative to iterative solvers exist and results of this article provide a reasonably good starting point for selecting an appropriate preconditioner.

Nevertheless, the recommendations presented here can only be a starting point. Ideally, the preconditioners would not depend on a dropping parameter t or at least it would be adjusted during factorization, so that a poor choice will not lead to failure or extreme deterioration of solve times. The ILUPACK preconditioners go somewhat in this direction as they attempt to keep the error of the inverse factors small. Indeed, these preconditioners seem to be somewhat less sensitive to variations in t than the ILU++ preconditioners, but they still require the user to provide parameters, which when chosen poorly will result in failure. As the preconditioners themselves seem to perform quite well, subsequent developments in this area will hopefully focus on strategies for selecting the threshold parameter t.

References

[1] Matthias Bollhöfer. A robust ILU with pivoting based on monitoring the growth of inverse factors. *Lin. Alg. Appl.*, 338:201–218, 2001.

[2] Matthias Bollhöfer. A robust and efficient ILU that incorporates the growth of the inverse triangular factors. *SIAM J. Sci. Comp.*, 25(1):86–103, 2003.

[3] Matthias Bollhöfer. ILUPACK. `http://www.math.tu-berlin.de/ilupack/`, Aug. 2007.

[4] Matthias Bollhöfer and Yousef Saad. Multilevel preconditioners constructed from inverse-based ILUs. *SIAM J. Sci. Comput.*, 25(2):715–728, 2003.

[5] Edmond Chow, Na Li, and Yousef Saad. Crout versions of ILU for sparse matrices. *SIAM J. Sci. Comput.*, 27(5):1627–1650, 2006.

[6] Edmond Chow and Yousef Saad. Experimental study of ILU preconditioners for indefinite matrices. *J. Comput. Appl. Math*, 86(2):387–414, 1997.

[7] Tim Davis. University of Florida Sparse Matrix Collection. `http://www.cise.ufl.edu/research/sparse/matrices`, `ftp://ftp.cise.ufl.edu/pub/faculty/davis/matrices`, Aug. 2007.

[8] E.D. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, 2002.

[9] Iain S. Duff and Jacko Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM J. Matrix Anal. Appl.*, 20(1):889–901, 1999.

[10] Iain S. Duff and Jacko Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM J. Matrix Anal. Appl.*, 22(4):973–996, 2001.

[11] Iain S. Duff and Stéphane Pralet. Strategies for scaling and pivoting for sparse symmetric indefinite problems. *SIAM J. Matrix Anal. Appl.*, 27(2):313–340, 2005.

[12] Nicholas I. M. Gould, Jennifer A. Scott, and Yifan Hu. A numerical evaluation of sparse solvers for symmetric systems. *ACM Transactions on Mathematical Software*, 33(2):10, June 2007.

[13] Michael Hagemann and Olaf Schenk. Weighted matchings for preconditioning symmetric indefinite linear systems. *SIAM J. Sci. Comput.*, 24(2):403–420, 2006.

[14] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comp.*, 20(1):359–392, 1998.

[15] George Kayrpis. METIS. `http://glaros.dtc.umn.edu/gkhome/views/metis/`, Aug. 2007.

[16] Jan Mayer. A multilevel Crout ILU preconditioner with pivoting and row permutation. To appear in *Numer. Linear Algebra Appl.*

[17] Jan Mayer. Symmetric permutations for I-matrices to delay and avoid small pivots. Preprint.

[18] Jan Mayer. ILUCP: a Crout ILU preconditioner with pivoting. *Numer. Linear Algebra Appl.*, 12(9):941–955, 2005.

[19] Jan Mayer. Alternative weighted dropping strategies for ILUTP. *SIAM J. Sci. Comput.*, 27(4):1424–1437, 2006.

[20] Jan Mayer. ILU++. `http://iamlasun8.mathematik.uni-karlsruhe.de/~ae04/iluplusplus.html`, Aug. 2007.

[21] Markus Olschowska and Arnold Neumaier. A new pivoting strategy for Gaussian elimination. *Lin. Alg. Appl.*, 220:131–151, 1996.

[22] Yousef Saad. ILUT: A dual threshold incomplete LU factorization. *Numer. Linear Algebra Appl.*, 1:387–402, 1994.

[23] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelphia, 2003.

[24] Yousef Saad. Multilevel ILU with reorderings for diagonal dominance. *SIAM J. Sci. Comput.*, 27:1032–1057, 2006.

[25] Olaf Schenk and Klaus Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Journal of Future Generation Computer Systems*, 20(3):475–487, 2004.

[26] Olaf Schenk and Klaus Gärtner. On fast factorization pivoting methods for sparse symmetric indefinite systems. *Elec. Trans. Numer. Anal.*, 23:158–179, 2006.

[27] Olaf Schenk and Klaus Gärtner. PARDISO. `http://www.computational.unibas.ch/cs/scicomp/software/pardiso/`, Aug. 2007.

[28] Olaf Schenk, Klaus Gärtner, and Wolfgang Fichtner. Efficient sparse LU factorization with left-looking strategy on shared memory multiprocessors. *BIT*, 40(1):158–176, 2000.

[29] Miron Tismenetsky. A new preconditioning technique for solving large sparse linear systems. *Linear Algebra Appl.*, 154-156:331–353, 1991.

IWRMM-Preprints seit 2005

Nr. 05/01	Götz Alefeld, Zhengyu Wang: Verification of Solutions for Almost Linear Complementarity Problems
Nr. 05/02	Vincent Heuveline, Friedhelm Schieweck: Constrained H^1-interpolation on quadrilateral and hexahedral meshes with hanging nodes
Nr. 05/03	Michael Plum, Christian Wieners: Enclosures for variational inequalities
Nr. 05/04	Jan Mayer: ILUCDP: A Crout ILU Preconditioner with Pivoting and Row Permutation
Nr. 05/05	Reinhard Kirchner, Ulrich Kulisch: Hardware Support for Interval Arithmetic
Nr. 05/06	Jan Mayer: ILUCDP: A Multilevel Crout ILU Preconditioner with Pivoting and Row Permutation
Nr. 06/01	Willy Dörfler, Vincent Heuveline: Convergence of an adaptive *hp* finite element strategy in one dimension
Nr. 06/02	Vincent Heuveline, Hoang Nam-Dung: On two Numerical Approaches for the Boundary Control Stabilization of Semi-linear Parabolic Systems: A Comparison
Nr. 06/03	Andreas Rieder, Armin Lechleiter: Newton Regularizations for Impedance Tomography: A Numerical Study
Nr. 06/04	Götz Alefeld, Xiaojun Chen: A Regularized Projection Method for Complementarity Problems with Non-Lipschitzian Functions
Nr. 06/05	Ulrich Kulisch: Letters to the IEEE Computer Arithmetic Standards Revision Group
Nr. 06/06	Frank Strauss, Vincent Heuveline, Ben Schweizer: Existence and approximation results for shape optimization problems in rotordynamics
Nr. 06/07	Kai Sandfort, Joachim Ohser: Labeling of n-dimensional images with choosable adjacency of the pixels
Nr. 06/08	Jan Mayer: Symmetric Permutations for I-matrices to Delay and Avoid Small Pivots During Factorization
Nr. 06/09	Andreas Rieder, Arne Schneck: Optimality of the fully discrete filtered Backprojection Algorithm for Tomographic Inversion
Nr. 06/10	Patrizio Neff, Krzysztof Chelminski, Wolfgang Müller, Christian Wieners: A numerical solution method for an infinitesimal elasto-plastic Cosserat model
Nr. 06/11	Christian Wieners: Nonlinear solution methods for infinitesimal perfect plasticity
Nr. 07/01	Armin Lechleiter, Andreas Rieder: A Convergenze Analysis of the Newton-Type Regularization CG-Reginn with Application to Impedance Tomography
Nr. 07/02	Jan Lellmann, Jonathan Balzer, Andreas Rieder, Jürgen Beyerer: Shape from Specular Reflection Optical Flow
Nr. 07/03	Vincent Heuveline, Jan-Philipp Weiß: A Parallel Implementation of a Lattice Boltzmann Method on the Clearspeed Advance Accelerator Board
Nr. 07/04	Martin Sauter, Christian Wieners: Robust estimates for the approximation of the dynamic consolidation problem
Nr. 07/05	Jan Mayer: A Numerical Evaluation of Preprocessing and ILU-type Preconditioners for the Solution of Unsymmetric Sparse Linear Systems Using Iterative Methods

Eine aktuelle Liste aller IWRMM-Preprints finden Sie auf:

www.mathematik.uni-karlsruhe.de/iwrmm/seite/preprints