An Assessment of Incomplete-LU Preconditioners for Nonsymmetric Linear Systems[1]

John R. Gilbert
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304, USA
E-mail: gilbert@parc.xerox.com
Sivan Toledo
School of Computer Science
Tel-Aviv University
Tel-Aviv 69978, Israel
E-mail: sivan@math.tau.ac.il

Keywords: Sparse nonsymmetric linear systems, iterative methods, direct methods, incomplete-LU preconditioners, pivoting.

Edited by: Marcin Paprzycki

Received: November 11, 1999 **Revised:** ???? **Accepted:** ????

We report on an extensive experiment to compare an iterative solver preconditioned by several versions of incomplete LU factorization with a sparse direct solver using LU factorization with partial pivoting. Our test suite is 24 nonsymmetric matrices drawn from benchmark sets in the literature.

On a few matrices, the best iterative method is more than 5 times as fast and more than 10 times as memory-efficient as the direct method. Nonetheless, in most cases the iterative methods are slower; in many cases they do not save memory; and in general they are less reliable. Our primary conclusion is that a direct method is currently more appropriate than an iterative method for a general-purpose black-box nonsymmetric linear solver.

We draw several other conclusions about these nonsymmetric problems: pivoting is even more important for incomplete than for complete factorizations; the best iterative solutions almost always take only 8 to 16 iterations; a drop-tolerance strategy is superior to a column-count strategy; and column MMD ordering is superior to RCM ordering.

The reader is advised to keep in mind that our conclusions are drawn from experiments with 24 matrices; other test suites might have given somewhat different results. Nonetheless, we are not aware of any other studies more extensive than ours.

1 Introduction

Black-box sparse nonsymmetric solvers, perhaps typified by the Matlab backslash operator, are usually based on a pivoting sparse LU factorization. Can we design a more efficient black-box solver that is based on an iterative solver with an incomplete LU preconditioner? This paper shows, using extensive experimental analysis, that the answer is no. An iterative solver with an incomplete LU preconditioner can sometimes be much more efficient than a direct solver in terms of both memory and time. But in most cases, the iterative solver is less less reliable and less efficient than a direct solver.

These conclusions are novel and correct, but one must keep in mind that they are drawn from a finite set of experiments. The conclusions are novel in the sense that no prior paper presented

[1]This research was supported in part by DARPA contract number DABT63-95-C-0087 and by NSF contract number ASC-96-26298.

a systematic study that supports these conclusions. We are aware that our conclusions coincide with long-held viewpoints of some researchers, but these viewpoints were never substantiated by systematic study before. Hence, the novelty lies not in the conclusions themselves, but in the fact that they are supported by evidence. Other researchers hold opposite viewpoints—that preconditioned iterative solvers are more reliable and efficient that direct solvers. These viewpoints are typically based on some theoretical justification and/or on experimental results. We point out that there is not much theory on the convergence rates of *nonsymmetric* preconditioned iterative solvers, and that success with some matrices does not invalidate our conclusions, since our claim that that iterative solvers are less effective in most cases, not in all cases. To summarize, we present an experimental study, and *we draw conclusions from our data*. It is conceivable that a different test suite would have suggested somewhat different conclusions.

Large sparse linear solvers can be classified into three categories. Some solvers are problem specific and are often built into applications. Such solvers exploit structural and numerical properties that typify linear systems arising from a narrow application domain, and many of them use information about the problem that is not part of the linear system (for example, geometric information). The second category of solvers can be described as toolkits (see, for example, [10]). These solvers, often in the form of numerical libraries that are designed to be called from application programs, offer a choice of algorithms that can be combined to create linear solvers. The user must decide which algorithms to use and how to set tuning parameters that these algorithms may have. Typical toolkits provide several iterative solvers and several preconditioners. The third category of solvers are black-box solvers. These solvers solve linear systems using few assumptions on the origins of the systems and little or no guidance from the user. Most sparse direct solvers fall into this category. Problem-specific solvers often achieve high performance but require considerable effort from experts. Black-box solvers cannot always achieve the same level of performance, but are robust and easy to use. Toolkits are somewhere in-between. This paper evaluates incomplete-LU preconditioners only as candidates for inclusion in black-box solvers; it is by now clear that nonsymmetric incomplete-LU preconditioners should be included in toolkits and in some problem-specific applications.

Large sparse nonsymmetric linear systems are often solved by direct methods, the most popular of which is based on a complete sparse LU factorization of the coefficient matrix. Iterative solvers, usually preconditioned Krylov-space methods, are sometimes more efficient than direct methods. Iterative solvers can sometimes solve linear systems with less storage than direct methods, and they can sometimes solve systems faster than direct methods. The efficiency of iterative methods, in terms of both space and time, depends on the preconditioner that is used. In this paper we focus on a popular class of so-called general-purpose preconditioners, those based on incomplete LU factorization with or without partial pivoting. We do not consider other classes of general-purpose preconditioners, such as those based on sparse approximate inverses (see, for example, Grote and Huckle [8]), and algebraic multilevel solvers (see, for example, Shapira [12]). We also do not consider domain-specific preconditioners, such as domain-decomposition preconditioners for linear systems arising from discretizations of PDE's.

Incomplete-factorization preconditioners are constructed by executing a sparse factorization algorithm, but dropping some of the fill elements. Elements can be dropped according to numerical criteria (usually elements with a small absolute value), or structural criteria (e.g., so-called levels of fill)[2]. Since some of the fill elements are dropped, the resulting factorization is sparser and takes less time to compute than the complete factorization. If this factorization preconditions the linear system well, it enables the construction of an efficient iterative solver. We have implemented an algorithm that can construct such preconditioners. On some matrices, the resulting preconditioners are more than 10 times sparser than a complete factorization, and the iterative solution

[2]We did not include level-of-fill dropping criteria for two reasons. First, we felt that the resulting preconditioners would be less effective and robust than those based on numerical dropping criteria. Second, level-of-fill criteria are *more difficult and computationally expensive* to implement in a pivoting factorization than numerical criteria since the level of every fill element must be kept in a data structure.

is more than 5 times faster than a direct solution. We plan to make our implementation, which can be used alone or as part of PETSc (a portable extensible toolkit for scientific computation [1]), publicly available for research purposes.

This strategy, however, can also fail. The algorithm can fail to compute a factorization due to a zero pivot, or it can compute a factorization that is unstable or inaccurate, which prevents the solver from converging. In other cases, the preconditioner can enable the iterative solution of the system, but without delivering the benefits that we expect. The running time can be slower than the running time of a direct solver, either because the iteration converges slowly or because the incomplete factorization is less efficient than a state-of-the-art complete factorization. The solver may need more space than a direct solver if the algorithm fails to drop many nonzeros, especially since an iterative solver cannot release the storage required for the matrix and needs storage for auxiliary vectors.

We have conducted extensive numerical experiments to determine whether incomplete factorizations can yield preconditioners that are reliable and efficient enough to be used in a black-box nonsymmetric linear solver. Our test cases are nonsymmetric linear systems that have been used to benchmark sparse direct solvers; all of them can be solved by complete sparse LU factorization with partial pivoting. The matrices range in size from about 1,100 to 41,000 rows and columns, and from about 3,700 to 1,600,000 nonzeros. Our main conclusion is that incomplete factorizations are *not* effective enough to be used in black-box solvers, even with partial pivoting. That is not to say that incomplete factorizations never produce effective preconditioners. In some cases they do. But in many cases state-of-the-art incomplete factorizations do not yield efficient preconditioners. Furthermore, in many other cases the resulting preconditioner is effective only within a small range of numerical dropping thresholds, and there are currently no methods for determining a near-optimal threshold. Therefore, current state-of-the-art incomplete factorizations cannot be used as preconditioners in iterative solvers that can be expected to be about as reliable and efficient as current direct solvers.

Our incomplete LU factorization algorithms are quite similar to Saad's ILTUP [11], but employ some additional techniques, which are described in Section 2. We describe our experimental methodology in Section 3. The discussion explains the structure of the experiments, the test matrices, and the hardware and software that were used. A summary of our experimental results is presented in Section 4. We discuss the results and present our conclusions in Section 5.

2 Pivoting Incomplete LU Factorizations

This section describes our algorithm for incomplete LU factorization with partial pivoting. The algorithm is similar to Saad's ILUTP [11], but with some improvements.

Our algorithm is a sparse, left-looking, column-oriented algorithm with row exchanges. The matrix is stored in a compressed sparse-column format, and so are L and U. The row permutation is represented by an integer vector.

At step j of the algorithm, sparse column j of A is unpacked into a full zero column v. Updates from columns 1 through $j-1$ of L are then applied to v. These updates collectively amount to a triangular solve that computes the jth column of U, and a matrix-vector multiplication that computes the jth column of L. The algorithm determines which columns of L and U need to update v, as well as an admissible order for the updates, using a depth-first search (DFS) on the directed graph that underlies L. This technique was developed by Gilbert and Peierls [7].

Once all the updates have been applied to v, the algorithm factors v, using threshold partial pivoting. Specifically, the algorithm searches for the largest entry v_m in v_L, the lower part of v (we use v_U to denote the upper part of v). If $|v_d| > \tau|v_m|$, where $0 \le \tau \le 1$ is the *pivoting threshold* and v_d is the diagonal element in v, then we do not pivot. Otherwise we exchange rows d and m. (In the experiments below, we use either $\tau = 1$, which is ordinary partial pivoting, or $\tau = 0$, which amounts to no pivoting). The exact definition of a diagonal element in this algorithm is explained later in this section.

After the column is factored, we drop small elements from v_L and v_U. We never drop elements that are nonzero in A.[3] The algorithm can drop elements using one of two different criteria: (1) the algorithm can drop all but the largest k elements in v_U and the largest k elements in v_L, or (2) the algorithm can drop from v_U all the elements that are smaller[4] than $\delta \max_{i \in U} |v_i|$, and from v_L the elements that are smaller than $\delta \max_{i \in L} |v_i|$, where δ is the *drop threshold*. When we drop elements using a drop threshold δ, we use the same value of δ for all the columns. When we drop elements using a fill count k, we set k separately for each column. The value of k for a column j is a fixed multiple of the number of nonzeros in the jth column of A. We refer to this method as a *column-fill-ratio* method. After elements have been dropped, the remaining elements of v are copied to the sparse data structures that represent L and U and the algorithm proceeds to factor column $j+1$.

Our dropping rules differ somewhat from Saad's ILUT and ILUTP, in that we do not drop small elements of U during the triangular solve. Doing so would require us to base the drop threshold on the elements of A_j rather than on the elements of U_j, which we prefer. Also note that we compute the absolute drop threshold for a column separately for v_L and for v_U. We expect separate thresholds to give relatively balanced nonzero counts for L and for U, which is difficult to guarantee otherwise since their scaling is often quite different.

Our algorithm uses one more technique, which we call *matching maintenance* that attempts to maintain a trailing submatrix with a nonzero diagonal. The technique is illustrated in Figure 1. Before we start the factorization, we compute a row permutation that creates a nonzero diagonal for the matrix using a bipartite perfect-matching algorithm (this algorithm returns the identity permutation when the input matrix has a nonzero diagonal). When the algorithm exchange rows (pivots), the nonzero diagonal can be destroyed. For example, if in column 1 the algorithm exchanges rows 1 and i (in order to pivot on A_{i1}), and if A_{1i} (which moves to the diagonal) is zero, then we may encounter a zero diagonal element when we factor column i. The element A_{1i} will certainly be filled, since both A_{11} and A_{ii} are nonzero. Therefore, whether we encounter a zero diagonal element or not depends on whether A_{1i} is dropped or not after it is filled. Since A_{1i} will fill, our technique simply marks it so that it is not dropped even if it is small. In effect, we update the perfect matching of the trailing submatrix to reflect the fact that the diagonal of column i is now A_{1i} instead of A_{ii}, which is now in U. If we exchange row i with another row, say l, before we factor column i, we will replace A_{1i} by A_{li} as the diagonal element of column i. This marked diagonal element is not dropped even if we end up pivoting on another element in column i, say A_{ij}, because its existence ensures that the diagonal element in column j will be filled in. The resulting diagonal elements may be small, but barring exact cancellation they prevent zero pivots, at the cost of at most one fill per column.

The goal of the matching maintenance is to prevent structural zero pivots at the cost of one fill element per column. Our experiments show, however, that in very sparse factorizations such as with $\tau = 1$ (which we denote by ILU(0)), exact *numerical* cancellations are common even when this technique is employed. When we replace the numerical values of the matrix elements with random values, the factorizations do not break down. This experiment shows that the technique does indeed prevent structural breakdowns. We were somewhat surprised that exact numerical cancellations are so common in practice, even when structural breakdown is prevented. It remains an open problem to find a similarly inexpensive way to guarantee against exact numerical breakdown.

Before concluding this section, we would like to comment on two techniques that are employed in state-of-the-art complete LU factorization codes but that are not included in our incomplete LU code. The first technique is called symmetric pruning [6]. This technique exploits structural symmetry by pruning the graph that the DFS searches for updating columns. The correctness of symmetric pruning depends on a complete factorization, so we could not use pruning in our code. The second technique is the exploitation of nonsymmetric supernodes [3] to improve the temporal locality in the factorization (and hence reduce cache misses). Maintaining supernodes

[3]We decided not to drop original nonzeros because we felt that dropping them might compromise the robustness of the preconditioner, but in some cases dropping original nonzeros may improve the efficiency of the preconditioner.

[4]All comparisons discussed in this section are of absolute values.

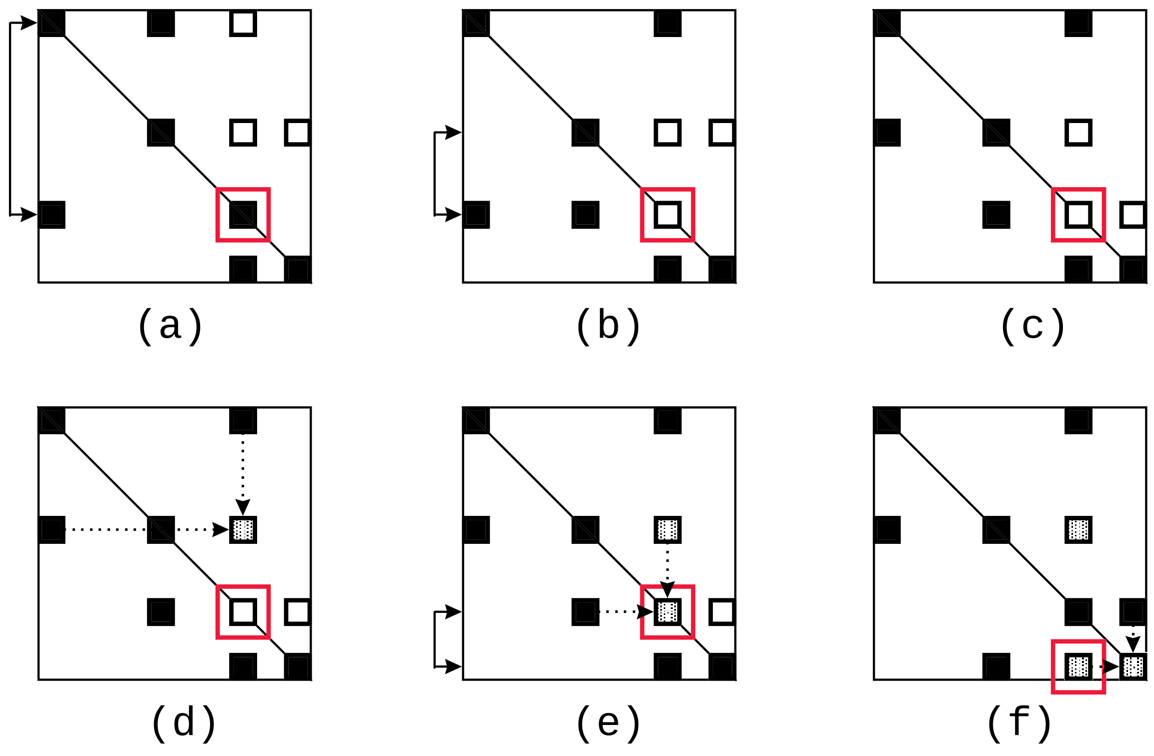

Figure 1: An example of the matching-maintenance method. Nonzero elements are represented by full squares, and zero elements by empty squares. A row exchange places a zero on the diagonal (a). This zero is marked as a diagonal element that must not be dropped, denoted by the enclosing (red) square. Another row exchange moves another zero to the diagonal (b). The new nonzero is marked and the previous one, which is now in U, is unmarked (c). The triangular solve fills in the first zero element, which is not dropped since we do not drop elements in U before the column factorization is complete (d). This fill element plus an original nonzero now cause the diagonal element to fill (e). A row exchange is performed, moving the element just filled off the diagonal. But since it is marked, it is not dropped (f), which ensures that the diagonal element in the last row will fill.

in an incomplete factorization requires a restriction on the dropping rules (a supernodal algorithm would need to drop or retain entire supernode rows). This restriction would have increased the density of the factors, and we estimated that the increased density would offset the savings in running time gained from supernodes. Still, this technique could perhaps enhance performance on some matrices.

We have implemented this algorithm as a modification to the GP code [7]. The modifications are mostly restricted to the driver subroutine and to the column factorization subroutine. The subroutines that perform the DFS and update the current column are essentially unmodified. (We had to slightly modify all the routines in order to implement a column ordering mechanism). The perfect-matching code is by Pothen and Fan [9].

3 Experimental Methodology

This section describes our experimental methodology. We describe the structure of the experiments, the test matrices that we use, and the software and hardware platforms that we used to carry out the experiments. The experiments are summarized in Table 1.

Structure of the experiments

Our experiments compare a direct sparse LU solver with partial pivoting, SuperLU [3], with an iterative solver. We used a transpose-free quasi-minimum-residual (TFQMR) Krylov-space method with the pivoting incomplete LU preconditioner described above (see Saad [11], for example, for background on the Krylov-space methods discussed in this paper). For each matrix A we construct a random solution vector x (with elements uniformly distributed between 0 and 1), and multiply A by x to form a right-hand side b. We then solve the resulting linear system using SuperLU, keeping track of the total solution time and the norm of the residual $A\hat{x} - b$, where $\hat{x}$ is the computed solution. We do not use iterative refinement. We then solve the same system several times using the Krylov-space iterative method with an incomplete-LU preconditioner, each time with a different value of the drop threshold. When the incomplete factorization breaks down due to zero pivots, we do not proceed with the iterative solver at all.

We chose TFQMR based on initial experiments that compared TFQMR, stabilized bi-conjugate gradients (BICGSTAB), and generalized minimum residual (GMRES), the last with restarts every 10, 20, and 40 iterations. These experiments, which are not reported here, showed that the overall performance and robustness of TFQMR and BICGSTAB are quite similar, with GMRES being less efficient. Since our goal is to evaluate iterative solvers as candidates for general-purpose black-box solvers, and not to compare different Krylov-space methods, we picked one solver for the experiments reported here.

The stopping criteria for the iterative solver are as follows. Convergence is defined as a residual whose 2-norm is at most 10^4 times the norm of the solution computed by SuperLU. While arbitrary, this choice reflects the fact that the accuracy achieved by a direct solver is often not needed in applications, while at the same time tying the required solution to the condition number of the system. Divergence is defined as a residual that grows by a factor of more than 10^4 relative to the initial residual. The solver also stops when the total solution time is more than 10 times the SuperLU solution time. We test for convergence after one iteration, 2, 4, 8, 16, and then every 16 iterations. This procedure reduces the overhead of convergence testing, while preventing the solver from iterating too many times when convergence is rapid. The convergence-testing subroutine computes true residuals.

Our time limit criterion and our convergence criterion cannot be implemented in applications, since they require the direct solution of the system. But they allow us to compare the iterative solver to the direct solver effectively without wasting too much computer time. (Even so, the experiments took about two weeks of computer time to complete.)

For each linear system, we ran four sets of experiments, two with partial pivoting (in both the complete and incomplete factorization), and two with no pivoting (sometimes called diagonal pivoting). The reference residual and running time used in the stopping criteria are always the ones from the SuperLU solver with partial pivoting. In two sets of experiments, one with and one without pivoting, we tested the drop-threshold preconditioner. In each set, we ran a direct solver and 33 iterative solvers, in which the drop threshold τ in the incomplete factorizations is set at $2^{-32}, 2^{-31}, \ldots, 2^{-1}, 1$. In the other two sets, also one with and one without pivoting, we tested column-fill-ratio preconditioners. We tested fill ratio $32, 16, 8, 4, 2,$ and 1. (A fill ratio r means that when a column of A has n nonzeros, the corresponding columns of L and U each retain their largest rn elements plus the diagonal element and all the original A elements. Because the original nonzeros are never dropped, and because some columns may not fill by a factor of r, the total number of nonzeros in $U + L$ may be somewhat smaller or larger than $r\mathrm{NNZ}(A)$.)

Most of the experiments were carried out using a column multiple-minimum-degree (MMD) ordering of the matrices, but we did run one set of experiments using a reverse-Cuthill-McKee (RCM) ordering. In this set, whose goal was to allow us to compare different orderings, we tested each matrix with two column-fill-ratio preconditioners, with ratios of 1 and 2.

We also ran three sets of experiments using symmetric-positive-definite (SPD) matrices. The first set was identical to the pivoting drop-threshold experiments carried out with nonsymmetric matrices. The other two sets compared an iterative solver specific to SPD matrices, denoted in Table 1 as CG+ICC, with a nonsymmetric iterative solver. These experiments are described more fully in Section 4.

Test Matrices

We performed the bulk of the experiments on a set of 24 test matrices, listed in Table 2. The ta-

Experiment	Figures	Method	Number of Matrices	Type of Matrices	Ordering	Pivoting	Drop thresholds	Col fill ratios
I	2,3	TFQMR+ILU	24	NS	MMD on A^TA	Y	$2^{-32}, 2^{-31}, \ldots, 2^{-1}, 1$	
II	—	TFQMR+ILU	24	NS	MMD on A^TA	Y		$32, 16, 8, 4, 2, 1$
III	2,3	TFQMR+ILU	24	NS	MMD on A^TA	N	$2^{-32}, 2^{-31}, \ldots, 2^{-1}, 1$	
IV	—	TFQMR+ILU	24	NS	RCM on A	Y		$2, 1$
V	4	TFQMR+ILU	12	SPD	MMD on A^TA	Y	$2^{-32}, 2^{-31}, \ldots, 2^{-1}, 1$	
VI	4	TFQMR+ILU	12	SPD	MMD on A^TA	N	$2^{-32}, 2^{-31}, \ldots, 2^{-1}, 1$	
VII	—	QMR+ILU	6	SPD	RCM on A	N	$2^{-16}, 2^{-15}, \ldots, 2^{-1}, 1$	
VIII	—	CG+ICC	6	SPD	RCM on A	N	$2^{-16}, 2^{-15}, \ldots, 2^{-1}, 1$	

Table 1: A summary of the experiments reported in this paper. The table shows the iterative and preconditioning methods that are used in each set of experiments, the number of matrices and their type (general nonsymmetric, NS, or symmetric positive definite, SPD), the ordering of the matrices, whether pivoting was used, and the parameters of the incomplete-LU preconditioners. The first six sets of experiments were carried out using our own incomplete-LU implementation. The last two experiments, VIII and IX, were carried out using Matlab.
Figures 2, 3, and 4 give detailed data from some of the experiments, the other results are described in the main text.

ble also lists the most important structural and numerical characteristics of the matrices, as well as whether pivoting was necessary for the complete and incomplete factorizations. The matrices are mostly taken from the Parallel SuperLU test set [4], where they are described more fully. The most important reason for choosing this set of matrices (except for availability) is that this is essentially the same set that is used to test SuperLU, which is currently one of the best black-box sparse nonsymmetric solvers. Therefore, this test set allows us to fairly assess whether preconditioned iterative methods are appropriate for a black-box solver.

We have also used a set of 12 symmetric positive-definite matrices in some experiments. Six of these matrices are from the Harwell-Boeing matrix collection and were retrieved from Matrix-Market, an online matrix collection maintained by NIST[5]. These include four structural engineering matrices (bcsstk08, bcsstk25, bcsstk27, bcsstk28), a power system simulation matrix (1108_bus), and a finite-differences matrix (gr_30_30). The other six matrices are image processing matrices contributed by Joseph Liu (den090, dis090, spa090, den120, dis120, spa120).

Software and Hardware Platforms

We used several mathematical libraries to carry out the experiments. The experiments were performed using calls to PETSc 2.0[6], an object-oriented library that implements several iterative linear solvers as well as numerous sparse matrix primitives [1]. PETSc is implemented in C and makes calls to the Basic Linear Algebra Subroutines (BLAS) to perform some operations on dense matrices and on vectors. PETSc includes several preconditioners, but it does not include a pivoting incomplete-LU preconditioner. We therefore added to PETSc two interfaces that call other libraries. The first interface enables PETSc to use the SuperLU library to order and factor sparse matrices. The second interface enables PETSc to use our modified version of the GP library to compute complete and incomplete LU factorizations.

SuperLU is a state-of-the-art library for sparse LU factorization with partial pivoting [3]. SuperLU achieves high performance by using a supernodal panel-oriented factorization, combined with other techniques such as panel DFS with symmetric pruning [6] and blocking for data reuse. It is implemented in C and calls the level 1

[5]Available online at `http://math.nist.gov/MatrixMarket`.

[6]Available online from `http://www.mcs.anl.gov/petsc`.

Matrix	N	NNZ	Structural Symmetry	Numerical Symmetry	Diagonal Dominance	Nonpivoting Direct	Nonpivoting Itererative
gre_1107	1107	5664	0.20	0.20	-1.0e+00		
orsirr_1	1030	6858	1.00	0.50	2.9e-04	Y	Y
mahindas	1258	7682	0.03	0.01	-1.0e+00	Y	
sherman4	1104	3786	1.00	0.29	2.0e-04	Y	Y
west2021	2021	7310	0.00	0.00	-1.0e+00		
saylr4	3564	22316	1.00	1.00	-6.1e-07	Y	Y
pores_2	1224	9613	0.66	0.47	-1.0e+00	Y	Y
extr1	2837	10969	0.00	0.00	-1.0e+00		
radfr1	1048	13299	0.06	0.01	-1.0e+00		
hydr1	5308	22682	0.00	0.00	-1.0e+00		
lhr01	1477	18428	0.01	0.00	-1.0e+00		
vavasis1	4408	95752	0.00	0.00	-1.0e+00		
rdist2	3198	56834	0.05	0.00	-1.0e+00		
rdist3a	2398	61896	0.15	0.01	-1.0e+00		
lhr04	4101	81067	0.02	0.00	-1.0e+00		
vavasis2	11924	306842	0.00	0.00	-1.0e+00		
onetone2	36057	222596	0.15	0.10	-1.0e+00		
onetone1	36057	335552	0.10	0.07	-1.0e+00		
bramley1	17933	962469	0.98	0.73	-1.0e+00	Y	Y
bramley2	17933	962537	0.98	0.78	-1.0e+00	Y	Y
psmigr_1	3140	543160	0.48	0.02	-1.0e+00		
psmigr_2	3140	540022	0.48	0.00	-1.0e+00	Y	
psmigr_3	3140	543160	0.48	0.01	-1.0e+00		
vavasis3	41092	1683902	0.00	0.00	-1.0e+00		

Table 2: The nonsymmetric matrices that are used in our experiments. The table shows the order N and number of nonzeros (NNZ) of the matrices, structural and numerical symmetry, diagonal dominance, and whether the matrices require pivoting in direct and iterative factorizations. The structural symmetry is the fraction of nonzeros whose symmetric matrix elements are also nonzeros, the numerical symmetry is the fraction of nonzeros whose symmetric elements have the same numerical value, and the diagonal dominance is defined as $\min_{i=1\ldots N}(|A_{ii}|/\sum_{j\neq i}|A_{ij}|) - 1$, so a matrix with nonnegative diagonal dominance is diagonally dominant, and a matrix with diagonal dominance -1 has a zero on the diagonal.

and 2 BLAS to perform computations on vectors and on dense submatrices. GP is a library for sparse LU factorization with partial pivoting [7]. GP is column oriented (that is, it does not use supernodes or panel updates). It uses column DFS, but no symmetric pruning. We modified GP to add the capability to compute incomplete factorizations with partial pivoting as explained in Section 2. GP is written in Fortran 77, except for some interface routines that are written in C.

We used the Fortran level-1 and level-2 BLAS. We used PETSc version 2.0.15. In SuperLU, we used the following optimization parameters: panels of 10 columns, relaxed supernodes of at most 5 columns, supernodes of at most 20 columns, and 2D blocking for submatrices with more than 20 rows or columns.

We ran the experiments on a Sun ULTRA Enterprise 1 workstation running the Solaris 2.5.1 operating system. This workstation has a 143 MHz UltraSPARC processor and 320 Mbytes of main memory. The processor has a 32 Kbytes on-chip cache, a 512 Kbytes off-chip cache, and a 288-bit-wide memory bus.

We used the Sunpro-3.0 C and Fortran 77 compilers, with the `-xO3` optimization option for C and the `-O3` optimization option for Fortran. Some driver functions (but no computational kernels) were compiled with the GCC C compiler version 2.7.2 with optimization option `-O3`.

4 Experimental Results

This section summarizes our results. This summary is quite long, and it is supported by many graphs that contain substantial amounts of information. This is a result of the complexity of the underlying data. We found that it was not possible to summarize the experiments concisely because each matrix or small group of matrices exhibited a different behavior. This complexity itself is part of our results, and we attempt to include enough information for readers to gauge it.

We begin with a broad classification of the matrices into those that require pivoting and those that do not. We then discuss each group separately. While most of the experiments were carried out with a column multiple-minimum-degree (MMD) ordering, we describe one set of experiments whose goal is to compare MMD to reverse-Cuthill-McKee (RCM) ordering. We also compare drop-threshold and column-fill-ratio factorizations with similar amounts of fill. We conclude the section with a discussion of two sets of experiments with symmetric-positive-definite matrices, whose goal is to determine whether the difficulties we encountered with the nonsymmetic matrices were due to the properties of the matrices, of the more general nonsymmetric solver, or of the incomplete-factorization paradigm itself.

General Classification of Matrices

In our experiments, matrices that are more than 50% structurally symmetric did not require pivoting for either the direct or the preconditioned iterative solvers. Matrices that are less than 50% structurally symmetric generally require pivoting for both the direct and the preconditioned iterative solvers, with two exceptions: mahindas and psmigr_2 (3% and 48% structurally symmetric, respectively). A nonpivoting direct solver worked on these two matrices (although the solutions produced were significantly less accurate than solutions obtained with pivoting factorizations), but the iterative solvers converged only with pivoting preconditioners. In both cases the nonpivoting iterative solver detected divergence and stopped. The 2-norms of the forward errors were about 7 orders of magnitude larger with the nonpivoting direct solver than with the pivoting solver for mahindas, and 5 orders of magnitude larger for psmigr_2. The 2-norms of the residuals were also larger by similar factors.

We believe that the 50% structural symmetry cutoff point for the need to pivot is significantly influenced by the set of test matrices that we used. We believe that there are matrices that arise in applications with more than 50% structural symmetry that do not require pivoting and matrices with less than 50% structural symmetry that do require pivoting.

Nonpivoting Factorizations

Figure 2 summarizes the results of experiments I and III the 6 matrices did not require a pivoting preconditioner. On 5 of these 6 matrices, the iterative solver with the best drop-threshold preconditioner was faster than SuperLU. On 3 of the

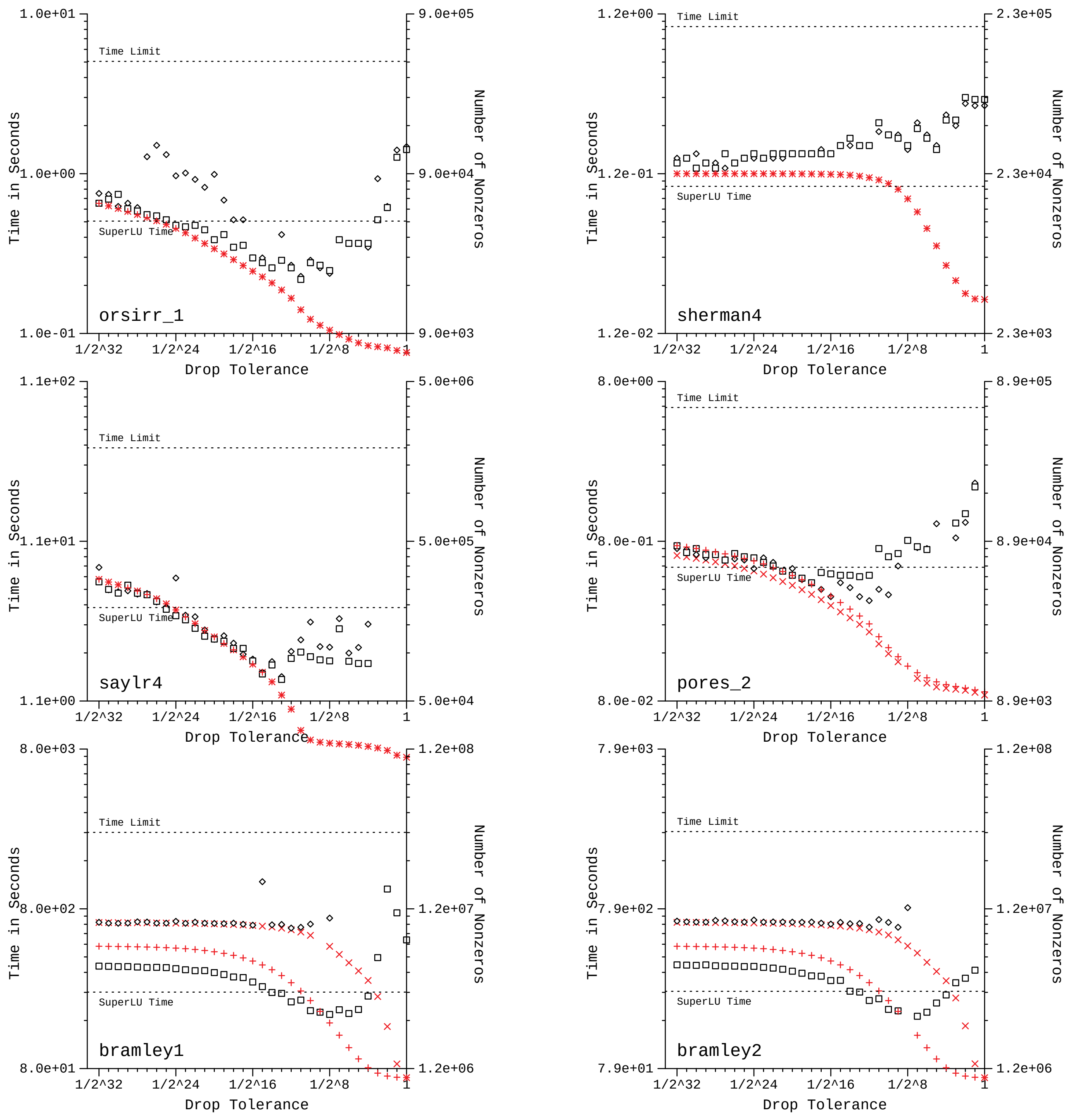

Figure 2: Experiments I and III. Running times (factorization+iteration) and nonzero counts for matrices that can be incompletely factored without pivoting, as a function of the drop threshold. The running times with pivoting are denoted by diamonds, and the running times without pivoting by squares. The nonzero counts are denoted by (red) x's for factorizations with pivoting, and by (red) crosses without pivoting. The y-axes are scaled so that the running time and nonzero count of the complete GP factorization with pivoting (and triangular solution, for time) would fall on the middle hash marks. The scale on both axes is logarithmic. The time limit for iterative solutions is 10 times the total factor-and-solve time for SuperLU.

6, even a pivoting preconditioner was faster than SuperLU. On the other 2 matrices in which a nonpivoting preconditioner was faster than SuperLU, bramley1 and bramley1, the pivoting preconditioners were not able to reduce the running time over either GP or SuperLU. On one matrix, sherman4, all the preconditioners converged, but none reduced the running time below either GP or SuperLU.

On 4 of the 6 matrices that did not require a pivoting preconditioner, the iterative solver converged with very sparse factorizations. On 3 of the 4, a factorization with no fill at all (ILU(0)) converged. On saylr4, factorizations with no fill or very little fill did not converge to an accurate solution. They did converge to a less accurate solution. The sparsest factorization that converged had only 117% of the fill of ILU(0) and only 5% of the fill of the complete factorization. On the two remaining matrices, bramley1 and bramley2, even the sparsest (ILU(0)) nonpivoting preconditioners converged, but all the pivoting preconditioners that converged were almost complete factorizations.

Generally speaking, the only failure mode for all 8 matrices that did not require a pivoting direct solver was exceeding the time limit. There were essentially no numerically zero pivots or unstable factorizations. In some cases we believe that a higher time limit would allow convergence; in some cases the solver has converged to a solution that was not accurate enough and could not further reduce the residual; and in some cases the solver exceeded the time limit without significantly reducing the residual at all. In two cases a single drop-threshold value produced an unstable factorization, once when pivoting (on pores_2), and once when not pivoting (on bramley2).

Pivoting Factorizations

We now discuss the results of experiments I and III with the 16 matrices that required pivoting for both complete and incomplete factorizations, as well as with the 2 matrices that did not require pivoting for a complete factorization, but did require pivoting incomplete factorizations. The results of these experiments are summarized in Figures 3a, 3b, and 3c.

On 7 of the 18 matrices, the iterative solver (with the best preconditioner) was faster than both SuperLU and GP (in 2 of these 7 cases the improvement over SuperLU was less than 20%). On 3 more matrices, the iterative solver was faster than GP but not faster than SuperLU (in one of the 3 cases the improvement over GP was less than 20%). On the remaining 8 matrices, the iterative solver did not reduce the solution time over either SuperLU or GP.

Twelve of the matrices converged with a preconditioner with 50% or less of the fill of a complete factorization. Only 7 converged with 40% or less, only 5 with 20% or less, and 2 with less than 10%. Only 2 matrices, psmigr_1 and psmigr_3, converged with an ILU(0) factorization.

Failure modes in the pivoting preconditioners on 12 of the 18 matrices included unstable factorizations that were detected as either numerically zero pivots during the factorization or divergence during the iterations (the 2-norm of the residual grows by a factor of 10^4 or more). Zero pivots were detected on 11 matrices, and divergence on 6. On the remaining 6 out of the 18 matrices, 2 matrices always converged, and on the other 4 the only failure mode was exceeding the time limit.

The Effect of Ordering on Convergence

Since RCM ordering produces significantly more fill than MMD in complete and nearly complete factorizations, we only tested RCM orderings on relatively sparse incomplete factorizations, namely, column-fill-ratio factorizations with ratios of 1 and 2. We now compare these RCM preconditioners, from experiment IV, with column-fill-raio MMD preconditioners with the same ratios from experiment II.

In only 18 of the 48 experiments (24 matrices with 2 fill ratios each), both orderings converged. In 7 more the MMD-ordered preconditioner converged but the RCM-one exceeded the time limit (which was identical for both orderings and based on the SuperLU time with an MMD ordering). There were no other cases.

When both converged, MMD was faster in 8 experiments and RCM in 10. But when MMD was faster, the RCM preconditioner took on average 207% more time to converge (that is, RCM was on average 3 times slower), whereas when RCM was faster, MMD took on average 47% more time (1.5 times slower). The MMD and RCM preconditioners had similar numbers of fill elements, but

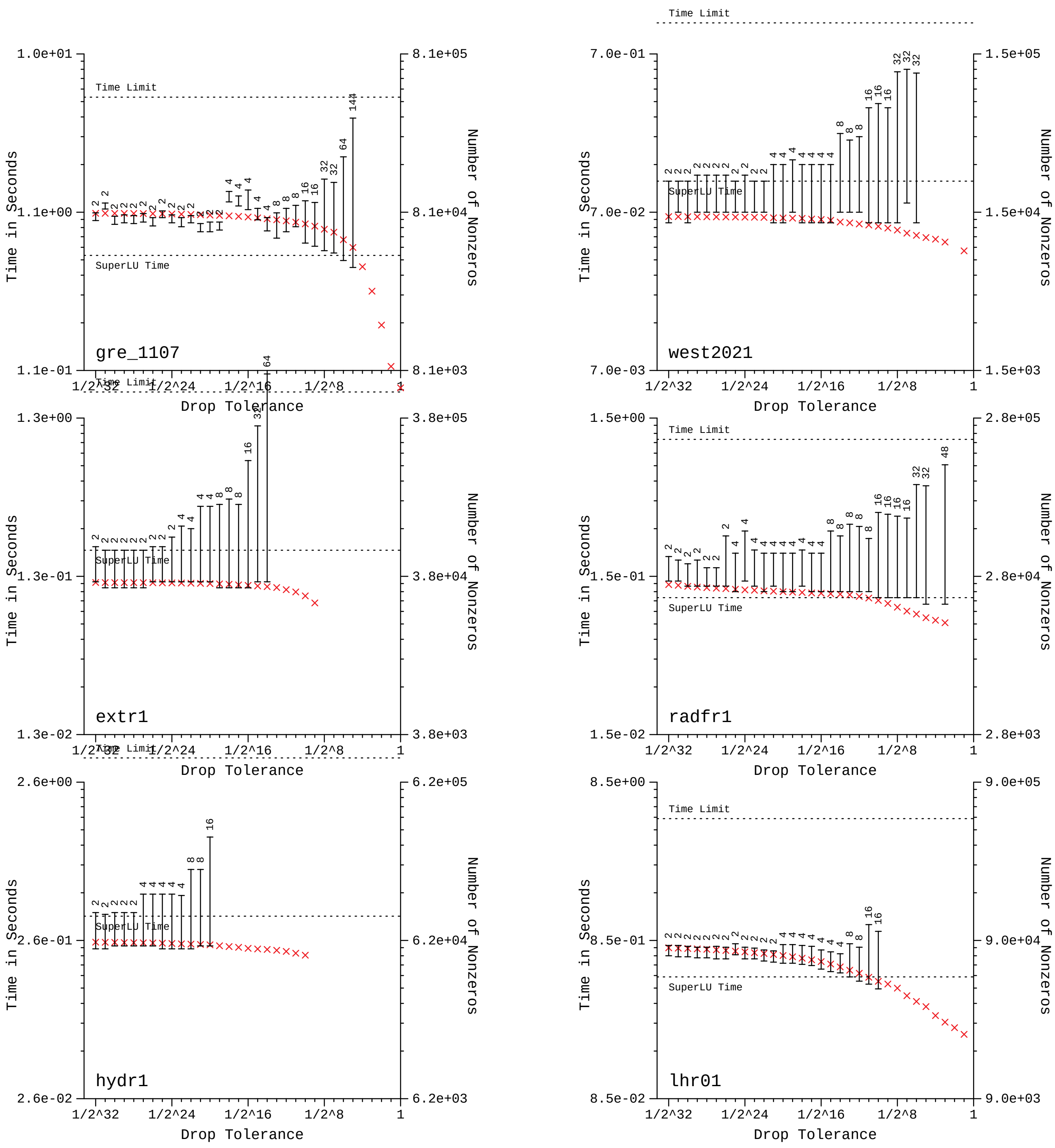

Figure 3a: Experiments I and III. Running times and nonzero counts for matrices that cannot be incompletely factored without pivoting, as a function of the drop threshold. The running times are denoted by vertical bars. The low end of the bar denotes the factorization time, and the high end shows the entire solution time (factorization+iteration). The nonzero counts are denoted by (red) x's. The y-axes are scaled so that the running time and nonzero count of the complete GP factorization with pivoting (and triangular solution, for time) would fall on the middle hash marks. The scale on both axes is logarithmic. The time limit for iterative solutions is 10 times the total factor-and-solve time for SuperLU.

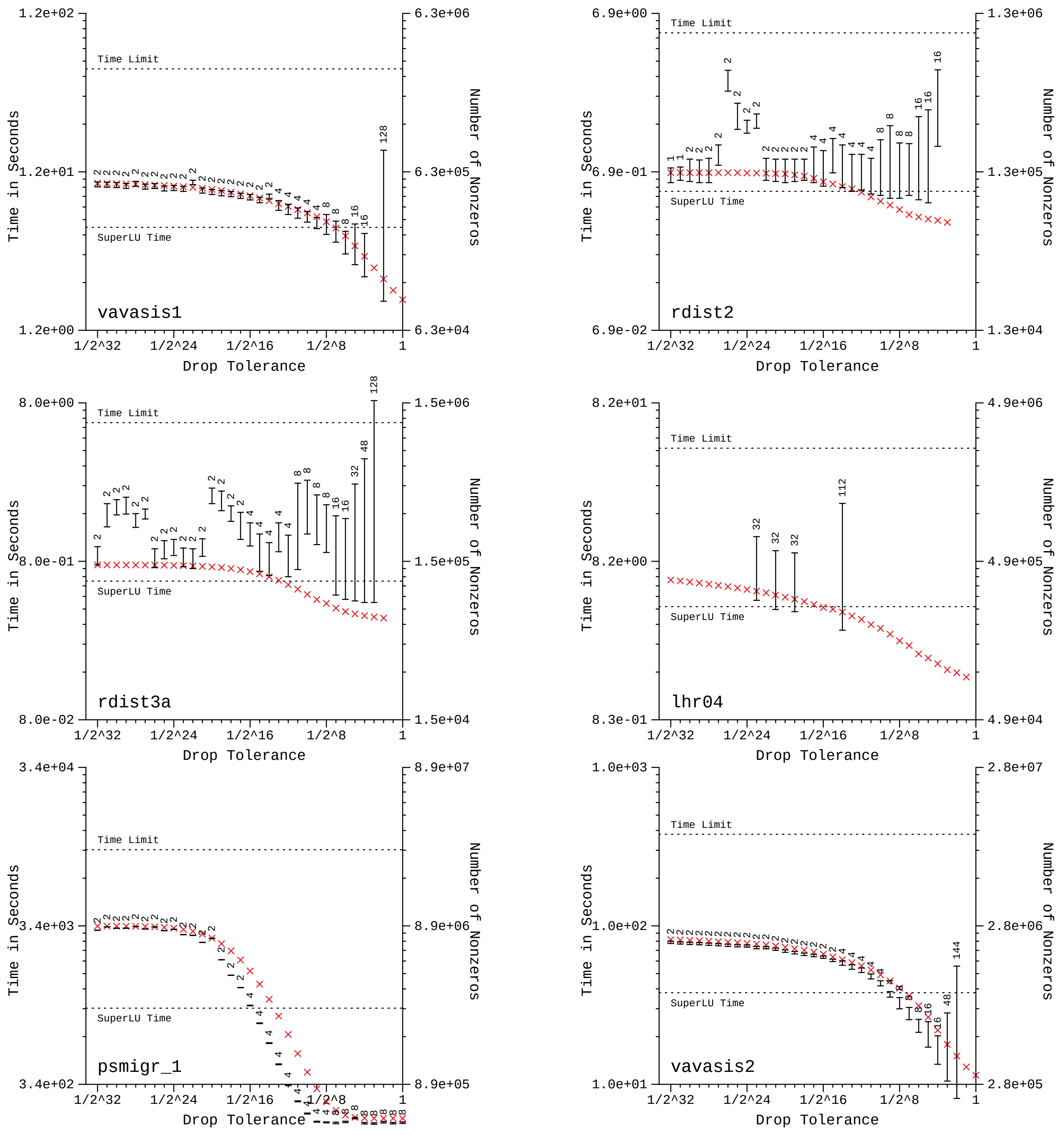

Figure 3b: Experiments I and III. Running times and nonzero counts for matrices that cannot be incompletely factored without pivoting (continued).

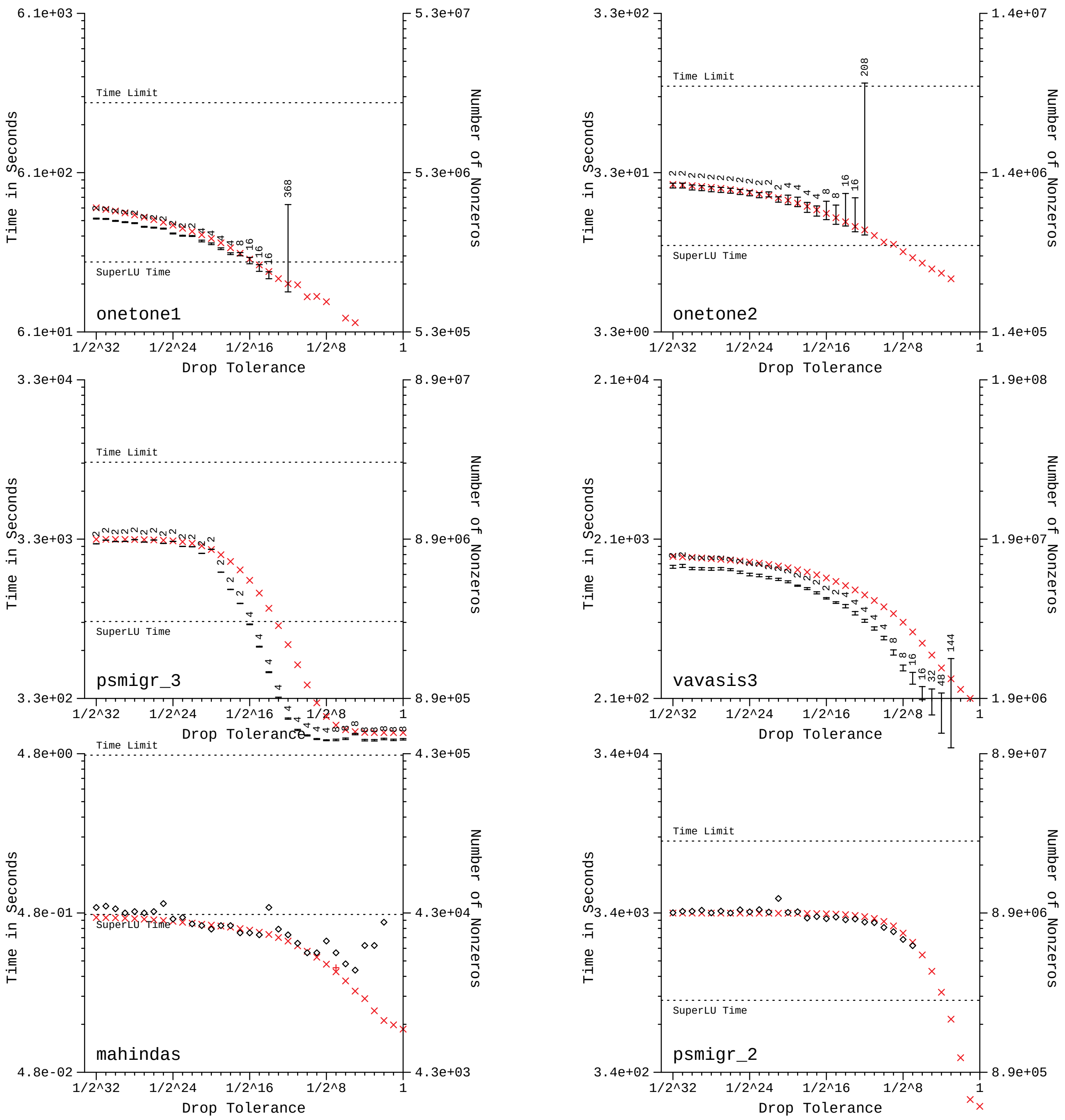

Figure 3c: Experiments I and III. Running times and nonzero counts for matrices that cannot be incompletely factored without pivoting (continued). SuperLU ran out of memory on vavasis3, so no SuperLU time is shown. The time limit for the iterative solution of vavasis3 was arbitrarily set at 10,000 seconds.

the MMD preconditioners were a little sparser on average in both cases.

Our experiments show that pivoting incomplete factorizations with column ordering based on an MMD ordering of A^TA usually converge faster than pivoting incomplete factorizations with column ordering based on an RCM ordering of A.

One possible reason for the difference in performance is that the MMD preconditioners retain a larger fraction of the fill of a complete factorization than the RCM ones. We expect an MMD ordering to yield a sparser complete factorization than an RCM ordering. Since the column-fill-ratio preconditioners had about the same number of fill elements with both orderings, more fill was dropped from the RCM preconditioners than from the MMD ones.

Column-Fill-Ratio Incomplete Factorizations

To compare the quality of drop-tolerance preconditioners and column-fill-ratio preconditioners, we matched each column-fill-ratio preconditioner from experiment II with a drop-tolerance preconditioner with a similar number of nonzeros from experiment I. Specifically, we paired with a column-fill-ratio preconditioner with k nonzeros a drop-tolerance preconditioner with between $0.9k$ and k nonzeros, if there was one. We broke ties by choosing the preconditioner with the largest number of nonzeros among all admissible ones.

Out of 144 drop tolerance preconditioners, 108 were paired. In 15 of the pairs neither preconditioner converged within the time limit. In 16 pairs only the drop-tolerance preconditioner converged. Among the 77 pairs in which both preconditioners converged, the column-fill-ratio preconditioners required, on average, factors of 21.7 more iterations and 2.67 more time to converge (time including both factorization and iterations). There were only 6 pairs in which the column-fill-ratio preconditioner converged faster. Among these 6, the column-fill-ratio preconditioners, required on average a factor of 0.87 less time to converge. We conclude that for a given number of nonzeros, a drop-tolerance preconditioner is usually dramatically better, and never much worse.

Symmetric Positive Definite Matrices

Although this paper only focuses on nonsymmetric matrices, we did perform some experiments on symmetric-positive-definite matrices. The goal of these experiments was not to assess iterative solvers for SPD matrices, but to answer two specific questions: (a) does our iterative solver perform better on SPD matrices than on nonsymmetric matrices, and if not, (b) can an iterative solver that exploits the properties of SPD matrices perform better?

We ran two sets of experiments. In the first set, consisting of experiments V and VI, we used exactly the same Krylov-space method and the same preconditioner, nonsymmetric LU with and without partial pivoting, to solve 12 SPD matrices. Our goal in this first set was to determine whether the behavior of our solver is significantly better when the matrices are SPD.

The results, which are described in Figures 4a and 4b, show that even SPD matrices can cause the solver difficulties. Out of the 12 matrices, only 4 can be solved with large drop tolerances. There is only one case (bcsstk08) of spectacular reduction in running time relative to the direct solver. Most of the failures are caused by exceeding the time limit, but in a few cases the factorization is unstable and causes the solver to diverge. There is no significant difference in performance between pivoting and nonpivoting factorizations.

In the second set, consisting of experiments VII and VII, we compared a symmetric and a nonsymmetric solver on 6 of the matrices (bcsstk08, bcsstk25, bcsstk27, bcsstk28, 1138_bus, and gr_30_30). We compared a conjugate gradient (CG) method using a drop-tolerance incomplete Cholesky preconditioner to a QMR method using a drop-tolerance incomplete LU preconditioner. The first solver exploits the fact that the matrices are SPD, while the second is quite similar to our nonsymmetric iterative solver, except that it does not pivot. We used Matlab 5.0 for this experiment, and symmetrically permuted the matrices using RCM ordering. We again set the required residual to be 10^4 times less accurate than a residual obtained by direct solution. We set the maximum number of iterations to 512.

The results of this experiment show no significant qualitative differences between the symmetric and the nonsymmetric solver. The symmetric solver was able to solve the matrices bcsstk08, bcsstk27, gr_30_30, and 1138_bus even with very large drop tolerances (more than 512 iterations were required on 1138_bus with the two sparsest factorizations). The symmetric solver failed to converge in 512 iterations with most of the values of the drop tolerance on the other two matrices. Reduction in running time relative to the direct solver was obtained only on bcsstk08. This behavior is similar to the behavior of our nonsymmetric solver.

5 Conclusions

The primary conclusion from our experiments is that iterative solvers with incomplete LU preconditioners can be very effective for some nonsymmetric linear systems, but they are not robust enough for inclusion in general-purpose black-box linear solvers.

Iterative solvers sometimes save a factor of about 10 in both time and space relative to a state-of-the-art direct sparse solver. But in most cases even the best drop-threshold value does not produce a very effective preconditioner. Also, to our knowledge there are no known techniques for determining an optimal or near-optimal drop-threshold value. Therefore, a black-box solver is likely to operate most of the time with sub-optimal drop thresholds, which can lead to slow convergence or no convergence. Out of hundreds of iterative solutions, few were more than 10 times faster than a direct solver, but many were more than 10 times slower.

Our experiments on SPD matrices, while limited, suggest that our primary conclusion remains valid even if we restrict our attention to SPD matrices, and perhaps even to SPD matrices solved by symmetric methods. These experiments, however, are limited in scope, and were only meant to indicate whether the nonsymmetry of the matrices or of the solvers caused the difficulties that we have reported. They were not meant to provide an evaluation of iterative solvers for SPD matrices and should not be used as such.

We also draw some secondary conclusions from the data on nonsymmetric matrices.

- First, pivoting in incomplete LU is necessary in many cases, even though we always begin by permuting the matrices to create a nonzero diagonal. Pivoting is necessary whenever pivoting is required for the direct solution, and it is necessary even for some systems that can be directly solved without pivoting. In other words, pivoting is more important in the incomplete case than in the complete case.

- Second, the best overall running times for the iterative solution of single linear systems (as opposed to multiple systems with the same matrix) are almost always achieved with around 8 to 16 iterations.

- Third, drop-tolerance preconditioners are more effective than column-fill-ratio preconditioners with a similar amount of fill. This is unfortunate, since column-fill-ration and other fixed-fill strategies allow solvers to tailor the preconditioner to the amount of available main memory.

- Fourth, MMD column ordering yields more efficient preconditioners than RCM column ordering. (Note that Duff and Meurant [5] showed that for SPD matrices, RCM is often a more effective ordering for incomplete-Cholesky preconditioners with no fill.)

Iterative solvers are suitable for toolkits for the solution of sparse linear systems. Toolkits implement multiple algorithms and enable the user to construct a solver that can efficiently solve a given problem. An iterative solver that works well on one matrix may be inefficient or even fail to converge on another. For example, Grote and Huckle [8] switch from right to left preconditioning in order to achieve convergence with a sparse-approximate-inverse preconditioner on pores2, and Chow and Saad [2] switch from row to column-oriented factorization to achieve convergence with lhr01. Chow and Saad also use a variety of other techniques to solve other systems. There are no established criteria that can guide an automatic system as to which solver is appropriate for a given matrix. Therefore, it is necessary to give the user control over which algorithms are used to solve a linear system, which is exactly what toolkits do.

Direct solvers, on the other hand, are suitable for black-box solvers. A single direct solver with a single ordering was reliable and efficient on all of

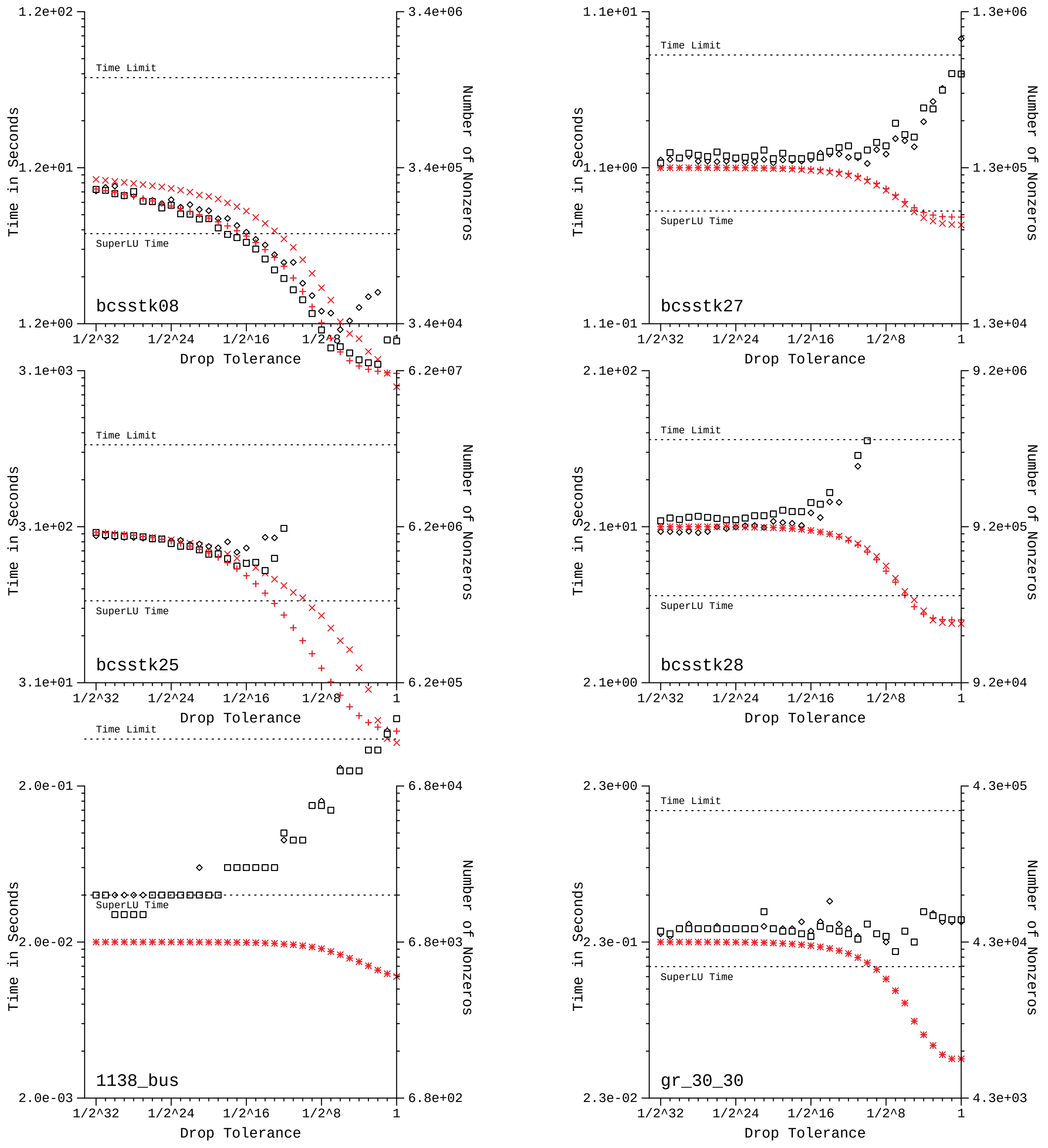

Figure 4a: Experiments V and VI. Running times and nonzero counts for SPD matrices ordered using MMD on $A^T A$, as a function of the drop threshold. The running times with pivoting are denoted by diamonds, and the running times without pivoting by squares. The nonzero counts are denoted by (red) x's for factorizations with pivoting, and by (red) crosses without pivoting. The y-axes are scaled so that the running time and nonzero count of the complete GP factorization with pivoting (and triangular solution, for time) would fall on the middle hash marks. The scale on both axes is logarithmic. The time limit for iterative solutions is 10 times the total factor-and-solve time for SuperLU.

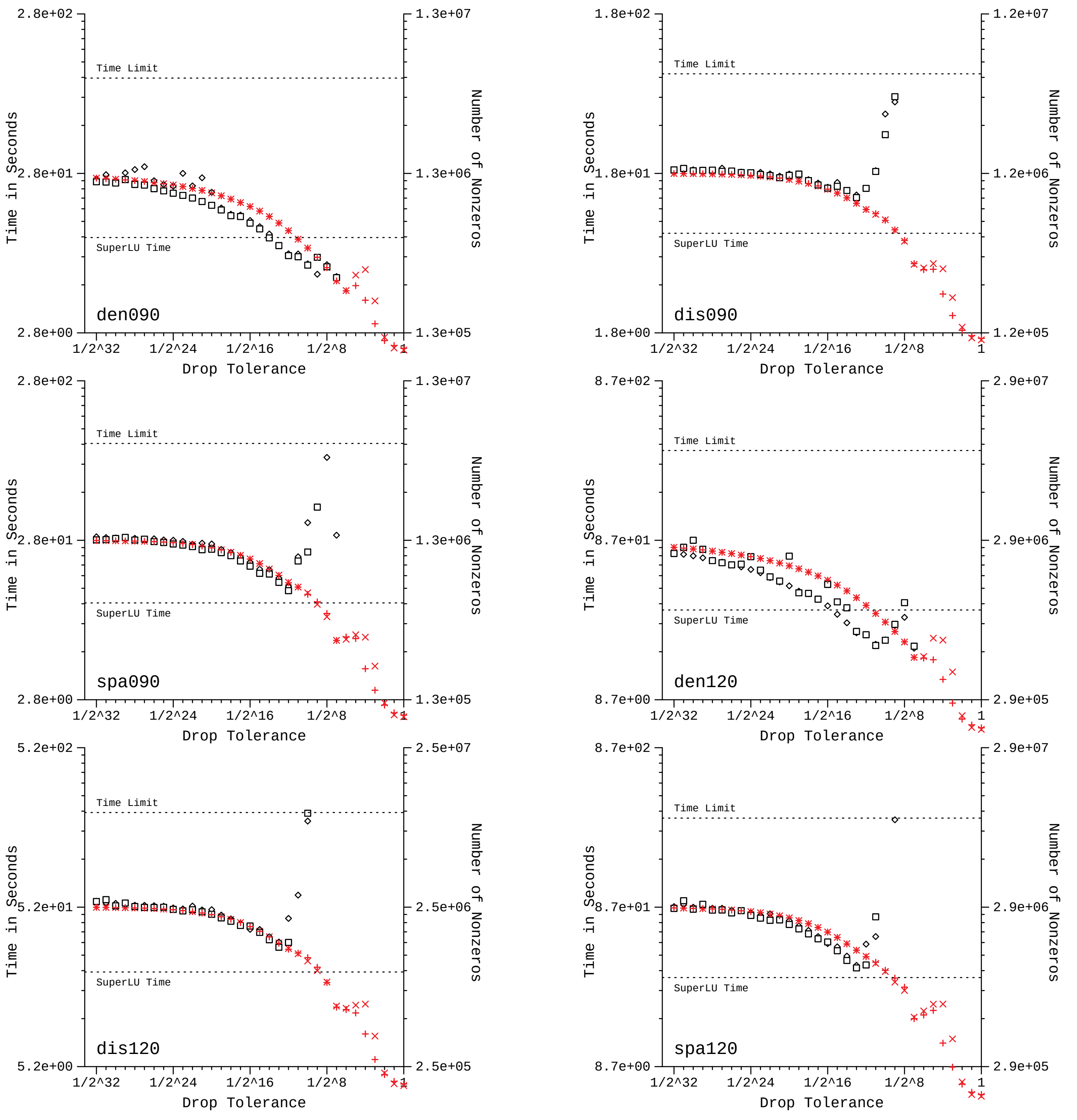

Figure 4b: Experiments V and VI. Running times and nonzero counts for SPD matrices ordered using MMD on A^TA (continued).

our test matrices. In comparison, our experiments have not turned up any single iterative solver (say, TFQMR with a specific drop-threshold preconditioner) that can rival the overall reliability and performance of this direct solver. While tuning a direct solver—by changing the ordering, for example—can sometimes improve its performance, we believe that direct solvers, even "right out of the box" with no tuning at all, are more reliable and more efficient than iterative solvers.

The discussion in the preceding paragraphs suggests that there are problems that can be solved, but not by black-box solvers. We did not include problems that are too large to be solved by a direct solver on a high-end workstation in our test set because we would not be able to compare direct and iterative solvers on them. Our experiments show that some problems can be solved by an iterative solver with a drop-threshold preconditioner with no or little fill, and that this solver requires significantly less memory than the direct solver. The direct solver would ran out of memory trying to solve similar but larger problems, but the iterative solver should be able to solve them. This implies that black-box solvers can solve all problems up to a certain size with reasonable efficiency, and that larger problems can sometimes be solved by more specialized solvers.

One potentially useful technique is to adaptively search for an efficient preconditioner, hoping not to waste too much time in the search. Two facts can guide us in designing the search. Since an efficient preconditioner usually yields a solution in 8–16 iterations, we can abort the solver after about 20 iterations, or if we encounter a zero pivot, and try to construct a new preconditioner. Since most of the solution time is spent in the factorization phase when the preconditioner is relatively dense, one should start the search with very sparse preconditioners, so that aborting and refactoring is not too expensive. One flaw in this idea is that some matrices do not fill very much (e.g., west2021), so each aborted iterative solution can be almost as expensive as a direct solution.

We believe that studying iterative solvers in the context of the reliability and performance of a direct solver is important. While comparisons of iterative solution techniques to one another can be very informative, they do not provide practitioners with specific practical advice. Since practitioners have the option to use direct solvers, which are generally reliable and efficient, they need to know whether the iterative solver under study outperforms state-of-the-art direct solvers. The knowledge that one iterative solver outperforms another is usually not sufficient for deciding to deploy it. We hope to see more direct/iterative comparative studies in the future, at least for nonsymmetric matrices, especially since SuperLU is freely available on NETLIB.

To summarize, we believe that incomplete LU preconditioners with partial pivoting are useful components in a toolkit for the iterative solution of linear systems, such as PETSc. Such preconditioners can be very effective in individual applications that give rise to a limited class of linear systems, so that the drop threshold and other parameters (e.g., ordering) can be tuned and the entire solver can be tested for reliability. But such iterative solvers cannot currently rival the reliability and performance of direct sparse solvers.

Finally, early responses to this paper convice us that more such studies are needed.

Acknowledgments

Thanks to Jim Demmel and Xiaoye Li for helpful discussions. Thanks to Barry Smith and the rest of the PETSc team for their dedicated support of PETSc. Thanks to Padma Raghavan and Esmond Ng for helpful discussions and for contributing the image processing matrices. Thanks to Alex Pothen for his perfect-matching code. Thanks to Cleve Ashcraft, Ed Rothberg, Yousef Saad, and Barry Smith for helpful comments on a draft of this paper.

References

[1] Satish Balay, William Gropp, Lois Curfman McInnes, and Barry Smith. PETSc 2.0 users manual. Technical Report ANL-95/11 — Revision 2.0.15, Argonne National Laboratory, 1996.

[2] Edmond Chow and Yousef Saad. Experimental study of ILU preconditioners for indefinite matrices. Technical Report UMSI 97/97, Supercomputer Institute, University of Minnesota, June 1997.

[3] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A supernodal approach to sparse partial pivoting.

Technical Report CSL 95/03, Xerox Palo Alto Research Center, September 1995.

[4] James W. Demmel, John R. Gilbert, and Xiaoye S. Li. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. Technical Report CSD–97–943, Computer Science Division, University of California, Berkeley, February 1997.

[5] Iain S. Duff and Gérard Meurant. The effect of ordering on preconditioned conjugate gradient. *BIT*, 29(4):635–657, 1989.

[6] S .C. Eisenstat and J. W. H. Liu. Exploiting structural symmetry in a sparse partial pivoting code. *SIAM Journal on Scientific and Statistical Computing*, 14:253–257, 1993.

[7] J. R. Gilbert and T. Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM Journal on Scientific and Statistical Computing*, 9:862–874, 1988.

[8] Marcus J. Grote and Thomas Huckle. Parallel preconditioning with sparse approximate inverses. *SIAM Journal on Scientific Computing*, 18(3):838–853, 1997.

[9] Alex Pothen and Chin-Ju Fan. Computing the block triangular form of a sparse matrix. *ACM Transactions on Mathematical Software*, 16(4):303–324, December 1990.

[10] Yousef Saad. Sparskit: A basic tool-kit for sparse matrix computations, version 2. Software and documentation available online from `http://www.cs.umn.edu/Research/arpa/SPARSKIT/sparskit.html`, University of Minnesota, Department of Computer Science and Engineering, 1994.

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

[12] Yair Shapira. A multi-level method for sparse linear systems. Technical Report Technical Report LA-UR-97-2551, Los Alamos National Laboratory, 1997.