An Experimental Evaluation of Iterative Solvers for Large SPD Systems of Linear Equations

Thomas George
Department of Computer Science
Texas A&M University
tgeorge@cs.tamu.edu

Anshul Gupta
Mathematical Sciences Department
IBM T. J. Watson Research Center
anshul@watson.ibm.com

Vivek Sarin
Department of Computer Science
Texas A&M University
sarin@cs.tamu.edu

ABSTRACT

Direct methods for solving sparse systems of linear equations are fast and robust, but can consume an impractical amount of memory, particularly for large three-dimensional problems. Preconditioned iterative solvers have the potential to solve very large systems with a fraction of the memory used by direct methods. The diversity of preconditioners makes it difficult to analyze them in a unified theoretical model. Hence, a systematic evaluation of existing preconditioned iterative solvers is necessary to identify the relative advantages of iterative methods and to guide future efforts. We present the results of a comprehensive experimental study of the most popular preconditioner and iterative solver combinations for symmetric positive-definite systems. A detailed comparison of the preconditioners, the iterative solver packages, and a state-of-the-art direct solver gives interesting insights into their strengths and weaknesses. We believe that these results would be useful to researchers developing preconditioners and iterative solvers as well as practitioners looking for appropriate sparse solvers for their applications.

1. INTRODUCTION

A fundamental step in many scientific and engineering simulations is the solution of a system of linear equations of the form $Ax = b$, where A is the sparse coefficient matrix, b is the right hand side vector (RHS) and x is the vector of unknowns. Large sparse linear systems involving millions and even billions of equations are becoming increasingly common in scientific, engineering, and optimization applications. These systems of equations can be solved using either *direct* or *iterative* methods. Direct methods are fast and robust, but are often unsuitable for large three-dimensional problems due to their prohibitive memory requirements. This limitation has spurred a lot of interest in iterative solvers and preconditioning methods due to their potential to lower the memory and computational cost of solving large sparse linear systems. To improve the rate of convergence of iterative methods, it is essential to use effective preconditioners, which can be problem dependent and may require problem-specific fine-tuning. Survey articles by Benzi [3] and Saad and van der Vorst [14] describe the relative strengths and weaknesses for a wide range of preconditioned iterative schemes from a theoretical perspective. There have been a few empirical studies in the past [5, 9, 4], however, these focused on a single preconditioner and were, therefore, somewhat limited in scope.

A typical practitioner has an overwhelming number of choices regarding solvers and preconditioners and their parameters in several black-box solver packages, such as PETSc [2], Trilinos [12], ILUPACK [13], Hypre [1], and many others [8]. However, choosing the appropriate scheme and fine-tuning the parameters for a particular linear system to be solved has remained a blend of art and science due to several reasons. The diversity of the preconditioners makes it difficult to analyze them in a unified theoretical model. There is often a significant variability in the different implementations for the same preconditioner, which limits the utility of a purely theoretical analysis. Finally, most preconditioners have a multitude of tunable parameters, and the best set of parameters for the same preconditioner can vary significantly not only between problems, but also from one implementation to another. Therefore, a thorough empirical evaluation can provide important insights into the relative strengths of different preconditioners, and that is the primary goal of this paper.

The main contributions of this paper are as follows. We present a detailed empirical evaluation of the most popular and promising preconditioned iterative schemes available in free black box solver packages on a collection of matrices drawn from a wide range of scientific applications. In each case, we use multiple values of each tunable parameter and measure the resulting performance in terms of the solution accuracy, computation time, and memory usage. We then analyze the results and present performance metrics for each preconditioner implementation that would be useful to a practitioner. These results help identify the preconditioner-parameter combinations most likely to yield good time or memory performance across a wide range of problems and the parameters that are most likely to improve performance upon fine-tuning. The experiments also indicate which preconditioners are most sensitive to changes in their parameter values and which are least sensitive. We compare the performance of different preconditioners with that of a direct solver with respect to both computation time and memory use. Last, but not the least, the paper presents a methodology for a systematic performance analysis of iterative solvers and preconditioners, which we believe can be useful in other similar contexts; for instance, for a similar study for unsymmetric or indefinite systems.

2. EXPERIMENTAL SETUP

In this section, we describe the setup for the empirical study, including the solver packages, preconditioner parameters, the suite of test matrices, the performance metrics used in the evaluation, hardware details, and the methodology for conducting the experiments and evaluating the results.

2.1 Software Packages

For our empirical study, we considered iterative solver packages that are most commonly used and cited in literature, such as PETSc [2] (release version 2.3.3-p0), Trilinos [12] (release version 8.0.3), and Hypre [1] (release version 2.0.0). We also included two packages, namely ILUPACK [13] (development version 2.2), and WSMP [11] (development version 7.12), which include promising research preconditioners that we found robust and stable enough for extensive testing. We used the direct solver from WSMP [10] for comparison with the iterative solvers.

2.2 Solvers

We used the Conjugate Gradient solver whenever possible because this study is limited to symmetric positive-definite systems. If a preconditioning scheme had the potential to result in a non-SPD preconditioner, then GMRES with restart values of 30, 65 and 100 was used.

2.3 Matrix Reordering

For preconditioners based on incomplete factorization, the matrices were first reordered using either the Reverse Cuthill Mckee ordering (RCM) or the Nested Dissection ordering (ND). In the case of ILUPACK, we used five built-in reordering schemes, namely, Nested Dissection (ND), RCM, Approximate Minimum Fill (AMF), Independent set (IND) and permutation for diagonal dominance (DDPQ).

2.4 Preconditioners and Parameters

For this study, we used general purpose preconditioners based on sparse approximate inverses (SAI) and variations of incomplete Cholesky or incomplete LU factorization (ILU). These preconditioners have been successfully used in black box solver packages for a large class of problems [5, 4]. We also used the algebraic multigrid preconditioner, which usually works best for matrices originating from elliptic PDEs, to test its utility as a general purpose preconditioner.

Table 1 lists the specific preconditioners that were used in our experiments. This table also lists all the values of the tunable parameters that were tried. In all, 621 different combinations of solvers, preconditioners, and parameters were tried. Due to space limitations, we are unable to provide a detailed description of the preconditioners and their parameters in this paper. These details can be found in the documentation of the packages [2, 12, 1, 13, 11].

2.5 Test Matrices

Table 2 lists the SPD matrices used in our experiments, along with their applications areas and sizes in terms of both dimension and the number of nonzeros. Most of the matrices are obtained from the University of Florida sparse matrix collection [6]. The remaining ones are from some of the applications that use WSMP [10].

2.6 System Details

The experiments were conducted on a single CPU of an IBM HPC cluster 1600 based on 1.9 GHz Power5+ processors. All the packages were compiled using IBM compilers xlf (Fortran), xlc (C) or xlC (C++) in 64-bit mode with the -O3 optimization flag. IBM's Engineering Scientific Subroutine library (ESSL) was linked in to provide BLAS routines. The page size for text and data was set to 64 KB. A wall time limit of 3 hours and memory limit of 16 GB was used.

2.7 Testing Procedure

The right hand side vector b was computed using an exact solution of all ones. Diagonal scaling was performed on all linear systems before starting the solution process. The initial guess of the solution for the iterative process was chosen to be a vector of all zeros. Right preconditioning was used since it was the default for all the packages except PETSc and it ensured a uniform convergence criteria based on the true residual for all the experiments. The iterations were stopped when the number of iterations reached 1000 or when the relative residual norm dropped below 10^{-8}. A trial was considered to have solved the problem successfully only if the relative error norm was less than 10^{-2}. A trial was also considered successful if the relative residual norm was less than 10^{-8} and the relative norm of the error was in the range [0.01, 0.1].

2.8 Methodology for Comparison

We now present our methodology for evaluating the collective and problem-specific performance of the preconditioners. Our choice of metrics was heavily influenced by the guidelines proposed for benchmarking optimization software by Dolan and More [7].

Let $\mathcal{S}$ be the set of m preconditioned solvers ($|\mathcal{S}| = m$), $\mathcal{P}$ be the set of n linear systems to be solved ($|\mathcal{P}| = n$) and let μ represent any performance measure that takes a specific value for each evaluation trial. We assume that a smaller value of μ is desirable. In Section 3, we evaluate the preconditioners when μ represents total solution time, preconditioner memory, memory-time product, or the sensitivity of a preconditioner to its parameters.

If each of the solvers is applied to all the problems, the performance measure values can be represented as an $n \times m$ matrix μ, where the element $\mu_{p,s}$ denotes the performance of solver s on problem p. These performance values $\mu_{p,s}$, can often be infinite or not well-defined due to solver failure and other practical limitations. A natural way to compare the various methods is to consider the normalized performance values $r_{p,s}$, otherwise known as the performance ratios. The performance ratio of a method s for a problem p is simply the ratio of the methods performanc $\mu_{p,s}$ to the best (least) performance value over all solvers for the same problem; i.e.,

$$r^{\mu}_{p,s} = \frac{\mu_{p,s}}{\min\limits_{s' \in \mathcal{S}} \mu_{p,s'}}.$$

Note that $r_{p,s} \geq 1$ for all (p, s) and is equal to 1 for at least one solver s for each problem p. It seems reasonable to expect that average performance ratio (APR) would be a fair indicator of the effectiveness of the method with respect to that performance metric.

In practice, however, APR is often not very useful since a single solver failure for a method can make its APR go to infinity, making it difficult to compare the different methods. A principled approach is to compare the performance of the methods both in terms of the number of problems solved as well as average performance ratio directly using the distribution of the performance ratios. To achieve this, we use the notion of a performance profile [7], which is a plot of the cumulative distribution of the performance ratios. Let $\rho^{\mu}_{s}(\tau)$ denote the cumulative distribution of the performance ratios of a solver method s with respect to the measure μ, i.e.,

$$\rho^{\mu}_{s}(\tau) = \frac{1}{n}|r^{\mu}_{p,s} \leq \tau|.$$

Package	Solver	Preconditioner	Orderings	Parameters
PETSc	CG	IC(K)	RCM, ND	*Level of fill:* 0, 1, 2 *Fill factor:* 3, 5, 8, 10
	GMRES (30,65,100)	ILUTP	RCM, ND	*Drop tolerance:* 1e-2, 3e-2, 1e-3, 5e-4 *Pivot threshold:* 0.0, 0.1 *Fill factor:* 3, 5, 8, 10
HYPRE	CG	IC(K)	RCM, ND	*Level of fill:* 0, 1, 2 *Fill factor:* 3, 5, 8, 10 *Maximum non-zeros per row:* 5, ∞
	GMRES (30,65,100)	ILUT	RCM, ND	*Drop tolerance:* 1e-2, 3e-2, 1e-3, 5e-4 *Fill factor:* 3, 5, 8, 10
	CG	SAI (PARASAILS)	-	*Number of levels:* 0, 1, 2 *Threshold:* 0, 0.01, 0.1, -0.75, -0.9 *Filter:* 0, 0.001, 0.005, -0.9
	CG	BoomerAMG	-	*Maximum number of levels:* 25 *Smoother* - Hybrid Gauss-Seidel/Jacobi *Number of aggressive coarsening levels:* 0, 10 *Coarsening schemes:* Falgout, HMIS, PMIS *Strong threshold:* 0.25, 0.5, 0.8. 0.9
Trilinos	CG	IC(K)	RCM, ND	*Level of fill:* 0, 1, 2
	CG	ICT	RCM, ND	*Drop tolerance:* 1e-2, 3e-2, 1e-3, 5e-4 *Fill factor:* 3, 5, 8, 10
	GMRES (30,65,100)	ILUT	RCM, ND	*Drop tolerance:* 1e-2, 3e-2, 1e-3, 5e-4 *Fill factor:* 3, 5, 8, 10
	CG, GMRES (30,65,100)	AMG (ML)	-	Default SA Default DD Default DD-ML
ILUPACK	CG	Multilevel ICT	RCM, ND, AMF, IND, DDPQ	*Drop Tolerance:* 1e-2, 3e-2, 1e-3, 5e-4 *Inverse Norm Estimate:* 10, 25, 100
WSMP	Auto-select CG/GMRES	ICT	Auto-select ND/RCM	Four cycles of self tuning

Table 1: Description of the package specific preconditioner parameters.

Thus, $\rho_s^{\mu}(\tau)$ denotes the fraction of the problems that can be solved by the solver s with a performance ratio of τ or less.

In order to compare the relative performance of the methods, we require a quantitative measure of the performance and for this purpose, we use the area under the performance profile curve (AUC). A critical choice in the AUC-based comparison is the upper limit for the performance ratio since this can significantly affect the AUC. In our study, we choose this threshold to be equal to 10. In other words, we assume that a trial that resulted in performance that was more than an order of magnitude worse than the best performance for a give problem was effectively a failure. Note that this is in addition to the failure criteria described in Section 2.7.

3. RESULTS AND DISCUSSION

In this section, we present the results of our empirical evaluation. We use the product of total solution time and the memory required for storing the coefficient matrix and preconditioner as our primary performance criteria. Henceforth, we will refer to this quantity as the *memory-time product*. The total solution time includes the time to create the preconditioner the time taken by the preconditioned CG or GMRES to obtain the approximate solution.

We chose memory-time product as a primary measure of the quality of a preconditioner implementation because both computation time and memory use appear to be inadequate measures of the quality of a preconditioner, when considered individually. For most preconditioners, there is a range of parameter choices in which there is a trade-off between solution time and memory consumption, although it is possible to make parameter choices that increase or decrease both time and memory simultaneously. The optimum operating point of a preconditioner for a given problem lies in a trade-off zone. As we will observe later in this section, the direct solver results in the overall fastest solution time for most problems in our test suite, albeit at the cost of a significantly high memory consumption. A preconditioner could simply emulate the direct solver and emerge as the fastest preconditioner. At the other extreme, preconditioners such as Jacobi, Gauss-Seidel, or IC(0), consume very little memory, but can take an impractically large number of iterations to converge. As a result, judging the quality of preconditioners based solely on their time or memory requirements simply yields winners that are extreme cases and are of little practical interest.

3.1 Best Default Solver Configurations

We begin by identifying the set of parameters for each preconditioner implementation that resulted in the best memory-time product over the entire problem suite. For each preconditioner-package combination, we solved all the problems in our test suite using all valid combinations of parameters shown in the last column of Table 1. We then chose the parameter combination that resulted in the maximum AUC (area under curve) with memory-time product as the performance metric μ as described in Section 2.7. Essentially, this amounts to plotting the performance profile curves for all the parameter combinations for a given preconditioner implementation using memory-time product as the performance criterion, and choosing the parameter combination corresponding to the curve with the maximum area under it. We do not show these plots because the large number of parameter combinations make them impossible to interpret visually. Instead, in Table 3, we report the parameter combination that resulted in the curve with the maximum area for each preconditioner implementation. Note that the

Matrix	N	NNZ	Application
90153	90153	5629095	Sheet metal forming
af_shell7	504855	17588875	Sheet metal forming
audikw_1	943695	77651847	Automotive crankshaft modeling
bcsstk25	15439	252241	Skyscraper simulation
bmwcra_1	148770	10644002	Automotive crankshaft modeling
bst-1	1017397	74144859	Structural analysis
bst-2	384012	28069776	Structural analysis
cfd1	70656	1828364	C.F.D. Pressure matrix
cfd2	123440	3087898	C.F.D. Pressure matrix
conti20	20341	1854361	Structural analysis
dfi1	89616	1474304	Linear programming
garybig	42459173	238142243	Circuit simulation
hood	220542	10768436	Automotive
inline_1	503712	36816342	Structural engineering
kyushu	990692	26268136	Structural engineering
ldoor	952203	46522475	Structural analysis
minsurfo	40806	203622	Minimum surface problem
msdoor	415863	20240935	Structural analysis
mstamp-2c	902289	70925391	Metal stamping
nastran-b	1508088	111614436	Structural analysis
nd24k	72000	28715634	3D mesh problem (ND problem set)
oilpan	73752	3597188	Structural analysis
parabolic_fem	525825	3674625	C.F.D. Convection-diffusion
qa8fk	66127	1660579	Stiffness Matrix for 3D acoustic problem
qa8fm	66127	1660579	Mass matrix for 3D acoustic problem
pga-rem-1	5978665	29640547	Power network analysis
pga-rem-2	1480825	7223497	Power network analysis
ship_003	121728	8086034	Structural analysis - Ship structure
shipsec5	179860	10113096	Structural analysis - Ship section
torso	201142	3161120	Human torso modeling

Table 2: SPD test matrices with their order (N), number of non-zeros (NNZ), and application area.

winning configurations reported in Table 3 are not simply the ones that resulted in the least overall memory-time product. By their very construction, the performance profiles also capture the robustness or the ability to solve a relatively large fraction of the total number of problems with reasonable performance.

Clearly, there are limitations to our methodology. First, it is impossible to test all parameter combinations by performing an exhaustive search over the parameter space. We have attempted to use as many combinations as were feasible within the ranges recommended by the authors of the packages that we are studying. Secondly, the winning parameter combination is a function of the test suite. We have attempted to collect a diverse set of problems; however, the best parameter combination can turn out to be different for a test suite with matrices with different properties. Within these limitations, the winning parameter combinations of Table 3 are good candidates for default values that have a high probability of yielding a small memory-time product for an arbitrary problem.

Having chosen the overall best parameter combinations for all the preconditioners used in this study, we now look at their relative performance with respect to memory, time, and memory-time product.

Figure 1 shows the memory profiles of the configurations shown in Table 3 for each preconditioner implementation. The memory profile of WSMP direct solver is also included in this figure. Hypre-BoomerAMG and PETSc-IC(0) appear to be most memory efficient; however, Hypre-BoomerAMG is significantly more robust and is able to solve 80% of the problems as compared to 60% for PETSc-IC(0). The next best group of curves includes those for Trilinos-ML, Hypre-PARASAILS, WSMP-ICT, and ILUPACK. Note that ILUT preconditioner implementations appear to have the worst memory profiles. This is primarily because (i) due to their unsymmetric nature, they need to store both triangular factors instead of just one, as in the case of symmetric preconditioners, and (ii) they often need to work with fairly low drop tolerances to be comparable with other preconditioners in terms of robustness. The memory profiles of ILUT preconditioners may look much better for unsymmetric matrices, which we do not study in this paper.

Figure 2 shows the time profiles of the configurations shown in Table 3 for each preconditioner implementation, along with that of the WSMP direct solver. The direct solver turns out to be the fastest solver for about 70% of the problems. This is followed by PETSc-IC(0), which is the fastest one for about 19% of the problems, which probably have good degree of diagonal dominance. However, understandably, PETSc-IC(0) does not do as well for more difficult problems and its time profile curve is soon surpassed by that of WSMP-ICT. Hypre-BoomerAMG, which is highly memory efficient, is seen to be slower than Hypre-PARASAILS, ILUPACK, WSMP-ICT, and the direct solver.

Figure 3 shows the memory-time product profiles of the configurations shown in Table 3 for each preconditioner implementation

Package-Preconditioner	Solver	Ordering	Preconditioner Parameters
petsc-ILUTP	GMRES-100	ND	Fill Factor: 10; Pivot Thresold: 0.01; Drop Tolerance: 0.0005
petsc-IC(K)	CG	RCM	Level of Fill: 0
hypre-ILUT	GMRES-100	ND	Fill Factor: 5; Drop Tolerance: 0.0005
hypre-IC(K)	CG	ND	Fill Factor: 3; Level of Fill: 2
hypre-AMG	CG	-	Coarsening: PMIS; Levels of Aggressive Coarsening: 0; Strong Threshold: 0.7
hypre-SAI	CG	-	Number of Levels: 2; Threshold: 0.1; Filter: 0.005
ilupack-MLICT	CG	RCM	Drop Tolerance: .001; Inverse Norm Estimate: 10
trilinos-ILUT	GMRES-30	RCM	Fill Factor: 5; Drop Tolerance: 0.0005
trilinos-IC(K)	CG	RCM	Level of Fill: 1
trilinos-ML	GMRES-30	-	Smoothed Aggregation Default Parameters
wsmp-ICT	AUTO	AUTO	Auto selected drop-tolerance and fill-factor

Table 3: Iterative solver configurations that resulted in the best overall performance with respect to memory-time product profile area.

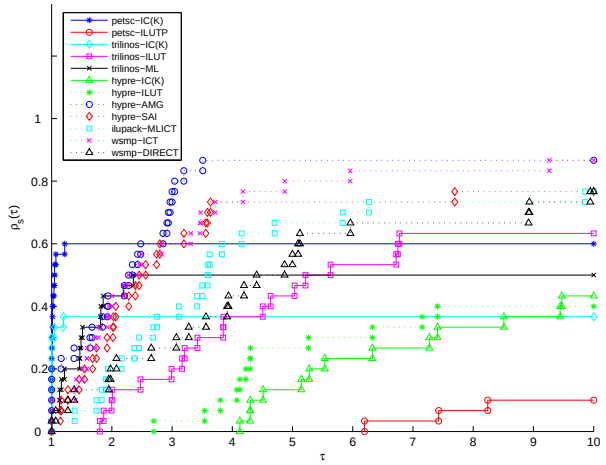

Figure 1: Memory performance profile curves for the direct solver and the best memory-time product configuration of the various preconditioner implementations shown in Table 3.

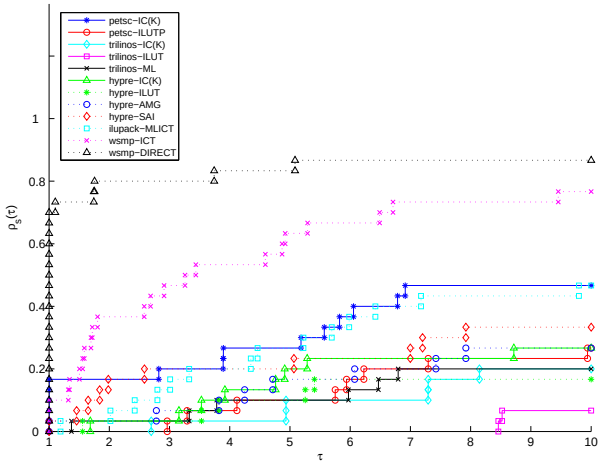

Figure 2: Time performance profile curves for the direct solver and the best time-memory product configuration of the various preconditioner implementations shown in Table 3.

and that of the WSMP direct solver. Surprisingly, the direct solver outperforms all the preconditioners even on the memory-time product criterion. In fact, the relative position of most memory-time product profiles is very similar to that of the corresponding time profiles in Figure 2, with the notable exception of Hypre-BoomerAMG, which moves up because of its excellent memory efficiency.

A comparison of the memory and time performance of the iterative solvers relative to WSMP's direct solver confirms the conventional wisdom that direct solvers are generally fast and robust, but require more memory resources. Conventional wisdom also holds that the preconditioned iterative solvers should outperform the direct solver on larger problems. In addition, the performance crossover point between iterative and direct solvers would be observed for relatively larger matrices that result from two dimensional physical problems as compared to three dimensional ones. Our results simply indicate that, although 40% of the problems in our test suite have more than half a million unknowns, the average problem size is still too small for most iterative solvers to outperform the direct solver in terms of solution time.

3.2 Problem Specific Parameter Selection

If all matrices arising in a user's application have similar characteristics, then, instead of using the default values of the parameters, the user would fine tune them to maximize the performance of a preconditioner for the application. Different preconditioners may have different degrees of tunability. In this section, we discuss the effect of problem-specific fine-tuning of parameters on the relative performance of various preconditioner implementations.

Figure 4 shows the memory profiles of all preconditioners when the parameter configuration for each problem was chosen individually to minimize its memory-time product. The most noteworthy observation from this figure is that problem specific fine-tuning results in remarkable improvements in memory use for most preconditioners, when compared with the best overall parameter configuration. This can be seen by comparing the memory profiles in Figures 1 and 4. The same direct solver results are used in the two figures, and all iterative solver curves move upwards with respect to the direct solver curve in Figure 4, when compared to Figure 1. Besides consuming less memory, most preconditioners are able to solve more problems successfully with problem-specific parameter tuning.

Figure 5 shows the time profiles of all preconditioners when the parameter configuration for each problem was chosen individually to minimize its memory-time product. Just like the memory pro-

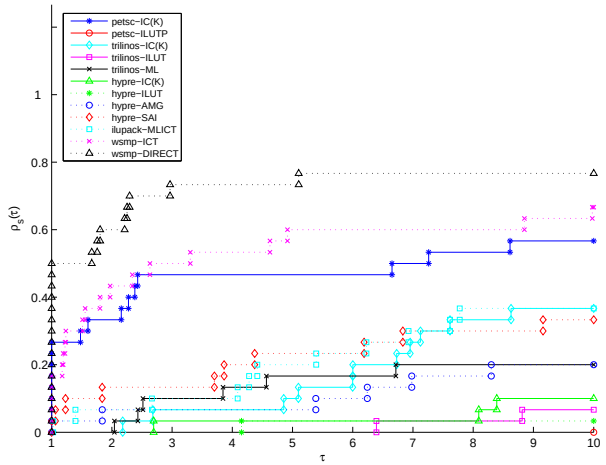

Figure 3: Memory-time product profiles of preconditioner configurations in Table 3.

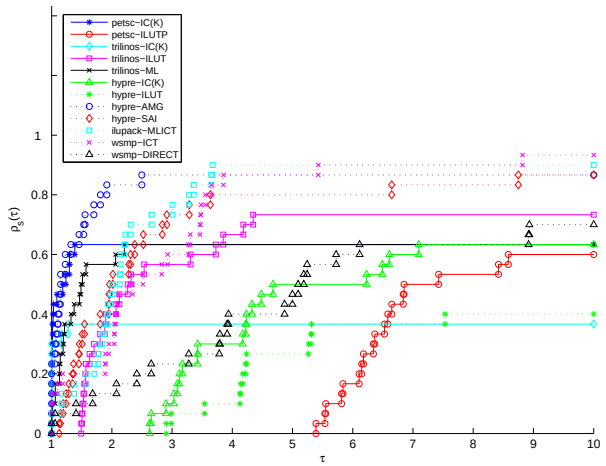

Figure 4: Memory performance profiles for the direct solver and the memory values corresponding to the best problem specific memory-time product configuration for each preconditioner.

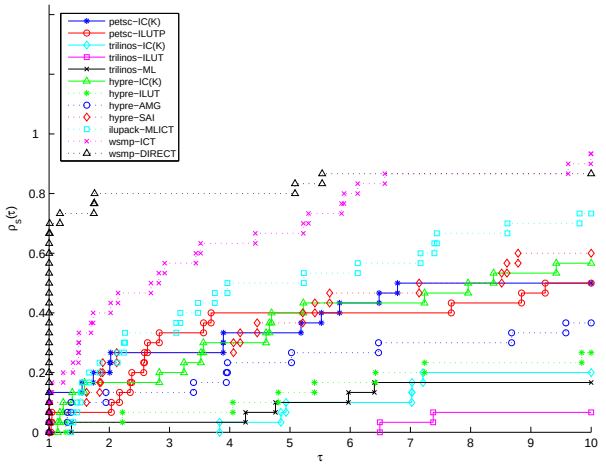

Figure 5: Time performance profiles for the direct solver and the time values corresponding to the best problem specific memory-time product configuration for each preconditioner.

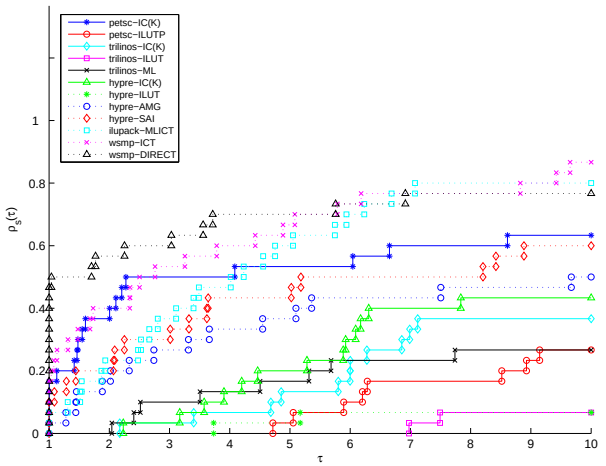

Figure 6: Memory-time product profiles for the direct solver and the best problem specific values for each preconditioner.

files, the time profiles of the preconditioners improve significantly when compared to those for the overall best parameter configuration in Figure 2. A comparison of the memory-time product profiles for the tuned parameters in Figure 6 with those for the overall best parameter configuration in Figure 3 shows a similar trend.

While fine-tuning the parameters improved the performance of all the preconditioners, it did not make a significant difference in the number of problems for which the direct solver turned out to have the fastest time or the best memory-time product. The direct solver was faster than all the 621 combinations of packages, solvers, preconditioners, and parameters that we tested for 70% of the matrices and had a better memory-time product for 47% of the matrices. The direct solvers, by the very nature of their computation, are able to exploit the memory hierarchy of modern microprocessor-based computers much more efficiently than the iterative solvers. Therefore, for sparse systems with similar numerical properties, they are expected to outperform iterative solvers for problems smaller than some threshold, despite generally higher operation counts and memory requirements. Our results tend to indicate that this threshold for most preconditioners occurs at fairly large problem sizes, and a majority of the problems in our test suite were smaller than this threshold.

Table 4 shows the memory and time required by the direct solver and by the iterative solver that resulted in the best memory-time product among all the package, preconditioner, and parameter combinations. For each iterative solver, the table also shows the parameter combination resulting in the best memory-time product. The values corresponding to the better memory-time product of the two are in bold font. The direct solver has the better memory-time product for 14 out of the 30 matrices. As expected, the iterative solvers do much better for large 3-D problems. Among the iterative solvers, PETSc-IC(0) and WSMP-ICT do best for 5 problems each, Hypre-PARASAILS for 3, ILUPACK for 2, and Hypre-BoomerAMG for 1.

Matrix	Iterative Solver Configuration	I. Mem (b)	I. Time (s)	D. Mem (b)	D. Time (s)
90153	wsmp, ICT w/ 3rd self-tuning step for drop-tolerance and fill factor	1.45e+008	5.37	**1.94e+008**	**3.85**
af_shell7	hypre, CG, SAI, Number of Levels (0), Threshold (0.1), Filter (0.005)	2.19e+008	139	**8.3e+008**	**16.3**
audikw_1	hypre, CG, SAI, Number of Levels (1), Threshold (0.1), Filter (0)	**1.37e+009**	**1110**	9.5e+009	1100
bcsstk25	wsmp, ICT w/ 2nd self-tuning step for drop-tolerance and fill factor	1.24e+007	1.17	**1.23e+007**	**0.122**
bmwcra_1	ilupack, CG, MLICT, RCM, Drop Tolerance (3e-2), Inverse Norm Est. (10)	1.36e+008	257	**5.68e+008**	**14.6**
bst-1	wsmp, ICT w/ 3rd self-tuning step for drop-tolerance and fill factor	3.03e+009	668	**3.21e+009**	**109**
bst-2	wsmp, ICT w/ 3rd self-tuning step for drop-tolerance and fill factor	2.78e+009	627	**2.35e+009**	**120**
cfd1	petsc, CG, IC(K), RCM, Level of Fill (0)	**1.8e+007**	**11.6**	1.58e+008	2.98
cfd2	petsc, CG, IC(K), ND, Level of Fill (0)	**3.06e+007**	**44.5**	3e+008	8.06
conti20	wsmp, ICT w/ 3rd self-tuning step for drop-tolerance and fill factor	5.89e+007	8.65	**6.4e+007**	**1.63**
dfi1	petsc, CG, IC(K), ND, Level of Fill (0)	1.61e+007	0.875	**1.91e+007**	**0.504**
garybig	wsmp, ICT w/ initial auto selected drop-tolerance and fill factor	**3.98e+009**	**1330**	—	—
hood	wsmp, ICT w/ 3rd self-tuning step for drop-tolerance and fill factor	**2e+008**	**5.05**	2.57e+008	4.1
inline_1	wsmp, ICT w/ 3rd self-tuning step for drop-tolerance and fill factor	2.08e+009	361	**1.5e+009**	**36.2**
kyushu	petsc, CG, IC(K), RCM, Level of Fill (0)	**2.58e+008**	**44.4**	9.21e+009	1520
ldoor	hypre, CG, SAI, Number of Levels (1), Threshold (-.75), Filter (-.9)	4.8e+008	283	**1.37e+009**	**32.9**
minsurfo	wsmp, ICT w/ 3rd self-tuning step for drop-tolerance and fill factor	**3.04e+006**	**0.0788**	1.05e+007	0.131
msdoor	wsmp, ICT w/ 3rd self-tuning step for drop-tolerance and fill factor	7.36e+008	49	**4.92e+008**	**8.31**
mstamp-2c	hypre, CG, SAI, Number of Levels (1), Threshold (-.75), Filter (-.9)	**6.68e+008**	**179**	—	—
nastran-b	hypre, CG, SAI, Number of Levels (1), Threshold (0.1), Filter (0.005)	**1.32e+009**	**1220**	8.62e+009	662
nd24k	ilupack, CG, MLICT, RCM, Drop Tolerance (1e-2), Inverse Norm Est. (100)	**1.53e+008**	**134**	2.75e+009	529
oilpan	wsmp, ICT w/ 3rd self-tuning step for drop-tolerance and fill factor	7.4e+007	3.73	**9.45e+007**	**1.74**
parabolic_fem	wsmp, ICT w/ 2nd self-tuning step for drop-tolerance and fill factor	9.27e+007	10.7	**2.31e+008**	**3.97**
qa8fk	petsc, CG, IC(K), RCM, Level of fill (0)	**1.65e+007**	**3.29**	1.89e+008	5.78
qa8fm	hypre, CG, AMG, PMIS, Aggregation Levels (0), Strong Threshold (0.5)	**1.57e+007**	**0.0975**	1.89e+008	6.18
pga-rem1	wsmp, ICT w/ 3rd self-tuning step for drop-tolerance and fill factor	**8.27e+008**	**97.4**	2.1e+009	65.4
pga-rem2	wsmp, ICT w/ 3rd self-tuning step for drop-tolerance and fill factor	2.51e+008	46.1	**6.27e+008**	**16.3**
ship_003	wsmp, ICT w/ 2nd self-tuning step for drop-tolerance and fill factor	**1.76e+008**	**20.6**	5.29e+008	20.8
shipsec5	ilupack, CG, MLICT, IND, Drop Tolerance (3e-2), Inverse Norm Est. (10)	**1.05e+008**	**20.3**	4.48e+008	17.7
torso	petsc, IC(K), RCM, Level of Fill (0)	**3.5e+007**	**6.99**	7.09e+008	35.5

Table 4: Table showing the overall best time (in seconds) and memory values (in bytes) corresponding to each problem for iterative and direct solvers. The bold values indicate the solver configuration for which the product of memory and time was the lowest.

3.3 Relative Strengths of Preconditioners and Sensitivity to Parameter Tuning

In Sections 3.1 and 3.2, we observed that different preconditioners and solvers display different strengths and weaknesses for the SPD matrices in our test suite. They have different degrees of robustness. Some are more memory efficient than others, while some are faster than others. In Section 3.2, we also saw that, as expected, most preconditioners performed significantly better when their parameters were permitted to be tuned to each coefficient matrix. However, different preconditioners displayed different degrees of improvement. Figure 7 displays all this relative information about the performance of various preconditioners by means of a single information-rich graphic.

Figure 7 has two sets of circles for each preconditioner. The green (dark) circles correspond to the default parameter configurations shown in Table 3. The yellow (light) circles correspond to problem-specific best parameters. The x- and y-coordinates of each circle are the areas of the time and memory profiles of the corresponding preconditioner derived from Figures 1 and 2, respectively. The size of each circle is proportional to the number of problems solved.

The height of a circle in Figure 7 is indicative of the memory efficiency of the corresponding preconditioner. Similarly, the distance from the y-axis towards the right is indicative of its speed. The size of the circle indicates robustness. The figure shows at the glance which solvers and preconditioners are most memory efficient and which ones are most time efficient. For example, the direct solver is very fast and robust, but is less memory efficient than many of the preconditioners. On the other hand, Hypre-BoomerAMG is very robust and memory efficient, but converges slowly.[1] For the default parameters, Hypre-PARASAILS and ILUPACK are progressively faster, but use more memory than Hypre-BoomerAMG. WSMP-ICT with default parameters is roughly as memory efficient as Hypre-PARASAILS, but significantly faster. Most threshold-based incomplete factorization preconditioners benefit a great deal from parameter tuning. This is evident from the fact the light circles corresponding to most preconditioners lie above and to the right of their dark counterparts. The most remarkable improvement can be seen in the case of ILUPACK, which outperforms Hypre-PARASAILS in both time and memory. With default parameters, ILUPACK was faster than Hypre-PARASAILS, but less memory efficient.

Whether with default or with fine tuned parameters, the best solvers lie on the periphery of the plot. In the absence of definitive knowledge of the best preconditioner for the application at hand, a user is likely to fare best by picking one of Hypre-BoomerAMG, ILUPACK, Hypre-PARASAILS, WSMP-ICT or WSMP-Direct solver, depending on the desired balance between computation time and memory. PETSc-IC(0) too emerges as a strong preconditioner

[1]Due to large CPU time requirements of the experiments, those with multiple combinations of Trilinos ML's parameters were still running at the time of submission of this paper. Therefore, Trilinos ML's results in this paper include only three trials. We expect both the default and the best-case performance of the Trilinos ML preconditioner to improve after we obtain results from all the trials, which we plan to include in the final version of the paper.

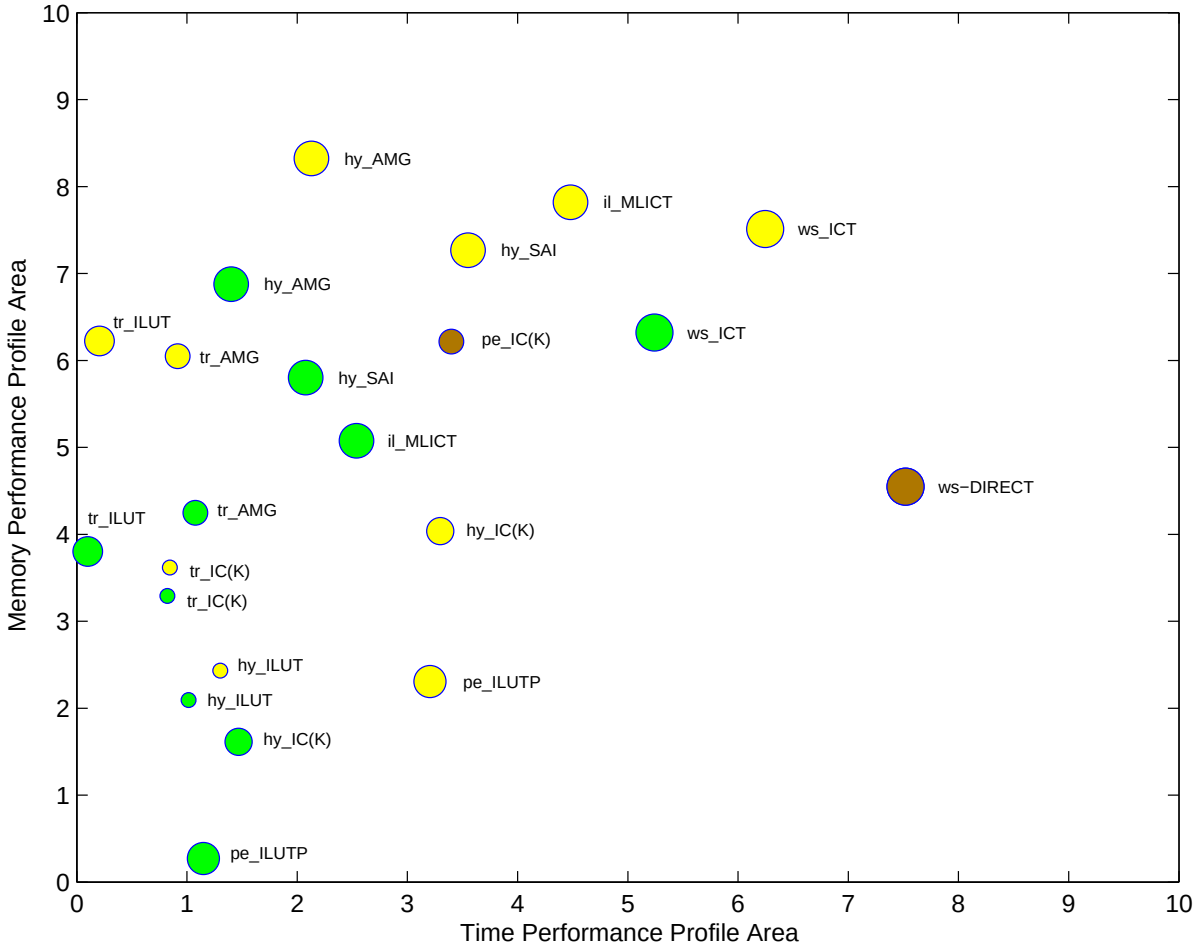

Figure 7: Plot of the time profile area versus the memory profile area for various preconditioner implementations. Each circle represents a preconditioner whose name consists of the first two letters of the name of the package followed by the type of preconditioner. The size of a circle is proportional to the number of problems solved. The green (dark) circles correspond to profile areas for the default parameter configuration and the yellow (light) ones correspond to profile areas for problem-specific best parameters.

in terms of memory and time efficiency, although it is able to solve fewer problems compared to the other leading preconditioners. Thus, at least one implementation of each of the major classes of preconditioners in this study can be regarded as the top choice, depending on the time and memory constraints.

Another interesting observation from Figure 7 is that the time, memory, and robustness of different implementations of the same underlying preconditioning method can be very different. Particularly, among threshold-based incomplete factorization methods, Trilinos ILUT is much more memory efficient than PETSc ILUTP, but is much slower. On the other hand, WSMP and ILUPACK, which also use preconditioning based in incomplete factorization, perform better in terms of time, memory, and robustness.

Figure 8 shows a more precise quantitative view of the sensitivity of different preconditioners to parameter tuning. The figure contains the profiles of the ratio of the memory-time product of the experimentally determined default parameter configuration of Table 3 to the problem-specific best memory-time product. This ratio is essentially the inverse of the improvement in memory-time product due to parameter tuning over that with the default parameter

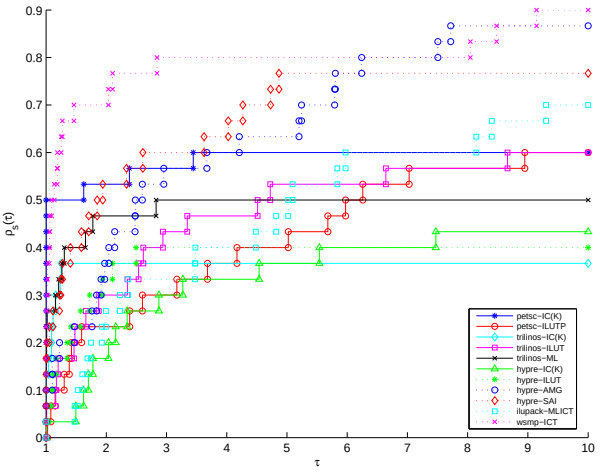

Figure 8: Profile curves for the ratio of default over problem specific best memory-time product values (inverse of sensitivity to parameter tuning).

configuration. From the figure, we observe that PETSc-IC(K) and WSMP-ICT are least sensitive to parameter tuning; i.e., typically, a user would be able to obtain performance close to the best that the preconditioner is capable of delivering for a given problem with either the default parameters or with experimenting with a few different parameter options. Recall, from Table 4, that these two preconditioners also solve the most number of problems with the smallest memory-time product. Since PETSc-IC(K) is a relatively simple preconditioner, it is not able to solve as many problems as WSMP's threshold based incomplete Cholesky factorization preconditioner, which is why it ranks lower when the area under the memory-time product profile is used as the comparison metric. However, for the applications for which it is suitable, PETSc's IC(0) implementation is an excellent choice that is efficient in terms of both time and memory.

3.4 Performance Impact of Tuning Different Parameters

In this section, we analyze the relative influence of the various parameter choices for the preconditioners on their time and memory performance. Intuitively, the effect of a parameter on a preconditioner's performance can be captured by measuring the variability in performance in response to changing that parameter while all others are kept fixed. To make this notion precise, we partition the set of all parameter configurations corresponding to a preconditioner implementation into subgroups such that only one parameter is varied in each subgroup. We then measure the standard deviation of the percentage by which that preconditioner's performance changes for each value of the varying parameter with respect to its performance with the experimentally determined default parameter configuration shown in Table 3. This is done for each solved problem, and then averaged over the entire problem set. Both time and memory performance are considered individually.

Figure 9 shows the fine-tuning effects of the various factors on the time and memory used for different preconditioners. The height of the bars indicate the variation in time and memory use in response to changing the corresponding parameter. Note that time and memory variations follow two different scales displayed on the left and right of the figure, respectively. Thus, for each preconditioner, we can observe which parameters have more influence on its time and memory performance than others. For example, the figure shows that the performance of Hypre-PARASAILS is much more sensitive to changing the filter parameter than the threshold. The figure also shows that different implementations of the same preconditioner can have different sensitivities to the same parameter. For example, the memory requirement of Trilinos ILUT is very sensitive to drop tolerance, but that of PETSc ILUTP is independent of the drop tolerance. This suggests that PETSc pre-allocates memory based on fill factor (hence the high sensitivity of PETSc ILUTP to fill factor), while Trilinos allocates it as the factor is generated, based on the number of nonzero entries that need to saved. This may partly explain why Trilinos ILUT is more memory efficient than PETSc ILUTP, but is also slower, as observed in Figure 7.

4. CONCLUDING REMARKS AND FUTURE WORK

We performed an extensive empirical evaluation of some commonly used preconditioned iterative methods available in free black box solver packages on a collection of matrices drawn from a wide range of scientific applications. For each package and preconditioner combination, we identify the best parameter choices using a performance profile based criterion [7] that takes into consideration the number of problems solved along with the time and memory usage across all the problems in the collection. Our experiments reveal parameter configurations that are good candidates for default configurations. For each preconditioner, we quantify benefits of parameter fine-tuning by comparing the best performance for each problem with the performance of our experimentally determined default parameters. Different preconditioners show different degree of tunability and tuning different parameters impacts the performance to different degrees. We provide a comparison of the performance of different iterative solvers relative to the direct solver that illustrates the successes and challenges in developing preconditioners for iterative solvers. The results also provide insight into the relative strengths and weaknesses of the various black box preconditioned iterative solver packages. We observed that different implementations of the same preconditioning method can vary widely in performance. A preconditioning algorithm may be more suitable for a particular class of problems than others. However, as general purpose methods, at least one implementation of each of the major classes of preconditioners in this study can be regarded as the top choice, depending on the time and memory constraints.

This study, admittedly, has its limitations. First, the experiments are conducted on a single processor. With a few exceptions, most preconditioners considered in this study have parallel implementations. Given that iterative solvers compare favorably with direct solvers for relatively large problems, they are more likely to be used on parallel computers than on a single processor. The computation time and memory use may scale at different rates for different preconditioners and their relative parallel performance may be substantially different from their serial performance. We plan to empirically study the scalability of some of the preconditioners evaluated in this paper. Another limitation of this study is that the results are derived from a test suite of only 30 problems. We kept the size of the test suite modest due to the sheer number of trials (621) for each matrix. Therefore, it is not clear how generalizable the results are. As part of our future work on this topic, we plan to assemble another test suite, conduct the same experiments, and check if we obtain similar results on the relative strengths and weaknesses of the preconditioners. We also plan to use clustering and other data-mining techniques to explore if a good choice of preconditioner and parameters can be predicted from the characteristics of the coefficient matrix as well as the underlying physical problem and those of the solution method that generates the matrix.

Acknowledgements. The authors wish to thank Matthias Bollhöfer, Robert Falgout, Michael Heroux, Marzio Sala, Heidi Thornquist, and Ulrike Yang for their assistance in installing and using their packages, answering queries, and suggesting the parameters to use and tune for the experiments. We also thank Felix Kwok for the initial installation of some of the packages and their driver programs.

5. REFERENCES

[1] Hypre, High Performance Preconditioners: Users Manual. Technical report, Lawrence Livermore National Laboratory,

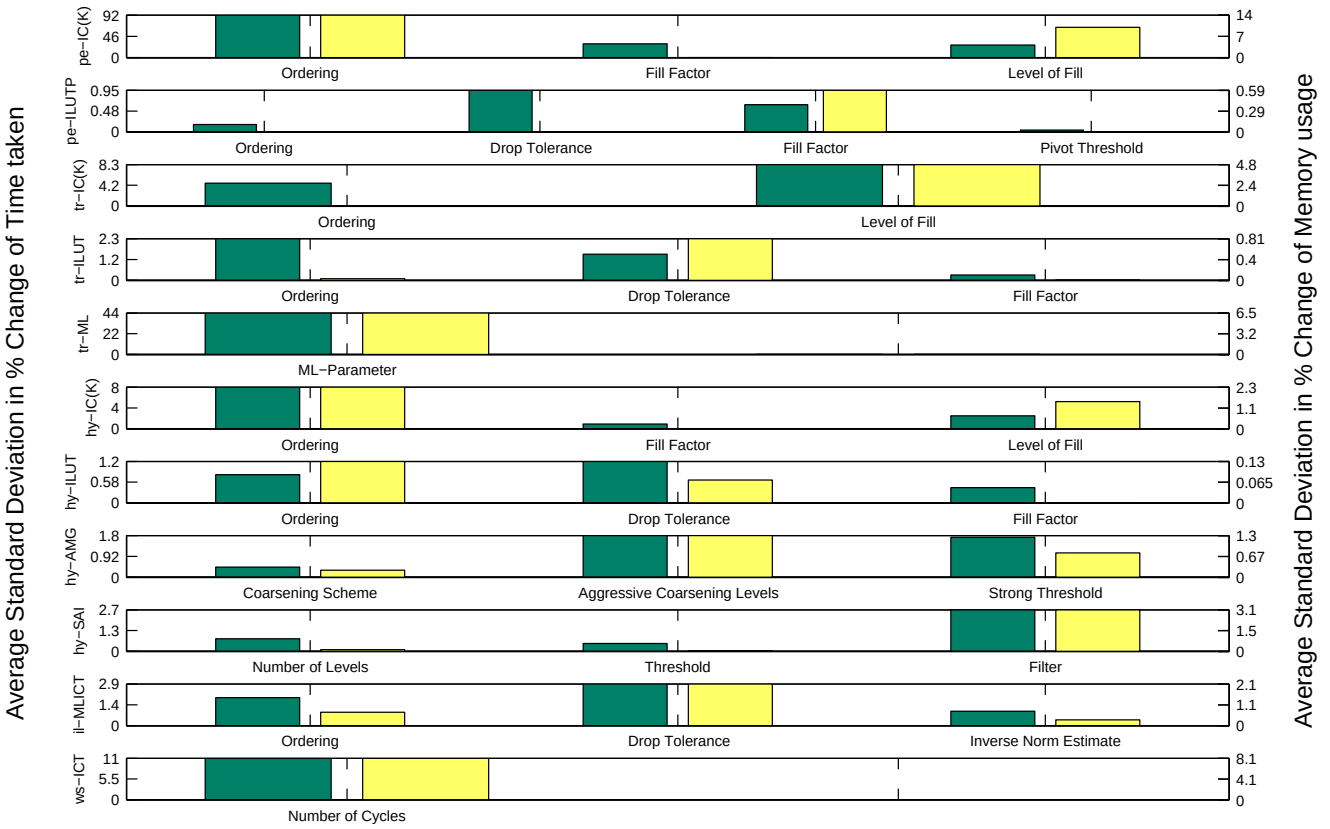

Figure 9: Average standard deviations of the variation in time and memory use of a preconditioner with different values of each parameter with respect to the time and memory use with the default configuration. The scales for the time and memory bars are different and are shown on the left and right, respectively.

2006.

[2] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc Web page, 2001. *http://www.mcs.anl.gov/petsc*.

[3] M. Benzi. Preconditioning Techniques for Large Linear Systems: A Survey. *Journal of Computational Physics*, 182(2):418–477, 2002.

[4] M. Benzi and M. Tuma. A Comparative Study of Sparse Approximate Inverse Preconditioners. *Applied Numerical Mathematics*, 30(2):305–340, 1999.

[5] E. Chow and Y. Saad. Experimental study of ILU preconditioners for indefinite matrices. *Journal of Computational and Applied Mathematics*, 86(2):387–414, 1997.

[6] T. A. Davis. The University of Florida Sparse Matrix Collection. Technical report, Department of Computer Science, University of Florida, Jan 2007.

[7] E. Dolan and J. Moré. Benchmarking Optimization Software with Performance Profiles. *Mathematical Programming*, 91(2):201–213, 2002.

[8] K. Dongarra and A. Buttari. Freely available software for linear algebra on the web, 2006. *http://www.netlib.org/utk/people/JackDongarra/la-sw.html*.

[9] J. R. Gilbert and S. Toledo. An assessment of incomplete-LU preconditioners for nonsymmetric linear systems. *Informatica*, 24:409–425, 2000.

[10] A. Gupta. WSMP: Watson sparse matrix package (Part-I: direct solution of symmetric sparse systems). Technical Report RC 21886, IBM T. J. Watson Research Center, Yorktown Heights, NY, November 16, 2000. *http://www.cs.umn.edu/~agupta/wsmp*.

[11] A. Gupta. WSMP: Watson sparse matrix package (Part-III: iterative solution of sparse systems). Technical Report RC 24398, IBM T. J. Watson Research Center, Yorktown Heights, NY, November 5, 2007. *http://www.cs.umn.edu/~agupta/wsmp*.

[12] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[13] Matthias Bollhöfer, Yousef Saad and Olaf Schenk. ILUPACK - preconditioning software package. Available online at *http://www.math.tu-berlin.de/ilupack*, January 2006.

[14] Y. Saad and H. van der Vorst. Iterative Solution of Linear Systems in the 20th Century. *Journal of Computational and Applied Mathematics*, 123(1-2):1–33, 2000.