RC24737 (W0901-110) January 30, 2009
Computer Science

IBM Research Report

An Empirical Analysis of Iterative Solver Performance for SPD Systems

Thomas George
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112

Anshul Gupta
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Vivek Sarin
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

An Empirical Analysis of Iterative Solver Performance for SPD Systems

Thomas George [*]
Department of Computer Science, Texas A&M University,
College Station, TX-77843-3112.

Anshul Gupta [†]
Department of Mathematical Sciences,
IBM T. J. Watson Research Center, Yorktown Heights, NY-10598.

Vivek Sarin [‡]
Department of Computer Science, Texas A&M University,
College Station, TX-77843-3112.

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 $A \in C^{m \times n}$, typically known as the coefficient matrix, is sparse, $b \in C^m$ is the right hand side vector (RHS) and $x \in C^n$ 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, combined with the need to solve ever increasing sizes of the linear systems has continued to fuel strong research efforts in iterative solvers and their preconditioning methods.

[*]tgeorge@cs.tamu.edu
[†]anshul@watson.ibm.com
[‡]sarin@cs.tamu.edu

The effectiveness and performance of most preconditioners is not only problem dependent, but also fairly sensitive to the choice of their tunable parameters. A typical practitioner has an overwhelming number of choices of solvers and preconditioners and their parameters in several black-box solver packages, such as PETSc [2], Trilinos [21], ILUPACK [25], Hypre [1], and many others [12]. 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. Survey articles by Benzi [3] and Saad and van der Vorst [28] describe the relative strengths and weaknesses of a wide range of preconditioned iterative schemes from a theoretical perspective. There have been a few empirical studies in the past [4, 6, 15], typically focused on a single preconditioner. Despite the widespread use of preconditioned iterative solvers, there has not been an extensive empirical evaluation of the readily available implementations of most common classes of preconditioners. This paper is an attempt to fill a part of this void for systems with symmetric positive definite coefficient matrices. To the best of our knowledge, it provides the most extensive practitioner-centric empirical evaluation to date of a number of popular and promising general purpose preconditioners available in black-box solver packages.

We expect our study to provide insights into the relative effectiveness of these preconditioners for problems from a variety of domains. Prospective users could benefit from our experimentation with the important parameters associated with the preconditioners. In addition to the users, the study could also provide guidance to the current and future developers of preconditioner and solver packages on ways and areas of potential improvement to their software. The main contributions of this work are as follows:

Benchmarking methodology: We introduce a methodology for a rigorous comparative evaluation of various preconditioners, including the use of some relatively simple but powerful metrics to facilitate a credible ranking of solver configurations (combinations of solver package, preconditioners, iterative method, and solver and preconditioner parameters). Notable among these metrics are memory-time product and area under the curve for performance profiles (Section 3).

Performance analysis infrastructure: We developed a semi-automated system for the collection, analysis, and visualization of relative performance data. It runs experiments and collects performance data (time, memory, error norm, etc.) for all combinations generated from a user specified a set of linear systems, a set of hardware configurations (number of CPUs and memory limits), and sets of values of various solver and preconditioner parameters. This is achieved via a data collection unit composed of both serial and parallel driver programs and associated scripts for some widely used solver packages. Subsequently, the analysis and reporting unit of the system performs various comparative and sensitivity analyses within and across pre-specified groups of solver configurations using the collected performance data.

Extensive empirical evaluation: Using the above system, we evaluate a suite of preconditioners based on the incomplete factorization, sparse approximate inverse, and the algebraic multilevel schemes available in packages such as PETSc, Trilinos, Hypre, Ilupack and WSMP. We compare the robustness, speed, and memory consumption of these preconditioners on a set of benchmark problems and present results that can serve as guidance to practitioners. For

packages that provide support for parallel execution, we collect and present performance data on multiple processors.

Good default configurations: For each combination of solver package and preconditioner, we identify the best overall choice of solver and preconditioner parameters on a suite of diverse problems. These observations can be used for choosing good *default* configurations for each package-preconditioner combination.

Fine tuning of parameters: In addition to determining good default configurations for each preconditioner implementation, we also study how sensitive the performance of a certain preconditioner is to parameter choices and which parameters have the greatest impact on performance. This analysis sheds light on the reliability of the default configuration and provides guidance for fine tuning the parameters to a specific problem or class of problems.

Choice of package-preconditioner combination: We simultaneously project the performance of all package-preconditioner combinations included in this study along three carefully chosen dimensions involving time, memory, and robustness to allow a ready comparison of the relative strengths of various implementations. We perform this comparison for the overall best set of parameters as well as for problem specific best set of parameters for each preconditioner implementation because their relative rankings can be different under the two scenarios.

Parallel performance: We extend our empirical comparison of various preconditioner implementations to up to 64 CPUs. In addition to traditional performance metrics like parallel efficiency and speedup, we also study impact of parallelism on the choice of parameters.

The remainder of the paper is organized as follows: Section 2 provides details of our experimental set up, including matrix collection, solver packages and the preconditioner included in the study, and the hardware used. Section 3 gives an overview of the various metrics that are used to rank the performance of individual solver configurations as well as that of package-preconditioner combinations. In section 4, we present the detailed empirical evaluation methodology and results. We describe our performance analysis framework and a prototype implementation in Section 5. Section 6 contains concluding remarks and possible directions for future work.

2 Experimental Setup

In this section, we present the details of our experimental setup. These include an introduction to the solver packages, preconditioners and their parameters, descpriptions of the test matrices and the hardware platform, and the specifics of our experimental approach.

2.1 Software Packages

We included the following packages in our study, which we believe are likely to be of most value to researchers and practitioners. These include well-established packages that include most commonly used preconditioners, as well as research packages with recently published general purpose preconditioners.

PETSc - Release Version 2.3.3-p0: PETSc [2], developed at Argonne National Laboratory, is implemented in C and has extensive documentation available for a new user with a plethora of informative examples demonstrating all the important aspects of the software package. The main goal of the PETSc project is to equip a user with the tools necessary for building scalable scientific

applications. PETSc provides efficient implementations for all the commonly used Krylov subspace methods as well as fixed pattern and threshold based incomplete factorization preconditioners. Even though a wide range of preconditioning schemes are available via interfaces to external packages, we were not able to configure PETSc to use external packages (except BlockSolve95 [23]) due to lack of support for 64-bit compilation.

Trilinos - Release Version 8.0.3: Trilinos [21] was developed at Sandia National laboratories and its main focus is to provide parallel solvers and libraries in an object oriented framework. Trilinos is composed of a number of self contained independently developed packages that could be used as a stand-alone application or in conjunction with other packages that support a minimal set of prerequisites for the interfaces. A suite of object oriented preconditioners are available in the Ifpack [30] and ML [13] packages. The AztecOO package, which provides an object oriented interface to the popular Aztec solver library and the recently released Belos package, contains implementations of the Krylov subspace methods. Ifpack supports a suite of Jacobi-style and incomplete factorization-based preconditioners whereas the ML package provides variants of algebraic multigrid type of preconditioners based on smoothed aggregation.

Hypre - Release Version 2.0.0: Hypre [1], developed at Lawrence Livermore National Laboratory, is designed primarily for the solution of large, sparse linear systems of equations on massively parallel systems. Hypre provides four different logical interfaces, namely, structured, semi-structured, finite element and linear algebraic. In addition to incomplete factorization based preconditioners (Euclid [22]), Hypre also has parallel implementations for approximate inverse based (ParaSails [7, 8]) and algebraic multigrid based (BoomerAMG [20]) preconditioners.

Ilupack - Dev. Version 2.2: Ilupack [25] was developed at Technische Universität Berlin and it contains implementations of inverse-based multilevel ILU preconditioners that controls the growth of the inverse triangular factors for both real and complex matrices. In addition to the standard static reordering schemes, it also includes the ARMS ordering schemes such as INDSET and ddPQ [27].

WSMP - Dev. Version 8.7: The Watson Sparse Matrix Package (WSMP) [17, 18], developed at IBM, contains serial and parallel sparse direct solvers as well as CG and GMRES solvers with new incomplete factorization based preconditioners [19].

2.2 Preconditioners and Parameters

We now describe the preconditioners that we use in our study. Table 1 provides a list of these preconditioners and the corresponding solvers. We study primarily three broad classes of preconditioners, namely incomplete factorization, sparse approximate inverse, and algebraic multigrid. Although the current study's scope is limited to symmetric positive definite systems, we do use the GMRES solver becuase it performed better than CG in certain cases. We also included the unsymmetric incomplete LU factorization preconditioner for packages in which we found it to perform better than the corresponding incomplete Cholesky factorization based preconditioner.

Level-of-fill based incomplete Cholesky factorization, IC(k): An important class of preconditioners for SPD systems is based on the incomplete Cholesky factorization of the coefficient matrix. There are several variations of incomplete factorization that differ in the rules for dropping entries to compute the incomplete factors. One scritctly positional criterion for dropping is based on what is known as the *level of fill*, which is a measure of "closeness" of a fill entry to the original entries in the coefficient matrix (please refer to the book by Saad [29] or the survey by Benzi [3] for a formal definition). In IC(k) factorization, all fill-in entries at levels exceeding k are dropped. An important advantage of IC(k) preconditioners is that the sparsity pattern can be determined a-priori by a symbolic factorization step and the cost of constructing the preconditioner is amortized

Package	Solver	Preconditioner
PETSc	CG	IC(k)
PETSc	CG	BlockSolve95
PETSc	GMRES (30,65,100)	ILUTP
Trilinos	CG	IC(k)
Trilinos	CG	ICT
Trilinos	GMRES (30,65,100)	ILUT
Trilinos	CG, GMRES (30,65,100)	Algebraic Multigrid (AMG)
Ilupack	CG	Multilevel-ICT
Hypre	CG	IC(k)
Hypre	CG	Algebraic Multigrid (AMG)
Hypre	CG	Sparse Approximate Inverse (SAI)
WSMP	CG/GMRES	ICT

Table 1: Preconditioners and solvers used in each package for SPD matrices. The numbers accompanying GMRES refer to the restart values.

when solving multiple linear systems with the same sparsity pattern. Often, depending on the implementation, when the parameter $k > 0$, it is supplemented with additional parameters to handle fill levels higher than 0. The following parameters are used in the the implementations of IC(k) that we study in this paper.

1. *Highest fill level,* k, is the fundamental parameter in all implementations of the IC(k) preconditioner and denotes the level beyond which all fill-ins are dropped. We experimented with values of 0, 1, and 2 for k for PETSc and Trilinos, and 0, 1, 2, 4, 6, and 8 for Hypre.

2. *Fill factor* specifies an upper bound on the amount of memory used by the preconditioner for levels of fill greater than zero. A fill factor γ denotes that the preconditioner would not use more that γ times the number of nonzeros in the original matrix. This parameter is used only in PETSc, but we emulate it in Hypre to permit a one-to-one comparison with PETSc. We used four different values, 3, 5, 8, and 10 in our experiments.

3. *Maximum nonzeros per row* is used in Hypre's implementation of IC(k) to control memory usage. It denotes the maximum number of nonzeros allowed per row in addition to those in the original matrix. We experimented with values of 5 and ∞ (i.e., no limit) for this parameter. In addition, we used four other values to emulate fill factors of 3, 5, 8, 10 by using a value of $(f - 1)$ times the average number of nonzeros per row in the original matrix to emulate a fill factor of f.

4. *Drop threshold* can be specified in Hypre's IC(k) implementation to restrict the amount of fill by dropping factor entries below the threshold. However, we do not use this parameter in our study because it is listed as a partly implemented feature in the documentation.

5. *Diagonal perturbation parameters* can be used in Trilinos to increase diagonal dominance of the matrix and to reduce the condition number of the preconditioner for highly illconditioned matrices. We did not use these parameters in our experiments because we found that their use resulted in excessive execution times.

BlockSolve95 [23] implements a hybrid strategy that incorporates ideas from both level-based and threshold based incomplete factorization preconditioners. The sparsity of the factors are guaranteed by retaining only the largest elements such that the memory usage is no larger than that required by a ILU(0) preconditioner. BlockSolve95 has the added advantage that no parameters need to be specified.

Threshold based incomplete factorization (ICT, ILUT, and ILUTP: ICT and ILUT use incomplete Cholesky and LU factorization, respectively, that controls fill-in by means of a dual dropping strategy based on a numerical threshold and an upper limit on the number of fill-ins in each row or column. The user typically specifies a value for the *drop tolerance* (τ) and a *fill factor* (f). A new fill-in is dropped if its magnitude is less than τ times a chosen metric, such as the corresponding diagonal value, or any norm of the row/column under consideration. After dropping based on τ, if a row or column of the incomplete factor is still left with more entries that the permissible limit of f times the original number of entries, then the smallest excess entries are dropped. An experimental study [6] of ILUT type preconditioners revealed a number of weaknesses, some of which can be mitigated by adding pivoting to ILUT and performing what is typically known as ILUTP preconditioning [29].

The various parameters used in PETSc and Trilinos for this class of preconditioners are listed below.

1. *Drop tolerance* specifies the numerical threshold used to drop the fill-in values. Lower values for drop tolerance result in more accurate preconditioners but result in higher memory consumptions, and vice versa.

2. *Fill factor* specifies an upper bound on the number of nonzeros in each row or column of the preconditioner.

3. *Pivot threshold* is the same as the pivot threshold used conventionally in sparse exact factorization and controls growth in the factors. It determines the smallest permissible magnitude of a diagonal entry relative to the largest entry in the corresponding column.

Inverse based multilevel-ICT (MLICT): This class of preconditioners, available in Ilupack, attempts to address the robustness and efficiency issues of ILUT preconditioners. Ilupack uses a hierarchical ILUT factorization strategy combined with static pre-ordering and diagonal pivoting such that the norms of the inverses of the triangular factors are bounded below a user specified threshold. The incomplete factorization process is performed for each column of L and column of U under consideration if the estimated norm of the inverse of the resulting triangular factor is below the prescribed bound. Otherwise, the associated row and column is pushed to the end [5]. The main user controlled parameters are described below.

1. *Drop tolerance* is similar to the one used in ILUT based preconditioners.

2. *Norm of inverse estimate* provides an upper bound on the norm of the inverse of the triangular factors.

Sparse Approximate Inverse (SAI): The problems inherent in using incomplete factorization based variants are partially addressed by preconditioners based on sparse approximate inverses [4]. Depending on the algorithms used for finding the sparse inverse, approximate inverse based preconditioners could be fairly robust in practice and easily parallelizable. However, these preconditioners usually incur a high initial setup cost and the efficacy and the cost of applying the preconditioner depends on the choice of the sparsity pattern. Hypre ParaSails preconditioner uses

a-priori knowledge of sparsity patterns and Frobenius norm minimization to generate an approximate inverse. For SPD matrices, a symmetric factored approximate preconditioner is generated. ParaSails uses three main parameters for controlling the accuracy and the cost of the preconditioner, and these are described below.

1. *Threshold* controls the sparsification of the coefficient matrix such that it can be used to generate the *a-priori* sparsity pattern by dropping elements that are below the specified value. The range of values for *threshold* is [0,1]. One can also specify a negative value for *threshold* such that its absolute value dictates the percentage of nonzeros that must be dropped. The exact value for *threshold* is determined automatically in this case.

2. *Number of levels* controls the memory usage of the resulting preconditioner. ParaSails uses *a-priori* sparsity patterns that are powers of sparsified matrices. For example, if a value of 2 is used for the number of levels, then the sparsity pattern corresponds to the power of 3 of the sparsified matrix. Typical values are 0, 1 and 2 with the default value being 1.

3. *Filter* is a numerical threshold used to reduce the cost of applying the preconditioner by further dropping elements from the computed approximate inverse. This parameter works similar to *threshold* and one could also specify a negative value if it has to be determined automatically based on a percentage of sparsity that is desired. For examples, if $filter = -0.9$, then the threshold is calculated such that 90% of the non zeros in the computed preconditioner are dropped.

BoomerAMG: Another preconditioner that has generated a lot of interest for use in black-box solvers is Algebraic Multigrid (AMG). BoomerAMG is the parallel implementation of the classical AMG [26] available in Hypre and it can be used either as a solver or as a preconditioner. BoomerAMG has a number of parameters which allow the user to choose multiple coarsening algorithms, relaxation and interpolation schemes. For our experiments, we used the preconditioners with multiple values for the most important parameters, which are described in detail below.

- *Smoothers* - Hybrid symmetric Gauss-Seidel/Jacobi [1] was the smoother of choice since CG was the default solver used for SPD matrices.

- *Coarsening schemes* - We experimented with three different coarsening schemes namely, Falgout (FALG), Parallel Modified Independent Set (PMIS) and Hybrid Modified Independent Set (HMIS) coarsening schemes.

- *Number of levels for aggressive coarsening* (AGG) - The value given to this parameter sets the number of levels for which aggressive coarsening must be applied starting from the finest level. We experimented with values of 10 and 0 (i.e., no aggressive coarsening) for all the three coarsening schemes.

- *Interpolation type* - For the Falgout coarsening scheme, we used the recommended multi-pass interpolation for use with aggressive coarsening and the classical interpolation while not using aggressive coarsening. For the PMIS and HMIS coarsening schemes, we used the "extended classical interpolation" scheme as recommended in the user guide.

- *Strong threshold* - The value specified for this parameter has the potential to affect the effectiveness of both coarsening and interpolation schemes. Strong threshold value determines whether two points are strongly or weakly connected. High values lead to cheaper but less

effective preconditioners whereas low values result in an expensive preocnditioner with better convergence properties. Since the choice of strong threshold values is highly problem dependent, we experimented with multiple values of 0.25, 0.5, 0.7 and 0.9.

ML: Trilinos-ML preconditioner is a parallel implementation of the smoothed aggregation approach for AMG. Similar to BoomerAMG, ML also has a large number of parameters that can be fine-tuned. There are four sets of default parameters for use with problems arising from different domains. We experimented with the default parameters corresponding to classical smooth aggregation (SA), classical two level aggressive coarsening (DD), three level aggressive coarsening (DD-ML) [13]. In addition to the values suggested in the user manual, we also experimented with other values for the following parameters.

- *Number of Levels* - For each recommended choice of the default set of parameters (SA, DD or DD-ML), we experimented with multiple values for the number of levels. This parameter affects the memory and time required for creating the preconditioner but could provide savings in the form of faster convergence or increased robustness.
- *Number of Smoother Sweeps* - Using higher number of smoother sweeps could affect the time required for convergence. A higher number of smoother sweeps results in an increased computational cost per iteration, but has the potential to reduce the number of iterations required.
- *Maximum Size of Coarse Grid* - This parameter specifies the lower limit on the size of the coarse grid. ML will coarsen the grid further only if the grid size for a level is higher than this value. Changing this parameter may or may not affect the convergence properties and its effect is dependent on the total number of levels as well as the aggregation scheme.

Table 2 lists the specific preconditioners and the values of the tunable parameters that were experimented with. In all, 897 different combinations of solvers, preconditioners, and parameters were tried for the single processor case. The total number of solver configurations including all the serial and parallel cases aggregate to 4643.

2.3 Choice of solvers

There are mainly two classes of iterative solvers, namely, stationary and non-stationary. Stationary iterative methods such as Jacobi, SOR involve some form of splitting of the coefficient matrix and the solution at the jth iteration is represented as a linear combination of the solution of the $j-1^{th}$ iteration. Non-stationary methods include popular Krylov subspace methods such as CG and GMRES that are characterized by the subspaces in which the solution iterate x_j lies. Different Krylov subspace methods differ in the criteria they use for selecting a vector in the subspace. For example, the criterion could be the minimization of the A-norm of the error (CG) or it could be the minimization of the two norm of the residual over the Krylov subspace (GMRES). Even though stationary methods are simple to implement, it is widely accepted [3] that they are not very effective for many problems of practical interest. Therefore, for most packages, we restrict our experiments to CG, which is one of the most popular solvers for symmetric positive definite (SPD) systems. However, if the preconditioning scheme had the potential to result in non-SPD preconditioners, then GMRES with restart values of 30, 65 and 100 was used.

Package	Solver	Preconditioner	Orderings	Parameters
PETSc	CG	BlockSolve95	RCM, ND	-
	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, 4, 6, 8 *Maximum nnz per row:* 5, F3, F5, F8, F10, ∞
	CG	ParaSails	RCM, ND NONE	*Number of levels:* 0, 1, 2 *Threshold:* 0, 0.01, 0.1, -0.75, -0.9 *Filter:* 0, 0.001, 0.05, -0.9
	CG	BoomerAMG	RCM, ND NONE	*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)	ML	RCM, ND NONE	*SA Number of Levels:* 6, 10 *DD Number of Levels:* 1, 2 *DD-ML Number of Levels:* 3, 5 *SA,DD,DD-ML Smoother sweeps:* 1, 2 *SA Maximum coarse size* 16, 32 *DD,DD-ML Maximum coarse size* 64, 128
ILUPACK	CG	Multilevel ICT	RCM, ND, AMF INDSET, 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 RCM/ND	Eight cycles of self tuning

Table 2: Description of the package specific preconditioner parameters. For Hypre-IC(k), values of F3, F5, F8, F10 for maximum number of nonzeros allowed per row correspond to a fill factor of 3, 5, 8 and 10 respectively.

2.4 Matrix Reordering

Previous studies [29] have shown that a suitable reordering of the coefficient matrix can reduce the fill and potentially have a significant impact on the performance for incomplete factorization based preconditioners. Hence, wherever applicable, the matrices were first re-ordered using either the Reverse Cuthill Mckee ordering (RCM) [9] or the Nested Dissection ordering (ND) [14, 16]. 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) [27].

2.5 Test Matrices

Since our objective is to detect general performance trends among the different preconditioned iterative solver schemes and our analysis is purely empirical, it is imperative that the test matrices represent a spectrum of the problems for which computational simulations are extensively used. To this effect, we chose the matrices from a wide range of applications spanning fluid dynamics, sheet metal forming, electric circuit simulation, chemical process simulation, optimization etc. The details of the SPD matrices are shown in Table 3. Most of the matrices are obtained from Tim Davis' collection [10] and the remaining ones are obtained from some of the applications that use

WSMP.

Matrix	N	NNZ	Application
90153	90153	5629095	Sheet metal forming
af_shell7	504855	17588875	Sheet metal forming
algor-big	1073724	84317460	Static stress analysis
audikw_1	943695	77651847	Automotive crankshaft modeling
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
garybig	42459173	238142243	Circuit simulation
G3_circuit	1585478	7660826	Circuit simulation
hood	220542	10768436	Automotive
inline_1	503712	36816342	Structural engineering (F.E.M)
kyushu	990692	26268136	Structural engineering (F.E.M)
ldoor	952203	46522475	Structural analysis
msdoor	415863	20240935	Structural analysis
mstamp-2c	902289	70925391	Metal stamping
nastran-b	1508088	111614436	Structural analysis
nd24k	72000	28715634	3D mesh problems (ND problem set)
oilpan	73752	3597188	Structural analysis
parabolic_fem	525825	3674625	C.F.D, Convection-diffusion
pga-rem-1	5978665	29640547	Power network analysis
pga-rem-2	1480825	7223497	Power network analysis
qa8fk	66127	1660579	F.E.M Stiffness matrix for 3D acoustic problem
qa8fm	66127	1660579	F.E.M Mass matrix for 3D acoustic problem
ship_003	121728	8086034	Structural analysis - Ship structure
shipsec5	179860	10113096	Structural analysis - Ship section
thermal2	1228045	8580313	Steady state thermal problem
torso	201142	3161120	Human torso modeling (F.E.M)

Table 3: SPD test matrices with their order (N), number of non-zeros (NNZ) and the application area of origin

2.6 Performance Measurement Methodology

We now describe our methodology for evaluating all the preconditioned iterative solvers, performance metrics and the adjustments made for performing a fair comparison.

2.6.1 Evaluation Steps

In order to make the evaluation as uniform as possible, we adhered to the following steps in all the driver programs.

- When using more than one processor, ParMetis [24] is used to partition the rows of the matrix and distributed appropriately.
- We then reorder the local matrix on each processor according to a pre-determined reordering scheme and set the right hand side vector b as a vector of all ones.
- Diagonal scaling is performed on the linear system before starting the solution process.
- The initial guess of the solution for the iterative process is the zero vector.
- We used right preconditioning since it was the default for all the packages except PETSc and it allowed us to have a uniform convergence criteria based on the true residual for all the experiments. This choice of right preconditioning was also influenced by a similar study conducted on ILU preconditioners [6].
- The iterations are stopped when the number of iterations reaches 1000 or when the relative residual norm drops below $1e-5$.
- After the solver iterations have completed, the solution and the linear system is scaled back and all the relevant performance metrics are computed.

2.7 Hardware Specifics

2.7.1 Compilation/Libraries

All the packages were compiled using IBM compilers xlf (Fortran), xlc (C) or xlC (C++) in 64-bit mode with the -O3 optimization flag. The Engineering Scientific Subroutine library (ESSL) was linked in to provide BLAS routines. The page size for text and data was set to 64 KB.

2.7.2 Platform

The experiments were conducted on up to 64 processors on an IBM HPC cluster 1600, based on the Power5+ IBM processor running the 64-bit version of AIX (version 5.3). Each of the p5-575 nodes on the cluster has 16, 1.9 Ghz Power5+ processors. A memory limit of 24 GB per node and a wall time limit of 4 hours was used for each empirical trial involving a single matrix and a solver configuration.

2.7.3 Performance Metrics

For each evaluation trial, (i.e., each application of preconditioned iterative solver to a matrix), we measure the following performance metrics.

Time taken (in sec), i.e., the total time required for creating the preconditioner and actually solving the linear system. We measure this using timing calls before and after the appropriate routines. In the case of multiple processor runs, the reported time is the *maximum* among all the processors. All the experiments are performed using a public distributed-shared memory machine and there could be slight variations in the computational time depending on the load on the machine. Therefore, the reader should be wary of giving undue consideration to minor variations (under 10%) in the timing measurements.

Memory usage (in bytes), i.e., the amount of memory allocated on the heap during the preconditioner creation phase. Since the packages are designed by different groups, they vary in their implementation. For example, there is a single call to set up the preconditioner and the Krylov subspace vectors for GMRES in Hypre and PETSc, whereas, the other packages had separate memory allocation phases for the preconditioner and solver. For the case of PETSc, we overcame this problem by initially using a restart of zero during the preconditioner creation stage and then setting it to the desired value just before the solution phase. In the case of multiple processor runs, we compute the *cumulative* memory usage across all the processors. If GMRES is used as the iterative solver, we also add the memory required for storing the subspace vectors ($8 \times numRows \times restart$) to the total observed memory.

Relative error norm, i.e., the ratio of L2 norm of the final error to that of the initial error. For computing the error, we use an approximation of the actual solution obtained using a direct solver.

Failure. In general, a method is considered to have been a failure for a particular problem if any of the following is true.

1. The total time is above 4 hours.
2. The final relative error norm is above 0.02.
3. The memory consumption per node exceeded the limit specified (16 GB while using up to 8 processors and 24 GB when using the entire node).

3 Benchmarking Methodology

In this section, we present the metrics chosen for evaluating the collective and problem-specific performance of the solver configurations used in this study. Our choice of metrics was heavily influenced by the guidelines proposed for benchmarking optimization software as described in [11].

3.1 Performance Data

Let $\mathcal{S}$ be the set of preconditioned solver configurations ($|\mathcal{S}| = m$), $\mathcal{P}$ be the set of linear systems/problems to be solved ($|\mathcal{P}| = n$) and let μ represent any performance measure that takes a specific value for each evaluation trial, i.e., application of a solver configuration to a problem. Examples of performance measures include time taken, sensitivity to fine-tuning, memory usage, etc. We assume that each of the solver configurations is applied to all the problems so that the performance measure values can be represented as a $n \times m$ matrix μ, where the ps^{th} element corresponds to the performance $\mu_{p,s}$ of solver configuration s with respect to problem p. These performance values $\mu_{p,s}$, often, are not well-defined due to solver configuration failure and other practical limitations. Without loss of generality, we assume that lower values of performance values are desirable and subsequently represent ill-defined values corresponding to solver configuration failures with a very high value (∞). Lastly, the solver configurations also happen to be partitioned into disjoint groups. To be precise, there exists a mapping from each solver configuration $s \in \mathcal{S}$ to a *configuration group* $c \in \mathcal{C}$. In this study, for the single processor scenario, the set $\mathcal{S}$ consists of all the 897 solver configurations. If we consider a configuration group to be a package-preconditioner combination, then there are 11 different groups in the set $\mathcal{C}$ corresponding to each specific preconditioner implementation (PETSc-IC(K), Trilinos-IC(K), Hypre-ParaSails etc.). For example, in the case of Hypre-ParaSails, there are 99 different solver configurations corresponding to the various

combinations of solver, ordering and preconditioner parameters that belong to this group. These partitionings will be more relevant in the context of problem specific performance analysis. Table 4 shows an artificial example of performance data (time taken) for a small set of solver configurations and problems.

Solver Configuration	Configuration Group	$\mu_{p,s}$			$r^{\mu}_{p,s}$		
		p_1	p_2	p_3	p_1	p_2	p_3
s_1	c_1	7	2	3	3.5	1	1
s_2	c_2	3	4	6	1.5	2	2
s_3	c_1	2	∞	8	1	∞	8/3

Table 4: An example of performance data with solver configuration failures represented with ∞.

3.2 Collective Performance

Given the performance data for a particular measure, it is straightforward to compare the effectiveness of the methods with respect to a single problem. Specifically, we assume that methods with lower performance values are better. However, comparing methods based on their collective performance requires calibration across the problems. A natural solution in this scenario is to consider the normalized performance values $r_{p,s}$, otherwise known as the performance ratios of the methods for each problem:

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

i.e., ratio of the actual performance value of solver configuration s to the best (least) value over all solver configurations for the problem p. Note that $r_{p,s} \geq 1$ for all (p, s) and is equal to 1 for at least one solver configuration s for each problem p. It seems reasonable to expect that the average performance ratio η_s of each method given below would be a fair indicator of the effectiveness of the method with respect to that performance metric

$$\eta^{\mu}_{s} = \frac{1}{n} \sum_{p=1}^{n} r^{\mu}_{p,s} \ .$$

In practice, however, η_s is often not very useful since a single failure for a solver configuration s can make its average performance ratio η_s ill-defined, making it difficult to compare the different methods. One simplistic solution for handling this issue is to only consider problems that have well defined performance ratios, but this would not be fair to methods that actually solve the harder problems not solved by all the methods. A more 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 performance profiles, which we now describe.

3.3 Performance Profile

A performance profile [11], simply defined, 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 configuration s with respect to the measure μ:

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

$\rho_s^\mu(\tau)$, therefore, denotes the fraction of the problems that can be solved where the available resources are within a factor of τ times that required in the best scenario for each problem.

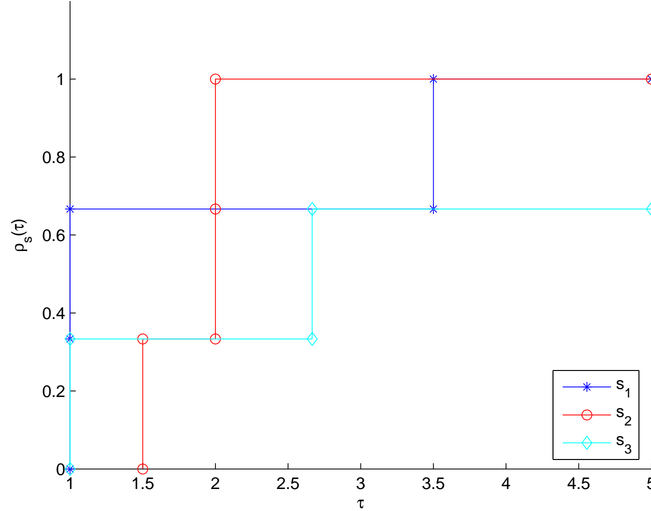


Figure 1: Performance profile example.

Figure 1 shows the performance profiles for the data in Table 1. This plot reveals valuable information that might not be apparent from average performance ratio even when it is well defined. Specifically, one can compare methods under circumstances where a user specifies an additional success threshold θ on performance ratio, i.e., any solver configuration that results in performance value that is θ times greater than that of the best possible value is considered a failure.

For example, it is clear that if one were restricted to a performance ratio of $\theta = 1.3$, then s_1 is clearly the best method followed by s_3 and then by s_2 since they solve 2, 1 and 0 problems respectively. The problems solved with a performance ratio > 1.3 are effectively equivalent to failures in this case. On the other hand, if the threshold was chosen to be $\theta = 4$, then both s_1 and s_2 are equally good since they both solve all the three problems and that too with identical average performance ratio $= 11/6$. Comparing with s_3 is somewhat tricky since it only solves 2 out of the 3 problems but has a lower average performance ratio $=8/6$.

3.4 Solver Quality Measure

If a performance profile plot consists of numerous curves, then it might not be possible to visually determine the relative ordering or pick the best solver configuration. The situation is exacerbated when the candidate solver configurations are fairly close to each other. To address this issue and to eliminate human intervention in the early analysis stages where solver configurations number in hundreds, we propose to use the area under the performance profile curves for ranking the various solver configurations. The area under the curve (AUC) provides a longitudinal summary of multiple assessments at all possible performance ratios of interest. For a performance ratio threshold of θ, the only τ values of interest are those from 1 to θ and so the relevant area under curve $AUC_s^\mu(\theta)$ is the area under the cumulative distribution curve upto θ, which is given by

$$AUC_s^\mu(\theta) = \theta - \frac{1}{n}\sum_{p \in \mathcal{P}} \min(\theta, r_{p,s}^\mu) \ .$$

This formulation, ignores all the performance ratios of solver-configurations outside the range of the threshold θ and effectively considers those as failures. For the example in Figure 1, we can now readily compute this AUC measure for the solver configurations s_1, s_2 and s_3 for any finite θ. In particular, we will notice that AUC of s_3 is comparable to that of s_1 and s_2 for smaller values of θ and becomes progressively worse as the threshold increases.

Note that for the special case where there are no solver configuration failures and all performance ratios are well defined, the area under curves turns out to be directly related to average performance ratios when the performance ratio threshold θ is set to the maximum performance ratio among the competing solver configurations.

The area under the performance profile curve of a method can readily account for failures and is a good indicator of the effectiveness of that method. Hence, we use the AUC values to determine the "best" method with respect to some of the important aspects of the solver configuration that are likely to be of interest to practitioners, e.g., time taken, memory usage, sensitivity of time/memory usage with respect to fine-tuning, etc. Note that one could perform a similar comparison where the performance metrics μ is itself a combination of the core metrics such as time and memory depending on the user requirements. A critical choice in the AUC-based comparison is the threshold for performance ratio θ since this can significantly affect the AUC and thereby, the relative quality of the different methods. In the current study, we choose this threshold to be equal to 10 both for time taken, memory usage as well as the sensitivity metrics. 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 given problem was effectively a failure. Note that this is in addition to the failure criteria described in 2.7.3.

3.5 Problem Specific Performance

The notion of problem specific performance is relevant mainly in the context of solver configuration groups that are amenable to fine-tuning. Each configuration group considered in this study involves all the solver configurations with the same preconditioner implementation i.e., a package-preconditioner combination. The basic premise is that a configuration group be evaluated not in terms of a single "best" member (e.g., specific parameter configuration for a preconditioner), but rather in terms of how all of its members can collectively solve the entire set of problems. In other words, there is no penalty if the individual members are not robust, as long as each problem is solved efficiently by some solver configuration in the group. Such a metric is relevant in scenarios where one can afford to fine-tune the parameters to obtain the best possible performance for each problem.

Formally, we define *best problem-specific performance* $\mu_{p,c}^{PSB}$ of a configuration group c for problem p as the best performance value among all the solver configurations in c, i.e.,

$$\mu_{p,c}^{PSB} = \min_{s \in c} \mu_{p,s} \ .$$

Note that $\mu_{p,c}$ is an aggregation of the performance values of the member solver configurations. This is in contrast to a possible alternate criterion that directly considers the performance values of a particular winning member solver configuration (for example, the solver configuration with the best performance profile area across all problems).

3.5.1 Fine-tuning Gain

An important issue in the context of problem-specific performance analysis is related to the benefits of fine-tuning. Specifically, one would like to quantitatively measure how sensitive each solver

configuration group is to fine-tuning. In order to do this, we assume that each solver configuration group $c \in \mathcal{C}$ has a default configuration given by the member solver configuration $def(c)$. The benefits of fine-tuning within this configuration group can then be captured in terms of new metric *fine-tuning gain* $\nu_{p,c}$ defined as

$$\nu_{p,c} = \frac{\mu_{p,c}^{PSB}}{\mu_{p,def(c)}} .$$

As one can readily observe, the fine-tuning gain is defined for every configuration group and problem and can be considered a meta performance metric associated with the configuration groups. However, it is different from the core performance metric μ in the sense that is not defined for each solver configuration-problem trial. This fine-tuning gain can be viewed as both an indicator of potential benefits for searching within the solver group (i.e., tweaking parameters) as well as the in-adequateness of the default solution. When the default solution corresponds to the one with best collective performance, the fine-tuning gain also reveals the extent of specialization in the configuration group.

3.6 Configuration Group Quality Measures

There are two important criteria for evaluating solver groups in the context of fine-tuning. The first one is the actual performance assuming fine-tuning can be done, which is captured by the problem-specific performance values $\mu_{p,c}^{PSB}$. However, to compare a given set of solver groups, one would have to aggregate these problem-specific performance values over all the problems. The issues involved in this aggregation (ill-defined values, calibration across problems) are identical to those described in Section 3.3. Hence, the area under the problem-specific performance profile curves is a natural choice for a quantitative comparison of various solver groups.

3.7 Parallel Performance

In the case of multiple processor runs, the performance metrics of interest could be different from that obtained using a single processor. Therefore, a user might be interested in knowing how the relative performance of the solvers observed in a serial environment change in a parallel setting. For this purpose, we consider each multi-processor run to be part of a different hardware group. The various solvers are evaluated in each of these hardware groups separately and the AUC of performance profiles are used to study the behavior of the solvers across various hardware configurations. An important performance metric in a parallel scenario is the efficiency of the respective parallel implementations. Efficiency is computed as $\epsilon = \frac{1}{np} Time^{sp}/Time^{np}$ where $Time^{sp}$ is the best sequential time and $Time^{np}$ represents the time observed for np processors. A relatively high efficiency for large processors could either suggest that the solver can be parallelized efficiently or that the serial implementation is not optimal. Similarly, a low value of efficiency could suggest the existence of expensive sequential components or a poor parallel implementation. Therefore, a decision regarding whether a package is "highly" efficient or not cannot be made just by analyzing the efficiency values. Nevertheless, a comparison with similar implementations across different hardware groups could give an indication about the relative robustness and parallel performance.

4 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* (MTP). The total solution time includes the time to create the preconditioner and the time taken by the preconditioned CG or GMRES to obtain the approximate solution.

We chose MTP 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. However, for the sake of readers interested in winners specific to memory, time and robustness, the winning parameter combination can be readily obtained from Tables 6-10.

The remainder of this section is organized as follows. First, for each configuration group i.e., package-preconditioner, we analyze the performance of the various parameters and report on the best parameter combination over all the problems with respect to time, memory, and MTP. We then present the effects of parameter fine-tuning on the performance within each configuration group while using multiple number of processors. This is followed by a comparison of the default and problem specific best performance of the various configuration groups. Lastly, we comment on the relative importance of fine-tuning on performance among the configuration groups as well as the efficiency of the parallel implementations.

4.1 Performance Within Configuration Groups

We first begin by describing the various configuration groups in this study. These configuration groups are obtained from five general purpose preconditioner classes - level based incomplete factorization (IC(k)), threshold based incomplete factorization (ICT/ILUT), algebraic multigrid (AMG), and sparse approximate inverse (SAI). From Table 2, one can readily observe that these preconditioner classes correspond to concrete implementations spread among different packages and thus different configuration groups. We identify each of these configuration groups using the package and preconditioner class names, e.g., Trilinos-ILUT for both the ICT and ILUT implementations with different fill factors and drop tolerances. Our first goal is to determine the set of parameters for each configuration group that resulted in the best MTP over the entire problem suite. This parameter is chosen as the parameter combination that resulted in the maximum AUC with MTP as the performance metric μ as described in Section 3. Essentially, this amounts to plotting the performance profile curves for all the parameter combinations for a given preconditioner implementation using MTP as the performance criterion, and choosing the parameter combination corresponding to the curve with the maximum area under it. We perform detailed analysis only for the serial case and present interesting trends in the parallel case.

For the confguration groups composed of a package and preconditioner, we choose the best configuration with respect to the time taken, memory usage, robustness and MTP *across all the problems*. Robustness is measured in terms of the percentage of the problems solved as determined by the failure criterion in Section 2.7.3. In the case of time, memory and MTP, the winning

configuration is identified by examining the area under the respective profile curves with ties broken arbitrarily. 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. Tables 6-10 list the winning parameter configurations with respect to the time taken, memory usage, and MTP for each processor configuration. We also highlight interesting dependencies between time/memory usage and various elements of the solver configurations via performance profile curves (Figures 2-19) for relevant groups of solver configurations. For the sake of brevity, we use acronyms to describe the parameter choices both in the tables and legends of the figures. The reader can refer to Table 5 for an explanation of these acronyms.

Parameter Name	Values	Acronyms
Level of fill	0, 1, 2, 4, 6 ,8	LF0, LF1, LF2, LF4, LF6, LF8
Fill factor	3, 5, 8, 10	FILL3, FILL5, FILL8, FILL10
Maximum nonzeros per row	5, ∞	Nz5, NzINF
Drop tolerance	1e-2, 3e-2, 1e-3, 5e-4	DT1e-2, DT3e-2, DT1e-3, DT5e-4
Pivot threshold Inverse Norm Estimate	0.0, 0.1 10, 25, 100	NP, P IE10, IE25, IE100
Number of ParaSails levels	0, 1, 2	PLev0, PLev1, PLev2
Threshold	0, 0.01, 0.1, -0.75, -0.9	Th0, Th.01, Th.1, Th-.75, Th-.9
Filter	0, 0.001, 0.05, -0.9	Flt0, Flt.001, Flt.05, Flt-.9
Coarsening schemes	Falgout, Hybrid MIS, Parallel MIS	FALG, HMIS, PMIS
Strong threshold	0.25, 0.5, 0.7, 0.9	ST.25, ST.5, ST.7, ST.9
ML parameters	SA, DD, DD-ML	SA, DD, DD-ML
Smoother sweeps	1, 2	SS1, SS2
Maximum coarse size	1, 2	MCS16, MCS32, MCS64, MCS128
Number of ML levels	2, 3, 4, 5, 6, 10	Lev2, Lev3, Lev4, Lev5, Lev6, Lev10

Table 5: List of acronyms used to denote various parameter choices.

4.1.1 Level-based Incomplete Factorization

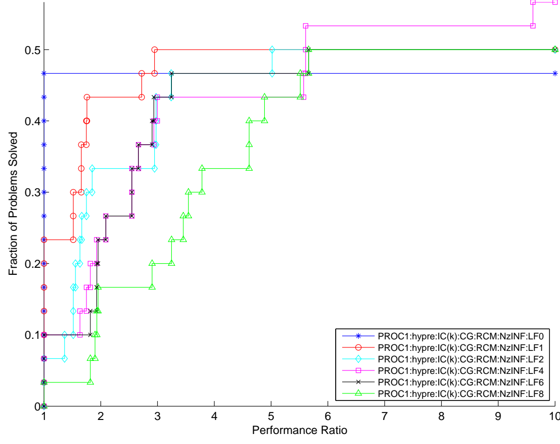
We experimented with the IC(k) implementations available in three packages PETSc, Trilinos and Hypre. Experiments were conducted with values of 0, 1, 2 for level of fill (LF) for matrices ordered using RCM and ND for PETSc and Trilinos. For higher values of level of fill, we experimented with values of 3, 5, 8 and 10 for the fill factor for PETSc and Hypre. Trilinos did not provide a user controlled parameter for manipulating the fill factor. For the Hypre-IC(k) preconditioner, we also experimented with values of 4, 6, and 8 for the level of fill and values of 5 and ∞ for the maximum number of nonzeros per row that is allowed in addition to the fill-in resulting from a level of fill 0 factorization. Table 6 summarizes the winning configurations with respect to time, memory, robustness and MTP. We also include the BlockSolve preconditioner results even though it can be considered as a hybrid of level based and threshold based incomplete factorization preconditioners.

Hypre:IC(k)

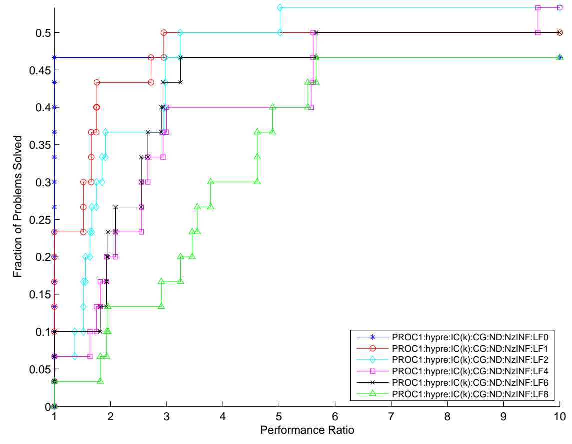
Figure 2 shows the variation of memory and time performance profiles for Hypre-IC(k) for different levels of fill K when no limit is placed on the fill factor (i.e., infinity). Increasing the level

Preconditioner	Memory Winner	Time Winner	MTP Winner
PETSc-IC(k)	CG,RCM,LF0 (1)	CG,RCM,LF0 (1)	CG,RCM,LF0 (1)
Trilinos-IC(k)	CG,RCM,LF0 (1 2 8 16 32 64) CG,ND,LF0 (4)	CG,RCM,LF0 (1 2 8 32) CG,RCM,LF1 (16) CG,RCM,LF2 (64) CG,ND,LF0 (4)	CG,RCM,LF0 (1 2 8 16 32 64) CG,ND,LF0 (4)
Hypre-IC(k)	CG,RCM,Nz5,LF1 (4 8) CG,RCM,Nz5,LF2 (2) CG,RCM,Nz5,LF4 (1 32 64) CG,ND,Nz5,LF4 (16)	CG,RCM,FILL1,LF4 (32) CG,RCM,FILL3,LF4 (1) CG,RCM,FILL10,LF2 (2 4) CG,RCM,FILL10,LF4 (64) CG,ND,Nz5,LF4 (16) CG,ND,FILL8,LF1 (8)	CG,RCM,FILL5,LF1 (4) CG,RCM,NzINF,LF0 (16) CG,ND,FILL5,LF1 (32) CG,ND,FILL8,LF1 (8) CG,ND,NzINF,LF0 (1 2 64)
PETSc-BlockSolve	CG,RCM (2 4 8 64) CG,ND,LF0 (1 16 32)	CG,RCM (1 2 8 64) CG,ND,LF0 (4 16 32)	CG,RCM (1 2 4 8 64) CG,ND,LF0 (16 32)

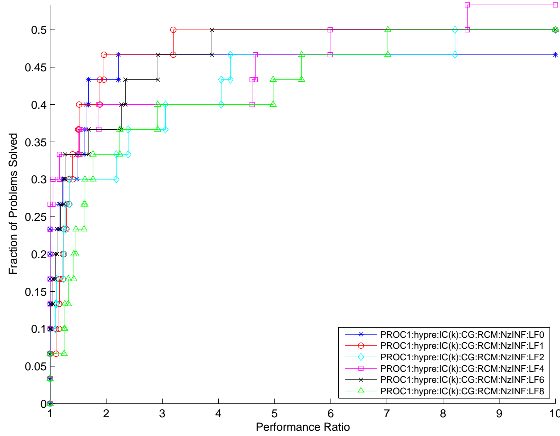
Table 6: Iterative solver configurations that resulted in the best performance with respect to time profile area, memory profile area, and MTP profile area for IC(k) preconditioners in PETSc, Hypre and Trilinos as well as Petsc-BlockSolve. The numbers enclosed by () denotes the processor number corresponding to the solver configurations. The configuration names provide details on the parameters such as fill factor (FILL), maximum number of additional non zeros allowed per row (Nz) and the level of fill (LF). Table 5 provides details on the parameter acronyms.



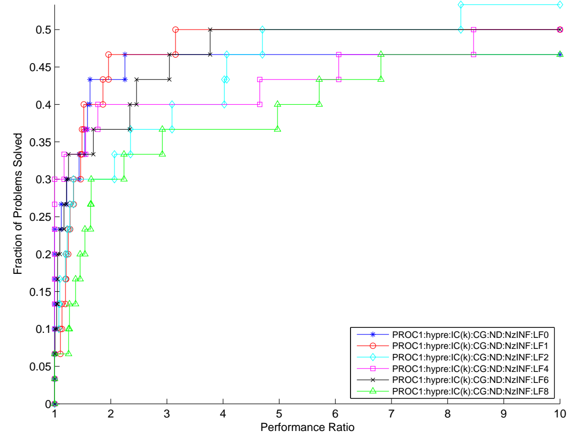
(a) Memory performance profile corresponding to infinite fill factor and RCM ordering



(b) Memory performance profile corresponding to infinite fill factor for ND ordering



(c) Time performance profile corresponding to infinite fill factor for RCM ordering



(d) Time performance profile corresponding to infinite fill factor for ND ordering

Figure 2: Memory and time performance profile curves for Hypre-IC(k) solver configurations for fill factor of ∞ in the serial case. The legends provide details on the solver (CG), ordering (RCM, ND) and parameters such as the maximum number of additional non zeros allowed per row (NzINF) and the level of fill (LF). Table 5 provides more details.

of fill solves more problems for both ordering schemes, but it ends up consuming more resources for problems which could have been solved using lower levels. The increase in memory usage is clearly seen in Figures 2(a) and 2(b), wherein level 0 (LF0) setting solves around 45% of the problems using the least memory. However, in the case of time profiles in 2(c) and 2(d), the variation across the levels is slightly lower.

Figure 3 shows the memory and time profiles for the best Hypre-IC(k) configurations corresponding to each fill factor and ordering combination. Most of the curves merge for the same ordering and fill factor, suggesting that the level of fill has a higher influence on the performance with respect to both memory and time.

PETSc: IC(k)

Figure 4 shows the memory and time profiles for all the PETSc-IC(k) configurations in Table 2 for RCM and ND ordering. The best configuration with respect to time, memory, and robustness for both orderings turned out to be the one with level of fill set to 0 (LF0). Using higher levels of fill, in fact, resulted in a reduction in the number of problems that were solved which is in stark contrast to that observed with the Hypre-IC(k) preconditioner.

PETSc: BlockSolve Figure 5 shows the memory and time profiles for the PETSc-BlockSolve configurations for RCM and ND ordering in the serial and 16 processor case. The memory and time profiles indicate that ordering did not make any significant impact for the serial case whereas in the 16 processor case RCM ordering could not solve as many problems as ND ordering.

Trilinos: IC(k)

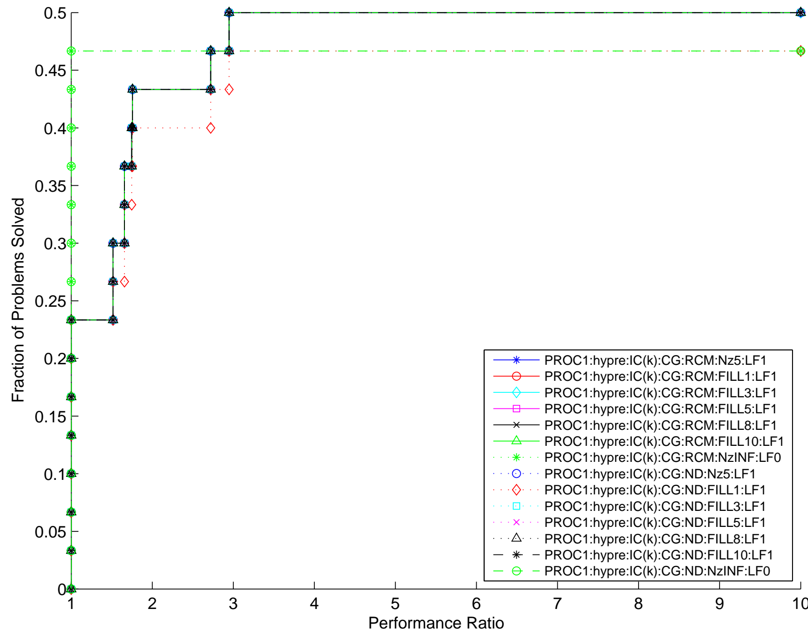
Figure 6 shows the memory and time profiles of all the RCM and ND ordered configurations for Trilinos-IC(k). Using higher levels of fill did not affect the number of problems solved for either of the orderings which is different from what was observed for both PETSc and Hypre IC(k) preconditioner. The time profile curves overlapped for the various levels of fill and in the case of memory, there is only a slight difference for both RCM and ND ordering.

4.1.2 Threshold-based Incomplete Factorization

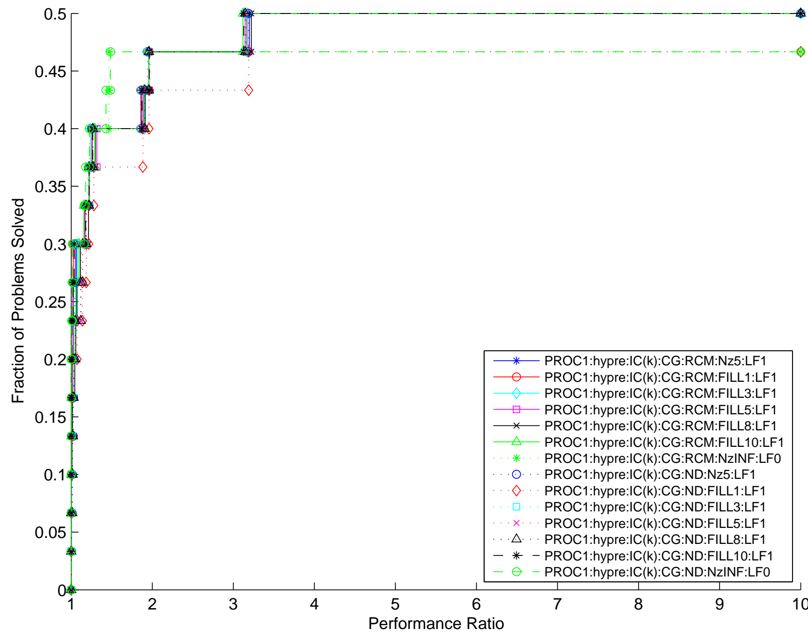
Typically, for SPD matrices, ILUT is not the best choice of preconditioner since the preconditioners tend to be unsymmetric and non SPD. Among the packages in this study only WSMP, Ilupack, and Trilinos provided an implementation of the threshold based incomplete Cholesky preconditioner (ICT). While experimenting with the ICT and ILUT preconditioners in Trilinos, we observed that for most SPD problems in our test suite, the ILUT preconditioner used in combination with the GMRES solver performed better with respect to time and robustness than ICT with CG as the solver. Therefore, we also experimented with ILUT preconditioners in PETSc and Trilinos. Only PETSc had an implementation in which the user could specify a pivoting threshold. Hypre-ILUT results have been omitted since the user manual strongly recommends against using it until all the functionality has been fully implemented and tested. Similar to IC(k) preconditioners, we experimented with multiple values of 3, 5, 8 and 10 for the fill factor. Since the use of drop tolerance could lead to unsymmetric preconditioners, we used GMRES with restarts of 30, 65 and 100 for solving the resulting preconditioned system except for ICT in Trilinos where CG was used. In addition, we also investigated the effect of drop tolerance by experimenting with values of 3e-2, 1e-2, 1e-3 and 5e-4 for each value of fill factor. The configurations that were the memory and time winners for each set of package-solver-ordering combination are shown in Table 7 and the key observations in each package are listed below.

PETSc: ILUTP

Figure 7 shows the performance profile curves for the memory and time profiles for configurations with restart 100 and fill factor 5. For both orderings, higher values of drop tolerance (1e-2 and

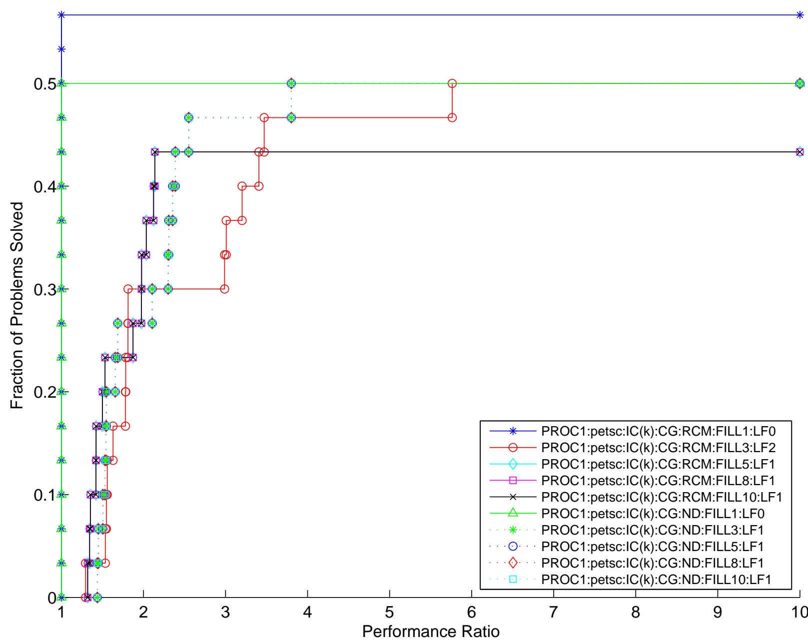


(a) Memory performance profile corresponding to the best level of fill parameters

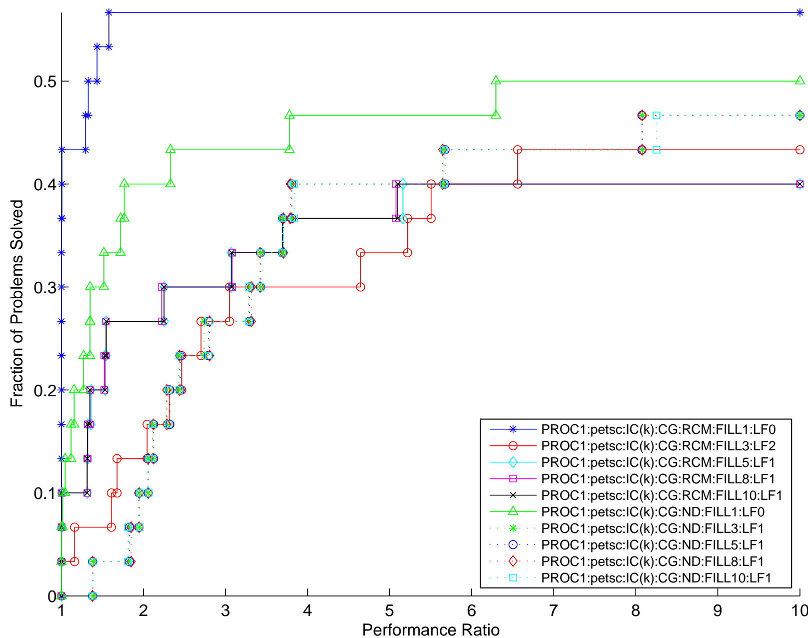


(b) Time performance profile corresponding to the best level of fill parameters

Figure 3: Memory and time performance profile curves for the best MTP Hypre-IC(k) solver configurations corresponding to each fill factor and ordering combination in the serial case. The legends provide details on the solver (CG) and parameters such as the level of fill (FILL) or maximum number of additional non zeros allowed per row (NzINF) and the level of fill (LF). Table 5 provides more details.

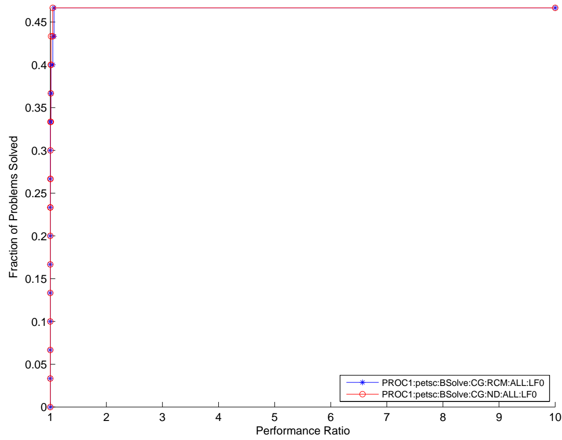


(a) Memory performance profile

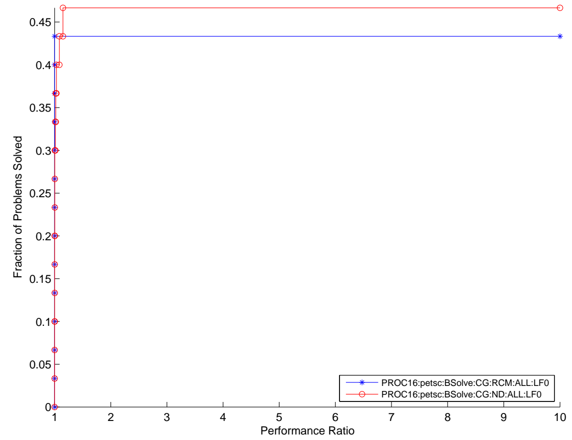


(b) Time performance profile

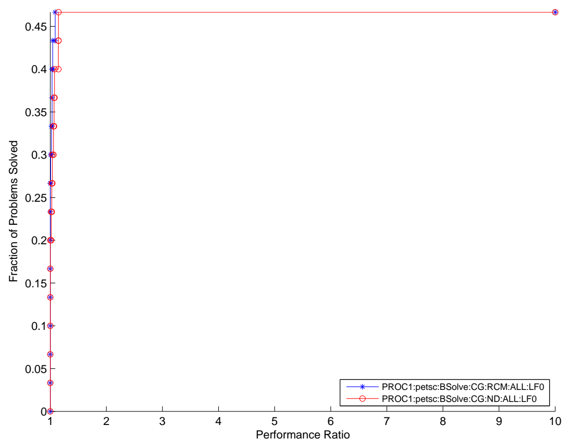
Figure 4: Memory and time performance profile curves for all PETSc-IC(k) solver configurations. The legends provide details on the solver (CG), ordering (RCM, ND) and parameters such as fill factor (FILL) and level of fill (LF). Table 5 provides more details.



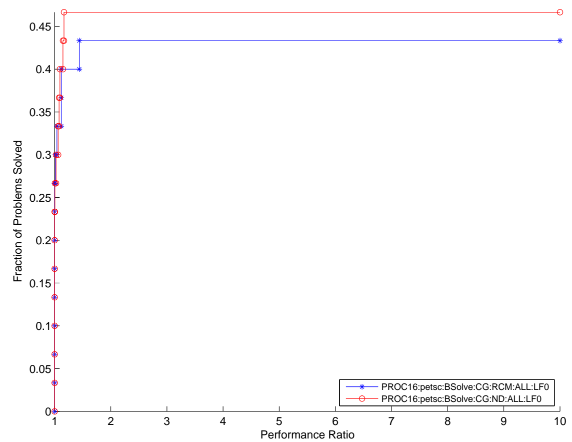
(a) Memory performance profile (single processor)



(b) Memory performance profile (16 processor)

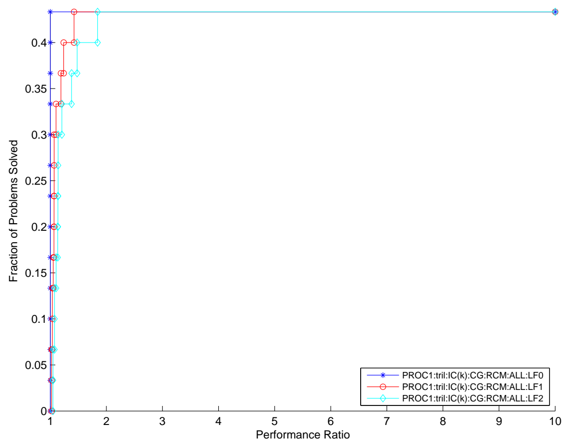


(c) Time performance profile (single processor)

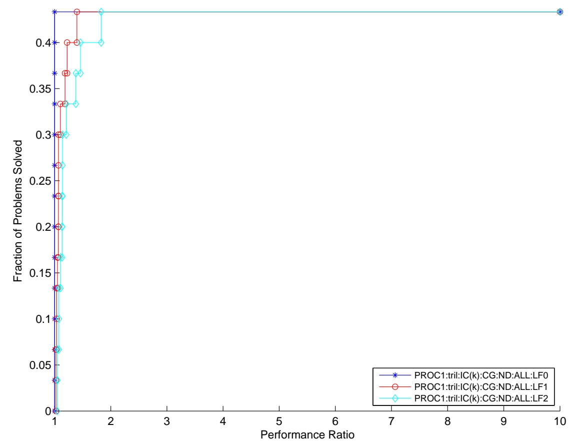


(d) Time performance profile (16 processor)

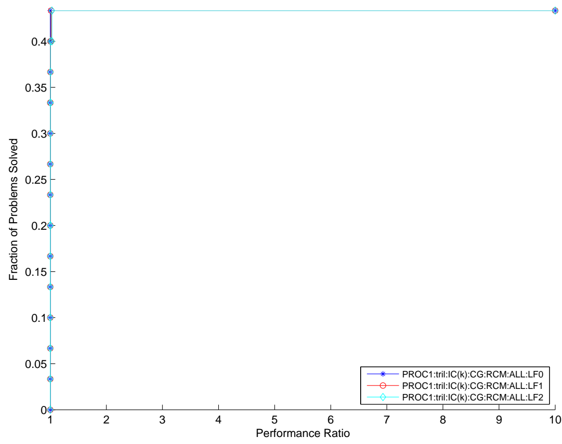
Figure 5: Memory and time performance profile curves for PETSc-BlockSolve configurations in the serial and 16 processor case. The legends provide details on the solver (CG) and ordering (RCM, ND). Table 5 provides more details.



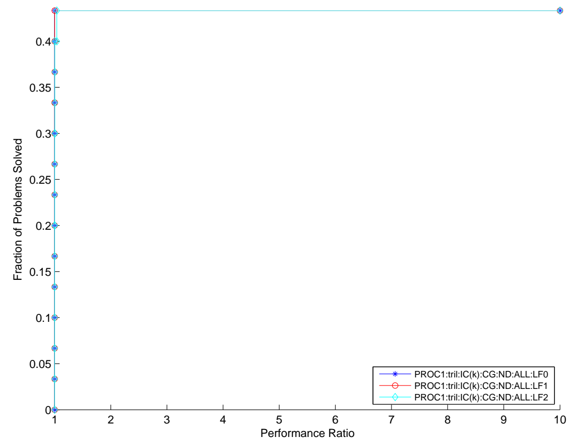
(a) Memory performance profile for RCM ordering



(b) Memory performance profile for ND ordering



(c) Time performance profile for RCM ordering



(d) Time performance profile for ND ordering

Figure 6: Memory and time performance profile curves for Trilinos-IC(k) solver configurations. The legends provide details on the solver (CG), ordering (RCM, ND) and parameters such as level of fill (LF). Table 5 provides more details.

Preconditioner	Memory Winner	Time Winner	MTP Winner
PETSc-ILUT	GMRES65,RCM FILL5,DT1.0e-03,P0.01 (1)	GMRES100,RCM FILL5,DT1.0e-03,P0 (1)	GMRES65,RCM FILL5,DT1.0e-03,P0 (1)
Trilinos-ILUT	GMRES30,RCM FILL3,DT5.0e-04 (2) GMRES30,RCM FILL3,DT1.0e-03 (1) GMRES30,RCM FILL8,DT5.0e-04 (16) GMRES30,RCM FILL10,DT5.0e-04 (8) GMRES30,ND FILL3,DT5.0e-04 (4) GMRES30,ND FILL8,DT5.0e-04 (32 64)	GMRES30,RCM FILL10,DT5.0e-04 (64) GMRES30,ND FILL8,DT5.0e-04 (32) GMRES65,RCM FILL3,DT5.0e-04 (1) GMRES65,RCM FILL3,DT1.0e-03 (4) GMRES65,RCM FILL8,DT5.0e-04 (16) GMRES100,RCM FILL3,DT5.0e-04 (2) GMRES100,RCM FILL10,DT5.0e-04 (8)	CG,RCM FILL3,DT3.0e-02 (64) GMRES30,RCM FILL3,DT5.0e-04 (2) GMRES30,RCM FILL3,DT1.0e-03 (1 4) GMRES30,RCM FILL10,DT5.0e-04 (8) GMRES30,RCM FILL10,DT1.0e-03 (16) GMRES30,ND FILL8,DT5.0e-04 (32)

Table 7: Iterative solver configurations that resulted in the best performance with respect to time profile area, memory profile area, robustness (maximum number of solved problems) and MTP for ILUT preconditioners in PETSc, Hypre and Trilinos. The numbers enclosed by () denotes the processor number corresponding to the solver configurations. The configuration names provide details on the parameters such as fill factor (FILL), drop tolerance (DT), pivot threshold (P), and the level of fill (LF). Table 5 provides details on the parameter acronyms.

3e-2) were poor choices in terms of the number of problems solved whereas lower values(1e-3 and 5e-4) were significantly better. Figure 8 shows the memory and time profiles for the best solver configurations corresponding to each solver and ordering combination. Increasing the restart value results in solving more number of problems for both the orderings. As the fill factor is increased from 3 to 5, there is an increase in the number of problems solved for low drop tolerance values, however, for higher values of fill factor (8, 10) there is a drop in the profile areas since the prescribed memory limit was exceeded for some of the problems.

Trilinos: ICT/ILUT

Figure 9 shows the memory and time profiles for the best configurations with respect to MTP profile area for each solver-ordering combination. For Trilinos-ICT (solver choice is CG), using higher fill factor values or lower values of drop tolerance resulted in preconditioners that were more expensive to construct whereas there was no improvement in the number of problems solved. On the other hand, for Trilinos-ILUT used along with GMRES, a high value of fill factor helped in solving more problems for low values of drop tolerance. The most robust configuration for the ILUT preconditioner in Trilinos, in combination with GMRES, could solve more number of problems in comparison to using CG and Trilinos-ICT.

Ilupack: MLICT Experiments were performed for five different built-in reordering schemes (RCM, AMF, INDSET, PQ and METISN) in Ilupack, four different values for drop tolerance (1e-2, 3e-2, 1e-3,5e-4), and three different values of the norm of inverse estimate (10, 25, 100). Table 8 shows the drop tolerance and inverse norm estimate combinations that performed the best for each ordering.

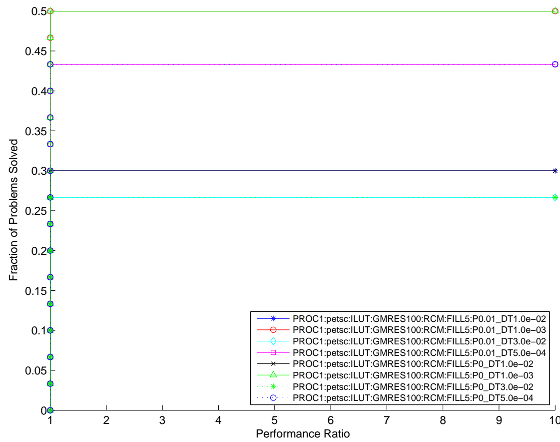
Preconditioner	Memory Winner	Time Winner	MTP Winner
Ilupack-MLICT	CG,RCM,IE10 DT1.0e-03,IE10 (1)	CG,AMF DT1.0e-03,IE10 (1)	CG,RCM DT3.0e-02,IE100 (1)

Table 8: Iterative solver configurations that resulted in the best performance with respect to time profile area, memory profile area robustness (maximum number of solved problems) and MTP for the MLICT preconditioner in Ilupack. The numbers enclosed by () denotes the processor number corresponding to the solver configurations.The parameter names provide information on the drop tolerance values (DT) and norm of inverse estimate values (IE). Table 5 provides details on the parameter acronyms.

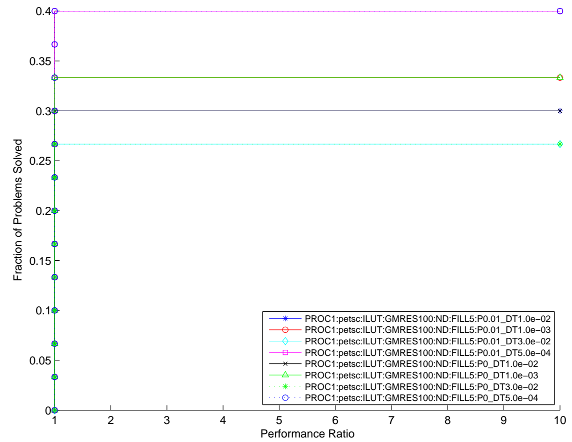
Figure 10 shows the memory and time profiles corresponding to all the different configurations for RCM ordering, which is the best ordering with respect to MTP. Figure 11 shows the memory and time profiles for the best MTP winning configuration for each ordering. Visual inspection of the time and memory profile curves in Figure 10 reveals that the configurations form two groups based on their drop tolerance values. Specifically, the configurations corresponding to lower drop tolerance values (1e-3, 5e-4) tend to require more computational effort, while the ones with higher drop tolerance values (1e-2, 3e-2) require relatively less time and memory. Figure 11suggests that a combination of a higher drop tolerance with a higher value of inverse norm estimate can provide significant benefit in terms of robustness and performance rather than just decreasing the drop tolerance.

4.1.3 Multigrid Methods

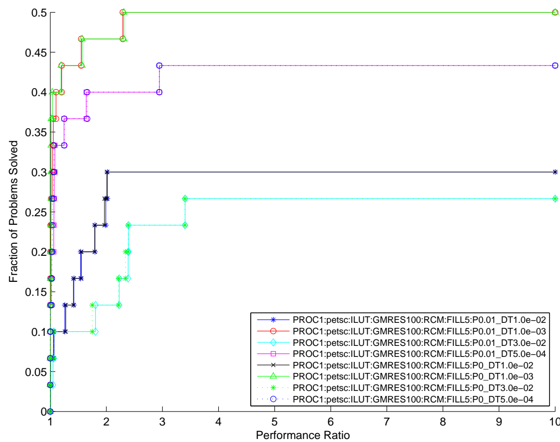
Multigrid preconditioners typically have a large number of parameters that need to be fine tuned. For the BoomerAMG preconditioner, we used the default values suggested in the user manual for



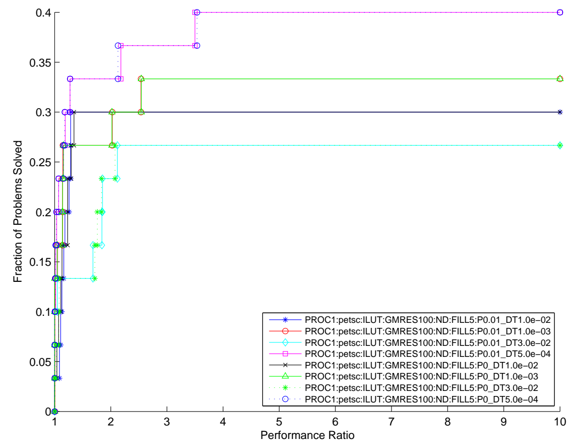
(a) Memory performance profile for RCM ordering



(b) Memory performance profile for ND ordering

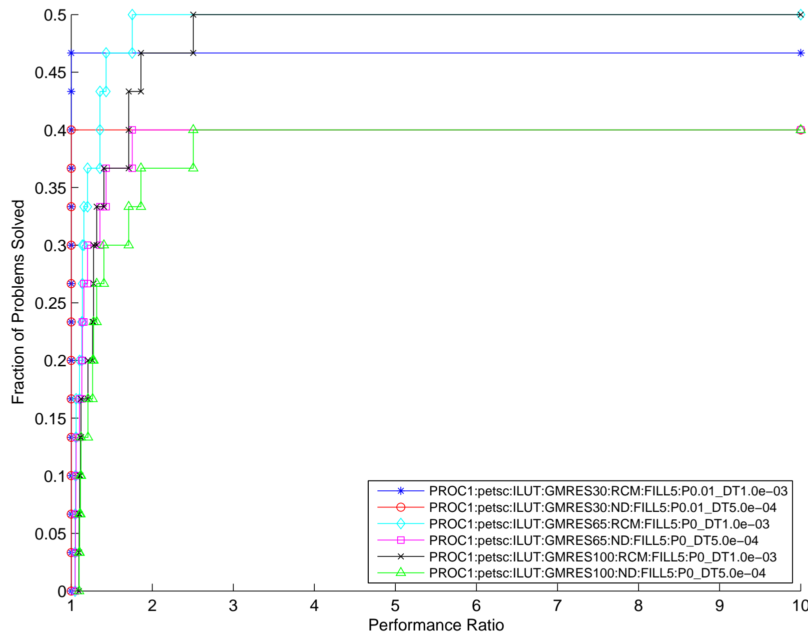


(c) Time performance profile for RCM ordering

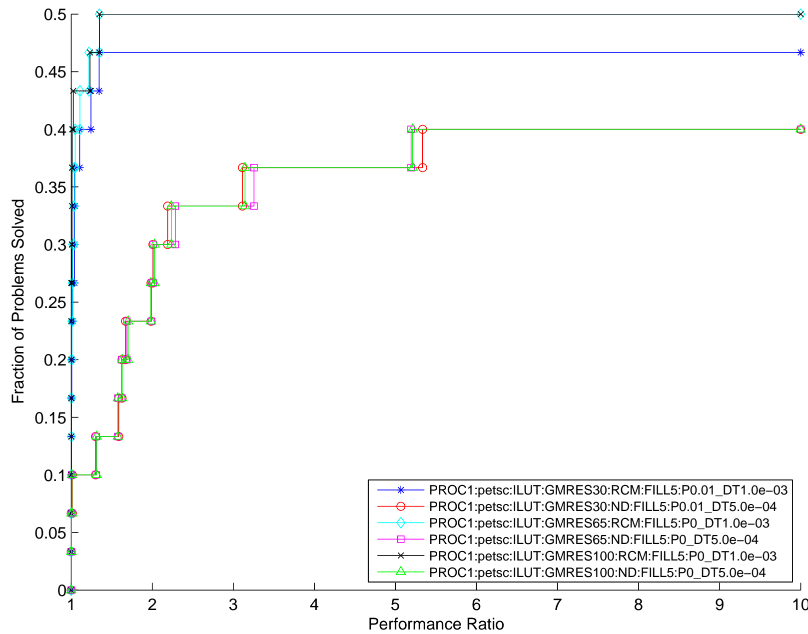


(d) Time performance profile for ND ordering

Figure 7: Memory and time performance profile curves for petsc-ILUTP solver configurations using GMRES with restart 100 (GMRES100) and fill factor of 5 (FILL5). The legends provide details on the ordering (RCM, ND), pivoting threshold (P) and drop tolerance(DT). Table 5 provides more details.

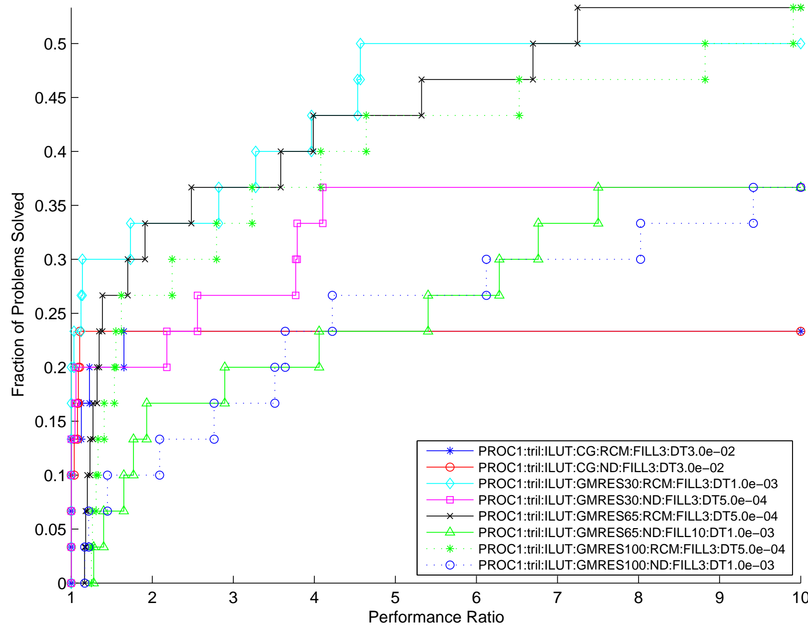


(a) Memory performance profile

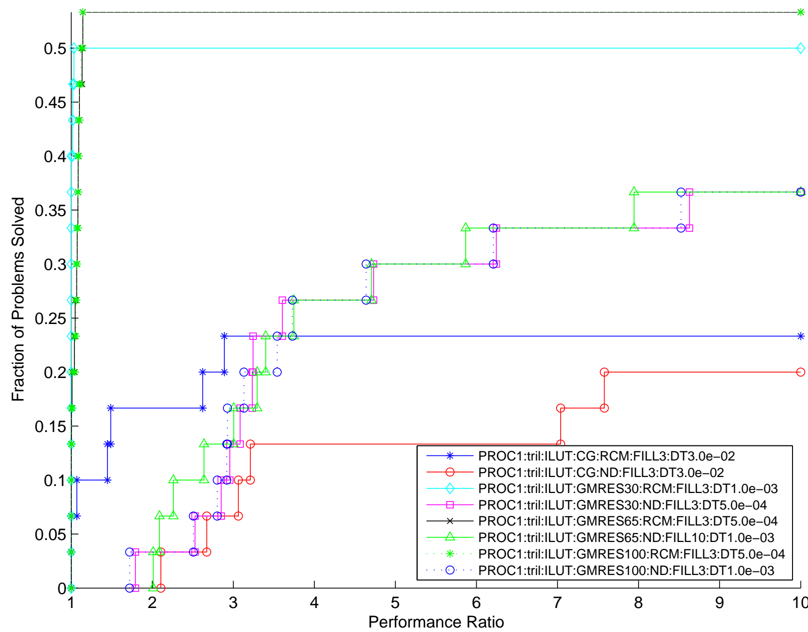


(b) Time performance profile

Figure 8: Memory and time performance profile curves for PETSc-ILUTP solver configurations that resulted in the best MTP profile area with respect to solver and ordering in the serial case. The legends provide details on the solver (GMRES(30, 65, 100)), ordering (RCM, ND), fill factor (FILL), pivoting threshold (P) and drop tolerance(DT). Table 5 provides more details.

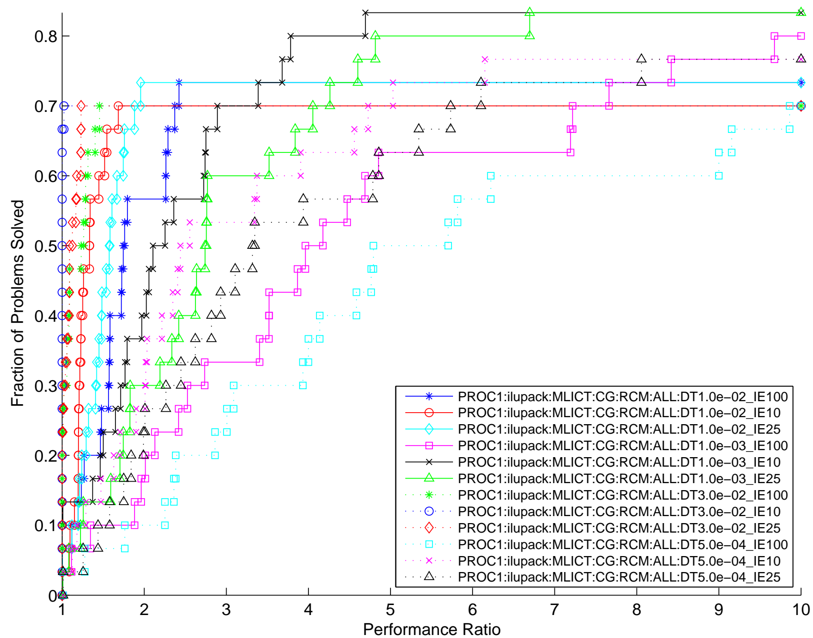


(a) Memory performance profile

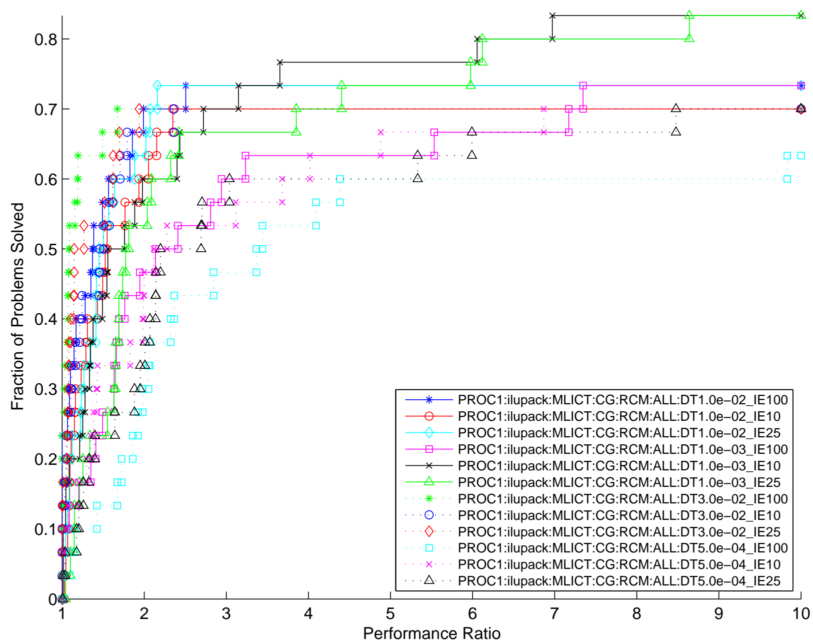


(b) Time performance profile

Figure 9: Memory and time performace profile curves for Trilinos-ILUT solver configurations that resulted in the best MTP profile area with respect to solver and ordering. The legends provide details on the solver (CG,GMRES(30, 65, 100)), ordering (RCM, ND), fill factor (FILL) and drop tolerance values (DT). Table 5 provides more details.

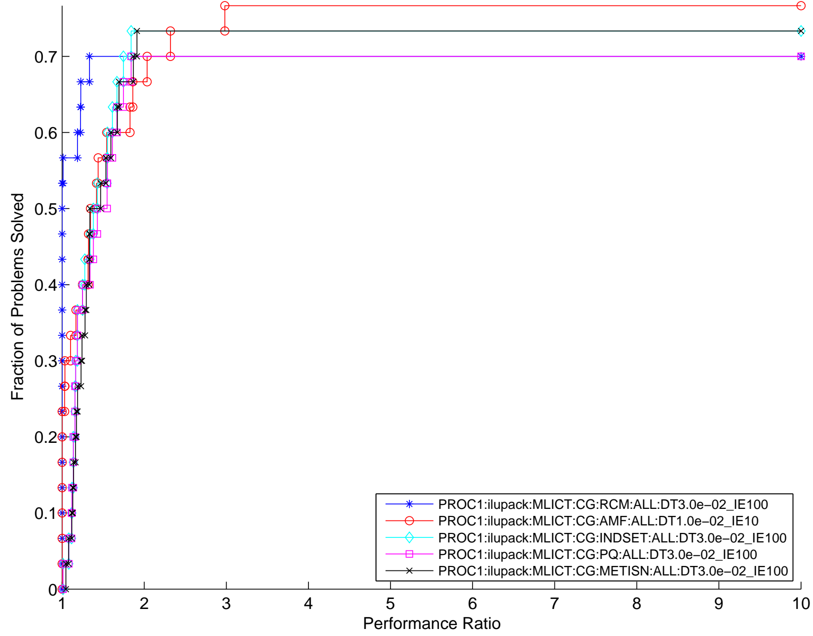


(a) Memory performance profile corresponding to the various parameters for RCM ordering

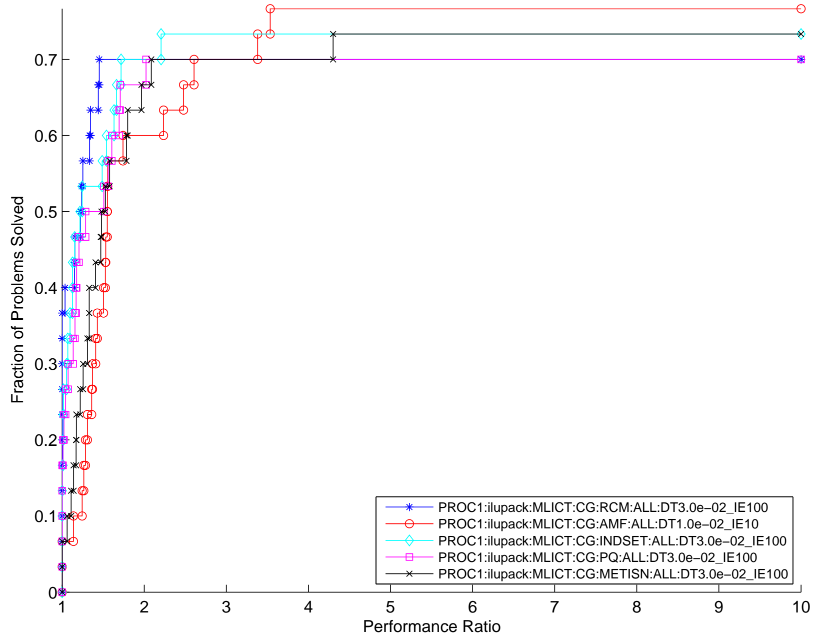


(b) Time performance profile corresponding to the various parameters for RCM ordering

Figure 10: Memory and time performance profile curves for Ilupack-MLICT solver configurations of drop tolerance and norm of inverse estimates. The legends provide details on the solver (CG), ordering (RCM), drop tolerance values (3e-2,1e-2,1e-3,5e-4) and norm of inverse estimates (10,25,100).



(a) Memory performance profile



(b) Time performance profile

Figure 11: Memory and time performance profile curves for Ilupack-MLICT solver configurations that resulted in the best MTP profile area with respect to ordering. The legends provide details on the solver (CG), ordering (RCM, ND, AMF, IND, ddPQ), drop tolerance values (DT) and norm of inverse estimates (IE). Table 5 provides details on the parameter acronyms.

most of the parameters in the driver program. However, we did experiment with a few important parameters such as the coarsening schemes, maximum number of levels for aggressive coarsening and *strong threshold*. Other parameters that were considered include the use of aggressive coarsening for 10 levels and four values for the strong threshold. For the ML preconditioner in Trilinos, we experimented with multiple values for a few parameters in the default set for smoothed aggregation (SA), two level domain decomposition (DD) and three level aggressive coarsening (DD-ML) as described in [13]. Table 9 shows the best configurations for this class of preconditioners.

Hypre:BoomerAMG

Figures 12 and 13 show the effect of aggressive coarsening and strong threshold values for Falgout and PMIS coarsening schemes for the best ordering scheme. The performance profile curves for the HMIS scheme is similar to that for the PMIS scheme and is therefore not shown. A comparison of the memory and time profiles for the best configurations with respect to MTP for the various coarsening schemes and ordering is shown in Figure 14.

- **Strong Threshold.** In Figure 12, irrespective of the ordering, the value of 0.25 for strong threshold resulted in heavy memory usage for most problems when used without aggressive coarsening. The use of higher values of strong threshold helped in solving a majority of the problems using relatively lesser resources.
- **Aggressive Coarsening.** Another important trend that can be observed in Figures 12 and 13 is that the use of aggressive coarsening can provide substantial reduction in the computational requirements with respect to both time and memory.
- **Coarsening Scheme.** With regards to robustness, all the three coarsening schemes turned out to be equally good with at least one parameter combination that solved the maximum fraction of problems (80%). However, PMIS was the better coarsening scheme with respect to memory and time.

Trilinos: ML

Although we experimented with multiple values for the various parameters, there was hardly any change in the performance metrics observed for Trilinos-ML. The memory and time profiles for the best MTP parameters for the various solver and ordering combinations for the default SA, DD and ML-DD parameter groups are shown in Figure 15. Only the SA parameters were used with CG and this combination could solve only about 40% of the problems (See Figure 15).

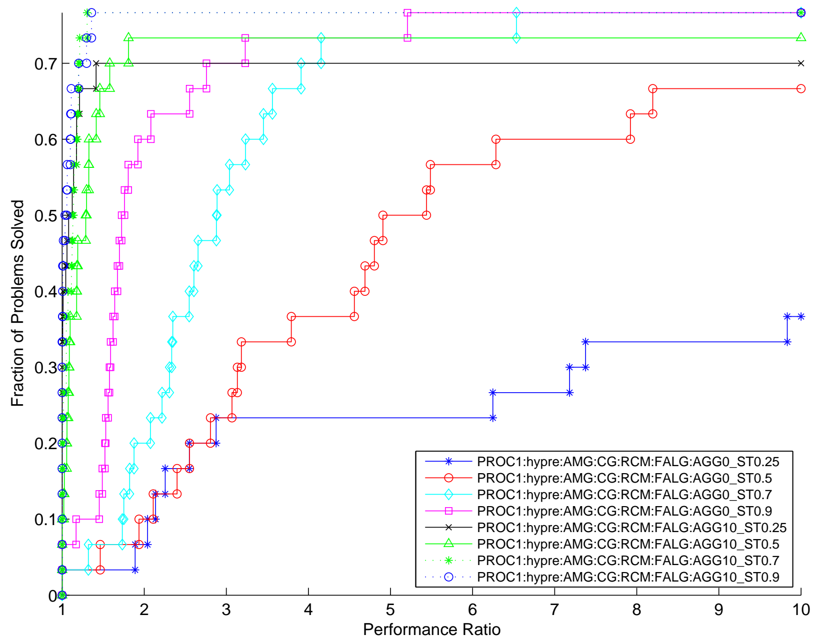
4.1.4 Sparse Approximate Inverse

For the ParaSails preconditioner in Hypre, we experimented with multiple threshold values (0, 0.01, 0.1, -0.75,-0.9) and filter values (0, 0.001, 0.05, -0.9) for three different levels as suggested by the user manual. Figures 16, 17 and 18 show the time and memory profiles for the different parameter configurations corresponding to each of the levels (0, 1, 2). Table 10 summarizes the configurations that resulted in the best performance profiles in our experiments. The memory and time profiles corresponding to best MTP parameters are shown in Figure 19.

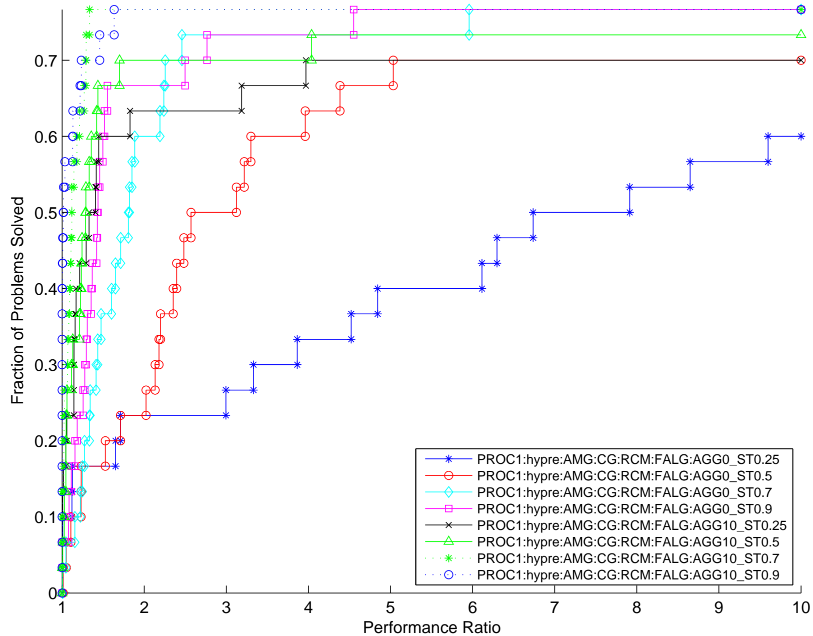
- **Threshold Values.** A visual inspection of the performance profiles in Figures 16, 17 and 18 shows the evolution of a partitioning of the 11 parameter configurations into two broad groups. When the number of levels is 0, all the memory and time profiles (shown in Figures 16) are clustered for performance ratios in the range $[1, 2]$ whereas for the case with number of levels set to 2, there is a clear clustering of the time as well as memory profile curves into

Preconditioner	**Memory Winner**	**Time Winner**	**MTP Winner**
Hypre-BoomerAMG	CG,RCM,HMIS AGG0,ST0.9 (32) CG,RCM,FALG AGG10,ST0.7 (4 8) CG,RCM,FALG AGG10,ST0.9 (2) CG,ND,FALG AGG10,ST0.9 (1 16) CG,NONE,FALG AGG0,ST0.9 (64)	CG,RCM,PMIS AGG0,ST0.9 (32) CG,RCM,FALG AGG10,ST0.7 (4 8) CG,RCM,FALG AGG10,ST0.9 (1 2) CG,ND,HMIS AGG10,ST0.9 (16) CG,NONE,PMIS AGG0,ST0.9 (64)	CG,RCM,FALG AGG10,ST0.7 (4 8) CG,RCM,FALG AGG10,ST0.9 (1 2 32) CG,ND,FALG AGG10,ST0.9 (64) CG,NONE,FALG AGG10,ST0.9 (16)
Trilinos-ML	CG,RCM,ML-SA MCS16,SS1,Lev10 (2) CG,ND,ML-SA MCS16,SS1,Lev6 (8) CG,NONE,ML-SA MCS16,SS1,Lev6 (64) CG,NONE,ML-SA MCS32,SS1,Lev6 (16 32) GMRES30,RCM,ML-DD MCS128,SS1,Lev2 (1) GMRES30,ND,ML-DD MCS128,SS1,Lev2 (4)	CG,NONE,ML-SA MCS32,SS2,Lev10 (32) GMRES30,RCM,ML-SA MCS16,SS2,Lev6 (1) GMRES30,NONE,ML-SA MCS16,SS1,Lev6 (16) GMRES65,RCM,ML-SA MCS16,SS2,Lev10 (2) GMRES65,ND,ML-DD MCS64,SS1,Lev2 (4) GMRES65,NONE,ML-DD MCS128,SS1,Lev2 (64) GMRES100,ND,ML-SA MCS16,SS2,Lev6 (8)	CG,RCM,ML-SA MCS16,SS1,Lev10 (2) CG,ND,ML-SA MCS16,SS1,Lev6 (8) CG,NONE,ML-SA MCS16,SS2,Lev6 (16 64) CG,NONE,ML-SA MCS32,SS2,Lev10 (32) GMRES30,RCM,ML-DD MCS64,SS2,Lev4 (1) GMRES30,ND,ML-DD MCS128,SS1,Lev2 (4)

Table 9: Iterative solver configurations that resulted in the best performance with respect to time profile area, memory profile area, robustness (maximum number of solved problems) and MTP for AMG preconditioners in Hypre and Trilinos. The numbers enclosed by () denotes the processor number corresponding to the solver configurations. For Hypre, the configuration names provide details on the parameters such as coarsening schemes (FALG,PMIS,HMIS), number of levels of aggressive coarsening (AGG) and strong threshold values (ST). For Trilinos, the best among the three default parameters (SA, DD, DD-ML) along with the number of lvels (Lev), number of smoother sweeps (SS) and maximum coarse grid size (MCS) is reported. Table 5 provides details on the parameter acronyms.

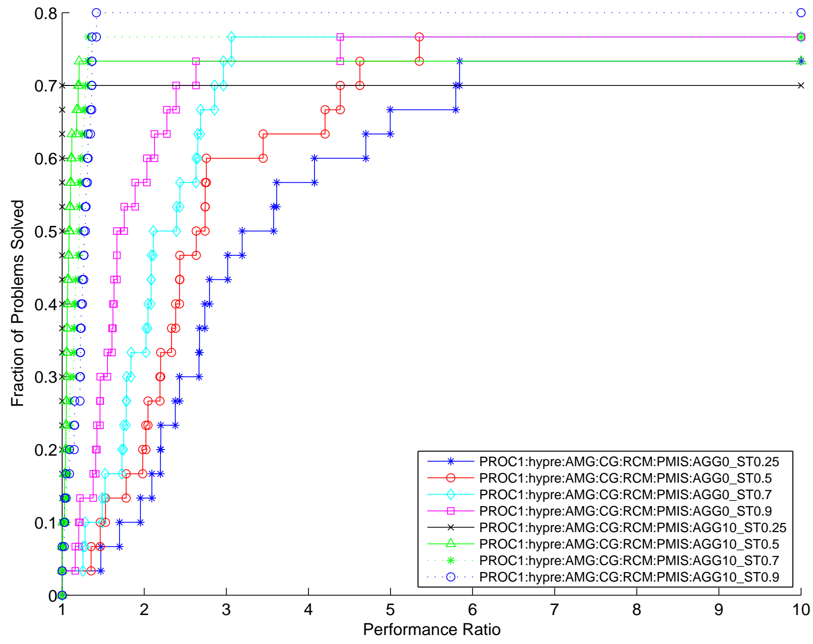


(a) Memory performance profile corresponding to the various parameters for RCM ordering

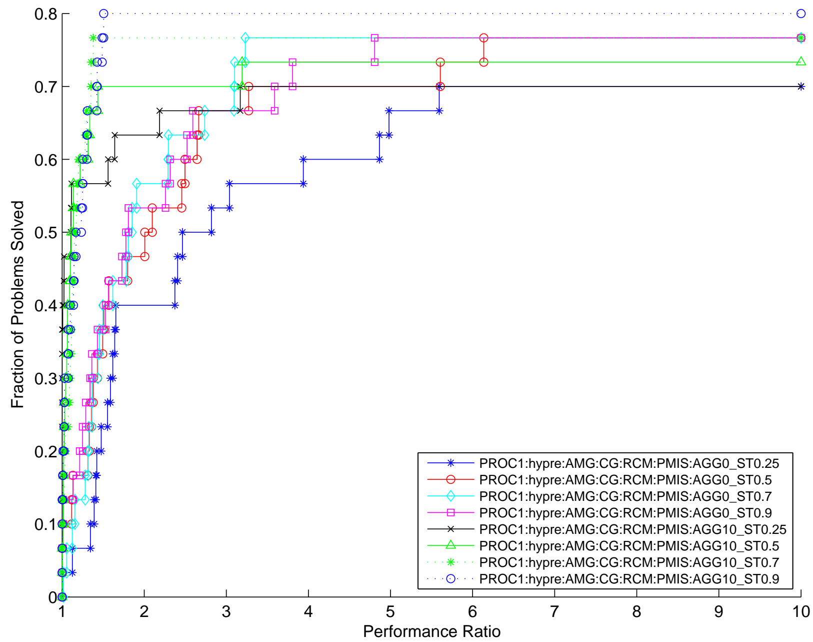


(b) Time performance profile corresponding to the various parameters for RCM ordering

Figure 12: Memory and time performance profile curves for Hypre-BoomerAMG solver configurations for the Falgout (FALG) coarsening scheme. The legends provide details on the solver (CG), ordering (RCM), number of levels of aggressive coarsening (AGG) and strong threshold values (ST). Table 5 provides more details.

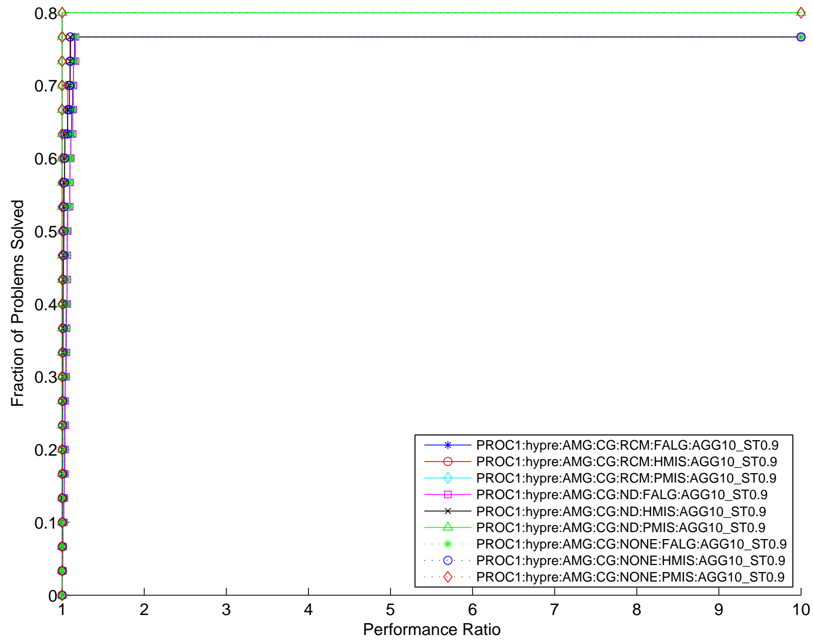


(a) Memory performance profile corresponding to the various parameters

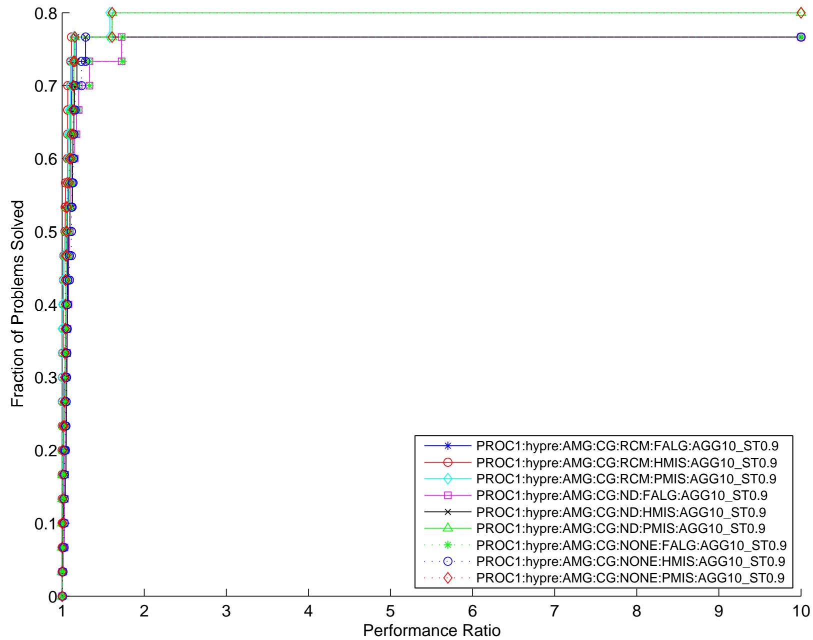


(b) Time performance profile corresponding to the various parameters

Figure 13: Memory and time performance profile curves for Hypre-BoomerAMG solver configurations for the PMIS coarsening scheme. The legends provide details on the solver (CG), ordering (RCM), number of levels of aggressive coarsening (AGG) and strong threshold values (ST). Table 5 provides more details.

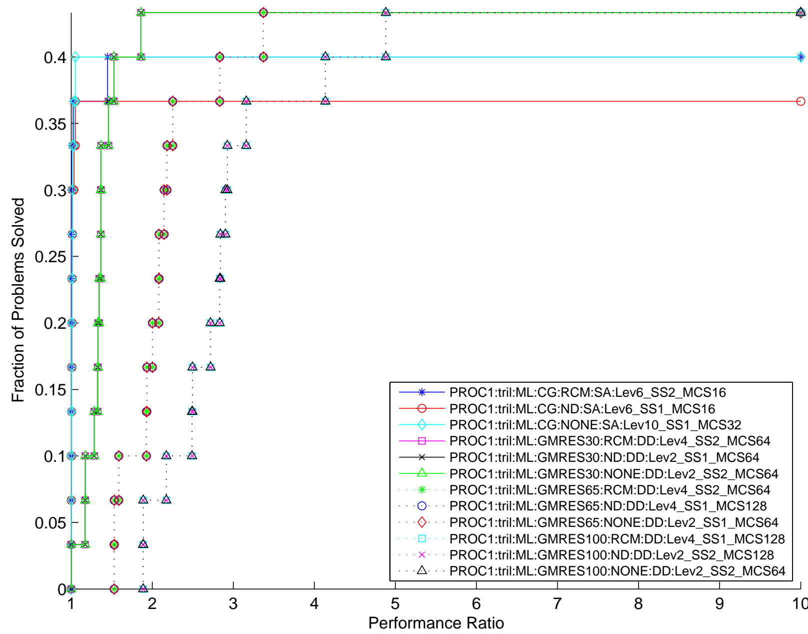


(a) Memory performance profile

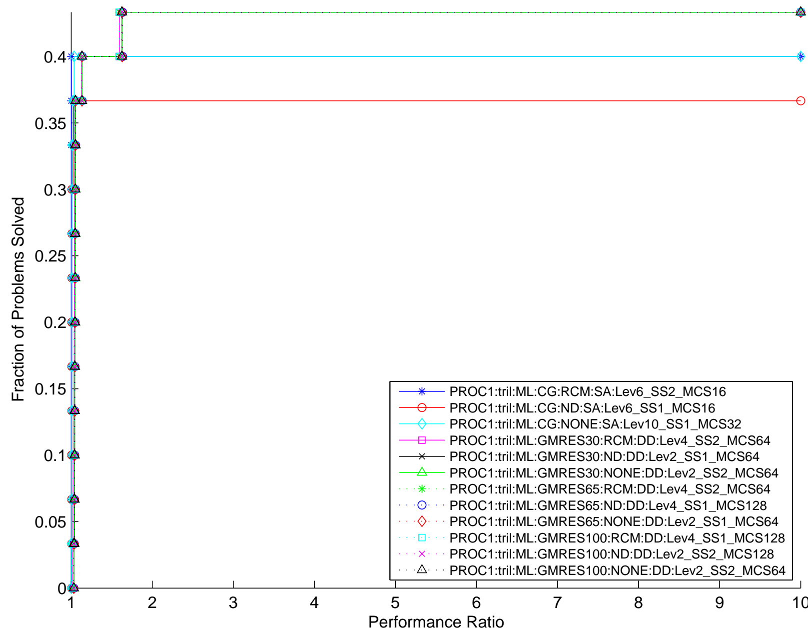


(b) Time performance profile

Figure 14: Memory and time performance profile curves for Hypre-BoomerAMG solver configurations that resulted in the best MTP profile area with respect to the coarsening scheme and ordering. The legends provide details on the solver (CG), ordering (RCM, ND, NONE), coarsening scheme (PMIS, HMIS,FALG), number of levels of aggressive coarsening (AGG) and strong threshold values (ST). Table 5 provides more details.



(a) Memory performance profile



(b) Time performance profile

Figure 15: Memory and time performance profile curves for Trilinos-ML solver configurations that resulted in the best MTP profile area with respect to solver and ordering. The legends provide details on the solver (CG, GMRES(30, 65, 100)), ordering (RCM, ND, NONE), default set of ML parameters (SA, DD,DD-ML), number of levels (Lev), number of smoother sweeps (SS) and the maximum size of the coarse grid (MCS). Table 5 provides more details.

Preconditioner	Memory Winner	Time Winner	MTP Winner
Hypre-ParaSails	CG,ND,PLev1 Th0.1,Flt0.05 (1 2 4)	CG,ND,PLev1 Th0.1,Flt0 (32)	CG,ND,PLev1 Th0.1,Flt0.001 (32)
	CG,ND,PLev2 Th0.1,Flt0.001 (8 16 32 64)	CG,ND,PLev1 Th0.1,Flt0.05 (2 4 8 16 64)	CG,ND,PLev1 Th0.1,Flt0.05 (2 4 8 16 64)
		CG,NONE,PLev1 Th0.1,Flt0.05 (1)	CG,NONE,PLev1 Th0.1,Flt0.05 (1)

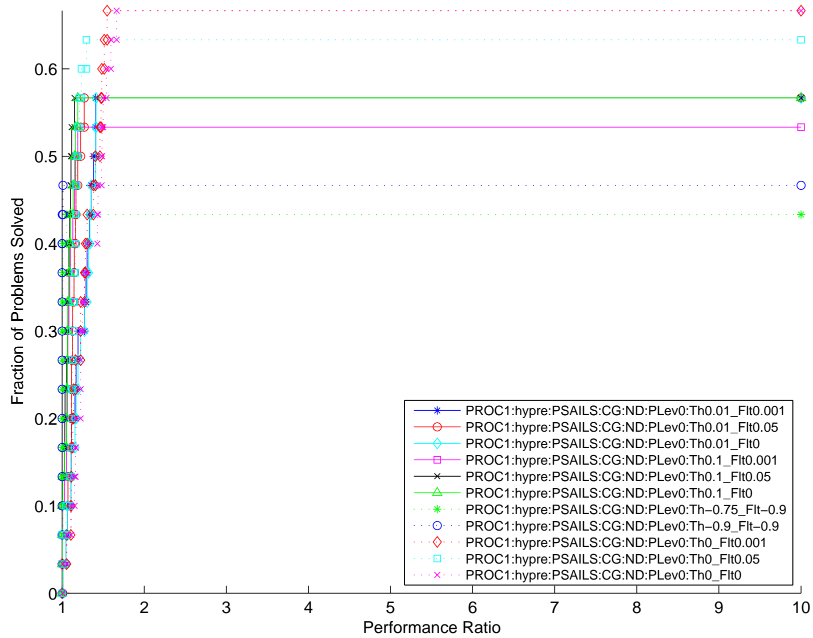
Table 10: Iterative solver configurations that resulted in the best performance with respect to time profile area, memory profile area, robustness (maximum number of solved problems) and MTP for the ParaSails preconditioner in Hypre. The numbers enclosed by () denotes the processor number corresponding to the solver configurations. The configuration names provide details on parameters such as number of levels (Lev), threshold (Th) and filter (Flt). Table 5 provides details on the parameter acronyms.

two broad groups (shown in Figure 18). On analyzing the configurations of the profiles in the two clusters, we observe that the group that required more computational effort (both time and memory) corresponds to lower threshold values of 0.0 and 0.01 whereas the group with better performance corresponds to a higher threshold value of 0.1. The negative threshold values suggested by the authors in the user manual, which have a different interpretation from the non negative values, solved the fewest number of problems in comparison to others.

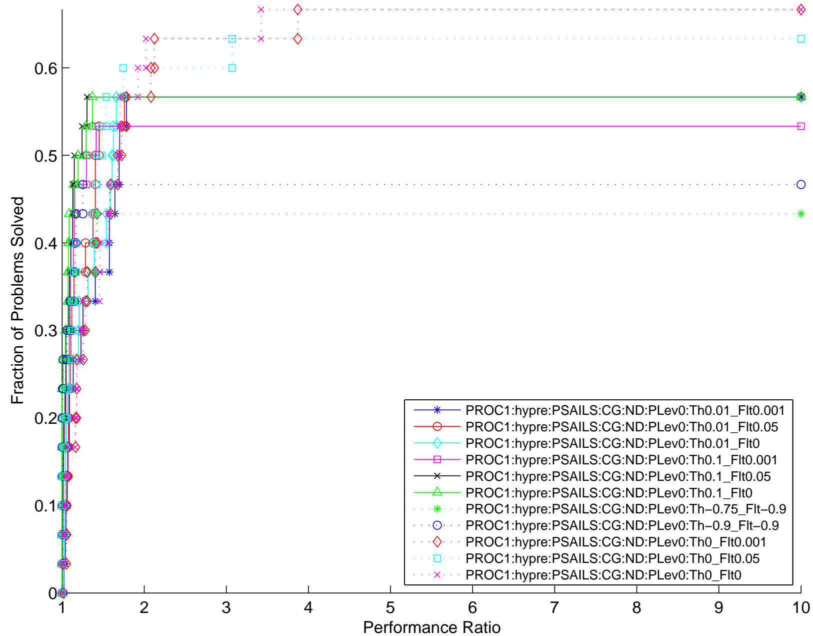
- **Number of Levels.** From Figure 19(a), we observe that the best configuration with number of levels set to 0 solves the maximum number of problems using the minimum memory i.e., the maximum value on the y axis corresponding to the performance ratio of 1. However, this comes at a cost in terms of the robustness when compared with the case where the number of levels is 2. A combination of higher values of threshold and filter with higher number of levels can solve more number of problems at the expense of using slightly more memory. A similar trend is also seen with respect to time in Figure 19(b).

4.2 Variation of Processor Specific Default Configurations

Some of the overall best configurations reported in Tables 6-10 indicate that the solver configurations are different for different processor settings. In certain cases, there is quite a bit of variation among the recommended configurations and finding a general trend among the various parameter choices becomes harder. To gauge the effect of processor specific default configuration, we computed the MTP performance profiles for each of the distinct default configurations corresponding to the configuration groups. Figures 20 and 25 show MTP performance profile plots for the distinct default configurations in the 16 processor case. The time profile curves for Trilinos IC(k) and ML in Figures 21 and 23 show that the processor specific default configurations do not vary much with respect to MTP. The MTP profile curves corresponding to default configurations in the case of 32 and 64 processors for Trilinos-ILUT as seen in Figure 22 shows siginificant variation in performance. There is also a slight difference in the MTP curves for BlockSolve, BoomerAMG and IC(k) because some of the processor specific default configurations did not solve as many problems as the others.

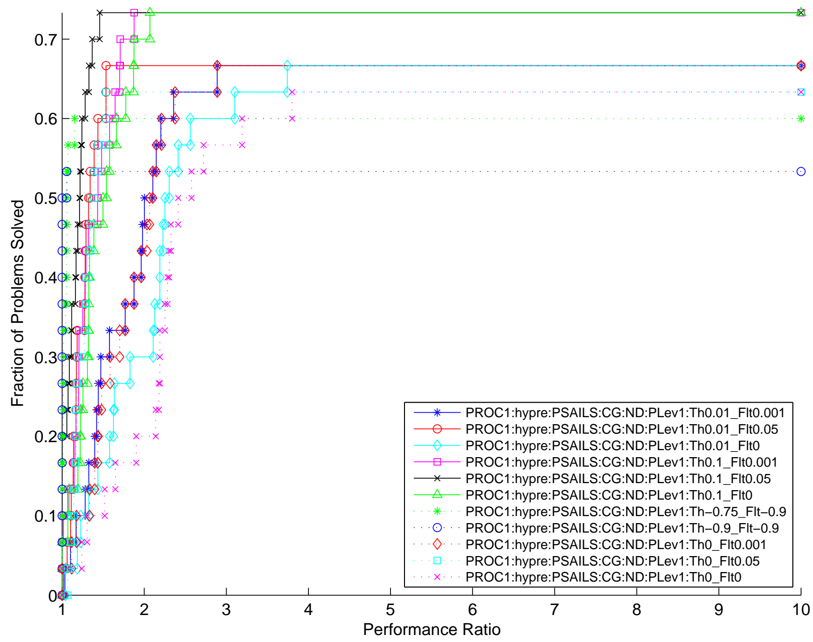


(a) Memory performance profile corresponding to the various parameters

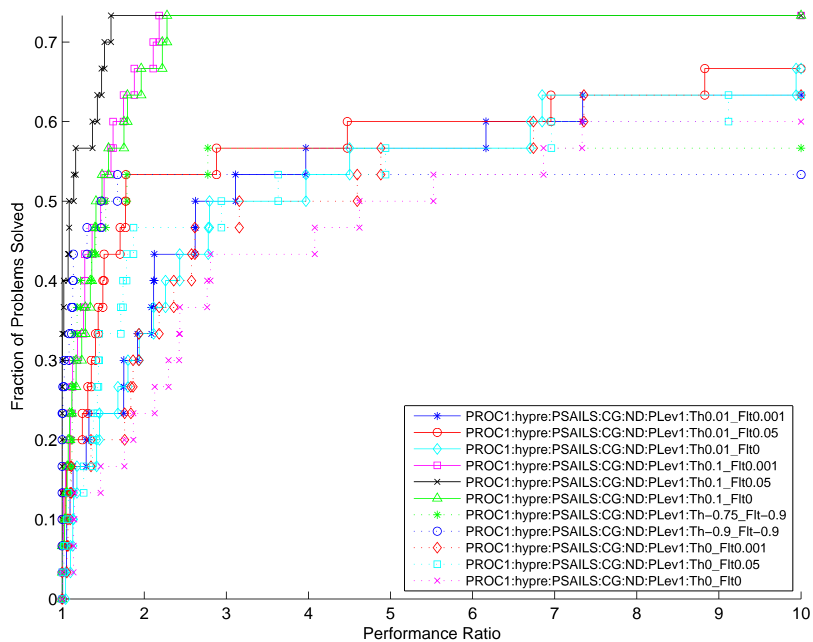


(b) Time performance profile corresponding to the various parameters

Figure 16: Memory and time performance profile curves for Hypre-ParaSails solver configurations of threshold and filter for number of levels 0. The legends provide details on the solver (CG), ordering (ND), number of levels (Lev), threshold (Th) and filter (Flt). Table 5 provides more details.

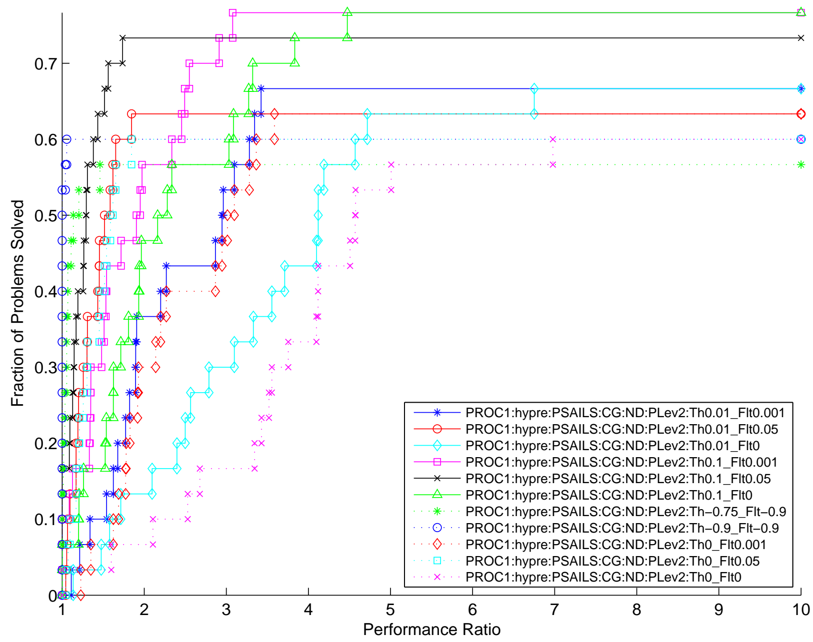


(a) Memory performance profile corresponding to the various parameters

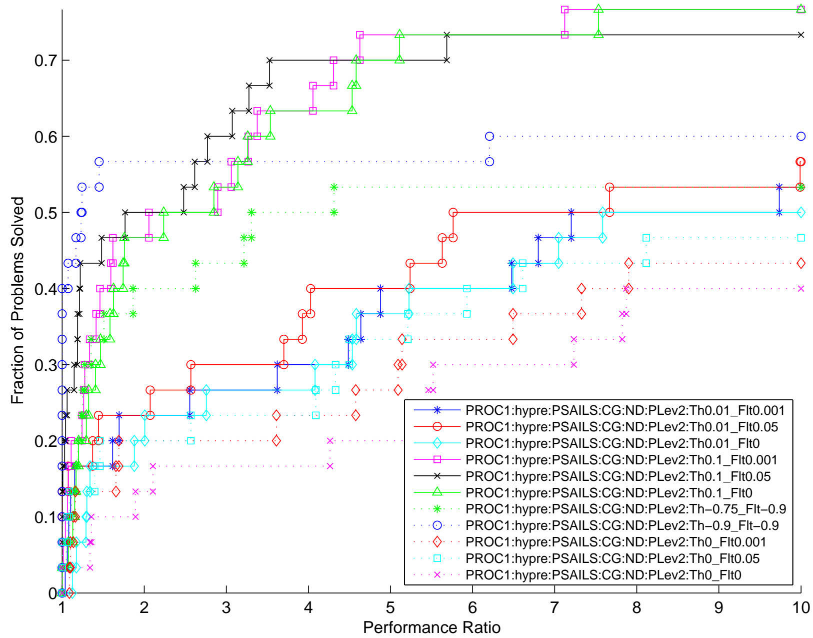


(b) Time performance profile corresponding to the various parameters

Figure 17: Memory and time performance profile curves for Hypre-ParaSails solver configurations of threshold and filter for number of levels 1. The legends provide details on the solver (CG), ordering (ND), number of levels (Lev), threshold (Th) and filter (Flt). Table 5 provides more details.

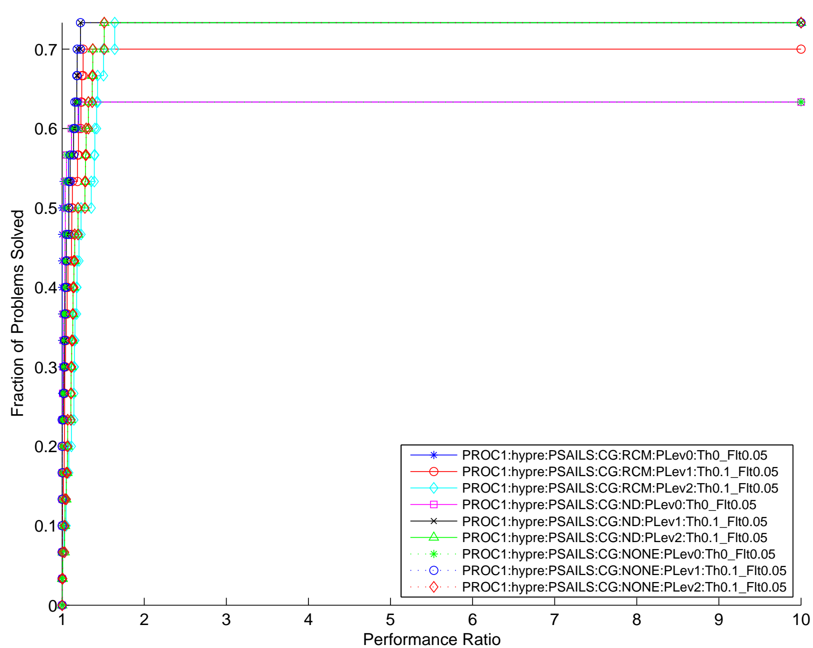


(a) Memory performance profile corresponding to the various parameters

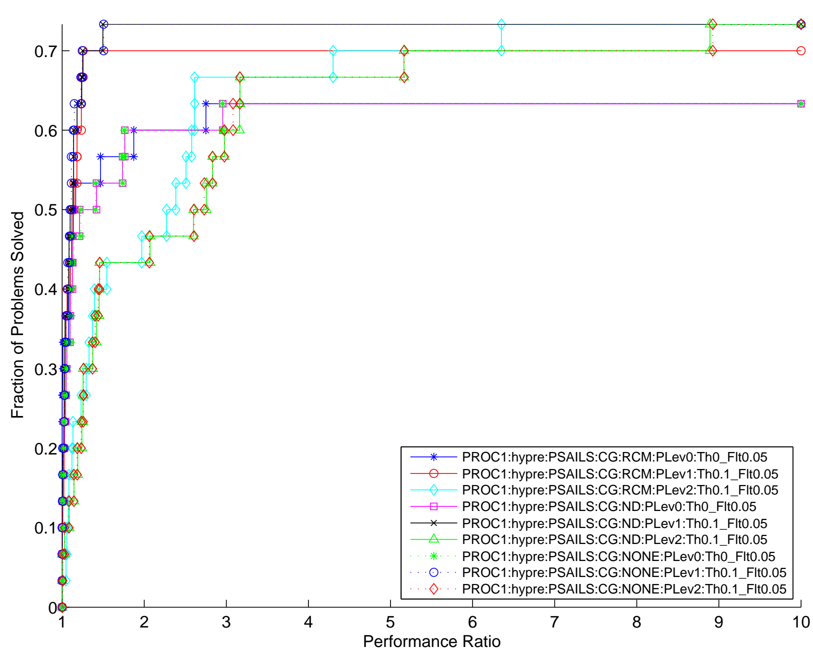


(b) Time performance profile corresponding to the various parameters

Figure 18: Memory and time performance profile curves for Hypre-ParaSails solver configurations of threshold and filter for number of levels 2. The legends provide details on the solver (CG), ordering (ND), number of levels (Lev), threshold (Th) and filter (Flt). Table 5 provides more details.



(a) Memory performance profile



(b) Time performance profile

Figure 19: Memory and time performance profile curves for Hypre-ParaSails solver configurations that resulted in the best MTP profile area with respect to solver and ordering. The legends provide details on the solver (CG), ordering (RCM, ND, NONE), number of levels (Lev), threshold (Th) and filter (Flt). Table 5 provides more details.

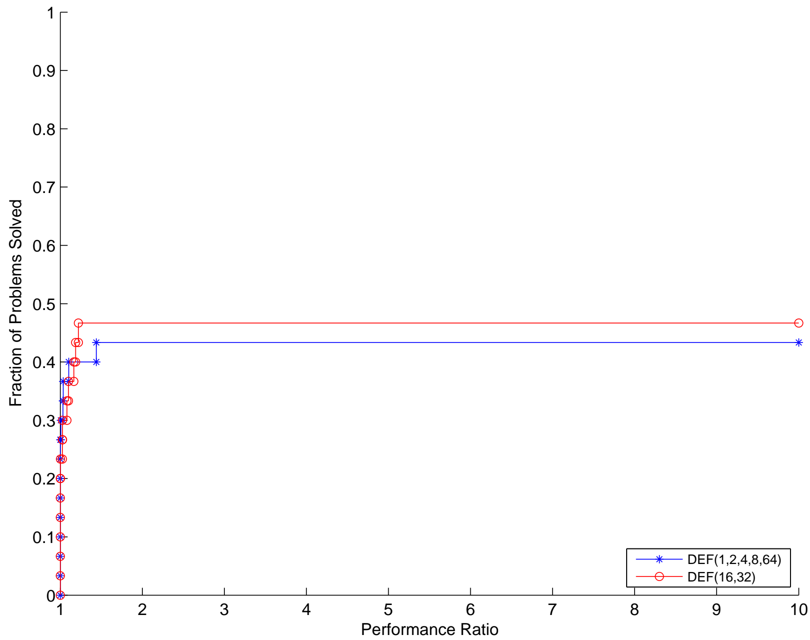


Figure 20: Memory time product profiles for the processor specific default configurations of PETSc-BlockSolve in the 16 procesor case.

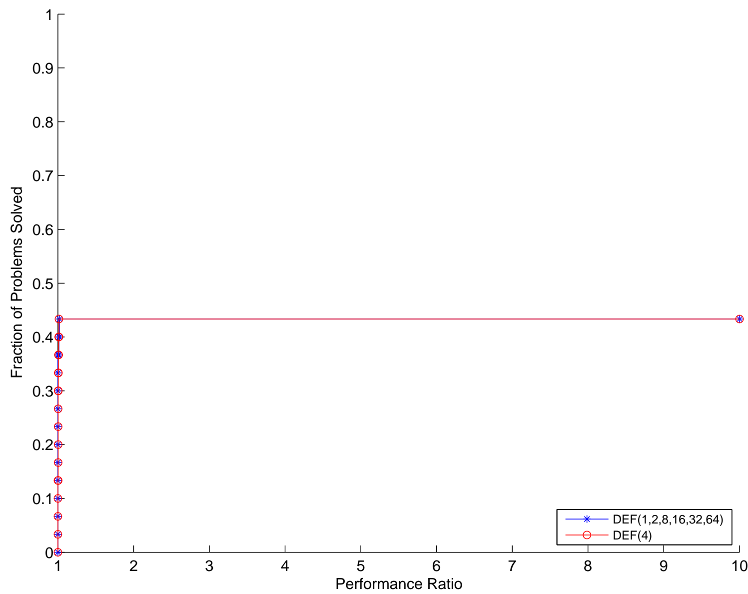


Figure 21: Memory time product profiles for the processor specific default configurations of Trilinos-IC(k) in the 16 procesor case.

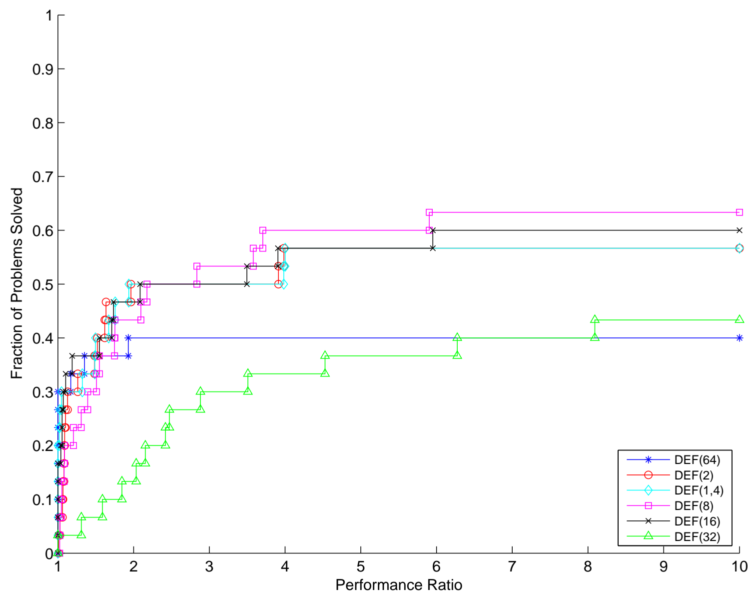


Figure 22: Memory time product profiles for the processor specific default configurations of Trilinos-ILUT in the 16 procesor case.

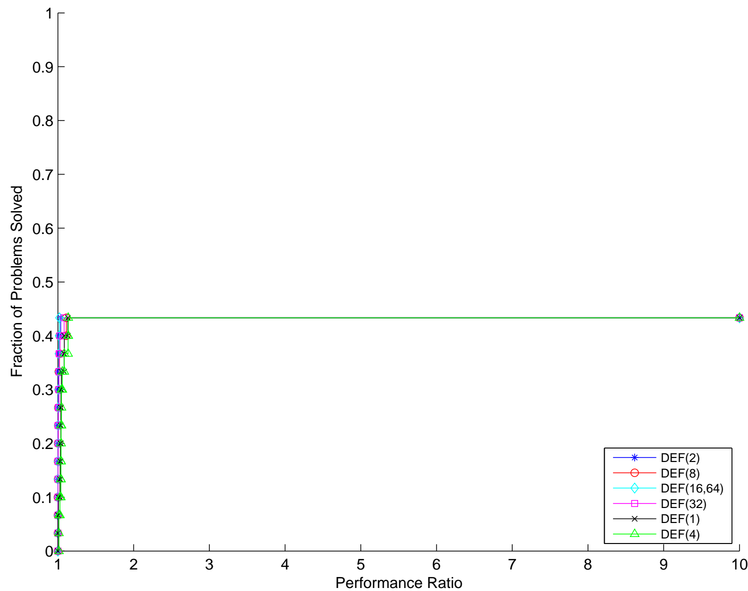


Figure 23: Memory time product profiles for the processor specific default configurations of Trilinos-ML in the 16 procesor case.

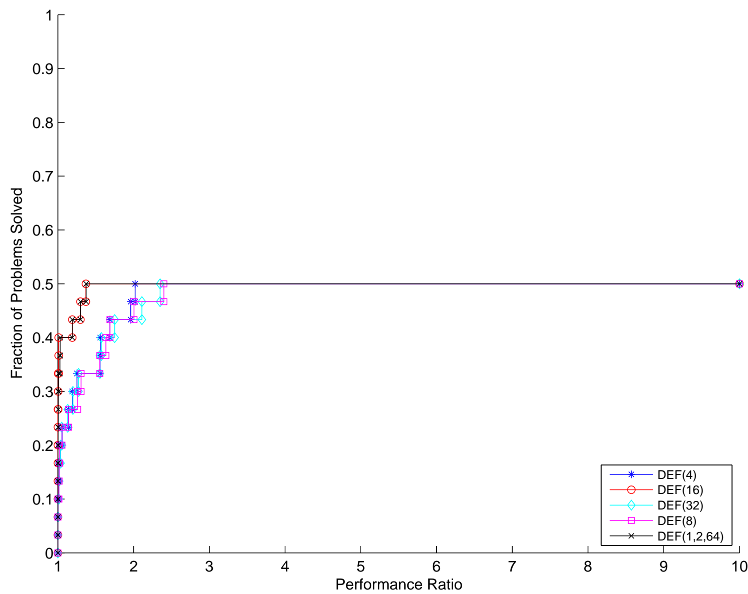


Figure 24: Memory time product profiles for the processor specific default configurations of Hypre-IC(k) in the 16 procesor case.

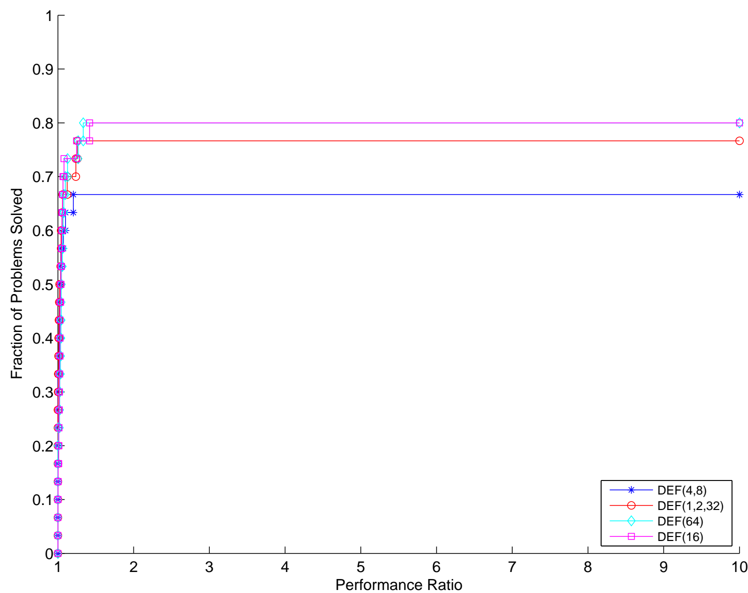


Figure 25: Memory time product profiles for the processor specific default configurations of Hypre-BoomerAMG in the 16 procesor case.

4.3 Performance Benefits of Fine Tuning

Most often, 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 26 shows the memory and time performance profile curves for the default and problem specific best (PSB) choices for the preconditioners that only had the serial implementation. The performance improvement is highest for Ilupack-MLICT and the effect is more pronounced in the case of time than memory. PETSc-IC(k) benefits the least with respect to both time and memory whereas fine-tuning does help ILUTP in solving an extra problem. Figures 27 and 28 show the performance variation between the PSB and default for PETSc-BlockSolve and WSMP-ICT respectively. For PETSc-BlockSolve, the PSB and default curves are almost identical since the only parameter involved in this case is the ordering scheme.

The memory and time variation of default and PSB configurations for Hypre-IC(k) is shown in Figure 29. We can see that there is considerable performance benefits due to fine-tuning for both memory and time. Although there is a slight drop in the number of problems solved for certain number of processors, in general the gap between the PSB and the default parameter curves reduces as we increase the number of processors. The corresponding curves for Trilinos-IC(k) in Figure 30 indicate that there was no significant improvement with respect to time with fine-tuning. The memory and time profiles in Figure 31 indicate that Trilinos-ILUT can benefit significantly with fine tuning in terms of both memory and time in comparison with the default configuration. For multiple processors, the main difference is in the number of problems solved.

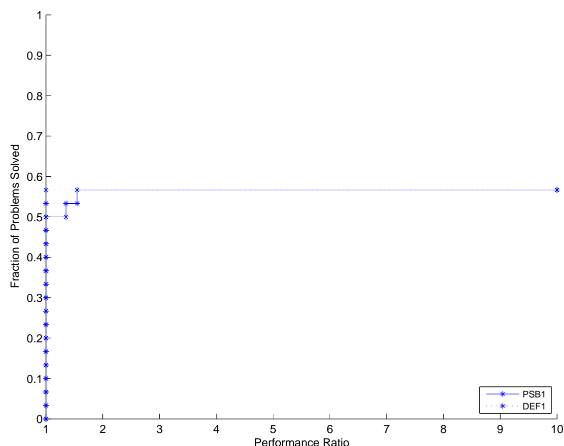
For Hypre-ParaSails, the memory and time profiles for the default and PSB configurations for multiple processors are shown in Figure 32. A comparison of the memory and time profiles shown in Figures 32(a) and 32(b) indicate that the benefits of parameter fine tuning is more pronounced in the case of time than for memory as seen by the separation between the default and PSB curves.

In the case of Hypre-BoomerAMG in Figure 33, the gap between the default and PSB profiles is minor in the case of both memory and time. For Trilinos-ML curves in Figure 34, memory benefits more with fine tuning in comparison to time.

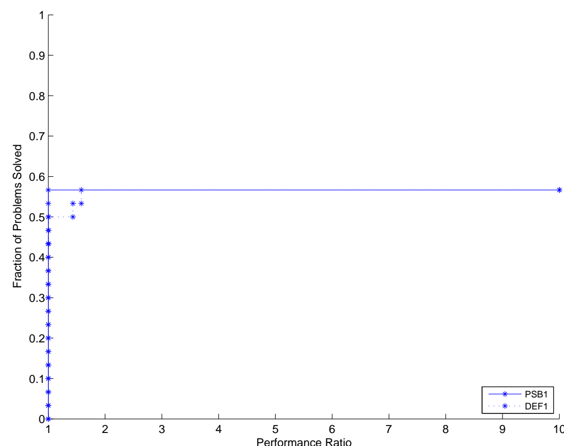
4.3.1 Relative Importance of Parameters on Performance

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 configurations similar to the ones reported in Tables 6 to 10. This is done for each solved problem, and then averaged over the entire problem set. Both time and memory performance are considered individually.

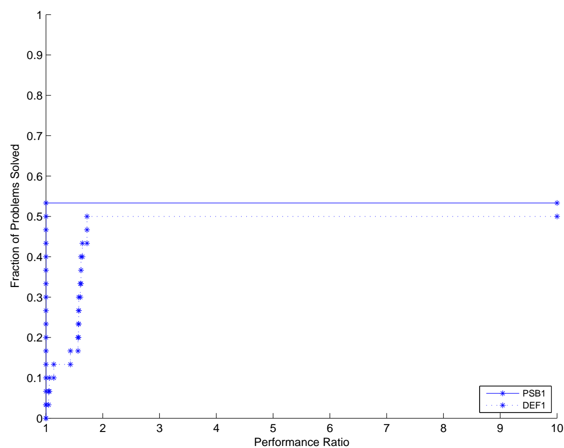
Figure 35 to shows the fine-tuning effects of the various factors on the time and memory used for different preconditioners for the single processor runs. The height of the bars indicate the variation in time and memory use in response to changing the corresponding parameter. Since the actual performance values for each problem are normalized by the default configuration values,



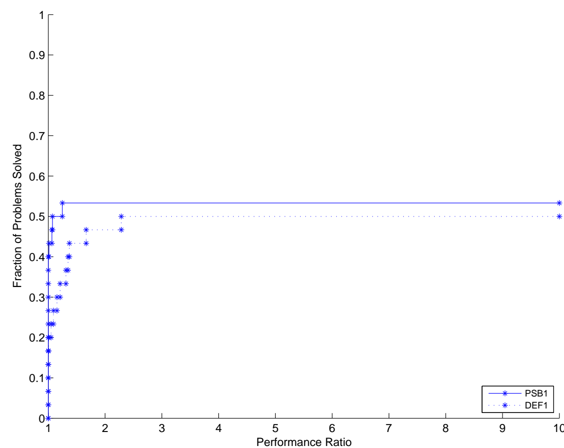
(a) Petsc IC(k) Memory Profile



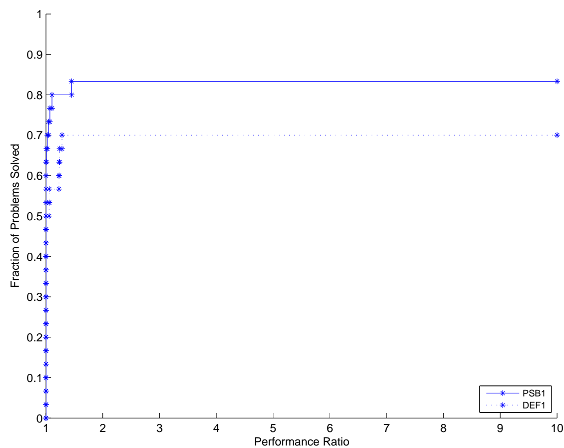
(b) Petsc IC(k) Time Profile



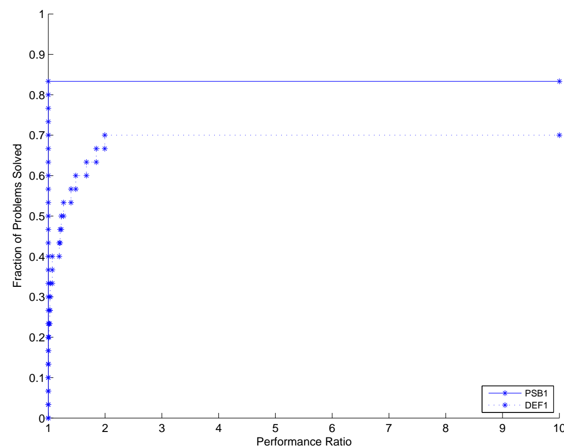
(c) Petsc ILUTP Memory Profile



(d) petsc ILUTP Time Profile

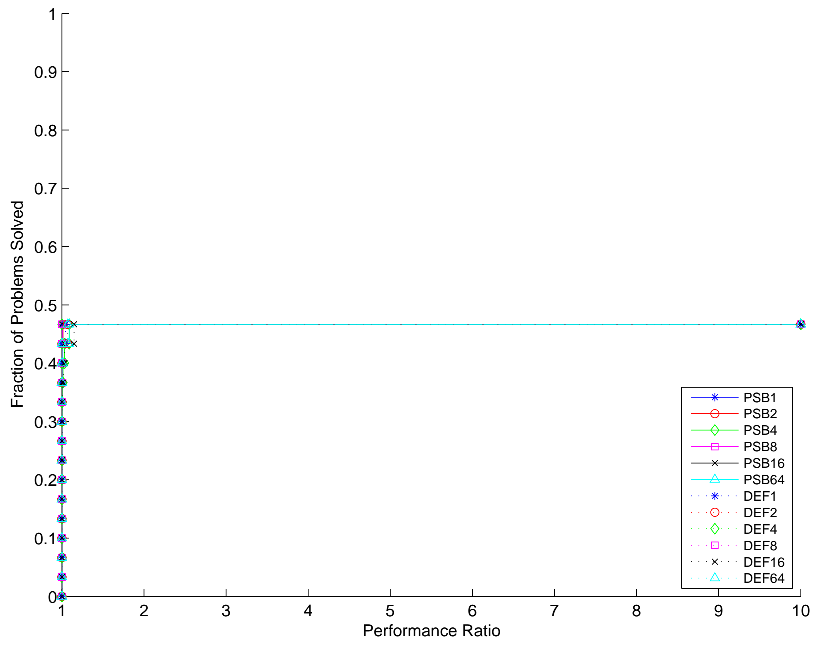


(e) Ilupack MLICT Memory Profile

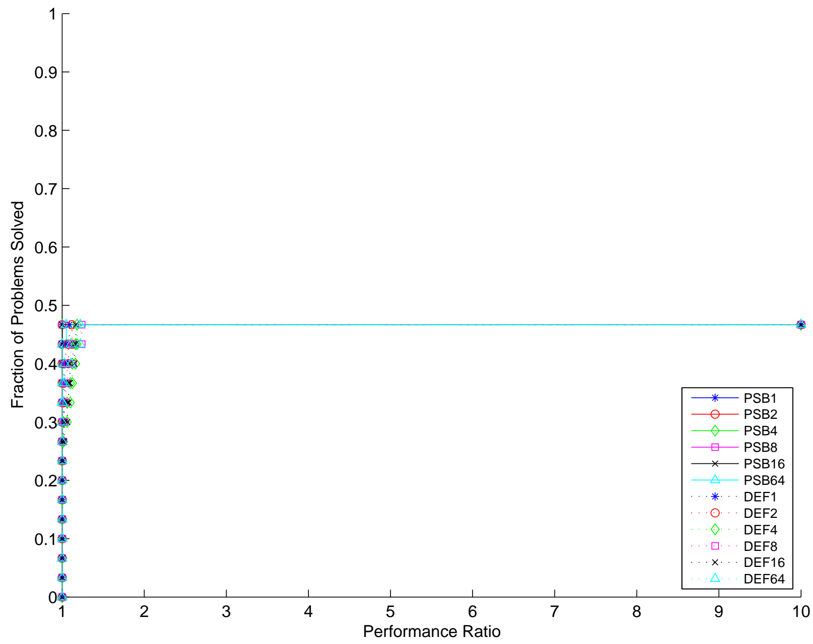


(f) Ilupack MLICT Time Profile

Figure 26: Memory and time performance profile curves for the default (DEF) and the problem specific best (PSB) configurations of PETSc-IC(k), PETSc-ILUTP and Ilupack-MLICT in the serial

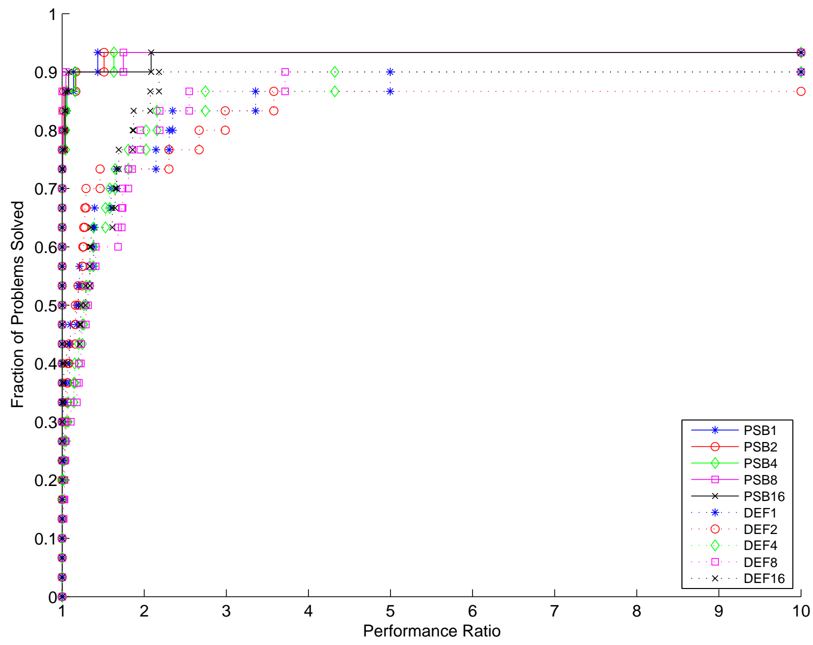


(a) Petsc BlockSolve Memory Profile

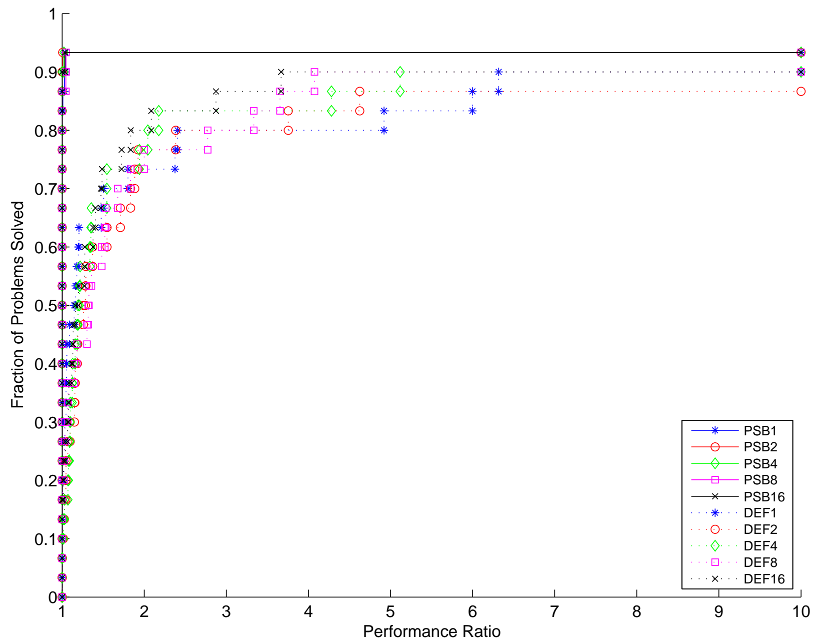


(b) Petsc BlockSolve Time Profile

Figure 27: Memory and time performance profile curves for the default (DEF) and the problem specific best (PSB) configurations of PETSc-BlockSolve for multiple processors.

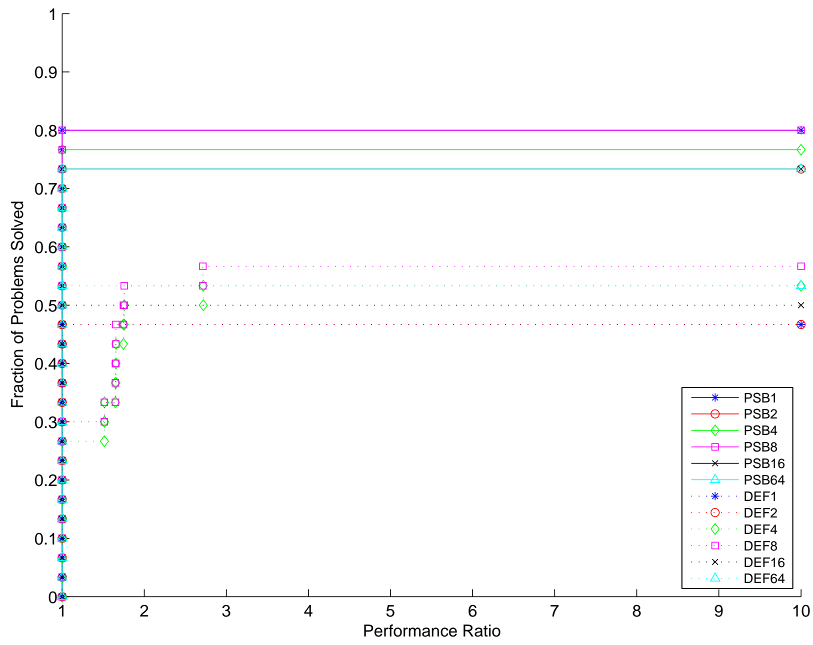


(a) WSMP ICT Memory Profile

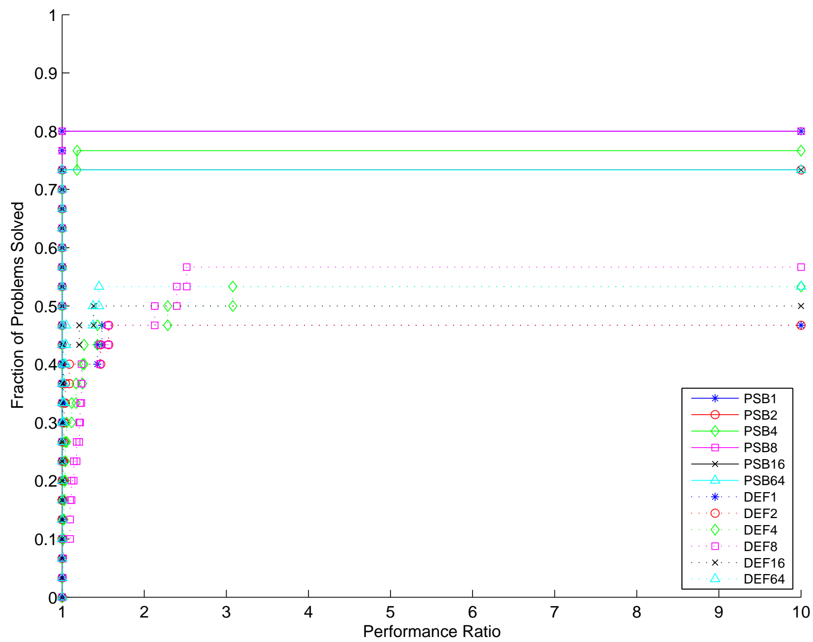


(b) WSMP ICT Time Profile

Figure 28: Memory and time performance profile curves for the default (DEF) and the problem specific best (PSB) configurations of WSMP-ICT for multiple processors.

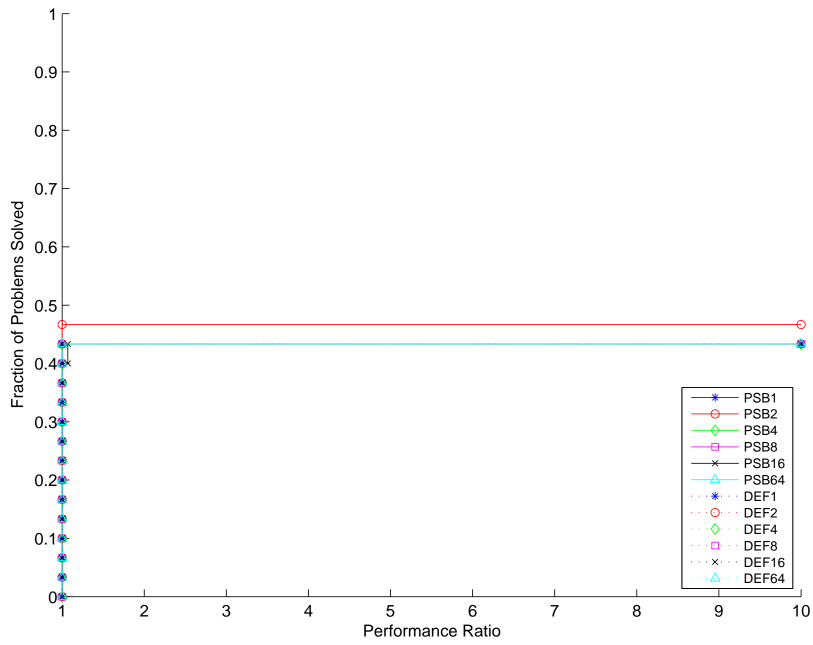


(a) Hypre IC(k) Memory Profile

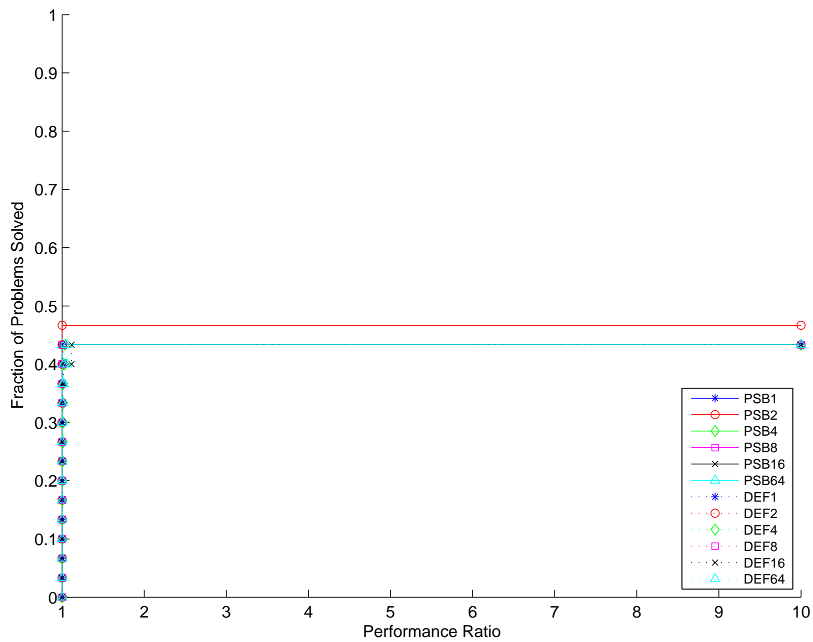


(b) Hypre IC(k) Time Profile

Figure 29: Memory and time performance profile curves for the default (DEF) and the problem specific best (PSB) configurations of Hypre-IC(k) preconditioner for multiple processors.

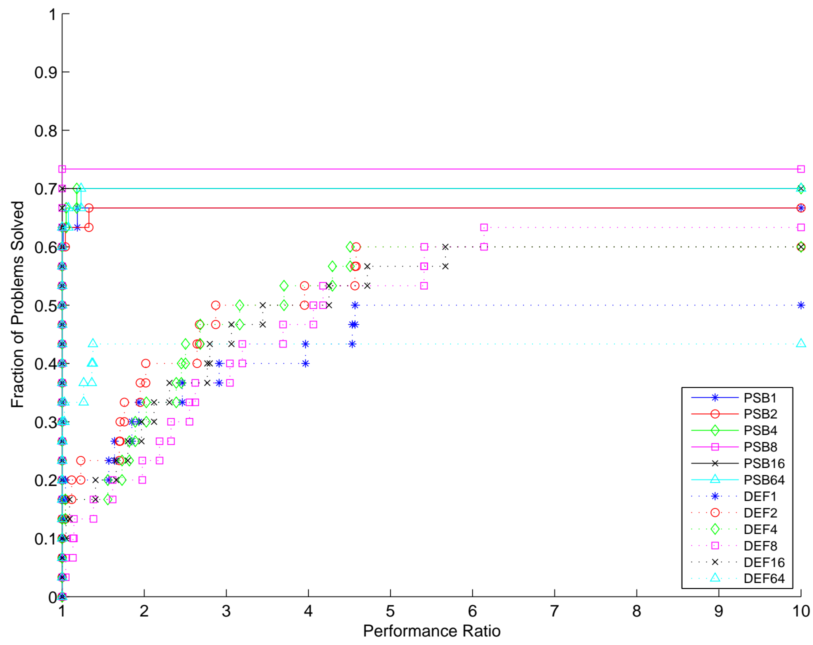


(a) Trilinos IC(k) Memory Profile

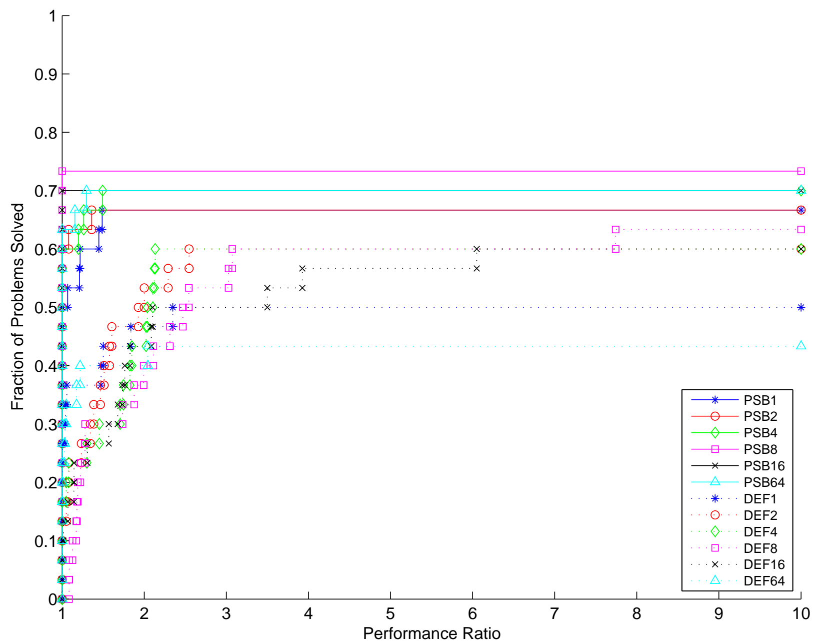


(b) Trilinos IC(k) Time Profile

Figure 30: Memory and time performance profile curves for the default (DEF) and the problem specific best (PSB) configurations of Trilinos-IC(k) preconditioner for multiple processors.

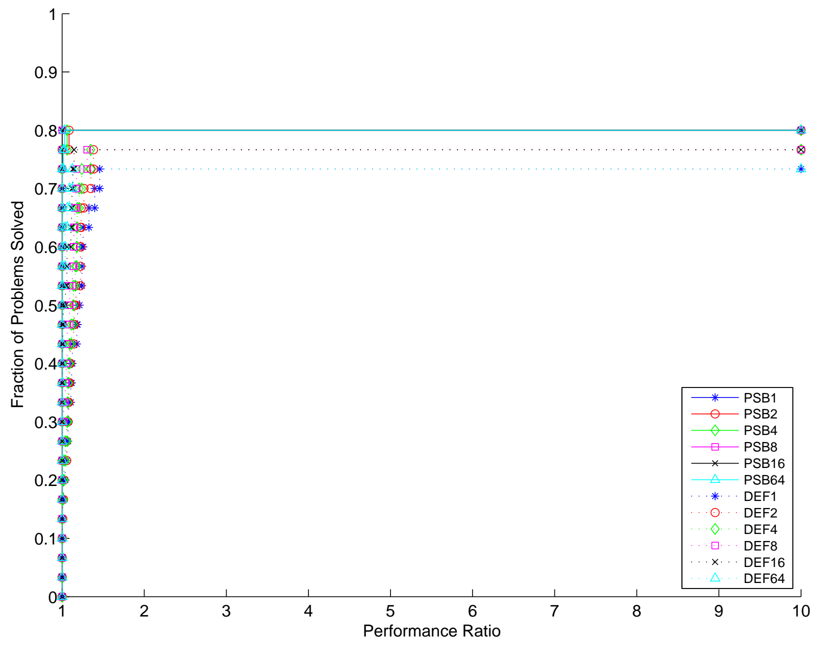


(a) Trilinos ILUT Memory Profile

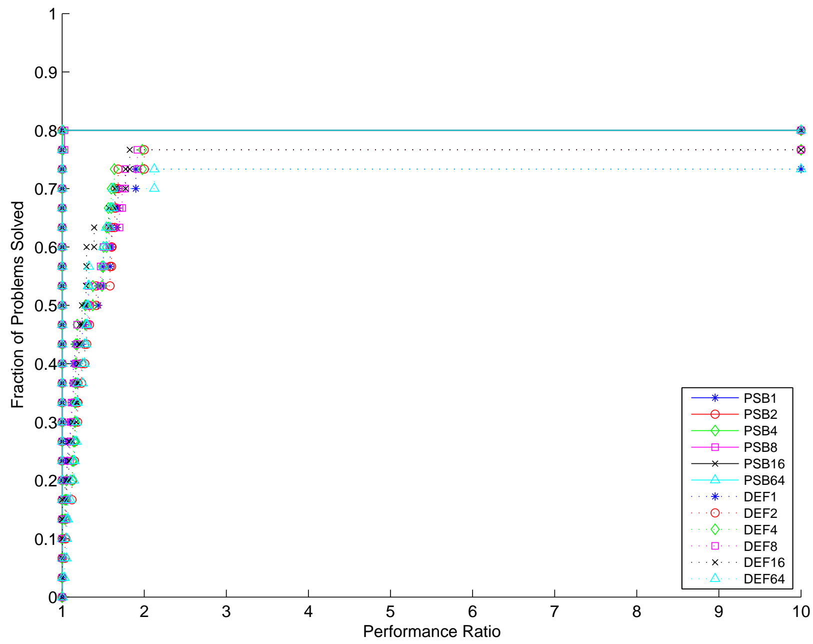


(b) Trilinos ILUT Time Profile

Figure 31: Memory and time performance profile curves for the default (DEF) and the problem specific best (PSB) configurations of Trilinos ILUT/ICT preconditioner for multiple processors.

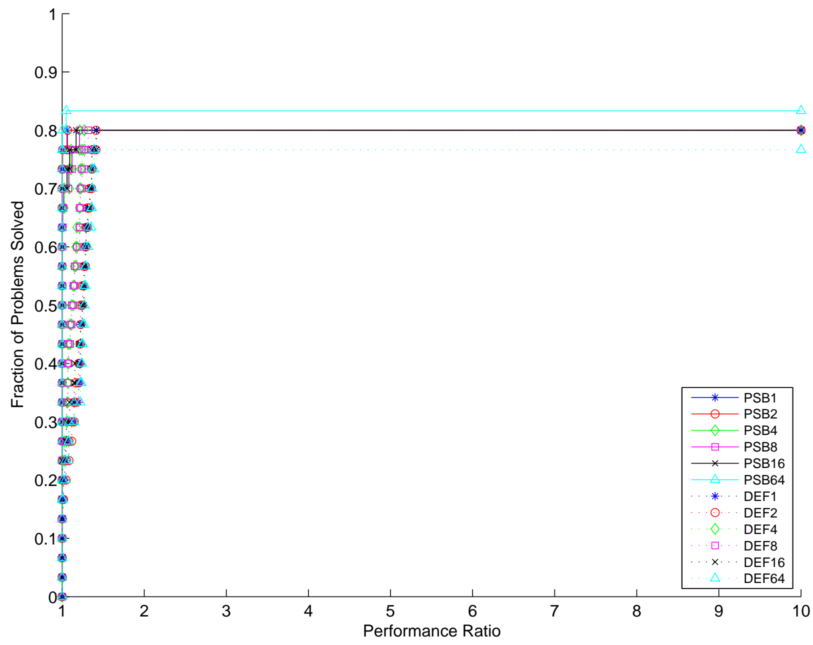


(a) Hypre ParaSails Memory Profile

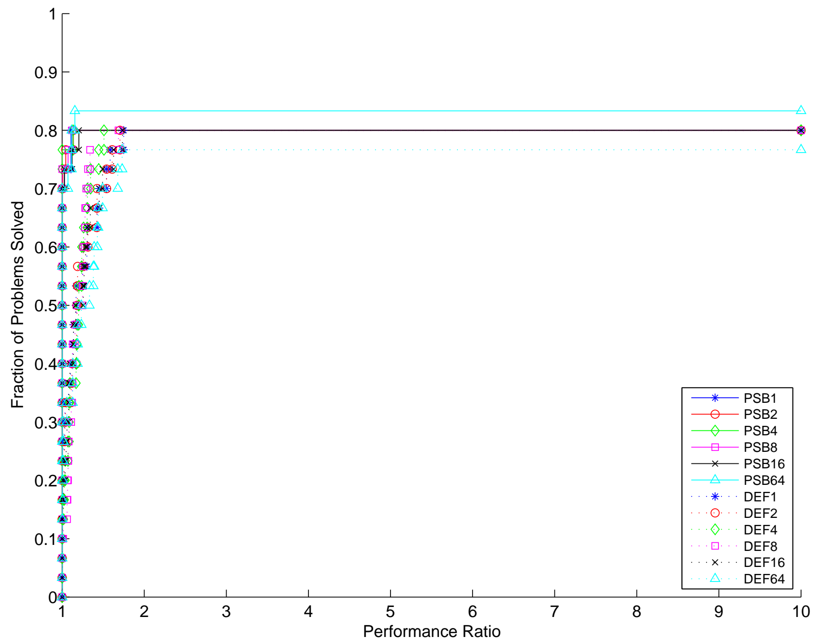


(b) Hypre ParaSails Time Profile

Figure 32: Memory and time performance profile curves for the default (DEF) and the problem specific best (PSB) configurations of Hypre-ParaSails preconditioner for multiple processors.

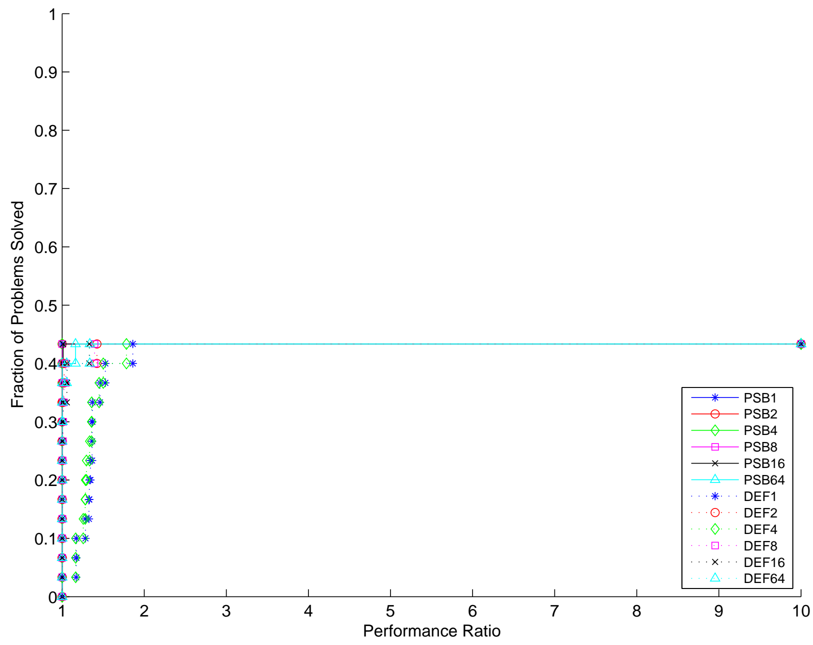


(a) Hypre BoomerAMG Memory Profile

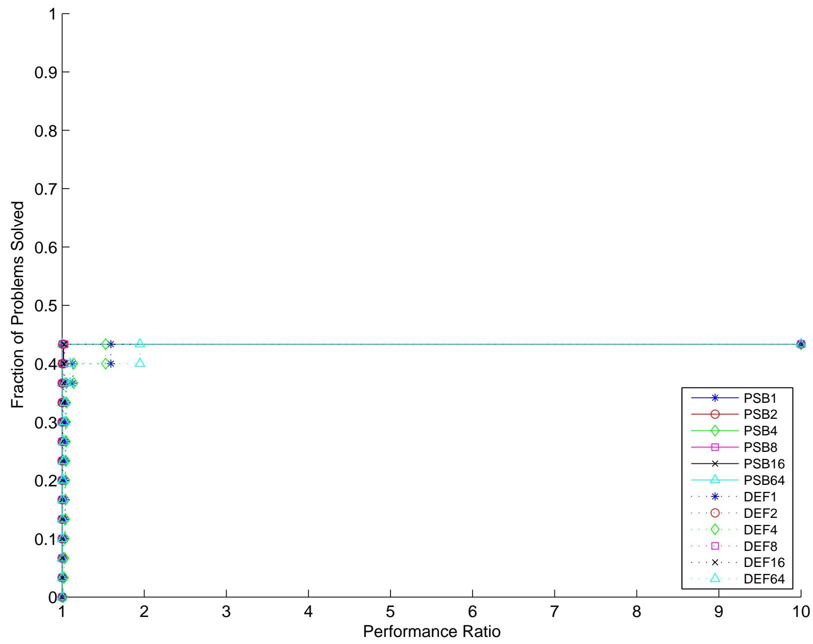


(b) Hypre BoomerAMG Time Profile

Figure 33: Memory and time performance profile curves for the default (DEF) and the problem specific best (PSB) configurations of Hypre-BoomerAMG preconditioner for multiple processors.

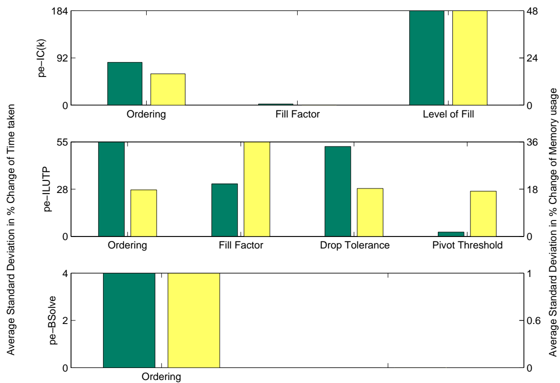


(a) Trilinos ML Memory Profile

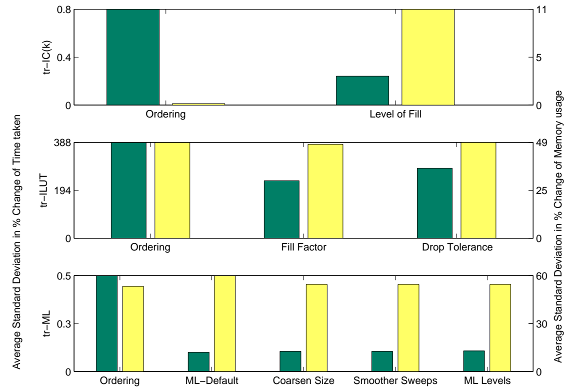


(b) Trilinos ML Time Profile

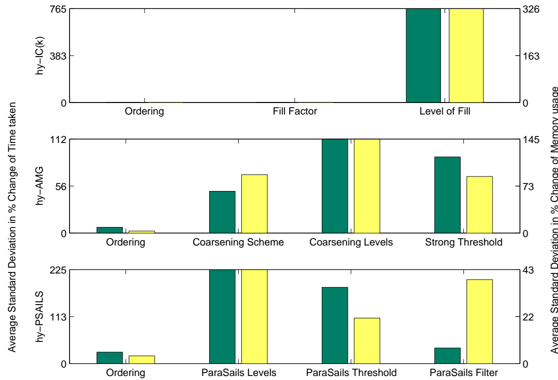
Figure 34: Memory and time performance profile curves for the default (DEF) and the problem specific best (PSB) configurations of Trilinos-ML preconditioner for multiple processors.



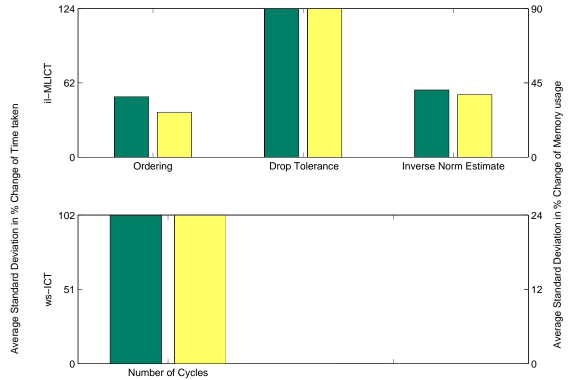
(a) Variation in time and memory for PETSc's IC(k), ILUTP and BlockSolve.



(b) Variation in time and memory for Trilinos' IC(k), ILUT and ML.



(c) Variation in time and memory for Hypre's IC(k), BoomerAMG and ParaSails.



(d) Variation in time and memory for Ilupack MLICT and WSMP ICT.

Figure 35: Average standard deviations of the variation in time and memory use of a preconditioner for different values of the fine-tunable parameters in the serial case. The scales for the time and memory bars are different and are shown on the left and right, respectively.

the values indicated on the y-axis give an indication of the percentage change in performance one could expect while varying 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 for memory and vice versa for time. 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.

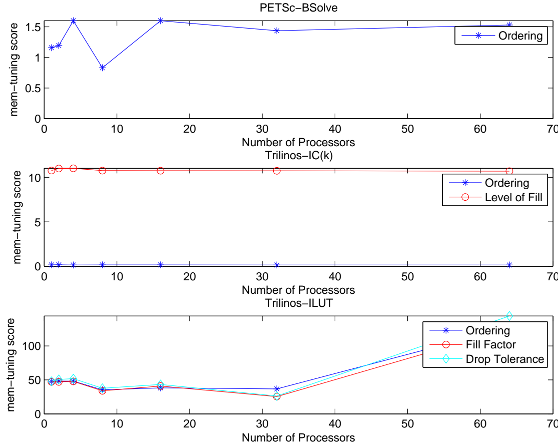
Figures 36 and 37 shows the effect of fine-tuning the various parameters on memory and time as the number of processors is increased. In Figures 36(a) and 36(b), we can see that ordering has a bigger impact on memory than for time in the case of PETSc-BlockSolve. However, the y-scales indicate that the variation is not much. For Trilinos-IC(k), the level of fill had a larger influence on memory than it had for time. There is only a slight increase in the time variations as we increase the number of processors. For Trilinos-ILUT, there is high variation in time for the all the parameters for low number of processors and it reduces with increasing number of processors. With respect to memory, we observe relatively less variation for the serial case, but steadily increases with increasing number of processors.

For Hypre-IC(k) in Figure 36(c) and 36(d), the level of fill is the most important parameter and except for the initial variation, it remains more or less flat for increasing number of processors. In the case of Trilinos-ML, even though there is some variation with respect to memory and time for the number of levels and ordering respectively, it is fairly small. In the case of Hypre-AMG, the gap between the parameters is maintained across the processors for both memory and time and ordering scheme had the least effect.

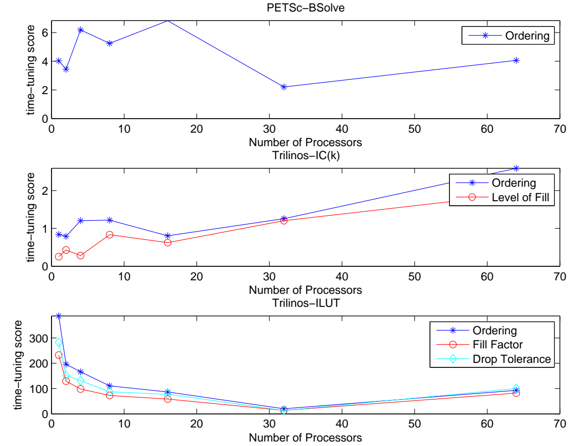
In the case of Hypre-ParaSails in Figure 37(a), the filter parameter had more impact on memory than other parameters, but the importance of this parameter decreases as with increasing number of processors. However, the y-axis scales indicate that the variation in memory is relatively small in comparison to the time variation shown in Figure 37(b). In the case of time, the threshold parameter has a significant impact on time and the effect is more when the number of processors is high. WSMP-ICT also benefits from fine tuning, but the effect is lower in case of multiple processors.

4.4 Comparison of Preconditioners Across Packages

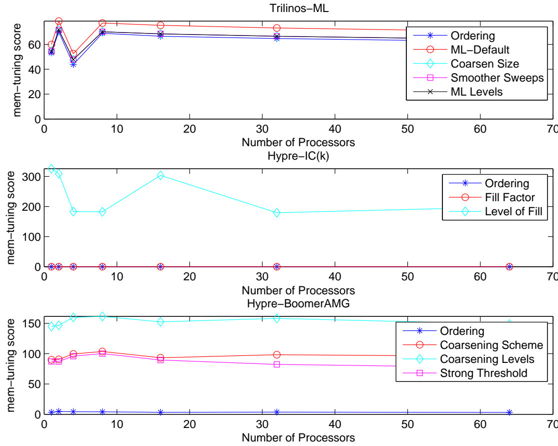
We now analyze the relative performance of the various preconditioner implementations. Specifically, we compare the various implementation under two scenarios. We first present the results of comparing the parameter configurations that performed best overall across all problems with respect to MTP for each preconditioner implementation (default). Then, we present the results corresponding to the best MTP parameter configuration for a preconditioner implementation for each problem (problem specific best). For both the default and problem specific scenarios, we present the results for the single processor as well as the 16 processor case. In addition, we also present the results on simultaneously projecting multiple performance metrics which helps in analyzing the relative effects of memory, time, robustness both in the serial and parallel case.



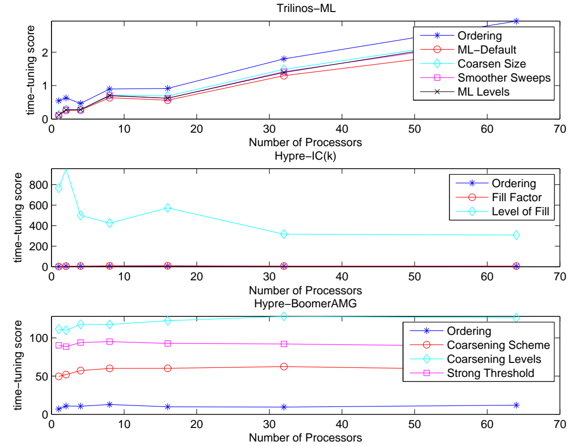
(a) Variation in Memory for PETSc-BlockSolve, Trilinos-IC(k), and Trilinos-ILUT



(b) Variation in Time for PETSc-BlockSolve, Trilinos-IC(k), and Trilinos-ILUT

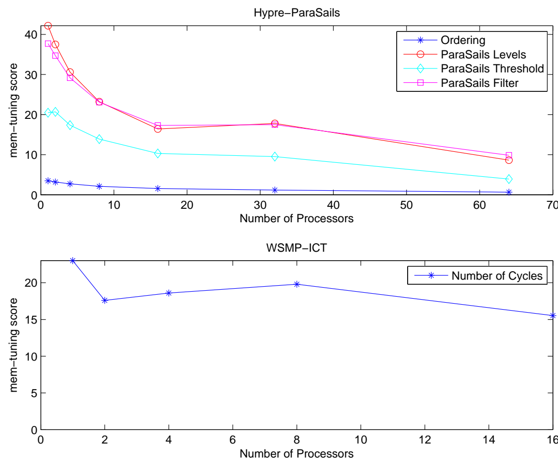


(c) Variation in Memory for Trilinos-ML, Hypre-IC(k), and Hypre-AMG

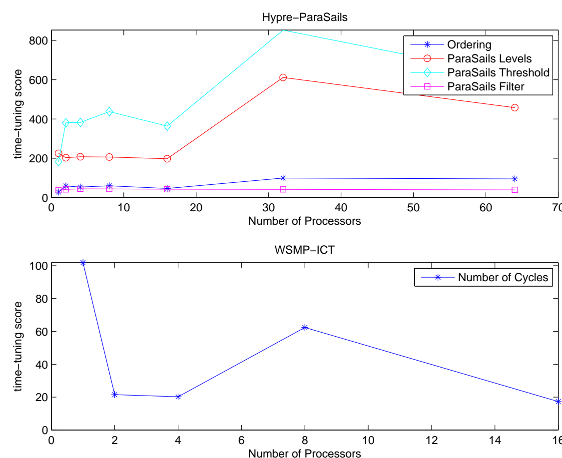


(d) Variation in Time for Trilinos-ML, Hypre-IC(k), and Hypre-AMG

Figure 36: Average standard deviations of the variation in memory and time use of the various preconditioners for different values of each parameter with respect to the default configuration in the parallel case. Each curve in the subplots corresponds to a parameter that is varied in our study.



(a) Variation in Memory for Hypre-ParaSails and WSMP-ICT



(b) Variation in Time for Hypre-ParaSails and WSMP-ICT

Figure 37: Average standard deviations of the variation in memory and time use of the various preconditioners for different values of each parameter with respect to the default configuration in the parallel case. Each curve in the subplots corresponds to a parameter that is varied in our study.

4.4.1 Best Default Configurations

The winning parameter combinations of Tables 11 and 12 are good candidates for default values that have a high probability of yielding a small memory-time product for an arbitrary problem in the case of 1 and 16 processors respectively.

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.

Figures 38 and 39 show the memory profiles of the configurations shown in Tables 11 and 12 respectively. The memory profile of WSMP direct solver is also included in these figures. Recall that the memory plotted here corresponds to the total memory needed for storing the non zeros in the linear system as well as the memory allocated in the heap during the preconditioner creation. Therefore, the memory efficiency reported here for a package-preconditioner curve could indicate a implementation specific weighted combination of both the factors.

For the single processor case in Figure 38, WSMP-ICT, Hypre-BoomerAMG, Hypre-ParaSails and Ilupack-MLICT appear to be the most memory efficient and robust. Even though PETSc-IC(k) and Trilinos-IC(k) preconditioners solves the maximum number of problems using the least memory, it is not as robust as the others and their respective curves flatten out fairly early. Note that ILUT preconditioner implementations appear to have one of 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. Another interesting observation is that the default profile curve of Hypre-IC(k) is worse than that of ILUT and the direct solver. This could be partly explained by the memory required to store the full linear system and also the high level of fill value (6) that is

Preconditioner	**Solver**	Ordering	**Preconditioner Parameters**
PETSc-IC(K)	CG	RCM	Fill factor (1), Level of fill (0)
PETSc-ILUT	GMRES65	RCM	Fill factor (5) ,Pivot threshold (0) Drop tolerance (0.001)
PETSc-BlockSolve	CG	RCM	-
Trilinos-IC(K)	CG	RCM	Level of fill (0)
Trilinos-ILUT	GMRES30	RCM	Fill factor (3), Drop tolerance (0.001)
Trilinos-ML	GMRES30	RCM	2-Level Domain decomposition, Number of levels (4) Smoother sweeps (2), Maximum coarse grid size (64)
Hypre-IC(K)	CG	RCM	Level of fill (0)
Hypre-BoomerAMG	CG	RCM	Coarsening Sceme (PMIS), Aggregate levels (10) Strong threshold (0.9)
Hypre-ParaSails	CG	NONE	Number of levels (1), Threshold (0.1), Filter (0.05)
Ilupack-MLICT	CG	RCM	Drop-tolerance (0.03), Inverse norm estimate (100)
Wsmp-ICT	Auto select (CG, GMRES)	Auto select (RCM, ND)	8th self tuning step for drop-tolerance and fill factor

Table 11: Iterative solver configurations that resulted in the best overall performance with respect to memory-time product profile area in the serial case.

Preconditioner	**Solver**	Ordering	**Preconditioner Parameters**
PETSc-BlockSolve	CG	ND	-
Trilinos-IC(K)	CG	RCM	Level of fill (0)
Trilinos-ILUT	GMRES30	RCM	Fill factor (10), Drop tolerance (.001)
Trilinos-ML	CG	NONE	Smothed aggregation, Number of levels (6) Smoother sweeps (2), Maximum coarse grid size (16)
Hypre-IC(K)	CG	RCM	Level of fill (0)
Hypre-BoomerAMG	CG	NONE	Coarsening Sceme (PMIS), Aggregate levels (10) Strong threshold (0.9)
Hypre-ParaSails	CG	ND	Number of levels (1), Threshold (0.1) , Filter (0.05)
Wsmp-ICT	Auto Select (CG, GMRES)	Auto select (RCM, ND)	8th self tuning step for drop-tolerance and fill factor

Table 12: Iterative solver configurations that resulted in the best overall performance with respect to memory-time product profile area in the 16 processor case.

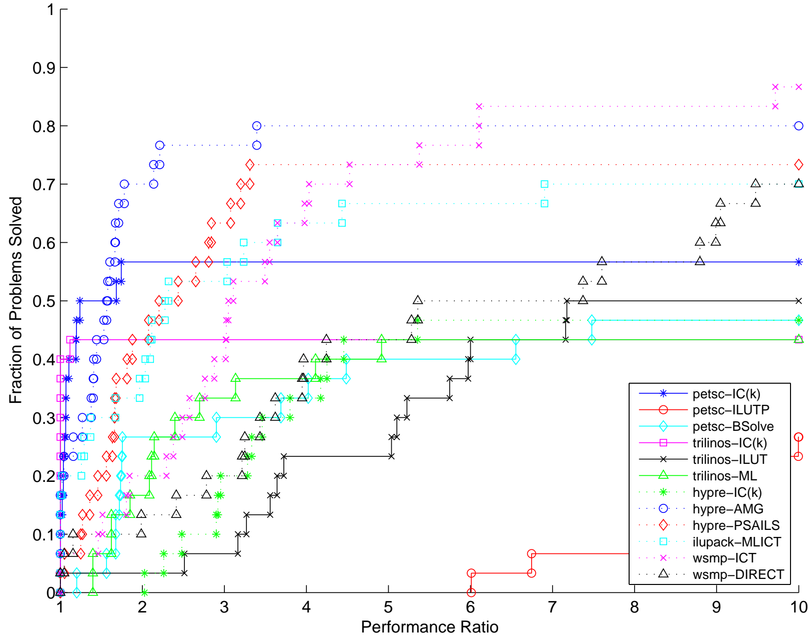


Figure 38: Memory performance profile curves for the direct solver and the best memory-time product configuration of the various preconditioner implementations shown in Table 11 (single processor case).

needed to solve more number of problems (thus maximizing the profile area) among the other less robust parameter choices for Hypre-IC(k). The analysis of the impact of level of fill on memory in Section 4.1 provides more details. For the 16 processor case in Figure 39, Hypre-BoomerAMG is the most memory efficient followed by WSMP-ICT. The relative ordering of other preconditioners remain the same except for Hypre-ParaSails which shows a higher memory usage that in the serial case.

Figures 40 and 41 show the time profiles of the configurations shown in Tables 11 and 12, along with that of the WSMP direct solver. In Figure 40, the direct solver turns out to be the fastest solver for about 70% of the problems in the serial case. This is followed by PETSc-IC(0), which is the fastest one for about 12% 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. For the 16 processor case in Figure 41, the direct solver is still the fastest for about 72% of the problems followed by WSMP-ICT and Hypre-BoomerAMG. Note that Hypre-BoomerAMG was one of the slowest in the single processor case. Almost all the preconditioners seem to close in the gap with the direct solver in the 16 processor case expect for WSMP-ICT whose serial performance relative to the direct solver was much better.

Figures 42 and 43 show the memory-time product profiles of the configurations shown in Tables 11 and 12 along with 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 in the serial case is very similar to that of the corresponding time profiles in Figures 40 whereas in the 16 processor case, Hypre-AMG moves up because of its excellent memory efficiency.

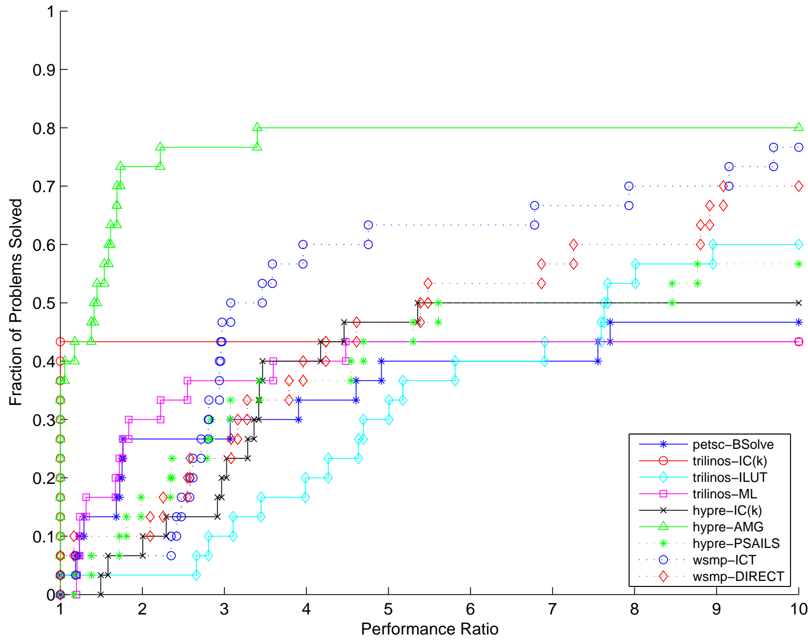


Figure 39: Memory performance profile curves for the direct solver and the best memory-time product configuration of the various preconditioner implementations shown in Table 12 (16 processor case).

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.

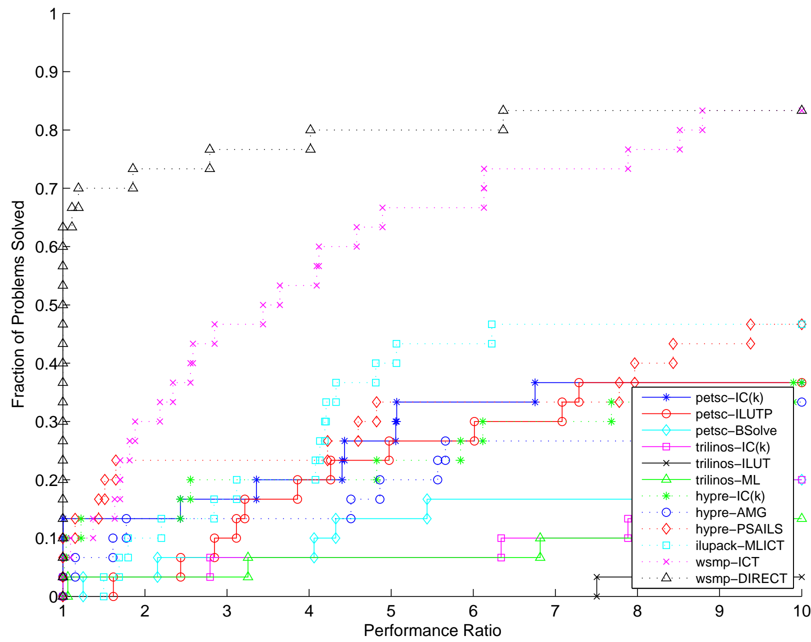


Figure 40: Time performance profile curves for the direct solver and the best time-memory product configuration of the various preconditioner implementations shown in Table 11 (single processor case).

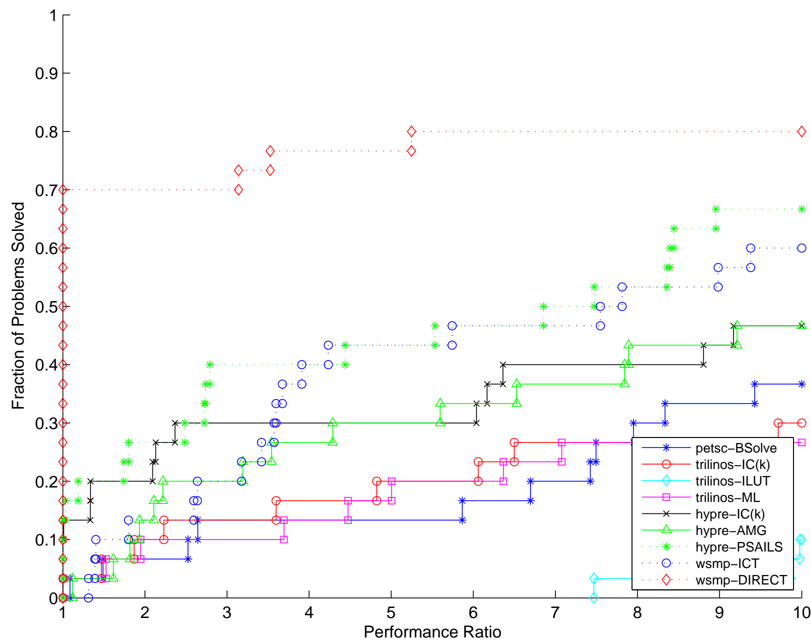


Figure 41: Time performance profile curves for the direct solver and the best time-memory product configuration of the various preconditioner implementations shown in Table 12 (16 processor case).

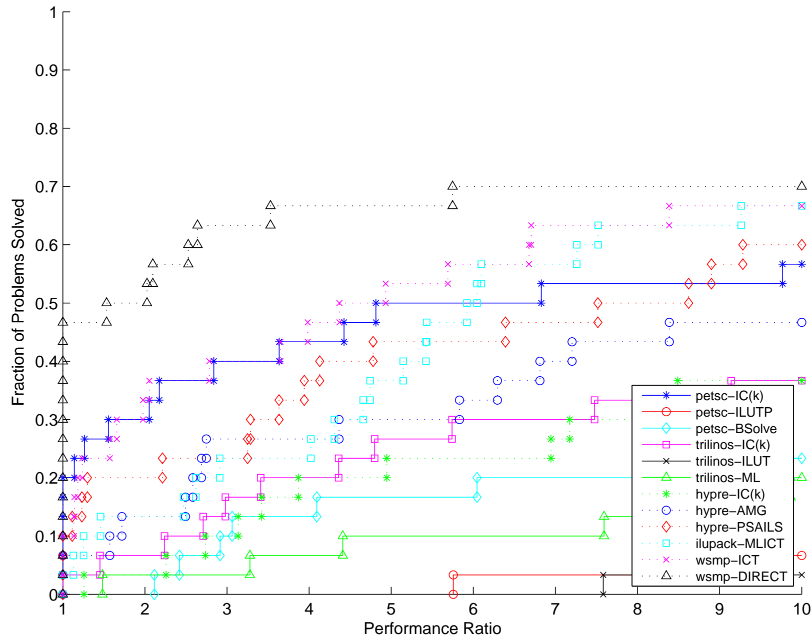


Figure 42: Memory-time product profiles of preconditioner configurations in Table 11.

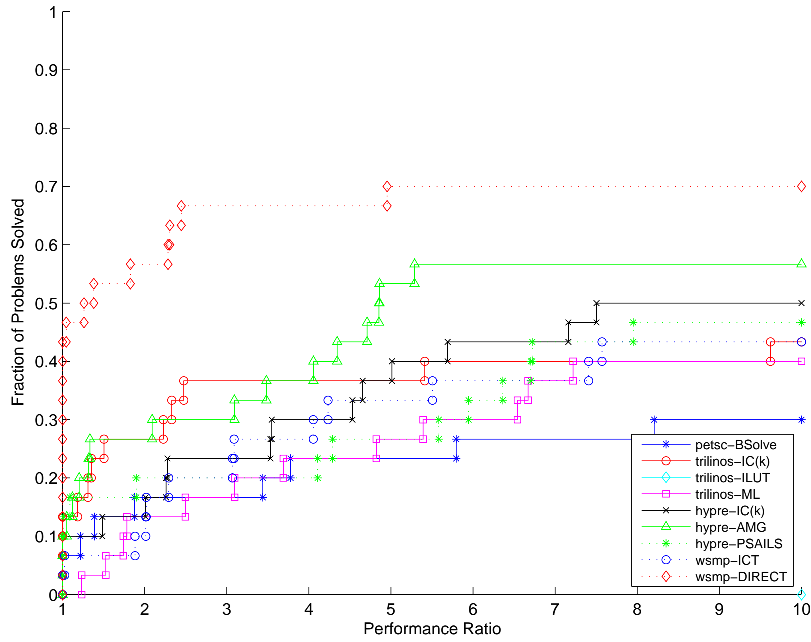


Figure 43: Memory-time product profiles of preconditioner configurations in Table 12.

4.4.2 Problem Specific Parameter Selection

If all the 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.

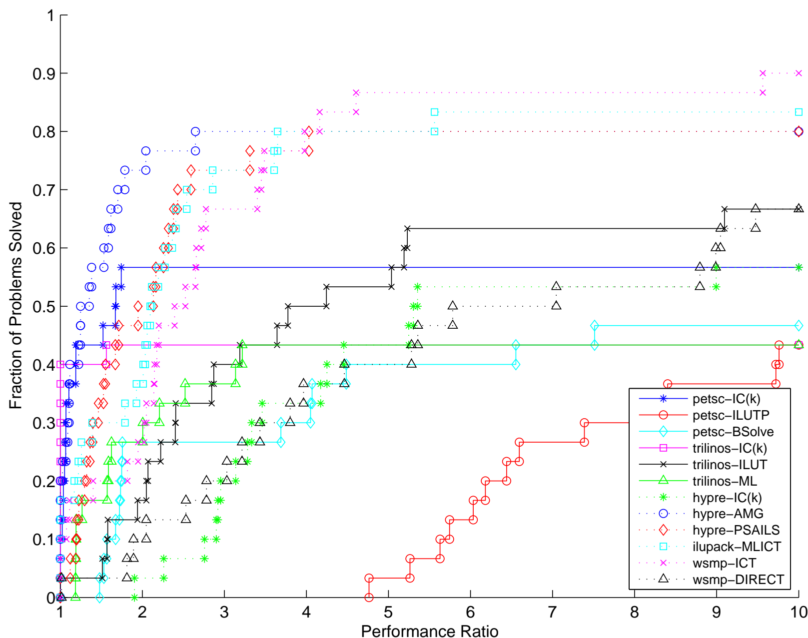


Figure 44: Memory performance profiles for the direct solver and the memory values corresponding to the best problem specific memory-time product configuration for each preconditioner (single processor case).

Figures 44 and 45 show the memory profiles for 1 and 16 processors respectively 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. For the single processor case, this can be seen by comparing the memory profiles in Figures 38 and 44. All iterative solver curves move upwards with respect to the direct solver curve in Figure 44, when compared to Figure 38. Besides consuming less memory, most preconditioners are able to solve more problems successfully with problem-specific parameter tuning. The most remarkable improvement with respect to memory occurs for Trilinos-ILUT and Hypre-IC(k). The improvement is even more drastic while using 16 processors as observed by comparing Figures 39 and 45. The higher memory efficiency of Trilinos-ILUT can also be explained by the fact that it is picked from a pool of both ICT and ILUT configurations.

Figures 46 and 47 show the time profiles of all the preconditioners when the parameter configuration for each problem was chosen individually to minimize its memory-time product. Just like the memory profiles, the time profiles of the preconditioners improve significantly when compared to those for the overall best parameter configuration in Figure 40. The most noted improvements in the serial case are for Ilupack-MLICT, Hypre-IC(k) and Hypre-ParaSails. While using 16 processors, a comparison of Figures 41 and 47 show that Hypre's BoomerAMG, IC(k) and ParaSails

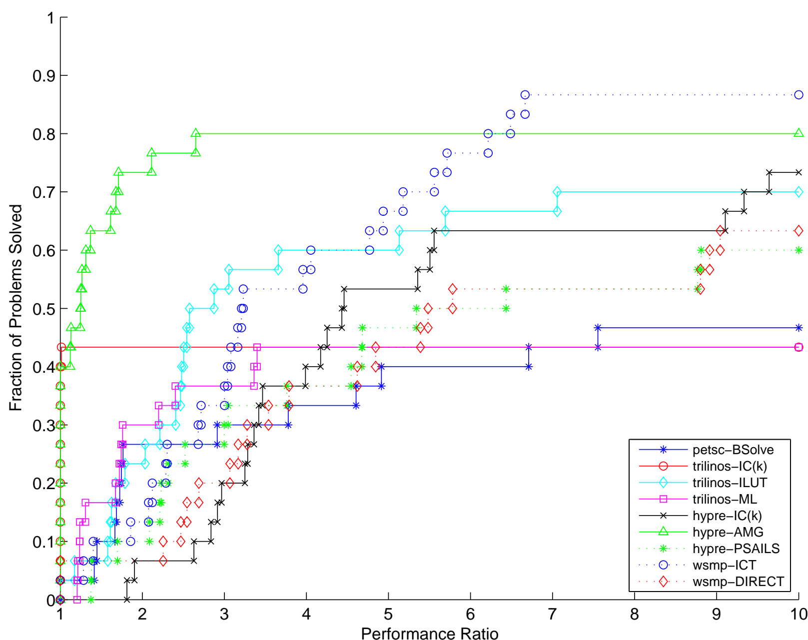


Figure 45: Memory performance profiles for the direct solver and the memory values corresponding to the best problem specific memory-time product configuration for each preconditioner (16 processor case).

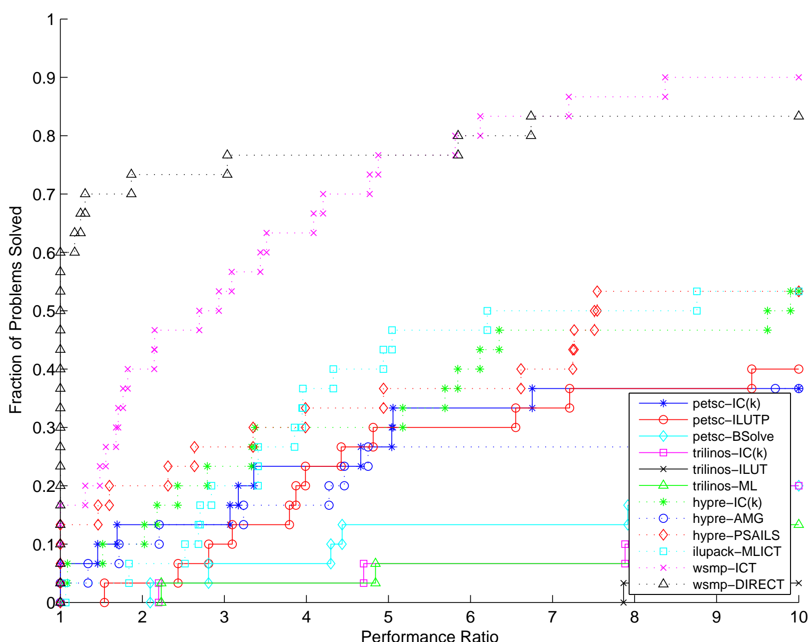


Figure 46: Time performance profiles for the direct solver and the time values corresponding to the best problem specific memory-time product configuration for each preconditioner (single processor case).

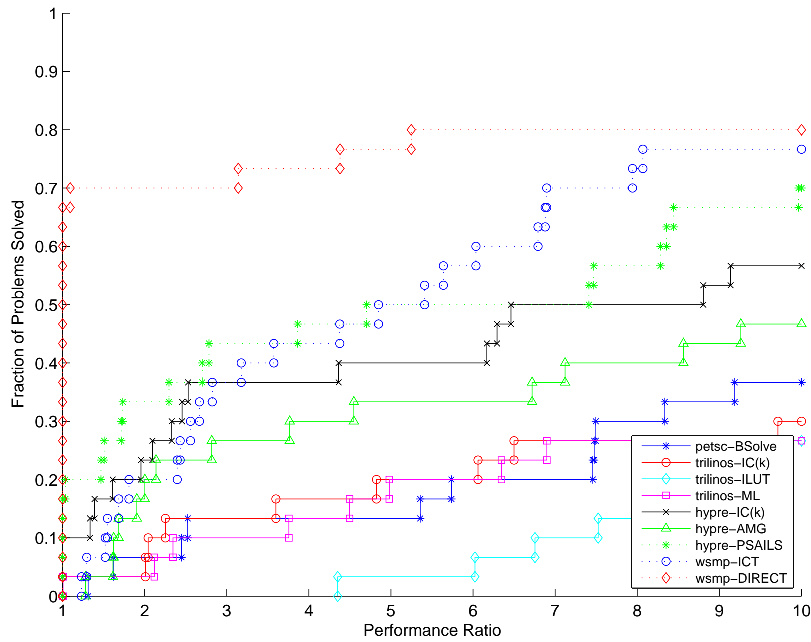


Figure 47: Time performance profiles for the direct solver and the time values corresponding to the best problem specific memory-time product configuration for each preconditioner (16 processor case).

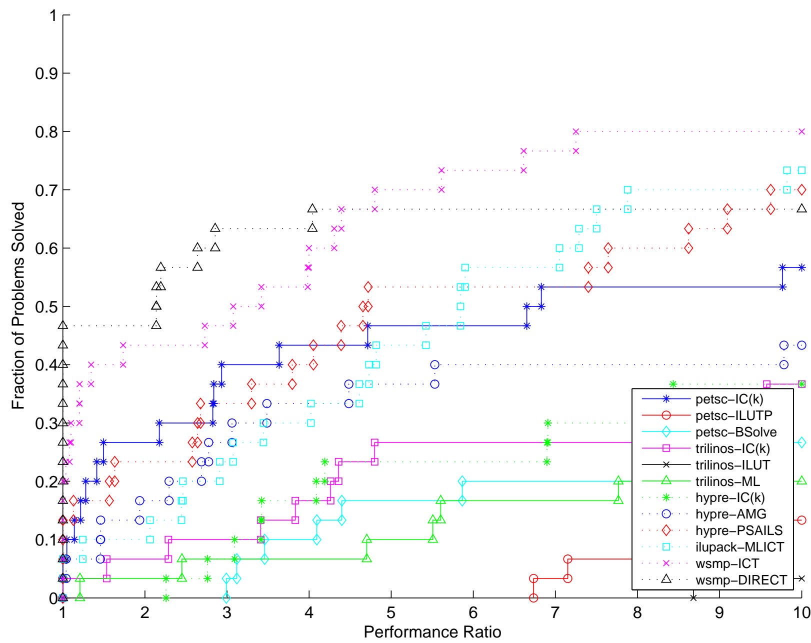


Figure 48: Memory-time product profiles for the direct solver and the best problem specific values for each preconditioner (single processor case) .

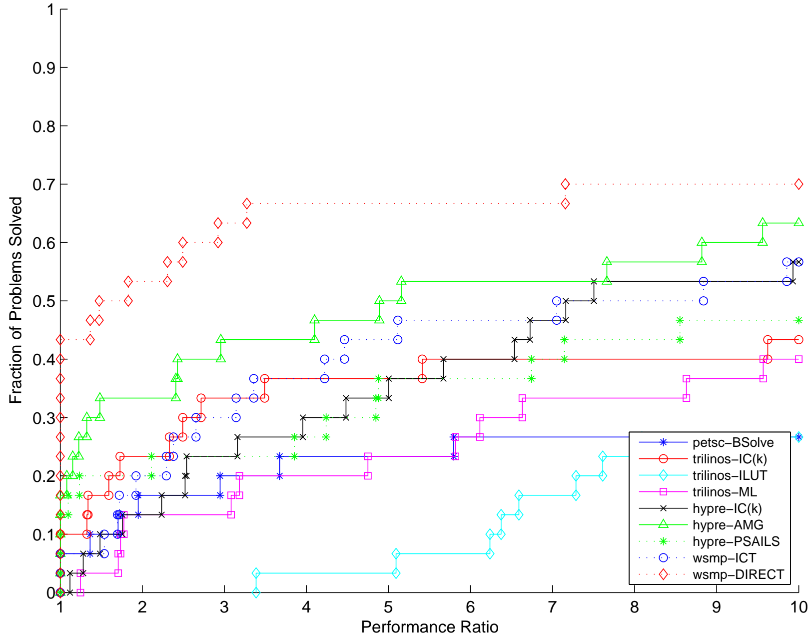


Figure 49: Memory-time product profiles for the direct solver and the best problem specific values for each preconditioner (16 processor case) .

reduces the time performance gap with WSMP-ICT. A similar comparison of the memory-time product profiles for the tuned parameters in Figures 48 and 49 with those for the overall best parameter configuration in Figures 42 and 43 also show a similar trend as observed for time.

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 897 combinations of packages, solvers, preconditioners, and parameters that we tested for about 70% of the matrices and had a better memory-time product for about 50% 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.

Tables 13 and 14 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 the serial and 16 processor case respectively. 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 15 out of the 30 matrices. As expected, the iterative solvers do much better for large 3-D problems. Among the iterative solvers, WSMP-ICT does best for 8 problems , PETSc-IC(k) for 3, and Hypre-Parasails and Ilupack for 2 . For the 16 processor case in Table 14, among the iterative solvers, Hypre-ParaSails and BoomerAMG do best for 5 problems, WSMP-ICT for 3 and PETSc-BlockSolve for 1.

Matrix	Iterative Parameter	I.Mem	I.Time	D.Mem	D.Time
90153	wsmp, ICT, 8th self tuning step	1.36e+08	3.66	**1.94e+08**	**2.47**
af_shell7	hypre, SAI, CG, RCM, PLev0, Th-.9, Flt-.9	2.05e+08	77.71	**8.3e+08**	**11.75**
algor-big	tril, IC(K), CG, RCM, LF0	**7.05e+08**	**1244.5**	-	-
audikw_1	hypre, SAI, CG, ND, PLev0, Th.1, Flt.001	**8.67e+08**	**466.29**	9.5e+09	87.00
bmwcra_1	wsmp, ICT, 6th self tuning step	3.34e+08	65.55	**5.68e+08**	**11.28**
bst-1	wsmp, ICT, 6th self tuning step	3.16e+09	423.15	**3.21e+09**	**86.78**
bst-2	ilupack, MLICT, CG, IND, DT1e-02, IE100	1.24e+09	1339.25	**2.35e+09**	**10.30**
cfd1	petsc, IC(K), CG, RCM, FILL1, LF0	**1.77e+07**	**8.08**	1.57e+08	2.41
cfd2	hypre, SAI, CG, RCM, PLev0, Th-.75, Flt-.9	**5.49e+07**	**16.73**	3.1e+08	6.35
conti20	wsmp, ICT, 5th self tuning step	5.87e+07	5.89	**6.4e+07**	**.96**
garybig	hypre, AMG, CG, ND, FALG, AGG0, ST.9	**6.66e+09**	**11597.5**	-	-
G3_circuit	wsmp, ICT, 8th self tuning step	**2.53e+08**	**35.50**	9.56e+08	2.11
hood	wsmp, ICT, 5th self tuning step	1.98e+08	3.91	**2.57e+08**	**2.51**
inline_1	wsmp, ICT, 7th self tuning step	2.2e+09	197.43	**1.5e+09**	**27.43**
kyushu	petsc, IC(K), CG, RCM, FILL10, LF2	**2.54e+08**	**39.73**	9.22e+09	1336.48
ldoor	hypre, SAI, CG, RCM, PLev0, Th-.9, Flt-.9	4.8e+08	169.34	**1.37e+09**	**22.45**
msdoor	wsmp, ICT, 6th self tuning step	5.14e+08	21.54	**4.92e+08**	**5.12**
mstamp-2c	hypre, SAI, CG, NONE, PLev2, Th.1, Flt.05	**7.26e+08**	**76.35**	-	-
nastran-b	hypre, SAI, CG, ND, PLev1, Th.01, Flt0	**1.08e+09**	**413.37**	8.62e+09	538.88
nd24k	ilupack, MLICT, CG, RCM, DT1e-02_IE100	**1.53e+08**	**65.24**	2.75e+09	412.18
oilpan	wsmp, ICT, 6th self tuning step	7.64e+07	2.19	**9.45e+07**	**1.02**
parabolic_fem	hypre, AMG, CG, RCM, HMIS, AGG10, ST.7	7.51e+07	9.36	**2.33e+08**	**2.90**
pga-rem1	wsmp, ICT, 7th self tuning step	3.23e+08	34.30	**7.42e+08**	**11.10**
pga-rem2	wsmp, ICT, 8th self tuning step	**4.96e+08**	**81.31**	1.98e+09	44.63
qa8fk	wsmp, ICT, 8th self tuning step	**1.64e+07**	**1.51**	1.93e+08	4.60
qa8fm	wsmp, ICT, 8th self tuning step	**9.7e+06**	**.17**	1.87e+08	4.21
ship_003	wsmp, ICT, 8th self tuning step	**1.63e+08**	**13.50**	5.29e+08	16.85
shipsec5	ilupack, MLICT, CG, RCM, DT3e-02, IE10	**1e+08**	**2.20**	4.48e+08	12.93
thermal2	hypre, AMG, CG, RCM, HMIS, AGG10, ST.25	1.6e+08	59.73	**4.84e+08**	**6.44**
torso	wsmp, ICT, 8th self tuning step	**3.33e+07**	**4.18**	6.77e+08	24.43

Table 13: Table showing the time (in seconds) and memory values (in bytes) corresponding to the best problem specific memory-time product for iterative and direct solvers in the single processor case. The bold values indicate the solver configuration for which the product of memory and time was the lowest.

4.4.3 Relative Strengths of Preconditioners and Sensitivity to Parameter Tuning

In Sections 4.1, we observed that different preconditioners and solvers display different strengths and weaknesses for the SPD matrices in our test suite. They also have different degrees of robustness. Some are more memory efficient than others, while some are faster than others. In Section **??**, 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. Figures 50 and 51 display all this relative information about the performance of various preconditioners by means of a single information-rich graphic.

Each figure has two sets of circles for each preconditioner. The green (dark) circles correspond to the default parameter configurations. The yellow (light) circles correspond to the corresponding 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 plotting the default and problem specific performance profiles in a single plot. The size of each circle is proportional to the

Matrix	Iterative Parameter	I.Mem	I.Time	D.Mem	D.Time
90153	petsc, BSolve, CG, RCM, LF0	7.74e+07	2.53	**1.94e+08**	**0.27**
af_shell7	hypre, AMG, CG, RCM, HMIS, AGG10, ST.9	2.6e+08	5.42	**8.49e+08**	**1.44**
algor-big	tril, ICK, CG, ND, LF0	**7.05e+08**	**85.34**	-	-
audikw_1	hypre, PSAILS, CG, ND, PLev1, Th-.75_Flt-.9	**1.12e+09**	**47.92**	-	-
bmwcra_1	hypre, AMG, CG, RCM, PMIS, AGG10_ST.25	9.87e+07	16.29	**5.71e+08**	**1.17**
bst-1	wsmp, ICT, AUTO, AUTO, 5th self tuning step	4.32e+09	92.66	**3.37e+09**	**11.48**
bst-2	-	-	-	**2.31e+09**	**7.32**
cfd1	hypre, AMG, CG, NONE, PMIS, AGG10_ST.25	**2.08e+07**	**1.05**	1.71e+08	0.37
cfd2	tril, ICK, CG, RCM, LF0	**2.82e+07**	**3.61**	3.14e+08	0.75
conti20	hypre, AMG, CG, ND, PMIS, AGG0_ST.25	2.11e+07	6.31	**6.7e+07**	**0.14**
garybig	-	-	-	-	-
G3_circuit	wsmp, ICT, AUTO, AUTO, 7th self tuning step	**2.2e+08**	**4.73**	9.31e+08	3.66
hood	wsmp, ICT, AUTO, AUTO, 8th self tuning step	2.11e+08	0.84	**3.03e+08**	**0.35**
inline_1	wsmp, ICT, AUTO, AUTO, 7th self tuning step	1.96e+09	4.49	**1.57e+09**	**3.64**
kyushu	tril, ICK, CG, RCM, LF0	**2.38e+08**	**28.75**	-	-
ldoor	wsmp, ICT, AUTO, AUTO, 8th self tuning step	1.21e+09	5.51	**1.51e+09**	**20.30**
msdoor	wsmp, ICT, AUTO, AUTO, 8th self tuning step	5.72e+08	5.01	**5.59e+08**	**0.73**
mstamp-2c	hypre, PSAILS, CG, ND, PLev0, Th.1_Flt0	**1.01e+09**	**1.70**	-	-
nastran-b	hypre, PSAILS, CG, NONE, PLev1, Th-.75_Flt-.9	**1.41e+09**	**61.26**	9.25e+09	66.95
nd24k	hypre, PSAILS, CG, ND, PLev1, Th0_Flt0	**5.37e+08**	**8.12**	2.7e+09	35.58
oilpan	wsmp, ICT, AUTO, AUTO, 2nd self tuning step	1.02e+08	1.45	**1.02e+08**	**0.13**
parabolic_fem	hypre, AMG, CG, NONE, HMIS, AGG10_ST.7	**7.56e+07**	**0.85**	2.38e+08	0.40
pga-rem1	wsmp, ICT, AUTO, AUTO, 7th self tuning step	2.88e+08	6.13	**7.33e+08**	**1.40**
pga-rem2	wsmp, ICT, AUTO, AUTO, 7th self tuning step	**4.78e+08**	**9.31**	2.01e+09	5.53
qa8fk	hypre, AMG, CG, RCM, PMIS, AGG10_ST.25	**1.84e+07**	**0.22**	1.89e+08	0.43
qa8fm	hypre, AMG, CG, ND, PMIS, AGG0_ST.5	**1.6e+07**	**0.05**	1.83e+08	0.41
ship_003	petsc, BSolve, CG, RCM, LF0	**1.18e+08**	**3.90**	5.44e+08	1.54
shipsec5	petsc, BSolve, CG, RCM, LF0	**1.51e+08**	**2.89**	5.05e+08	1.18
thermal2	hypre, AMG, CG, RCM, HMIS, AGG10_ST.25	1.6e+08	4.04	**4.92e+08**	**0.89**
torso	hypre, AMG, CG, NONE, PMIS, AGG10_ST.25	**4.25e+07**	**0.80**	6.8e+08	2.20

Table 14: Table showing the time (in seconds) and memory values (in bytes) corresponding to the best problem specific memory-time product for iterative and direct solvers in the 16 processor case. The bold values indicate the solver configuration for which the product of memory and time was the lowest.

number of problems solved.

The height of a circle in the figures 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, in Figure 50 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. For the default parameters, Hypre-Parasails and Ilupack are progressively faster, but use slightly more memory than Hypre-BoomerAMG. WSMP-ICT with default parameters is fairly memory efficient and 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 Hypre-IC(k). Ilupack also shows

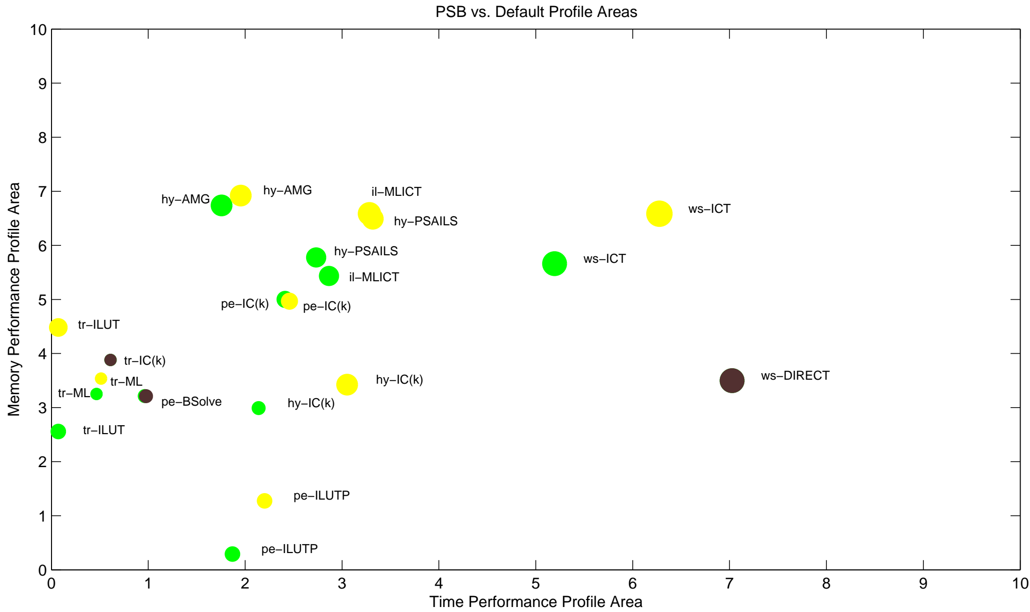


Figure 50: Plot of the time profile area versus the memory profile area for various preconditioner implementations (single processor case). Each circle represents a preconditioner whose name consists of the first two letters of the name of the package followed by the type of preconditioner. The size 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. If the yellow and green circles overlap, it is shown as brown.

significant improvement and outperforms Hypre-ParaSails in both time and memory. Another interesting observation from Figure 50 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 on incomplete factorization, perform better in terms of time, memory, and robustness.

Whether with default or with fine tuned parameters, the best solvers in the serial case 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 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.

In Figure 51, we compare the relative performance of the preconditioners in the 64 processor case. Just like the serial case, the direct solver is very fast but memory intensive in comparison to other preconditioners. Hypre-BoomerAMG is relatively more efficient with respect to both time and memory and the benefits of fine-tuning is is also more than that observed in 50. The relative

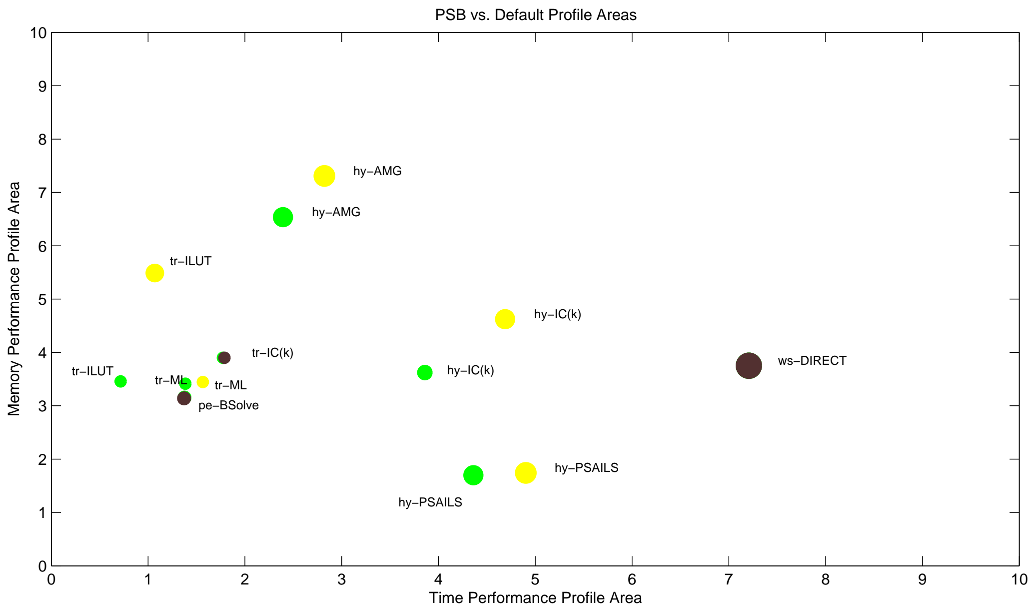


Figure 51: Plot of the time profile area versus the memory profile area for various preconditioner implementations (64 processor case). Each circle represents a preconditioner whose name consists of the first two letters of the name of the package followed by the type of preconditioner. The size of a 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. If the yellow and green circles overlap, it is shown as brown.

time efficiency of Hypre-ParaSails increases, but it comes at the expense of increased memory usage. Although we observed only memory improvements for Trilinos-ILUT in the single processor case, on 64 processors, there is a significant relative improvement in time. To summarize, Hypre-BoomerAMG and Trilinos-ILUT are highly memory efficient, WSMP-direct and Hypre-ParaSails are relatively fast whereas Hypre-IC(k) seems to balance both time and memory.

4.5 Parallel Efficiency

An important measure often reported for parallel implementations is the time efficiency across multiple processors. Most of these packages are used for large scale scientific simulations involving thousands of processors and might exhibit excellent weak scaling. Since the matrices in our test are of fixed size, we perform strong scaling studies. The efficiency obtained during a weak scaling study will be much better than that observed for our strong scaling study. Since efficiency is measured using the ratio of the serial time to the parallel time, the values depend heavily on the serial implementation of the respective preconditioners. Figure 52 shows the average time efficiency for all the solved problems while using the various parallel preconditioners. Since WSMP-ICT could only be run on a single node of 16 processors due to its shared memory implementation, we do not have results for WSMP-ICT on 32 and 64 processors.

The efficiency plots show a completely different scenario from that seen in the problem specific time profile plots comparing these preconditioners. Trilinos-ILUT exhibits super-linear speedup till 64 processors. Even though Trilinos-ILUT was memory efficient in the parallel case, the time profile

was one of the lowest in comparison to other preconditioners. Trilinos IC(K) and ML preconditioners also exhibit very little drop in efficiency for the problems that it could solve. Hypre ParaSails and AMG show similar drop in efficiencies. WSMP-ICT and WSMP-Direct shows the maximum drop in efficiency. This drastic drop could also be attributed to its superior time performance in the serial case in comparison to other preconditioners.

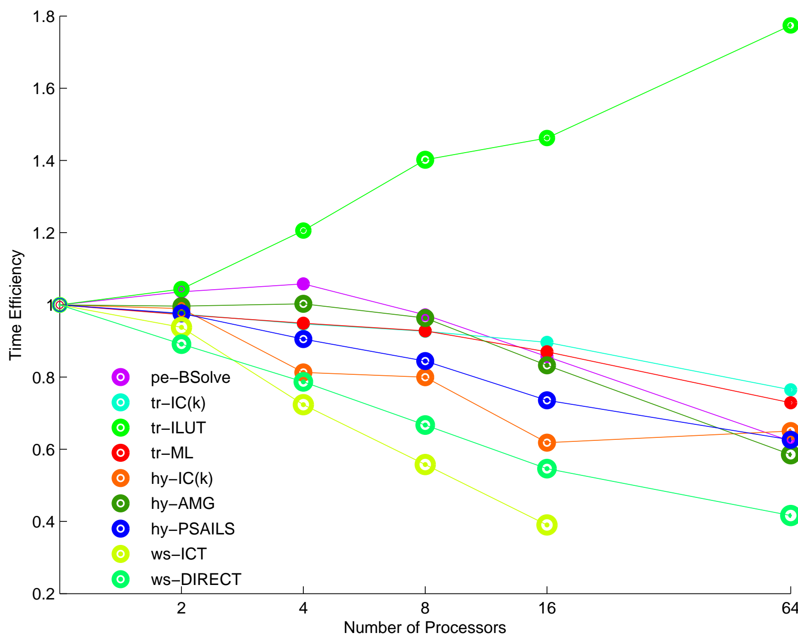


Figure 52: Average time efficiency corresponding to the problem specific best parameters of the various preconditioner implementations. The legend names consists of the first two letters of the name of the package followed by the type of preconditioner. The size of a circle is proportional to the number of problems solved.

5 Performance Analysis Infrastructure

In this section, we describe the performance analysis framework that we have developed to evaluate the various solver configurations. Figure 53 shows the various components of the framework which is composed of a performance data collection unit and multiple analysis and visualization units. Even though the implemented framework described is specific for the domain of preconditioned iterative solvers, it is readily extensible to other domains. We now describe each component in more detail.

- *Data Collection & Pre-processing Unit* consists of serial and parallel driver programs for some of the most widely used solver packages such as PETSc, Trilinos, Hypre, etc. The input to this component includes a set of linear systems (P), a set of hardware configurations (e.g., number of processors)(H), a subset of supported solver configurations (S), where each solver configuration includes details on the solver and its parameters, ordering scheme, preconditioner and its parameters, and finally a set of performance metrics (M). Other choices of interest include driver specific information such as the right hand side (RHS), initial solution, exact solution and, stopping criterion (e.g., the relative residual norm or maximum number of iterations of

the solver), each of which has a default value and can also be modified by the user. The system performs empirical trials for each possible setting, collects the specified metrics, and pre-processes them appropriately to generate the performance data (denoted by $PerfData$). It also generates a mapping ($ConfigMap$) from the solver configurations S to various key attributes such as the package-name, preconditioner-name, solver and the associated parameters, e.g., package:Hypre, preconditioner:IC(k), solver:CG, ordering:RCM,level-of-fill:1, etc. A grouping (G) of solver configurations is also created using the specified rules, with the default choice based on package-preconditioner combinations. The user can specify a group as a combination of any of the solver configuration components such as package, preconditioner, solver, ordering etc. For example, Figure 10 shows the performance profiles for groups created with the following specification: package:Ilupack, preconditioner:MLICT, solver:CG, and ordering:RCM.

- *Parameter Fine-tuning Analysis Unit* computes the variability in the performance due to a single parameter while keeping all others fixed. This analysis is especially relevant for solver configurations within each group (Section 4.3.1). It requires as input the performance data $PerfData$ as well as the solver configuration to parameter map $ConfigMap$ generated by the data collection & pre-processing unit in order to partition the configurations into subgroups that differ along a single parameter. The normalized plots generated by this unit (e.g., Figures 35 - 37) show the relative impact of the various parameters within each configuration group on memory and time performance across the various hardware configurations.

- *Intra Group Analysis Unit* computes good default solver configurations ($DEF - Configs$) that result in the maximum AUC among all the solver configurations within that group (as in Table 6 - 10). Although, the default option is to use MTP as the metric (m) for determining the winning solver configuration, the user can specify other metrics such as memory and time. A series of performance profile (PP) plots are created for each combination of metric, hardware configuration, and solver configuration group for comparative analysis at a fine-grained resolution(e.g., Figures 2 - 19). In addition, the effect of fine-tuning is captured by comparing the performance of the best default configuration and the problem specific best performance (PSB) (as in Figures 26 - 34). The performance metric values in both cases (DEF, PSB) are accumulated for each solver configuration group, problem, and hardware configuration and output ($GroupPerfData$) for further analysis.

- *Inter Group Analysis Unit* provides a coarse grain comparison of the different solver configuration groups based on group performance data $GroupPerfData$ corresponding to the best default configuration (DEF) as well as the problem specific best PSB). Performance profile plots are generated for each hardware configuration for both DEF and PSB values (as in Figures 40 - 49). In addition, we also present a multi-metric plot that simultaneously captures the trends up to three metrics (e.g., memory, time and robustness) in the PSB/DEF performance (as in Figures 50, 51). This provides a snapshot of the relative strengths and weaknesses of the various solver configuration groups with respect to the performance metrics under consideration.

6 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

P = {p}	:	Linear Systems	S = {s}	:	Solver Configurations
M = {m}	:	Metric (time, memory, MTP)	F = {f}	:	Fine-tunable Factors
H = {h}	:	Hardware Configurations	G = {g}	:	Solver Configuration Group
T = {t}	:	Type (DEF, PSB, DEF vs. PSB)	PP	:	Performance Profile

Figure 53: Overview of the performance analysis infrastructure. Boxes represent the proc
units, dotted ellipses represent the input and output data while the plots generated for visuali
are represented by solid ellipses.

range of scientific applications. For each package and preconditioner combination, we identify the best parameter choices using a novel performance profile based criterion 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 the 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 varying levels of tunability and optimizing individual parameters impacts the performance to different degrees. We provide a comparison of the performance of various iterative solver configurations relative to the direct solver, which 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. The results used for analysis 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 (4943) for each matrix. Therefore, it is not clear how generalizable the results are. However, the performance collection and reporting infrastructure we have developed is independent of the test suite and can be used on any set of test matrices from any domain. Application scientists can provide us with specific matrices in their domain and the results specific to those matrices can be generated with little effort. The methodology described in this paper can be helpful to researchers in the field of iterative methods and preconditioning for evaluating different aspects of new solver techniques in a systematic fashion. As part of our future work on this topic, we plan to set up an anonymous ftp site so that users can submit their matrices and obtain a report on the relative performance of each solver configuration group. 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, Edmond Chow, Jonathan Hu, 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.

References

[1] Hypre, High Performance Preconditioners: Users Manual. Technical report, Lawrence Livermore National Laboratory, 2006.

[2] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong 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] Matthias Bollhöfer and Yousef Saad. Multilevel preconditioners constructed from inverse-based ILUs. *SIAM J. Sci. Comput.*, 27(5):1627–1650, 2006.

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

[7] Edmond Chow. Parallel implementation and practical use of sparse approximate inverse preconditioners with a priori sparsity patterns. *International Journal of High Performance Computing Applications*, 15(1):56–74, 2001.

[8] Edmond Chow and Yousef Saad. Approximate inverse preconditioners via sparse-sparse iterations. *SIAM Journal on Scientific Computing*, 19(3):995–1023, 1998.

[9] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*, pages 157–172, New York, NY, USA, 1969. ACM.

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

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

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

[13] M.W. Gee, C.M. Siefert, J.J. Hu, R.S. Tuminaro, and M.G. Sala. ML 5.0 Smoothed Aggregation User's Guide. Technical Report SAND2006-2649, Sandia National Laboratories, 2006.

[14] Alan George and Joseph W. Liu. *Computer Solution of Large Sparse Positive Definite*. Prentice Hall Professional Technical Reference, 1981.

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

[16] Anshul Gupta. Fast and effective algorithms for graph partitioning and sparse-matrix ordering. *IBM Journal of Research and Development*, 41(1&2):171–184, 1997.

[17] Anshul 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.*

[18] Anshul 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.*

[19] Anshul Gupta and Thomas George. Adaptive techniques for improving the performance of incomplete factorization preconditioning. Technical Report RC 24598 (W0807-036), IBM T. J. Watson Research Center, Yorktown Heights, NY, July 7, 2008. To appear in *SIAM Journal on Scientific Computing.*

[20] Van Emden Henson and Ulrike Meier Yang. BoomerAMG: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics: Transactions of IMACS*, 41(1):155–177, 2002.

[21] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[22] David Hysom and Alex Pothen. "a scalable parallel algorithm for incomplete factor preconditioning". *"SIAM Journal of Scientific Computing"*, 22(6):2194–2215, 2000.

[23] Mark T. Jones and Paul E. Plassmann. An improved incomplete cholesky factorization. *ACM Trans. Math. Softw.*, 21(1):5–17, 1995.

[24] George Karypis and Vipin Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 35, Washington, DC, USA, 1996. IEEE Computer Society.

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

[26] J.W. Ruge and K. Stüben. Multigrid methods. 3:73–130, 1987. Frontiers in Applied Mathematics.

[27] Y. Saad and B. Suchomel. ARMS: An Algebraic Recursive Multilevel Solver for General Sparse Linear Systems. *Numerical Linear Algebra with Applications*, 9(5):359–378, 2002.

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

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

[30] M. Sala and M. Heroux. Robust Algebraic Preconditioners with IFPACK 3.0. Technical Report SAND-0662, Sandia National Laboratories, 2005.