Technical Report
RAL-TR-2005-005

A numerical evaluation of sparse direct solvers for the solution of large sparse, symmetric linear systems of equations

N I M Gould Y Hu J A Scott

May 6, 2005

Council for the Central Laboratory of the Research Councils

Enquires about copyright, reproduction and requests for additional copies of this report should be addressed to:

Library and Information Services
CCLRC Rutherford Appleton Laboratory
Chilton Didcot
Oxfordshire OX11 0QX
UK
Tel: +44 (0)1235 445384
Fax: +44(0)1235 446403
Email: library@rl.ac.uk

CCLRC reports are available online at:
http://www.clrc.ac.uk/Activity/ACTIVITY=Publications;SECTION=225;

ISSN 1358-6254

Neither the Council nor the Laboratory accept any responsibility for loss or damage arising from the use of information contained in any of their reports or in any communication about their tests or investigations.

RAL-TR-2005-005

A numerical evaluation of sparse direct solvers for the solution of large sparse, symmetric linear systems of equations

Nicholas I. M. Gould[1,2,3], Yifan Hu[4], and Jennifer A. Scott[1,2,3]

ABSTRACT

In recent years a number of solvers for the direct solution of large sparse, symmetric linear systems of equations have been developed. These include solvers that are designed for the solution of positive-definite systems as well as those that are principally intended for solving indefinite problems. The available choice can make it difficult for users to know which solver is the most appropriate for their applications. In this study, we use performance profiles as a tool for evaluating and comparing the performance of serial sparse direct solvers on an extensive set of symmetric test problems taken from a range of practical applications.

[1] Computational Science and Engineering Department, Rutherford Appleton Laboratory,
Chilton, Oxfordshire, OX11 0QX, England, UK.
Email: n.i.m.gould@rl.ac.uk & j.a.scott@rl.ac.uk

[2] Current reports available from "http://www.numerical.rl.ac.uk/reports/reports.shtml".

[3] This work was supported by EPSRC grants GR/R46641 and GR/S42170.

[4] Wolfram Research, Inc., 100 Trade Center Drive, Champaign, IL61820, USA.
Email: yifanhu@wolfram.com

Computational Science and Engineering Department
Atlas Centre
Rutherford Appleton Laboratory
Oxfordshire OX11 0QX

May 6, 2005.

1 Introduction

Solving linear systems of equations lies at the heart of many problems in computational science and engineering. In many cases, particularly when discretizing continuous problems, the system is large and the associated matrix A is sparse. Furthermore, for many applications, the matrix is symmetric; sometimes, such as in some finite-element applications, A is positive definite, while in other cases, including constrained optimization and problems involving conservation laws, it is indefinite.

A direct method for solving a sparse linear system $Ax = b$ involves the explicit factorization of the system matrix A (or, more usually, a permutation of A) into the product of lower and upper triangular matrices L and U. In the symmetric case, for positive definite problems $U = L^T$ (Cholesky factorization) or, more generally, $U = DL^T$, where D is a block diagonal matrix with 1×1 and 2×2 blocks. Forward elimination followed by backward substitution completes the solution process for each given right-hand side b. Direct methods are important because of their generality and robustness. Indeed, for the 'tough' linear systems arising from some applications, they are currently the only feasible solution methods. In many other cases, direct methods are often the method of choice because finding and computing a good preconditioner for an iterative method can be computationally more expensive than using a direct method. Furthermore, direct methods provide an effective means of solving multiple systems with the same A but different right-hand sides b because the factorization needs only to be performed once.

Since the early 1990s, many new algorithms and a number of new software packages that are designed for the efficient solution of sparse symmetric systems have been developed. Because a potential user may be bewildered by such choice, our intention in this paper is to compare the alternatives on a significant set of large test examples from many different application areas, and, as far as is possible, to make recommendations concerning the efficacy of the various algorithms and packages. This study is an extension of a recent comparison by Gould and Scott (2004) of sparse symmetric direct solvers in the mathematical software library HSL (HSL, 2004). This earlier study concluded that the best general-purpose HSL package for solving sparse symmetric systems is currently `MA57` (Duff, 2004). Thus the only HSL direct solver included here is `MA57`, but the reader should be aware that, for some classes of problems, other HSL codes may be more appropriate. For full details and results for the HSL symmetric solvers, the reader is referred to Gould and Scott (2003).

For ease of reference, all the sparse solvers used in this study are listed in Table 1.1. The release date of the version of the code used in our experiments is given. As far as we are aware, in each case, the most up-to-date version has been used. The codes will be discussed in more detail in Sections 2 and 3. We remark that a number of the packages offer versions for complex symmetric and/or Hermitian matrices, and some can be used for unsymmetric systems. Our experiments are limited to real symmetric matrices. Some of the packages have parallel versions (and may even have been written primarily as parallel codes); this study considers only serial codes and serial versions of parallel solvers.

Code	Date/version	Language	Authors
BCSLIB-EXT	11.2001, v4.1	F77	The Boeing Company
MA57	02.2005, v3.0.0	F77/F90	I.S. Duff, HSL
MUMPS	11.2003, v4.3.2	F90	P.R. Amestoy, I.S. Duff, J.-Y. L'Excellent, J. Koster, A. Guermouche and S. Parlet
Oblio	12.2003, v0.7	C++	F. Dobrian and A. Pothen
PARDISO	04.2005, v1.2.3	F77 & C	O. Schenk and K. Gärtner
SPOOLES	1999, v2.2	C	C. Ashcraft and R. Grimes
SPRSBLKLLT	1997, v0.5	F77	E.G. Ng and B.W. Peyton
TAUCS	08.2003, v2.2	C	S. Toledo
UMFPACK	04.2003, v4.1	C	T. Davis
WSMP	2004, V 4.4.30	F90 & C	A. Gupta and M. Joshi, IBM

Table 1.1: Solvers used in our numerical experiments. A '&' indicates both languages are used in the source code; 'F77/F90' indicates there is a F77 version and a F90 version.

Some of the solvers are freely available to academics while to use others it is necessary to purchase a licence. This information is provided in Table 1.2. For each code a webpage address is also given (or, if no webpage is currently available, an email contact is provided that may be used to obtain further information). Note that for non academic users, the conditions for obtaining and using a solver varies between the different packages and we advise interested users to refer to the webpage or contact the code's authors directly for full details.

2 An introduction to sparse symmetric solvers

Sparse direct methods solve systems of linear equations by factorizing the coefficient matrix A, generally employing graph models to try and minimize both the storage needed and work performed. Sparse direct solvers have a number of distinct phases. Although the exact subdivision depends on the algorithm and software being used, a common subdivision is given by:

1. An ordering phase that exploits structure.
2. An analyse phase (which is sometimes referred to as the symbolic factorization step) that analyses the matrix structure to (optionally) determine a pivot sequence and data structures for efficient factorization. A good pivot sequence significantly reduces both memory requirements and the number of floating-point operations required.

Code	Free to academics	Webpage / email contact
`BCSLIB-EXT`	×	`www.boeing.com/phantom/BCSLIB-EXT/index.html`
`MA57`	×	`www.cse.clrc.ac.uk/nag/hsl`
`MUMPS`	√	`www.enseeiht.fr/lima/apo/MUMPS/`
`Oblio`	√	`dobrian@cs.odu.edu` or `pothen@cs.odu.edu`
`PARDISO`	√	`www.computational.unibas.ch/cs/scicomp/software/pardiso`
`SPOOLES`	√	`www.netlib.org/linalg/spooles/spooles.2.2.html`
`SPRSBLKLLT`	√	`EGNg@lbl.gov`
`TAUCS`	√	`www.cs.tau.ac.il/∼stoledo/taucs/`
`UMFPACK`	√	`www.cise.ufl.edu/research/sparse/umfpack/`
`WSMP`	√	`www-users.cs.umn.edu/∼agupta/wsmp.html`

Table 1.2: Availability and contact details of the solvers used in our numerical experiments.

3. A factorization phase that uses the pivot sequence to factorize the matrix (some codes scale the matrix prior to the factorization).

4. A solve phase that performs forward elimination followed by back substitution using the stored factors. The solve phase may include iterative refinement.

Of the different phases, in a serial implementation, the factorization is usually the most time-consuming, while the solve phase is generally significantly faster. In many software packages, the first two phases are combined into a single user-callable subprogram. An introduction to sparse direct solvers is given in the book by Duff, Erisman and Reid (1986). Another useful reference for symmetric positive definite systems is George and Liu (1981).

2.1 Ordering choices

There are a number of different approaches to the problem of obtaining a good pivot sequence. An important class of ordering methods is based upon the minimum degree (MD) algorithm, first proposed by Tinney and Walker (1967). Variants include the multiple minimum degree (MMD) algorithm (Liu 1985) and the approximate minimum degree (AMD) algorithm (Amestoy, Davis and Duff, 1996, 2004). Other methods are based on nested dissection (ND), a term introduced by George (1973). Many of the recent packages include an explicit interface to the multilevel nested-dissection routine `METIS_NodeND` (or a variant of it) from the METIS package of Karypis and Kumar (1998, 1999). Other orderings include multisection (Ashcraft and Liu, 1998) and orderings based on local minimum fill (Tinney and Walker, 1967). The ordering options offered by the codes in this study are summarized in Table 2.1. An entry marked with * indicates the default (or recommended) ordering. Note that for MUMPS, the default is dependent on the size of the linear system and the packages SPOOLES and WSMP perform two orderings by default and select the better. By default, the latest version of MA57 automatically chooses whether to use AMD or METIS depending on the order of the system and the characteristics of the sparsity pattern; in some cases it will perform both orderings and use the one with the smallest predicted level of fill. A '√'in the column headed 'User' indicates the user may supply his or her own ordering. We note that if the user wishes to specify the ordering for the package SPRSBLKLLT, this can only be done if the matrix is preordered before entry; the other packages perform any necessary permutations on the input matrix using the supplied ordering. Packages that offer orderings that are not included elsewhere in the table have a '√' in the 'Other' column; further details of these are given in Section 3.

Code	Ordering options								Factorization
	MD	AMD	MMD	ND	METIS	MS	MF	User	Algorithm
BCSLIB-EXT	×	×	√	×	√*	×	×	√	Multifrontal
MA57	√	√*	×	×	√*	×	×	√	Multifrontal
MUMPS	√	√*	×	×	√*	√	√	√	Multifrontal
Oblio	×	×	√	×	√*	×	×	√	Left-looking, right-looking, multifrontal
PARDISO	×	×	√	×	√*	×	×	√	Left-right looking
SPOOLES	×	×	√	√*	×	√*	×	√	Left-looking
SPRSBLKLLT	×	×	√*	×	×	×	×	√	Left-looking
TAUCS	√	√	√	×	√*	×	×	√	Left-looking, multifrontal
UMFPACK	×	√*	×	×	×	×	×	×	Unsymmetric multifrontal
WSMP	×	×	×	√*	×	×	√*	√	Multifrontal

Table 2.1: Ordering options and factorization algorithm for the solvers used in our numerical experiments. MD = minimum degree; AMD = approximate minimum degree, MMD = multiple minimum degree; ND = nested dissection; METIS = explicit interface to METIS_NodeND (or variant of it); MS = multisection; MF = minimum fill). * indicates the default.

2.2 Factorization algorithms

Following the analyse phase, the factorization can be performed in many different ways, depending on the order in which matrix entries are accessed and/or updated. Possible variants include left-looking, right-looking, and multifrontal algorithms. The (supernodal) right-looking variant computes a (block) row and column at each step and uses them to immediately update all rows and columns in the part of the matrix that has not yet been factored. In the (supernodal) left-looking variant, the updates are not applied immediately; instead, before a (block) column k is eliminated, all updates from previous columns of L are applied together to the (block) column k of A. Hybrid left-right looking algorithms have also been proposed (see Schenk, Gärtner and Fichtner, 2000). The multifrontal method was first introduced by Duff and Reid (1983). It accumulates the updates; they are propagated from a descendant column j to an ancestor column k via all intermediate nodes on the elimination tree path from j to k. Further details of these variants may be found, for example, in the survey paper of Heath, Ng and Peyton (1991) and the book by Dongarra, Duff, Sorsensen and van der Vorst (1998). A useful overview of the multifrontal method is given by Liu (1992). The algorithm used by each of the codes involved in our tests is given in Table 2.1. Note that a number of the solvers (in particular, `Oblio` and `TAUCS`) offer more than one algorithm.

2.3 Pivoting for stability

For symmetric matrices that are positive definite, the pivot sequence may be chosen using the sparsity pattern alone, and so the analyse phase involves no computation on real numbers and the factorization phase can use the chosen sequence without modification. Moreover, the data structures are determined by the analyse phase and can be static throughout the factorization phase. For symmetric indefinite problems, using the pivot sequence from the analyse phase may be unstable or impossible because of (near) zero diagonal pivots. The disadvantage of using standard partial pivoting for stability is that symmetry is destroyed. Different codes try to address this problem in different ways. The simplest approach is to terminate the computation if a zero (or very small) pivot is encountered. Alternatively, the computation may be continued by perturbing near zero pivots. This allows the data structures chosen by the analyse phase to be used but may lead to large growth in the entries of the factors. The hope is that accuracy can be restored through the use of iterative refinement but, with no numerical pivoting, these simple static approaches are only suitable for a restricted set of indefinite problems.

A larger set of problems may be solved by selecting only numerically stable 1×1 pivots from the diagonal, that is, a pivot on the diagonal is only chosen if its magnitude is at least u times the largest entry in absolute value in its column, where $0 < u \leq 1$ is a threshold parameter set by the user. Potentially unstable pivots (those that do not satisfy the threshold test) will be delayed, and the data structures chosen during the analyse phase may have to be modified. This approach is used by the symmetric version of `MUMPS`. Success is still not guaranteed because if all the remaining (uneliminated) diagonal entries are zero the computation cannot continue.

To preserve symmetry and maintain stability, pivots may be generalised to 2×2 blocks. Again, different packages use different 2×2 pivoting strategies. The approach of `PARDSIO` is to use Bunch-Kaufmann pivoting (Bunch and Kaufmann, 1977) on the dense diagonal blocks that correspond to supernodes and, if a zero (or nearly zero) pivot occurs, it is perturbed. Since pivots are only chosen from within the supernodal block, numerical stability is not guaranteed but because there is no searching or dynamic reordering during the factorization, it is anticipated that this static pivoting strategy will have a substantial performance advantage over more robust approaches that search for a stable pivot and force the delay of any that are unstable. The stable approach is followed by MA57, which uses a modified version of the algorithm of Bunch, Kaufmann and Parlett (1976); details are given in Duff (2004). A threshold parameter $u \in (0, 0.5]$ must be selected. Values close to zero will generally result in a faster factorization with fewer entries in the factors but values close to 0.5 are likely to result in a more stable factorization. Oblio follows a similar approach to MA57. BCSLIB-EXT also uses 1×1 and 2×2 block pivots, again with a threshold parameter under the user's control. BCSLIB-EXT gives preference to 2×2 pivots; the algorithm is described in Ashcraft, Grimes and Lewis (1998). This paper also proposed using $k \times k$ block pivots to improve performance but none of the solvers in our study currently employs pivot blocks with $k > 2$. For the software developer, the main disadvantage of including full 2×2 pivoting is that it adds significantly to the complexity of the code (particularly in a parallel implementation).

We note that, when solving indefinite problems, all the codes used in this study by default select a tentative pivot sequence based upon the sparsity pattern alone (although PARDISO optionally uses the numerical values). Then during the factorization they either return an error message if the sequence cannot be used or modify it to allow the factorization to continue. An alternative approach is to work on the actual numbers and to combine the analyse and factorization phases. In such cases, the code is sometimes described as an analyse-factorize code. The software library HSL includes the analyse-factorize code MA67, which is primarily designed for the solution of symmetric indefinite problems. The results of our earlier study (Gould and Scott, 2004) found that overall MA67 was slower than the multifrontal code MA57, but MA67 was successful at solving some "tough" (highly ill-conditioned and singular) indefinite problems that MA57 struggled on. However, it is common to encounter the need to factorize and solve a sequence of sparse linear systems where the coefficient matrices change but their sparsity pattern remains fixed. A key advantage of designing a solver with separate analyse and factorize phases is that the work of choosing a pivot sequence does not have to be repeated.

The pivoting strategies offered by the codes used in this study are summarised in Table 2.2. Further details are given in Section 3. Although each of the codes may be used to solve positive-definite problems, some have an option that allows the user to indicate that the matrix is positive definite and, in this case, the code follows a logically simpler path. A '$\surd$' in the column headed 'Positive definite' indicates that the code either has such an option or is designed principally for positive-definite systems. A '×' in the 'Indefinite' column indicates that the documentation available with the code states it is designed for solving positive definite problems and is thus not intended for indefinite examples. In our numerical experiments, the latter codes will only be used to solve the positive definite problems.

Code	Positive definite	Indefinite
BCSLIB-EXT	√	Numerical pivoting with 1 × 1 and 2 × 2 pivots.
MA57	√	Numerical pivoting with 1 × 1 and 2 × 2 pivots.
MUMPS	√	Numerical pivoting with 1 × 1 pivots.
Oblio	√	Numerical pivoting with 1 × 1 and 2 × 2 pivots.
PARDISO	√	Supernode Bunch-Kaufmann within diagonal blocks.
SPOOLES	√	Fast Bunch-Parlett.
SPRSBLKLLT	√	×
TAUCS	√	$\times^*$
UMFPACK	×	Partial pivoting with preference for diagonal pivots.
WSMP	√	No pivoting.

Table 2.2: Default pivoting strategies offered by the solvers used in our numerical experiments. ∗ indicates numerical pivoting is to be included in a future release.

2.4 Out-of-core working

To solve very large problems using a direct solver it is usually necessary to work out-of-core. By holding the matrix and/or its factor in files, the amount of main memory required by the solver can be substantially reduced. In this study, only the solvers BCSLIB-EXT, Oblio, and TAUCS include an option for holding the matrix factor out-of-core. Oblio also allows the stack used in the multifrontal algorithm to be held in a file. BCSLIB-EXT is the most flexible. It offers the option of holding the matrix data and/or the stack in direct access files and, if a front is too large to reside in memory it is temporarily held in a direct access file. In addition, information from the ordering and analyse phases may be held in sequential access files. We anticipate that the facility for out-of-core working and out-of-core storage of the matrix factor will allow the solution of problems that are too large for the other codes to successfully solve with the memory available in our test environment. The penalty of out-of-core working is possibly slower factorize and solve times because of I/O overheads.

2.5 Other key features

We conclude this section by briefly highlighting some of the other key features of sparse direct algorithms that are offered by some or all of the solvers in this study. All the codes employ supernodal techniques that enable dense linear algebra routines to be used to improve efficiency of the factorization phase. All the packages except SPOOLES use high level Basic Linear Algebra Subprograms (BLAS) (Dongarra, DuCroz, Duff and Hammarling, 1990) and a number also employ LAPACK routines. Once the factors have been computed, they may be used to solve repeatedly for different right-hand sides b. Some codes offer the option of solving for more than one right-hand side at once because this enables them to take advantage of Level 3 BLAS in the solve phase (see Table 2.3).

A number of codes offer options for automatically scaling the matrix and/or automatically performing iterative refinement to improve the quality of the computed solution and to help assess its accuracy (again, see Table 2.3).

Code	Element	Scaling	Out-of	Iterative	Multiple	Complex	Hermitian
	entry		-core	refinement	rhs	symmetric	
`BCSLIB-EXT`	×	×	√	×	√	√	√
`MA57`	×	√	×	√	√	√	×
`MUMPS`	√	√	×	√	$\times^*$	√	×
`Oblio`	×	×	√	√	√	√	×
`PARDISO`	×	×	×	√	√	√	√
`SPOOLES`	×	×	×	×	√	√	√
`SPRSBLKLLT`	×	×	×	×	√	×	×
`TAUCS`	×	×	√	×	×	√	√
`UMFPACK`	×	√	×	√	×	√	√
`WSMP`	×	√	×	√	√	√	√

Table 2.3: Summary of other key features of the sparse direct solvers used in this study. $*$ indicates to be included in a future release.

When solving problems that arise from finite-element applications, it is often convenient not to assemble the matrix A but to hold the matrix as a sum of element matrices. The only code that allows A to be input in element form is `MUMPS`, although a number of packages (in particular, `BCSLIB-EXT`) offer the user more than one input format for the (assembled) matrix A.

A summary of the key features of the solvers in this study that have not already been included in earlier tables is given in Table 2.3.

3 Sparse symmetric solvers used in this study

In this section, we give a very brief description of the software packages listed in Table 1.1. We highlight some of the main features, with particular reference to the above introductory discussion.

3.1 `BCSLIB-EXT`

`BCSLIB-EXT` is a library of mathematical software modules for solving large sparse linear systems and large sparse eigenvalue problems. It includes multifrontal solvers that are designed both for positive definite and indefinite symmetric systems.

When factorizing indefinite problems, the sequence may be modified and both 1×1 and 2×2 block pivots are used with a user-controlled threshold parameter u (with default value 0.01). Modifying the pivot sequence may cause additional fill-in in the matrix factor beyond that predicted by the analyse phase. `BCSLIB-EXT` allows the user to set a parameter that will cause the factorization to abort if this fill-in exceeds a prescribed level. By default, the factorization also terminates if a zero pivot is encountered. Alternatively, a parameter may be set to allow the package to perturb a (nearly) zero pivot and continue the factorization. The user can also request that the computation terminates immediately a negative pivot is found. The size of the blocks used by the Level 3 BLAS routine `GEMM` during the factorization is controlled by parameters that may be reset by the user.

As already mentioned, a key feature of `BCSLIB-EXT` is its use of files to reduce the amount of main memory required by the package. The user can choose to hold the original matrix and/or the matrix factors in files. If there is not enough memory to hold the multifrontal stack and the current frontal matrix, the code will store the stack out-of-core. It will also perform an out-of-core frontal assembly and factorization step if the current frontal matrix does not fit in memory. The user can choose a minimum core processing option that forces out-of-core storage. In our tests, we provide the amount of storage recommended in the documentation and provide positive stream numbers for each of the files used by the code. In this case, if the amount of main storage we have provided is insufficient, the code will use sequential files for holding information from the ordering and analyse phases and may use one or more files during the factorization (and solve) phase.

3.2 `MA57`

`MA57` is part of the mathematical software library HSL (2004) and was designed by Duff (2004) to supersede the earlier well-known HSL multifrontal code `MA27` (Duff and Reid, 1983) for the solution of symmetric indefinite systems. Our earlier study (Gould and Scott, 2004) compared the performance of version 1.0.0 using an AMD ordering with that of the nested dissection ordering from the METIS package. Our findings were that for very large, positive-definite test problems (typically those of order $> 50,000$), it is generally advantageous to use the METIS ordering but for small and very sparse problems and also for many indefinite problems, using an AMD ordering with quasi-dense row detection is preferable. Based on our findings and experiments by, amongst others, Duff and Scott (2005), `MA57` has now been modified so that in the latest release (version 3.0.0) the default is for the code to automatically select to use either the AMD ordering with dense row detection or the METIS ordering based on the order of the system and characteristics of the sparsity pattern; for some problems, it computes both orderings and chooses the one with the smallest predicted level of fill.

During the factorization phase, when diagonal 1×1 pivots would be numerically unstable, 2×2 diagonal blocks are used. Note that, for a given threshold parameter u, the test for stability of 2×2 pivots is less severe than the test used in the earlier `MA27` code (details are given by Duff, 2004). If the problem is known to be positive definite, the user can set a control parameter that switches off threshold pivoting. In this case, if a sign change or a zero is detected among the pivots, an error exit will optionally occur. Alternatively, the user can choose to terminate the computation if any pivot is found to be of modulus less than a user-defined value.

Parameters that are under the user's control determine the size of the blocks used by the Level 3 BLAS routines during the factorization and solve phases. The (optional) iterative refinement is based on the strategy of Arioli, Demmel and Duff (1989). Estimates of the error are also optionally computed.

We have already observed that the default ordering in the version of the code used in this study differs from the one used in our earlier study (Gould and Scott, 2004). Since the earlier study there have been a number of other key changes to MA57. In particular, by default the new version scales the matrix using a symmetrized version of the HSL code MC64 (Duff and Koster, 1999). The aim is to put large entries in the diagonal blocks so as to restrict the number of pivots that are rejected for stability reasons during the factorization; details are given in Duff and Pralet (2004). The matrix is explicitly scaled internally to the package as are the right-hand side and the solution so that the user need not be concerned with this. Iterative refinement, if requested, is based on the original unscaled matrix. Static pivoting is now an option so that the factorization can be performed using the storage predicted by the analysis phase even if the matrix is not positive definite. Because static pivoting is not the default strategy, it is not used in our tests.

There is little difference between the speed of the Fortran 90 version of MA57 and the Fortran 77 version, because the former is essentially a Fortran 90 encapsulation of the latter. However, the Fortran 90 version does offer some additional facilities, and the user interface is simplified through the use of dynamic storage allocation. In our numerical experiments, the Fortran 77 version is used.

3.3 MUMPS

The MUMPS (MUltifrontal Massively Parallel Solver) package is designed and developed by Amestoy, Duff, L'Excellent and Koster (2001) (see also Amestoy, Duff and L'Excellent, 2000). It is a multifrontal code primarily intended for unsymmetric systems and for symmetric positive definite systems but it can also be used to solve many indefinite problems. Both C and Fortran 90 interfaces to MUMPS are available; in our numerical experiments, the Fortran 90 interface is used. MUMPS has been developed primarily as a parallel solver (originally targeted at distributed memory computers); in this study we use the sequential version.

MUMPS offers the user a wide range of options for choosing the pivot sequence (see Table 2.1). These include a version of AMD with automatic quasi-dense row detection (Amestoy, 1997) and an approximate minimum fill-in algorithm. The multisection ordering is implemented using the code PORD of Schulze (2001) and nested dissection using one of the graph partitioning packages METIS or SCOTCH (www.labri.fr/Perso/∼pelegrin/scotch/). By default, MUMPS automatically chooses the ordering algorithm depending on the packages installed, the size of the matrix, and the number of processors available. On a single processor, AMD is used for problems of size $n < 10^4$ and METIS_NodeND for larger problems.

As discussed in Section 2.3, when factorizing indefinite problems, MUMPS uses 1×1 pivots chosen from the diagonal. The factorization terminates with an error if at any stage no numerically stable pivots are available on the diagonal. Because this may mean some problems are not solved, at the authors' suggestion, we run both the symmetric and unsymmetric versions of MUMPS when testing indefinite examples.

Other features of the MUMPS package include facilities for use in domain decomposition, error analysis, optional iterative refinement using the approach of Arioli et al. (1989), and estimation of rank deficiency.

3.4 `Oblio`

`Oblio` is a sparse symmetric direct solver library developed by Dobrian and Pothen as an experimental tool (Dobrian, Kumfert and Pothen, 2000). Their goal was to create a "laboratory for quickly prototyping new algorithmic innovations, and to provide efficient software on serial and parallel platforms". The code is written in C++ using object-oriented techniques and is still being actively developed.

The most recent version (0.7) is able to solve both positive definite and indefinite systems. For indefinite problems, the user is offered so-called static LDL^T or dynamic LDL^T. In the former case, if a small pivot is encountered, it is perturbed to a value under the user's control, allowing the computation to continue. The default (which we use in our tests) is dynamic LDL^T. This employs a combination of 1×1 and 2×2 pivots. When a diagonal 1×1 pivot would be unstable, a search is made for a suitable 2×2 pivot. Thus searches for 1×1 and 2×2 pivots are interlaced.

For flexibility, `Oblio` implements three different sparse factorizations: left-looking, right-looking and multifrontal. For 2-dimensional problems the multifrontal option is recommended but for large 3-dimensional problems the user documentation reports the multifrontal factorization can be outperformed by the other two algorithms. The default algorithm is the multifrontal algorithm and this is used in our tests. The multifrontal version includes an out-of-core option. This allows the matrix factor and/or the stack to be held in files.

3.5 `PARDISO`

The `PARDISO` package of Schenk and Gärtner offers serial and parallel solvers for the direct solution of unsymmetric and symmetric sparse linear systems on shared memory multiprocessors. In this study, only the serial version for symmetric systems is used. `PARDISO` employs a combination of left- and right-looking Level 3 BLAS supernode techniques (Schenk et al., 2000, Schenk and Gärtner, 2004*b*) and is written using a combination of Fortran 77 and C source code. `PARDISO` is included in Intel Math Kernel Library (see `www.intel.com/software/products/mkl/features/dss.htm`).

The default ordering is a modified version of METIS; if the user does not wish to use this ordering, a fill-reducing ordering may be input. The user must set a parameter to indicate whether a Cholesky factorization or an LDL^T factorization is required. For indefinite problems, the current version includes Bunch-Kaufmann pivoting (Bunch and Kaufmann, 1977) applied to the dense diagonal supernode blocks. A modified version of the LAPACK routine `dsytf2` is used for factorizing these blocks. Pivots that are zero (or nearly zero) are perturbed so that pivots are not delayed beyond the current block. The amount by which pivots are perturbed in this static pivoting strategy is determined by a parameter under the user's control. The current version includes an option to perform preprocessing based on symmetric weighted matchings. The documentation states that this is very robust but, because it incurs an overhead and involves the numerical values of entries of the matrix (so that a new analyse may be required if the entries change, even if the sparsity pattern is unaltered), it is not the default and is not used in this study. Iterative refinement is offered, with the maximum number of steps controlled by a parameter set by the user. In our tests the default value of 0 is used. This means that iterative refinement is only used if pivots have been perturbed during the factorization. In this case, two steps of iterative refinement are performed.

We note that when calling PARDISO it is assumed that zero diagonal entries are stored explicitly in the list of matrix entries. For many indefinite examples, one or more of the diagonal entries is often not present within the sparsity pattern and the user must add explicit zeros. PARDSIO also requires that the upper triangular part of the matrix is entered by rows with the entries within each row ordered by increasing column index.

3.6 SPOOLES

SPOOLES is a library for solving sparse real and complex linear systems of equations, and may be used for both symmetric and unsymmetric problems. The package is written in C using an object-oriented design. Both serial and parallel versions are available. The serial version for real symmetric systems is used in our tests.

SPOOLES uses the Crout reduction variant of Gaussian elimination, which is a left-looking algorithm. In addition to MMD and generalized ND, the analyse phase offers a multisection ordering algorithm (Ashcraft and Liu, 1998). The default is to use the better of the nested dissection and multisection methods (although the user reference manual does comment on situations where the user may find it beneficial to select another choice).

To try and ensure stability of the factorization for indefinite problems, the entries of the triangular factor L are bounded by a user-supplied tolerance (in our tests we use the recommended value of 100 for this tolerance). The fast Bunch-Parlett algorithm described by Ashcraft et al. (1998) is used to choose 1×1 or 2×2 pivot blocks. SPOOLES is the only package tested that does not use the high level BLAS kernels; instead it performs operations within the factorization phase using multiple dot products.

We note that SPOOLES requires that the sparsity pattern of the input matrix includes the diagonal. For many indefinite examples, one or more of the diagonal entries is often not present within the sparsity pattern. In such cases, the user must include an explicit zero.

3.7 SPRSBLKLLT

SPRSBLKLLT was developed by Esmond Ng and Barry Peyton at Oak Ridge National Laboratory in the early 1990s for the solution of sparse symmetric positive definite systems. The pivot sequence is selected using the MMD algorithm; the implementation used is taken from the Waterloo sparse matrix package SPARSPAK (see sparse.uwaterloo.ca/∼jageorge/Sparspak/sparspak.html). The symbolic factorization subroutines are independent of any ordering algorithms.

SPRSBLKLLT implements a supernodal left-looking Cholesky factorization algorithm (details are given in Ng and Peyton, 1993). The symbolic factorization algorithm uses the results of Gilbert, Ng and Peyton (1994), which allow storage requirements to be determined in advance, regardless of the ordering strategy used. The performance of the package has been enhanced since it was first released by exploiting the memory hierarchy:

it splits supernodes into sub-blocks that fit into the available cache; and it unrolls the outer loop of matrix-vector products in order to make better use of available registers. A parameter that must be set by the user determines the maximum supernode size. The storage requirements depend on this parameter (large values increase the storage). Based on the limited documentation provided with the code, in our tests this parameter is set to 100.

3.8 TAUCS

TAUCS has been developed since 2001 by Sivan Toledo's research group in the Department of Computer Science at Tel-Aviv University as a platform for research on sparse linear solvers. TAUCS is designed to support the development of research codes by providing a library of fundamental algorithms and services, and to facilitate the maintenance and distribution of the resulting research codes. Toledo and his colleagues are still developing the package; a version for indefinite problems will be available in the future. TAUCS is currently used in Mathematica 5.

Both a multifrontal algorithm and a left-looking algorithm are implemented; the documentation states the latter is slower than the former but requires less memory. As well as MD, AMD, MMD, and METIS_NodeND, a no-fill ordering code for matrices whose graphs are trees is available. This is a special case of MD but is faster. METIS is recommended for large problems and was used in our tests. The current version of TAUCS is designed for positive definite symmetric problems and so numerical pivoting is not incorporated (although the package does include a general sparse LU factorization code with partial pivoting). An option exists to compute an incomplete LL^T factorization.

TAUCS is able to factorize a matrix whose factor is larger than the main memory by holding the factor out-of-core. The factor is held in multiple files, each at most 1 Gbyte in size (see Rotkin and Toledo, 2004 for details).

3.9 UMFPACK

The principal author of the sparse direct solver UMFPACK is Tim Davis of the University of Florida (Davis, 2003*a*, 2003*b*). The tested version (version 4.1) is written in C; the original code was developed by Davis and Duff in Fortran 77 (Davis and Duff, 1993). It is the only code included in this study that is primarily written for unsymmetric matrices, that is, it is the only code that requires the sparsity pattern of the whole matrix A. However, for symmetrically (or nearly symmetrically) structured matrices it offers a symmetric pivoting strategy and for this reason we have included it in this study. This also serves as a benchmark to illustrate how symmetric solvers compare to a state of the art unsymmetric package.

UMFPACK combines a column ordering strategy with a right-looking unsymmetric-pattern multifrontal numerical factorization. All pivots with zero Markowitz cost are eliminated first and placed in the LU factors. The analyse phase then automatically selects one of three ordering and pivoting strategies (*unsymmetric*, *2-by-2*, and *symmetric*). For symmetric matrices with a zero-free diagonal, the symmetric strategy is used. This computes a column ordering using AMD. No modification of the column ordering is made during the numerical factorization. A nonzero diagonal entry is selected as a suitable pivot if in magnitude it is at least u_1 times the largest entry in its column. Otherwise, an off-diagonal pivot is selected with magnitude at least u_2 times the largest entry in its column. u_1 and u_2 are parameters under the user's control with default values of 0.001 and 0.1, respectively. Thus strong preference is given to pivoting on diagonal entries. For symmetric indefinite problems with zeros on the diagonal, the so-called 2-by-2 strategy is attempted. This looks for a row permutation that puts nonzero entries onto the diagonal. The symmetric strategy is applied to the permuted matrix.

MATLAB, C and Fortran interfaces to the present version (4.1) are offered. An earlier version (4.0) appears as a built-in routine in MATLAB 6.5 and Mathematica 5. Version 2.2.1 by Davis and Duff is available as routine `MA38` within the software library HSL. Versions prior to 4.1 only offer the unsymmetric pivoting strategy and are thus not well suited for matrices with a symmetric nonzero pattern.

3.10 `WSMP`

The Watson Sparse Matrix Package (`WSMP`) was developed by Anshul Gupta of the IBM T. J. Watson Research Center. The package is written using Fortran 90 and C and includes direct solvers for both symmetric and unsymmetric systems. `WSMP` was primarily developed as a highly scalable parallel code that can be used in either a shared-memory multiprocessor or a message-passing environment. A serial version is available and is used in this study.

The analyse phase offers a minimum local fill ordering and an ordering based on recursive bisection. By default, both orderings are computed and the one that will result in the least fill-in is selected. The factorization phase implements a modified multifrontal algorithm. It is primarily designed for positive-definite systems but may be used for solving quasi-definite (Vanderbei, 1995) and indefinite problems provided the numerical factorization is stable irrespective of the pivot sequence. By default (and in our tests), if a zero (or very small) pivot is encountered, the factorization terminates with an error message. Alternatively, the user can request that near zero pivots are perturbed to allow the factorization to continue. The user must set a parameter to indicate whether an LL^T or LDL^T factorization is to be performed. A routine may be called to perform iterative refinement, with an option of using extended precision arithmetic.

`WSMP` requires that the sparsity pattern of the input matrix includes the diagonal. If one or more of the diagonal entries is not present, the user must add an explicit zero.

Further details of `WSMP` are given in Gupta, Karypis and Kumar (1997) and Gupta, Joshi and Kumar (2001). Currently, `WSMP` is available for use on AIX, SunOS, Tru64, HP-UX, and Linux platforms. Although `WSMP` libraries contain multithreaded code, the libraries are not thread-safe.

4 The test environment

4.1 The test set

Our aim in this study is to test the solvers on a wide range of test problems from as many different application areas as possible. In collecting test data we imposed only two conditions:

- The matrix must be of order greater than $10,000$.

- The data must be available to other users.

The first condition was imposed because our interest in this study is in large problems. The second condition was to ensure that our tests could be repeated by other users and, furthermore, it enables other software developers to test their codes on the same set of examples and thus to make comparisons with other solvers. Provided the above conditions are satisfied, we have included all square real symmetric matrices of order exceeding $10,000$ that were available in June 2003 in the Matrix Market (math.nist.gov/MatrixMarket/), the Harwell-Boeing Collection (Duff, Grimes and Lewis, 1989), and the University of Florida Sparse Matrix Collection (www.cise.ufl.edu/research/sparse/matrices), as well as a number of problems that were supplied to us by colleagues. The test set comprises 88 positive-definite problems and 61 numerically indefinite problems. We note that some of the indefinite problems are highly ill-conditioned and 5 are structurally singular. Of these matrices, those of order 50,000 or more are further classed as being in the *subset* of larger examples (there are 43 positive-definite and 30 indefinite examples in this category). Any matrix for which we only have the sparsity pattern available is included in the positive-definite set, and appropriate numerical values have been generated (see Section 4.6). Application areas represented by our test set include linear programming, structural engineering, computational fluid dynamics, acoustics, and financial modelling. A full list of the test problems together with a brief description of each is given in Gould, Hu and Scott (2005). The problems are all available from ftp://ftp.numerical.rl.ac.uk/pub/matrices/symmetric (and are also now part of the University of Florida Sparse Matrix Collection).

4.2 The performance profile

Benchmark results are generated by running a solver on a set $\mathcal{T}$ of problems and recording information of interest such as the computing time and memory used. In this study, we use a performance profile as a means to evaluate and compare the performance of the solvers on our test set $\mathcal{T}$.

Let $\mathcal{S}$ represent the set of solvers that we wish to compare. Suppose that a given solver $i \in \mathcal{S}$ reports a statistic $s_{ij} \geq 0$ when run on example j from the test set $\mathcal{T}$, and that the smaller this statistic the better the solver is considered to be. For example, s_{ij} might be the CPU time required to solve problem j using solver i. For all problems $j \in \mathcal{T}$, we want to compare the performance of solver i with the performance of the best solver in the set $\mathcal{S}$.

For $j \in \mathcal{T}$, let $\hat{s}_j = \min\{s_{ij}; i \in \mathcal{S}\}$. Then for $\alpha \geq 1$ and each $i \in \mathcal{S}$ we define

$$k(s_{ij}, \hat{s}_j, \alpha) = \begin{cases} 1 & \text{if } s_{ij} \leq \alpha \hat{s}_j \\ 0 & \text{otherwise.} \end{cases}$$

The *performance profile* (see Dolan and Moré, 2002) of solver i is then given by the function

$$p_i(\alpha) = \frac{\sum_{j \in \mathcal{T}} k(s_{ij}, \hat{s}_j, \alpha)}{|\mathcal{T}|}, \quad \alpha \geq 1.$$

Thus $p_i(1)$ gives the fraction of the examples for which solver i is the most effective (according to the statistic s_{ij}), $p_i(2)$ gives the fraction for which it is within a factor of 2 of the best, and $\lim_{\alpha \longrightarrow \infty} p_i(\alpha)$ gives the fraction for which the algorithm succeeded.

In this study, the statistics used are:

- The CPU times required to perform the analyse, factorize, and solve phases.
- The number of nonzero entries in the matrix factor.
- The total memory used by the solver.

4.3 Computing platform

The numerical results were all obtained on a Compaq DS20 Alpha server with a pair of EV6 CPUs; in our experiments only a single processor with 3.6 Gbytes of RAM was used. We compiled the codes with full optimisation; the vendor-supplied BLAS were used where applicable. All CPU reported times are in seconds and, where appropriate, include all I/O costs involved in holding the factors in direct-access files. A CPU limit of 30 minutes was imposed for each code on each problem; any code that had not completed after this time was recorded as having failed.

In all the experiments, double precision reals were used. Thus storage for a real was 8 bytes and for an integer was 4 bytes. Memory is measured using the C utility function `getrusage`. In particular, the maximum resident set size of the current process is measured. Extra memory required for setting up the test is subtracted.

4.4 Control parameters

Each of the sparse solvers used in our numerical experiments has a number of parameters that control the action. These are either assigned default values through a call to an initialisation subroutine or the values recommended in the user documentation are used. Unless otherwise stated, we use these defaults in each case, even if different codes sometimes choose a different value for essentially the same parameter. The main exception is the stability threshold parameter u (see Section 2.3). We remark that we decided early on in this study not to try and fine tune the input parameters for each solver on each problem. In some cases, performance would have been improved by tweaking. However, we felt that to do this for each individual code and all the problems in the test set would be an almost impossible task and, more importantly, our aim is to compare the codes from a common standpoint, that is, using the control settings chosen by the authors of the packages. Our experience is that many users (in particular, those who would regard themselves as non-experts) rely on the default settings and are reluctant to try other values (possibly because they do not feel confident about making other choices).

When testing the solvers on positive-definite problems the threshold parameter u is set to zero. This results in no numerical pivoting being performed. For our tests on numerically indefinite problems, for the codes that employ a stability threshold parameter, we run both with the code's default u value and with u set to 10^{-10}. Such a value is frequently used in optimization applications (Saunders, 1994, Gould and Toint, 2002), where speed is of the essence, and any instability is countered either by iterative refinement or ultimately by refactorization with a larger value of u.

MA57, MUMPS and BCSLIB-EXT use a default threshold $u = 0.01$, while UMFPACK has two threshold parameters with default values of 0.001 and 0.1 (see Section 3.9). When testing with a small threshold, both UMFPACK parameters are set to 10^{-10}.

4.5 Out-of-core working

Out-of-core options are offered by the packages BCSLIB-EXT, Oblio and TAUCS. In our tests, the out-of-core facilities are only used if this is the default. For Oblio and TAUCS, the user must decide explicitly if the out-of-core option is required. By default, BCSLIB-EXT switches automatically to out-of-core working if it finds that the user has provided insufficient workspace for the code to run in-core (see Section 3.1). We therefore anticipate that out-of-core working will be used by BCSLIB-EXT for some of our largest test examples.

4.6 Numerical values and scaling

Some of our test examples are not supplied with numerical values (only the sparsity pattern is available). For these cases, appropriate numerical values are generated. Reproducible pseudo-random off-diagonal entries in the range $(0, 1)$ are generated using the HSL routine FA14, while the i-th diagonal entry is set to $\max(100, 10\rho_i)$, where ρ_i is the number of off-diagonal entries in row i of the matrix, thus ensuring that the generated matrix is numerically positive definite.

In all our tests, right-hand side vectors b are computed so that the exact solution x (of the unscaled system) is $x = e \overset{\text{def}}{=} (1, 1, ..., 1)^T$.

If the input matrix has entries differing widely in magnitude, then an inaccurate solution may be obtained in the indefinite case and the accuracy may be difficult to assess in all cases. A number of the packages tested include an option for scaling the input matrix. We do not use these options unless scaling is performed by default (this is the case for MA57 and UMFPACK). To examine the effects of scaling on the codes that do not perform scaling by default, for each value of the threshold parameter u used, we run both with and without scaling of the matrix A and the corresponding right-hand side b using the HSL scaling routine MC30. For our positive-definite problems, scaling was found to make an insignificant difference and hence we report on the effects of scaling only for the indefinite examples.

4.7 Residuals and iterative refinement

A number of the solvers include routines for automatically performing iterative refinement. Unless the solver's default is to perform iterative refinement, we have not used these routines in this study (by default for indefinite problems PARDISO performs up to two steps of iterative refinement). Instead, once we have computed the approximate solution x, we perform one step of iterative refinement by computing the residual $r = Ax - b$ and then recalling the solve routine to solve $A\delta x = r$ for the correction δx.

For each right-hand side b and corresponding solution x, we compute the scaled residual

$$\|b - Ax\|_\infty / (\|A\|_\infty \|x\|_\infty + \|b\|_\infty)$$

A check is made after one step of iterative refinement that this residual is sufficiently small (in our tests, a residual greater than 0.0001 causes an error message to be returned). Note that the residual of the unscaled system is computed.

For nonsingular A, we also check the accuracy of the computed solution. Some of the systems are highly ill-conditioned and for these the norm of the error $x - e$ was large for some solvers. A positive warning flag is set in this case, but we do not count this as a failure provided the scaled residual is small.

5 Results

Since some of the solvers we are examining are specifically designed for positive-definite problems (and may be unreliable, or even fail, on indefinite ones), we will discuss the positive-definite and indefinite cases separately. Moreover, as the competing algorithms have different design goals, we consider it worth examining each of the solution phases (analyse, factorize, solve) both separately and ultimately together.

Full details of the statistics generated by each solver are given in an accompanying technical report (Gould et al., 2005).

5.1 Positive-definite examples

Overall the reliability of the solvers for positive-definite examples was excellent. All failed to solve the problem audikw_1 because of a lack of space required to hold its factors[1], but for the majority this was the only failure. UMFPACK was the solver with the largest number of failures, caused either by the CPU time limit being exceeded or by a lack of space. This is the only solver for which extra precautions must be taken to guarantee stability, because it permits off-diagonal pivoting.

We first present the performance profile for the analyse time for the ten solvers in Figure 5.1.1 . It is immediately apparent that the solvers that use (or select) variants of the minimum-degree strategy (SPRSBLKLLT, UMFPACK and MA57) have a faster ordering than those that employ a dissection-based strategy. Furthermore, there is little to choose between the speed at which the variants of the latter orderings are computed. The most

[1] In the case of BCSLIB-EXT, which permits out-of-core factorization, the run was terminated through excessive CPU time. However, subsequent experiments showed that BCSLIB-EXT was able to solve the problem if sufficient time (roughly 2.5 CPU hours) was allowed.

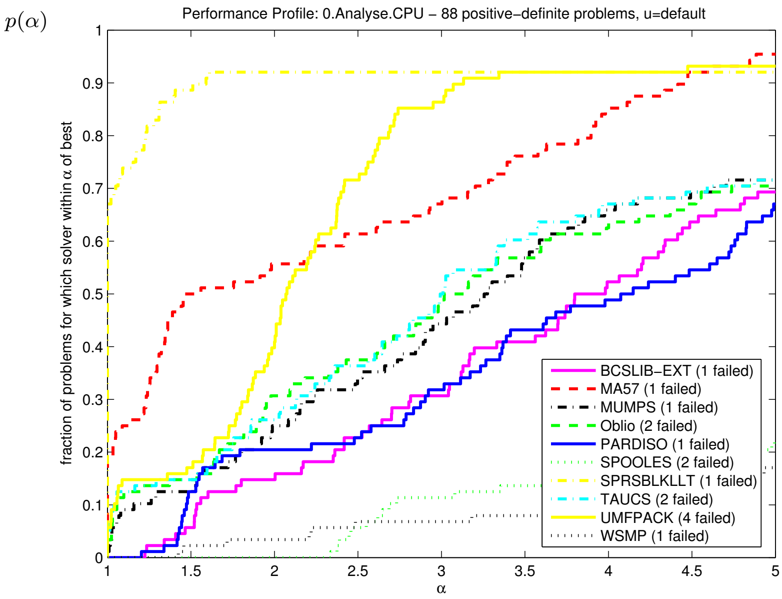

Figure 5.1.1: Performance profile, $p(\alpha)$: CPU time for the analyze phase (positive-definite problems).

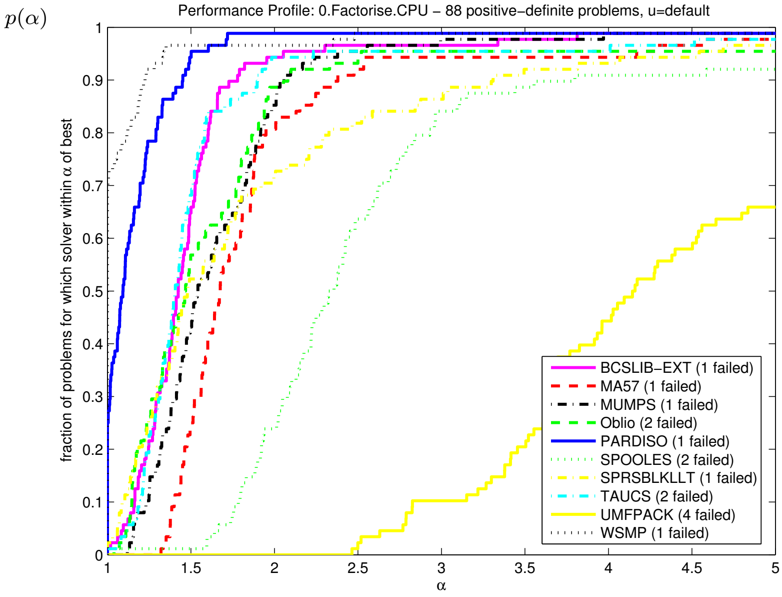

Figure 5.1.2: Performance profile, $p(\alpha)$: CPU time for the factorization phase (positive-definite problems).

expensive strategies are those employed by SPOOLES and WSMP, both of which compute two orderings and the select the best.

When it comes to the factorization, we see in Figure 5.1.2 that the careful analysis strategy adopted by WSMP pays off. Over the complete set of positive definite examples, the other codes (with the exception of UMFPACK and SPOOLES) are broadly comparable. Interestingly, the differences between left/right-looking and multifrontal factorizations do not seem as significant as we had anticipated might be the case. UMFPACK is slower because it is essentially an unsymmetric solver and we believe that SPOOLES is not competitive because it does not use high level BLAS. We also see in Figure 5.1.3 that, in our computing environment, the fastest factorization is generally closely tied to the number of nonzeros in the generated factors—note here that this statistic was not available for BCSLIB-EXT. The unsymmetric solver UMFPACK does not appear to be competitive in the symmetric case, and this agrees with our observations in our earlier paper (Gould and Scott, 2004) concerning the unsymmetric HSL code MA48.

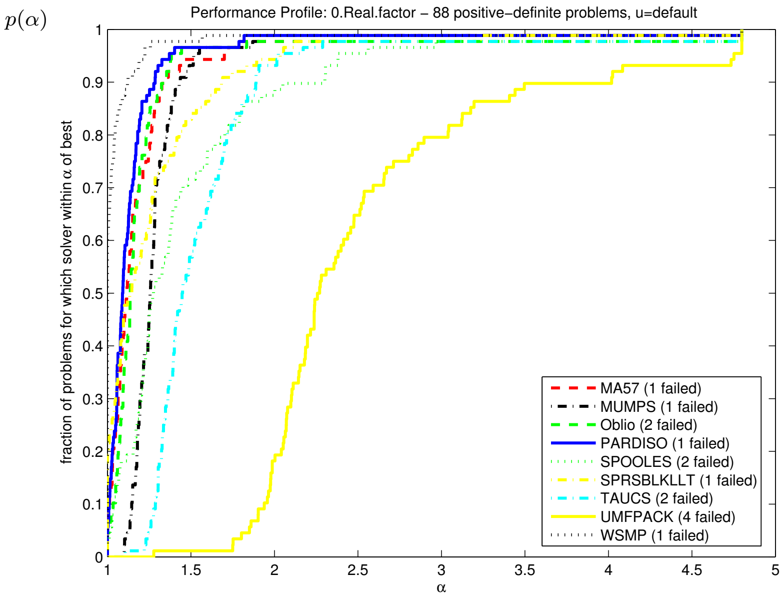

Figure 5.1.3: Performance profile, $p(\alpha)$: Number of entries in the factors (positive-definite problems).

Having computed the factors, the performance profiles for solving for a single right-hand side are illustrated in Figure 5.1.4. Here there is a reasonable correlation between sparsity in the factors and time taken, with PARDISO, BCSLIB-EXT, and MA57 generally the faster codes. The only slight surprise is that, although WSMP produces the sparsest factors, its solve time is solver than most of the other codes.

In Figure 5.1.5 we present the performance profile for the CPU time for a single solution (that is, the CPU time for analysing, factorizing and solving for a single right-hand side) for the ten solvers under consideration. As one might expect, there is little to choose between the best solvers, since most use broadly similar orderings and there is no need for

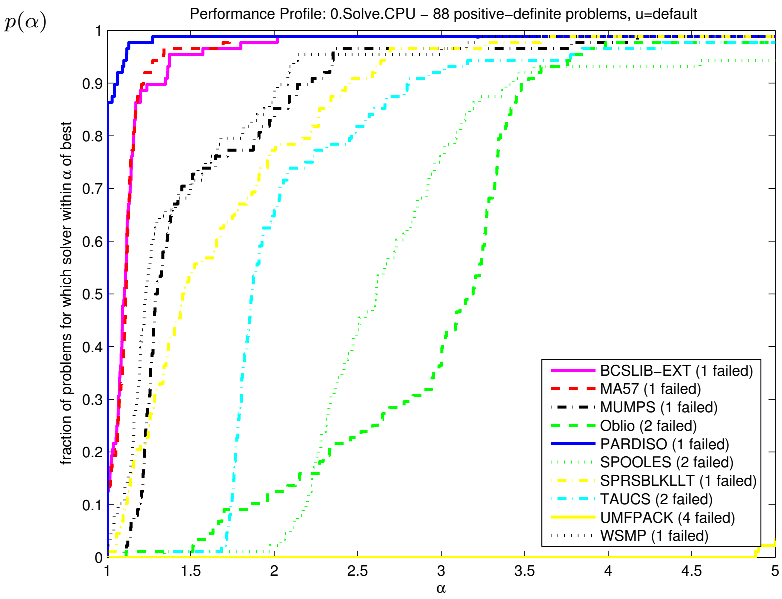

Figure 5.1.4: Performance profile, $p(\alpha)$: CPU time for the solution phase (positive-definite problems).

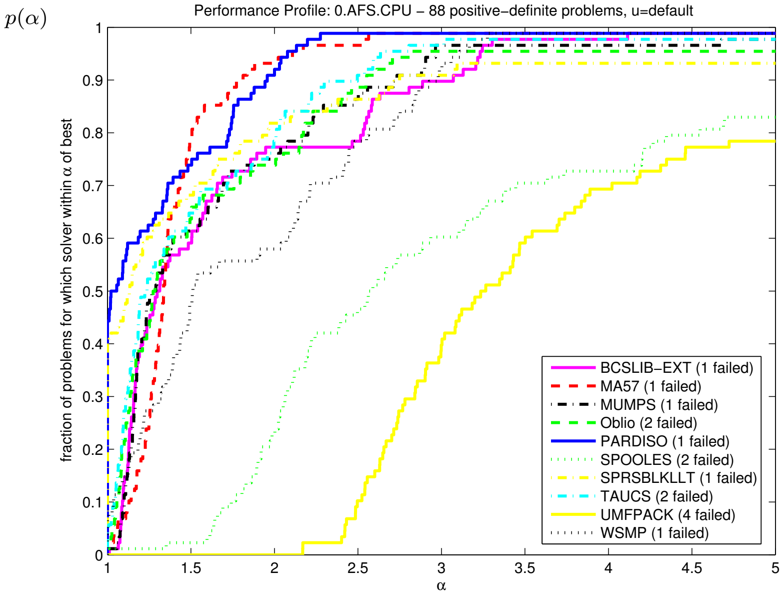

Figure 5.1.5: Performance profile, $p(\alpha)$: CPU time for the complete solution (positive-definite problems).

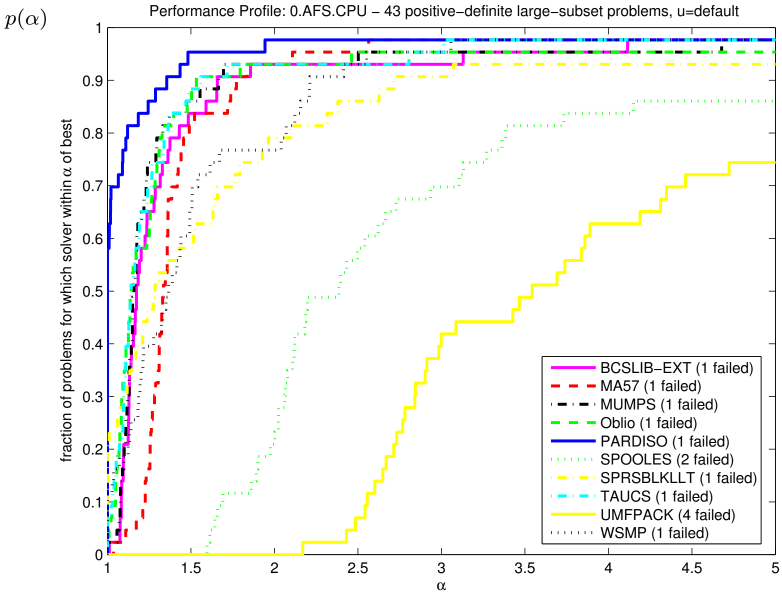

Figure 5.1.6: Performance profile, $p(\alpha)$: CPU time for the complete solution (large positive-definite subset problems).

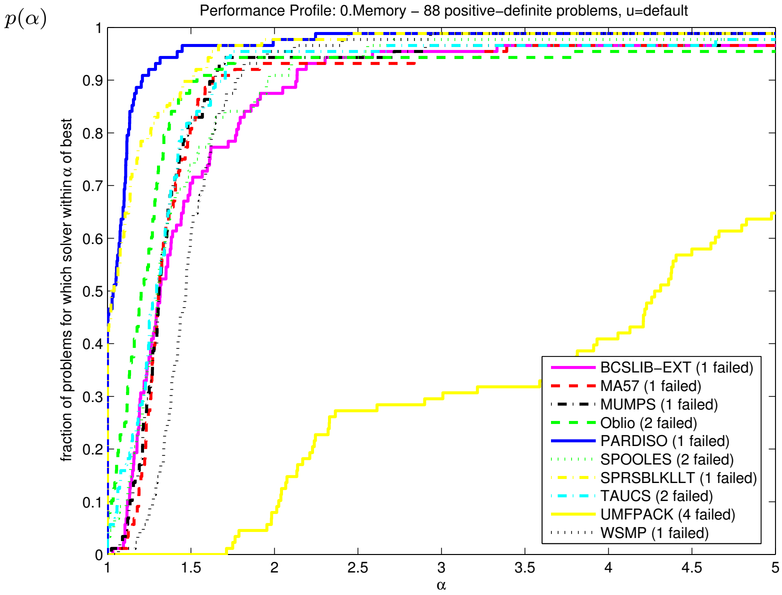

Figure 5.1.7: Performance profile, $p(\alpha)$: Memory used (positive-definite problems).

numerical pivoting. PARDISO and MA57 appear to perform marginally better than the rest, but SPRSBLKLLT, TAUCS, OBLIO, MUMPS, and BCSLIB-EXT are all close. The slow analyse and solve times clearly affect WSMP. Only SPOOLES and UMFPACK do appear uncompetitive. For the subset of larger problems, Figure 5.1.6 reinforces this trend, with all the codes apart from SPOOLES and UMFPACK being within a factor of 2 of the fastest code on approximately 80 per cent of the large test problems.

In Figure 5.1.7 we also compare the total memory used. We had expected that the multifrontal solvers would require significantly more memory than the other codes but our results suggest that there is generally little to distinguish between any of the symmetric solvers from this perspective, with perhaps a slight edge for PARDISO.

5.2 Indefinite examples

We now turn to indefinite problems, for which numerical pivoting is important. We need to assess the effects of the different ordering and pivoting strategies. Note that SPRSBLKLLT and TAUCS were not designed for indefinite problems and thus are omitted from these tests. Moreover, as discussed in Section 2.3, many of the other solvers only offer limited forms of pivoting, and thus give no stability guarantees. At its authors' suggestion, we include results for both the symmetric (here denoted by MUMPS) and unsymmetric (MUMPS_US) versions of MUMPS (note that the unsymmetric version includes off-diagonal pivoting).

Although, in our companion paper (Gould et al., 2005), we report on the results of four different pre-scaling/pivoting strategies, here we largely restrict our attention to the default strategy. The first thing to note is that the general reliability on indefinite problems is far below that for the definite case. Indeed, only three of the solvers (MA57, PARDISO and UMFPACK) had a success rate of 90% or better, while some of the others failed on 25% or more of the problems—admittedly some of the latter issued strong warnings in their documentation about possible limitations (including not being able to factorize singular systems and not performing numerical pivoting). All the solvers failed on the problem SPARSINE because of a lack of space or they exceeded our CPU limit.

We start by presenting in Figure 5.2.1 the performance profile for the analyse times. The conclusions are broadly as for the definite case, with those solvers that use (or select) variants of the minimum-degree strategy being faster than those opting for dissection orderings.

Now examining the factorize times (see Figure 5.2.2), we see a significant gap between PARDISO and the remaining solvers. Recall that PARDISO employs static pivoting and thus does not need to alter the ordering suggested by the analyse phase to complete its factorization. One might anticipate a lack of robustness with such an approach; what we find is that, by using the default iterative refinement, all the problems pass our residual test (see Section 4.7), but for a small number of problems (notably crystk02 and crystk03) the scaled residuals are significantly larger than those obtained using other solvers. Interestingly, the gap in performance is less pronounced when comparing the numbers of entries in the factors (see Figure 5.2.3), with MA57 the runner up—again the statistics for BCSLIB-EXT are not available. As one might predict, Figure 5.2.4 indicates a high correlation between the total memory used and the numbers of nonzeros in the factors (see Figure 5.2.3), with PARDISO requiring the least memory, followed by MA57 well

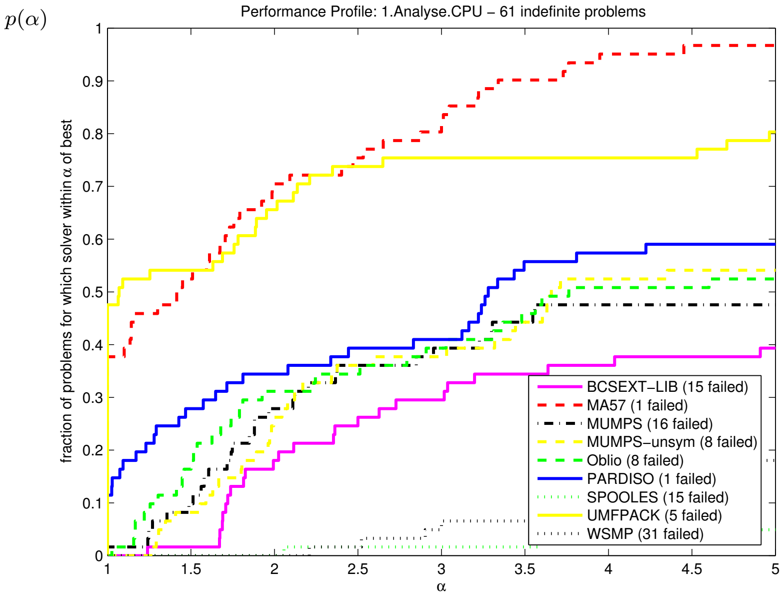

Figure 5.2.1: Performance profile, $p(\alpha)$: CPU time for the analyze phase (indefinite problems).

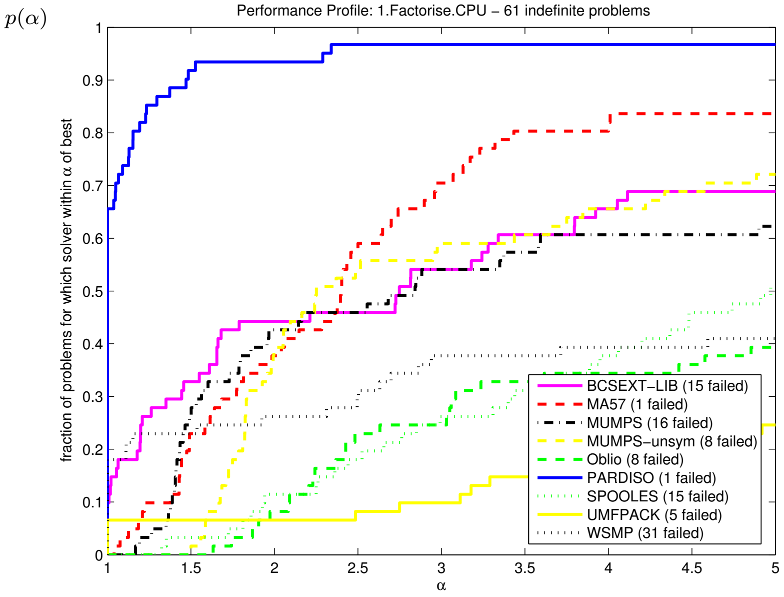

Figure 5.2.2: Performance profile, $p(\alpha)$: CPU time for the factorization phase (indefinite problems).

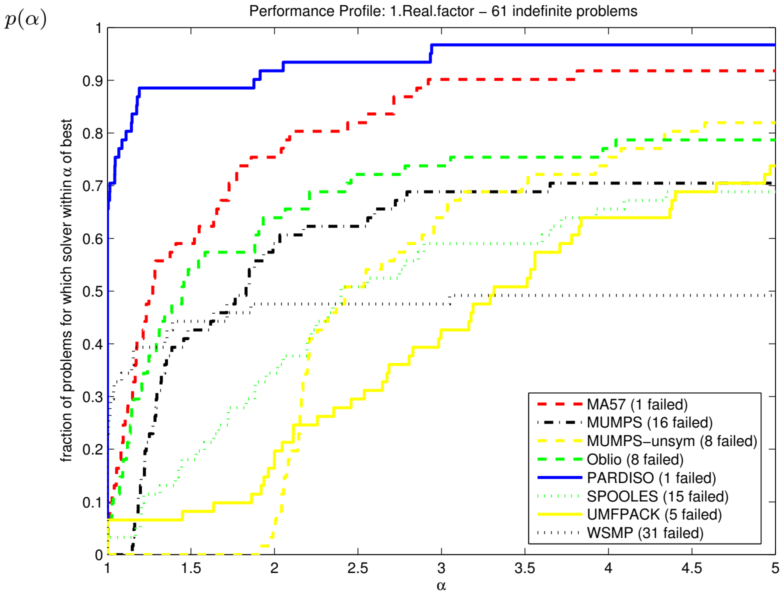

Figure 5.2.3: Performance profile, $p(\alpha)$: Number of entries in the factors (indefinite problems).

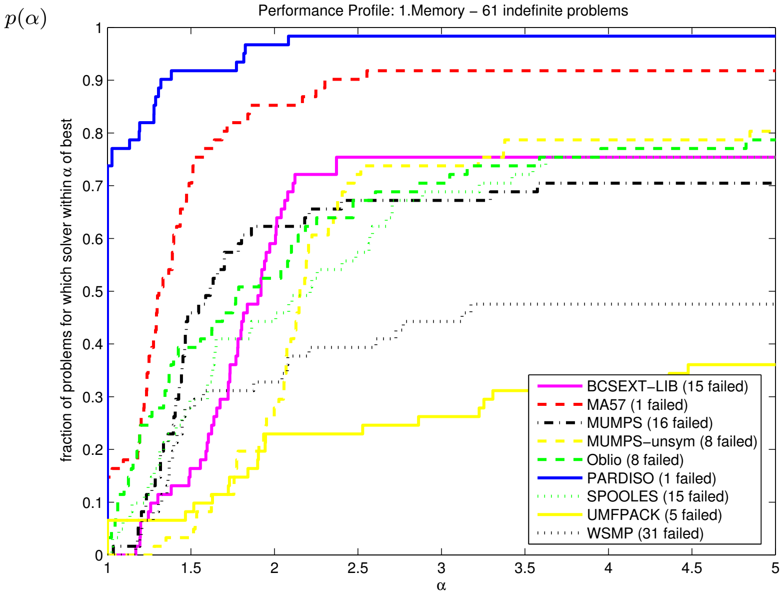

Figure 5.2.4: Performance profile, $p(\alpha)$: Memory used (indefinite problems).

above the rest.

Of course, there is some penalty to be paid for using a potentially less stable factorization, and that is that iterative refinement is a necessary precaution when using the generated factors to solve $Ax = b$. This is apparent in Figure 5.2.5. Now MA57 is a clear winner (with BCSLIB-EXT also performing well on the problems it solved within the CPU time limit), while PARDISO, which performs iterative refinement when pivots have been perturbed, is slower. A closer investigation of the detailed results show that, if pivots have been perturbed during the PARDISO factorization, the corresponding solve can be up to three times slower than the comparable MA57 solve precisely because of the possible two extra "refinement" steps taken.

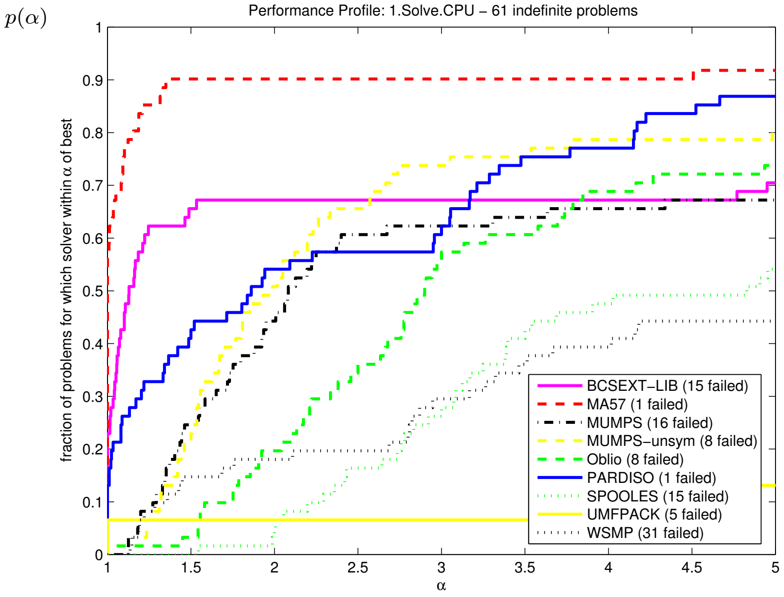

Figure 5.2.5: Performance profile, $p(\alpha)$: CPU time for the solution phase (indefinite problems).

If a complete solution (analyse-factorize-solve) is the primary concern, Figure 5.2.6 indicates a clear preference for MA57 and PARDISO. In terms of CPU time, there is little to choose between the two. Of the remaining solvers, BCSLIB-EXT and the two variants of MUMPS perform best, with the unsymmetric version of MUMPS proving more reliable than the symmetric version because it incorporates off-diagonal pivoting. This trend is reinforced when the subset of larger problems is considered (see Figure 5.2.7).

We also investigated using a small stability threshold parameter (see § 4.4). In some cases, this improved the quality of the factorization (reduced both the CPU time and numbers of nonzeros in the factors), but at the cost of lower overall reliability for some solvers (for example, without employing iterative refinement, MA57 did not solve three additional problems with the required accuracy and for UMFPACK a further twelve failures occurred). But for other solvers (BCSLIB-EXT) and MUMPS) there were fewer failures overall either because, with the smaller threshold, the solver completed within our time limit or

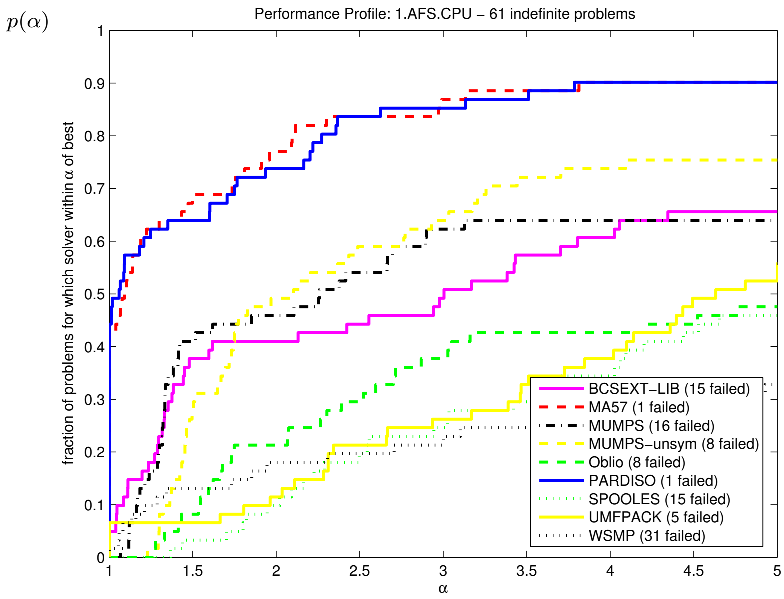

Figure 5.2.6: Performance profile, $p(\alpha)$: CPU time for the complete solution (indefinite problems).

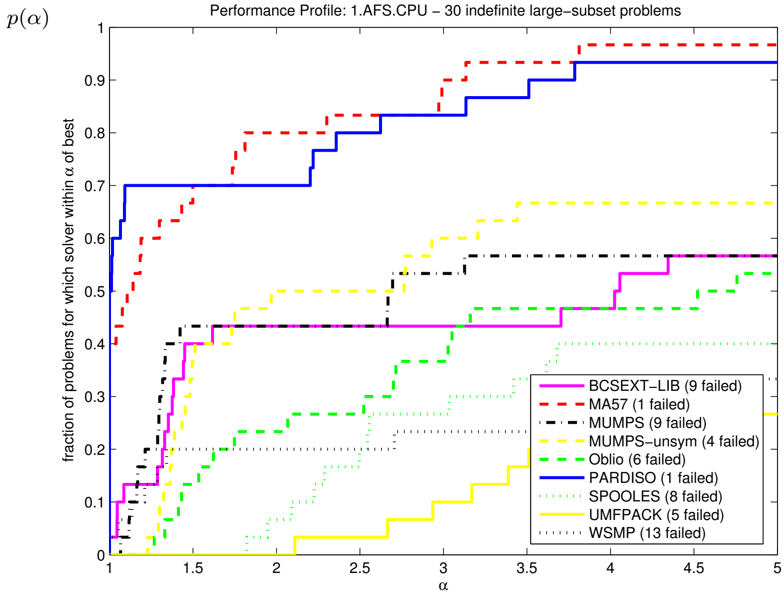

Figure 5.2.7: Performance profile, $p(\alpha)$: CPU time for the complete solution (large indefinite subset problems).

because less space was required because of fewer delayed pivots. Using external scaling (see § 4.6) did not appear to offer a significant or consistent advantage.

6 Concluding remarks

In this paper, we have compared a number of currently-available software packages for the direct solution of real symmetric sparse linear systems of equations. Although there are detailed differences, all the methods we have considered broadly comprise three phases: an analysis of the sparsity pattern with a view to reordering the variables to reduce fill-in, a (static or dynamic) factorization of the reordered matrix, and a solution of the given system using forward- and back-substitution. The interaction between all three phases is crucial for a reliable and fast solution. Thus although minimum-degree-based analysis phases generally appear to be faster than dissection-based ones, for are test problems the resulting factors are generally less sparse which (negatively) influences the speed of both the subsequent factorization and solve phases.

For positive-definite systems, we find in general that there is little in terms of reliability and efficiency to distinguish between the leading competitors (BCSLIB-EXT, MA57, MUMPS, Oblio, PARDISO, SPRSBLKLLT, TAUCS and WSMP). Nevertheless, if many factorizations of matrices with identical sparsity patterns but differing values are required, WSMP and PARDISO are the strongest candidates, while if many solutions for a given matrix are needed BCSLIB-EXT, MA57 and PARDISO can be recommended. For indefinite problems, there are no strong stability guarantees without pivoting, and this is reinforced by the high percentage of failures for some algorithms in this case. The leading contenders here are MA57 and PARDISO. The former is more cautious with its factorization phase (and consequently the latter is faster), but such caution pays off in a faster solution phase as there is less need to resort to iterative refinement to correct for poor residuals. Both of these codes are being actively developed; indeed, both codes have been significantly improved since we started work on this study, partly as a result of feedback from us. The careful use of static pivoting within PARDISO (see Schenk and Gärtner, 2004*a*) is surprisingly effective and currently under investigation by the authors of other packages (see, for example, Duff and Pralet, 2005).

As we discussed in Section 4.4, we have limited our experiments to running each of the packages with its default (or recommended) settings. Clearly, for many problems it may be possible to get an individual code to run faster and produce sparser factors by tuning the control parameters to the problem (indeed, a particular parameter choice may enable a code to succeed where we report a failure). From the brief descriptions of the codes and their key features given in Sections 2 and 3, it should be apparent that some of the codes offer the user a large number of parameters that can be used to tune the code for particular applications. Notable examples are BCSLIB-EXT and MA57. In addition, Oblio offers the user the possibility of trying different factorization algorithms.

A further limitation of this study is that all our experiments were performed using a single computing platform. Although our main concern is how the codes perform relative to each other rather than individual CPU timings, clearly there could be some variation in performance on different computing platforms. Of course, some of our reported statistics such as the number of entries in the factors and memory usage are independent of the platform.

We readily concede that this paper is merely a snap-shot of an evolving field, and that perhaps a different picture will emerge in the not-too-distant future. Nevertheless, since the solution of linear systems is a vital component in many areas of scientific computation, we believe that our paper will be useful to both software developers and (potential) users as a guide to current state-of-the-art of sparse direct solvers.

Acknowledgements

We would like to thank the authors of the solvers used in this study who supplied us with copies of their codes and documentation, helped us to use the software, answered our queries, and commented on a draft of this paper. In particular, we are grateful to Patrick Amestoy, Cleve Ashcraft, Tim Davis, Florin Dobrian, Iain Duff, Jean-Yves L'Excellent, Anshul Gupta, John Lewis, Esmond Ng, Alex Pothen, Stephane Pralet, Olaf Schenk, Sivan Toledo, and David Wah. Our thanks also to those who supplied test problems, including Mario Arioli, Christian Damhaug, Tim Davis, Anshul Gupta, Alison Ramage, Olaf Schenk, Miroslav Tuma, and Andy Wathen.

References

P.R. Amestoy. Recent progress in parallel multifrontal solvers for unsymmetric sparse matrices. *in* 'Proceedings of the 15th World Congress on Scientific Computation, Modelling and Applied Mathematics, IMACS 97, Berlin', 1997.

P.R. Amestoy, T.A. Davis, and I.S. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Analysis and Applications*, **17**, 886–905, 1996.

P.R. Amestoy, T.A. Davis, and I.S. Duff. Algorithm 837: Amd, an approximate minimum degree ordering algorithm. *ACM Trans. Mathematical Software*, **30**(3), 381–388, 2004.

P.R. Amestoy, I.S. Duff, and J.Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods in Appl. Mech. Eng*, **184**, 501–520, 2000.

P.R. Amestoy, I.S. Duff, J.Y. L'Excellent, and J. Koster. A fully asynchronmous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Analysis and Applications*, **23**, 15–41, 2001.

M. Arioli, J.W. Demmel, and I.S. Duff. Solving sparse linear systems with sparse backward error. *SIAM J. Matrix Analysis and Applications*, **10**, 165–190, 1989.

C. Ashcraft and J.W.H. Liu. Robust ordering of sparse matrices using multisection. *SIAM J. Matrix Analysis and Applications*, **19**, 816–832, 1998.

C. Ashcraft, R.G. Grimes, and J.G. Lewis. Accurate symmetric indefinite linear equation solvers. *SIAM J. Matrix Analysis and Applications*, **20**, 513–561, 1998.

J.R. Bunch and L. Kaufmann. Some stable methods for calculating inertia and solving symmetric linear systmes. *Mathematics of Computation*, **31**, 163–179, 1977.

J.R. Bunch, L. Kaufmann, and B.N. Parlett. Decomposition of a symmetric matrix. *Numerische Mathematik*, **27**, 95–110, 1976.

T.A. Davis. Algorithm 832: UMFPACK, an unsymmetric-pattern multifrontal method. *ACM Trans. Mathematical Software*, **30**(2), 196–199, 2003*a*.

T.A. Davis. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. *ACM Trans. Mathematical Software*, **34**(2), 165–195, 2003*b*.

T.A. Davis and I.S. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. Technical Report RAL-93-036, Rutherford Appleton Laboratory, 1993.

F. Dobrian, G.K. Kumfert, and A. Pothen. The design of sparse direct solvers using object-oriented techniques. *in* H. Langtangen, A. Bruaset and E. Quak, eds, 'Advances in Software Tools in Scientific Computing', Vol. 50 of *Lecture Notes in Computational Science and Engineering*, pp. 89–131. Springer-Verlag, 2000.

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

J.J. Dongarra, J. DuCroz, I.S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Mathematical Software*, **16**(1), 1–17, 1990.

J.J. Dongarra, I.S. Duff, D.C. Sorsensen, and H.A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. SIAM, 1998.

I.S. Duff. MA57– a new code for the solution of sparse symmetric definite and indefinite systems. *ACM Trans. Mathematical Software*, **30**, 118–154, 2004.

I.S. Duff and J. Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM J. Matrix Analysis and Applications*, **20**, 889–901, 1999.

I.S. Duff and S. Pralet. Strategies for scaling and pivoting for sparse symmetric indefinite problems. Technical Report RAL-TR-2004-020, Rutherford Appleton Laboratory, 2004.

I.S. Duff and S. Pralet. Towards a stable static pivoting strategy for the sequential and parallel solution of sparse symmetric indefinite systems. Technical Report RAL-TR-2005-007, Rutherford Appleton Laboratory, 2005.

I.S. Duff and J.K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, **9**, 302–325, 1983.

I.S. Duff and J.A. Scott. Towards an automatic ordering for a symmetric sparse direct solver. Technical report, Rutherford Appleton Laboratory, 2005. To appear.

I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, England, 1986.

I.S. Duff, R.G. Grimes, and J.G. Lewis. Sparse matrix test problems. *ACM Trans. Mathematical Software*, **15**, 1–14, 1989.

A. George. Nested dissection of a regular finite-element mesh. *SIAM J. Numerical Analysis*, **10**, 345–363, 1973.

A. George and J.W.-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, New Jersey, U.S.A., 1981.

J.R. Gilbert, E. Ng, and B.W. Peyton. An efficient algorithm to compute row and column counts for sparse Cholesky factorization. *SIAM J. Matrix Analysis and Applications*, **15**, 1075–1091, 1994.

N.I.M. Gould and J.A. Scott. Complete results for a numerical evaluation of HSL packages for the direct solution of large sparse, symmetric linear systems of equations. Numerical Analysis Internal Report 2003-2, Rutherford Appleton Laboratory, 2003. Available from www.numerical.rl.ac.uk/reports/reports.shtml.

N.I.M. Gould and J.A. Scott. A numerical evaluation of HSL packages for the direct solution of large sparse, symmetric linear systems of equations. *ACM Trans. Mathematical Software*, pp. 300–325, 2004.

N.I.M. Gould and Ph.L. Toint. An iterative working-set method for large-scale non-convex quadratic programming. *Applied Numerical Mathematics*, **43**(1–2), 109–128, 2002.

N.I.M. Gould, Y. Hu, and J.A. Scott. Complete results for a numerical evaluation of sparse direct solvers for the solution of large, sparse, symmetric linear systems of equations. Numerical Analysis Internal Report 2005-1, Rutherford Appleton Laboratory, 2005. Available from www.numerical.rl.ac.uk/reports/reports.shtml.

A. Gupta, M. Joshi, and V. Kumar. WSMP: A high-performance serial and parallel sparse linear solver. Technical Report RC 22038 (98932), IBM T.J. Watson Reserach Center, 2001. www.cs.umn.edu/˜agupta/doc/wssmp-paper.ps.

A. Gupta, G. Karypis, and V. Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed Systems*, **8**(5), 502–520, 1997.

M. Heath, E. Ng, and B. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, **33**, 420–460, 1991.

HSL. A collection of Fortran codes for large-scale scientific computation, 2004. See http://www.cse.clrc.ac.uk/nag/hsl/.

G. Karypis and V. Kumar. METIS: A software package for partitioning unstructured graphs, partitioning meshes and computing fill-reducing orderings of sparse matrices - version 4.0, 1998. See http://www-users.cs.umn.edu/ karypis/metis/.

G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, **20**, 359–392, 1999.

J.W.H. Liu. Modification of the Minimum-Degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, **11**(2), 141–153, 1985.

J.W.H. Liu. The multifrontal method for sparse matrix solution: theory and practice. *SIAM Review*, **34**, 82–109, 1992.

E.G. Ng and B.W. Peyton. Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM Journal on Scientific Computing*, **14**(5), 1034–1056, 1993.

V. Rotkin and S. Toledo. The design and implementation of a new out-of-core sparse Cholesky factorization method. *ACM Transactions on Mathematical Software*, **30**(1), 2004.

M.A. Saunders. Sparse matrices in optimization, 1994. Presented at Sparse Days at St Girons, International meeting on Sparse Matrix Methods, St Girons, France. See http://www.stanford.edu/group/SOL/talks/saunders-stgirons.ps.

O. Schenk and K. Gärtner. On fast factorization pivoting methods for sparse symmetric indefinite systems. Technical Report CS-2004-004, Department of Computer Science, University of Basel, Switzerland, 2004*a*.

O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Journal of Future Generation Computer Systems*, **20**(3), 475–487, 2004*b*.

O. Schenk, K. Gärtner, and W. Fichtner. Efficient sparse LU factorization with left-right looking strategy on shared memory multiprocessors. *BIT*, **40**(1), 158–176, 2000.

J. Schulze. Towards a tighter coupling of bottom-up and top-down sparse matrix ordering methods. *BIT*, **41**, 800–841, 2001.

W.F. Tinney and J.W. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proc. IEEE*, **55**, 1801–1809, 1967.

R.J. Vanderbei. Symmetric quasidefinite matrices. *SIAM Journal on Optimization*, **5**(1), 100–113, 1995.