HSL_MI28: an efficient and robust limited-memory incomplete Cholesky factorization code

JENNIFER SCOTT, Rutherford Appleton Laboratory
MIROSLAV TŮMA, Institute of Computer Science, Academy of Sciences of the Czech Republic

This paper focuses on the design and development of a new robust and efficient general-purpose incomplete Cholesky factorization package HSL_MI28, which is available within the HSL mathematical software library. It implements a limited memory approach that exploits ideas from the positive semidefinite Tismenetsky-Kaporin modification scheme and, through the incorporation of intermediate memory, is a generalization of the widely-used ICFS algorithm of Lin and Moré. Both the density of the incomplete factor and the amount of memory used in its computation are under the user's control. The performance of HSL_MI28 is demonstrated using extensive numerical experiments involving a large set of test problems arising from a wide range of real-world applications. The numerical experiments are used to isolate the effects of scaling, ordering and dropping strategies so as to assess their usefulness in the development of robust algebraic incomplete factorization preconditioners and to select default settings for HSL_MI28. They also illustrate the significant advantage of employing a modest amount of intermediate memory. Furthermore, the results demonstrate that, with limited memory, high quality yet sparse general-purpose preconditioners are obtained. Comparisons are made with ICFS, with a level-based incomplete factorization code and, finally, with a state-of-the-art direct solver.

Categories and Subject Descriptors: G.1.3 [**Numerical Linear Algebra**]: Linear systems (direct and iterative methods); G.1.3 [**Numerical Linear Algebra**]: Sparse, structured, and very large systems (direct and iterative methods)

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Symmetric and positive definite linear systems, preconditioning, incomplete Cholesky, iterative methods

ACM Reference Format:
Jennifer Scott and Miroslav Tůma, 2013. HSL_MI28: an efficient and robust limited-memory incomplete Cholesky factorization code. *ACM Trans. Math. Softw.* V, N, Article A (YYYY), 19 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Incomplete Cholesky (IC) factorizations have long been an important tool in the armoury of methods for the numerical solution of large sparse symmetric linear systems $Ax = b$. Many papers, books and reports on the development and performance of incomplete factorizations as preconditioners for iterative methods for solving problems from a wide range of practical applications have been published over the last 50 years

This work is supported by EPSRC grant EP/I013067/1 and by the Grant Agency of the Czech Republic Project No. P201/13-06684S. Travel support from the Academy of Sciences of the Czech Republic is also acknowledged.
Author's addresses: J. Scott, Scientific Computing Department, STFC Rutherford Appleton Laboratory, Harwell Oxford, Oxfordshire, OX11 0QX, UK, jennifer.scott@stfc.ac.uk; M. Tůma, Institute of Computer Science, Academy of Sciences of the Czech Republic, Pod Vodárenskou věží 2, 18207 Praha 8 - Libeň, tuma@cs.cas.cz.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© YYYY ACM 0098-3500/YYYY/-ARTA $15.00
DOI: http://dx.doi.org/10.1145/0000000.0000000

(for an introduction or overview see, for instance, [Benzi 2002; Saad 1996; Scott and Tůma 2013] and the long lists of references therein). A number of software packages have also been developed and made available; the recent study by George, Gupta and Sarin [2011; 2012] provides an insightful empirical analysis of some widely-used iterative solvers that include an IC factorization preconditioner option.

An incomplete Cholesky factorization takes the form LL^T in which some of the fill entries (entries that were zero in A) that would occur in a complete factorization are ignored. There are many different types of incomplete factorization. An important class are level-based $IC(\ell)$ methods in which the location of permissible fill entries using only the sparsity pattern of A is prescribed in advance. The aim is to partially mimic the way in which the pattern of A is developed during the complete factorization. A symbolic factorization phase is used to assign each potential fill entry a level and an entry is only permitted in the factor if its level is at most ℓ. Unfortunately, while entries of the error matrix $E = A - LL^T$ are zero inside the prescribed sparsity pattern, outside they can be very large, and the pattern of $IC(\ell)$ (even for large ℓ) may not guarantee that L is a useful preconditioner (particularly if the entries of A do not decay significantly with distance from the diagonal). Furthermore, although $IC(1)$ can be a significant improvement over $IC(0)$, the fill-in resulting from increasing ℓ can be prohibitive in terms of both storage requirements and the time to compute and then apply the preconditioner.

Another widely-used class of IC factorizations are threshold-based $IC(\tau)$ methods in which the locations of permissible fill entries are determined in conjunction with the numerical factorization of A; entries of the computed factors or intermediate quantities that in magnitude are less than a prescribed drop tolerance τ are discarded. Success depends on being able to choose a suitable τ but this can be highly problem dependent. Intuitively, using a small τ is more likely to produce a high quality preconditioner (measured in terms of the iteration count of a preconditioned Krylov subspace method) than using a larger τ. But as the fill-in increases as τ is reduced, there is generally a trade-off between sparsity and quality. Improvements in robustness through extending the preconditioner structure by adding simple patterns or by reducing τ have their limitations and, importantly, for algebraic preconditioning, do not address the problem of memory consumption.

A straightforward approach to overcome the memory problem is to simply prescribe the maximum number of entries allowed in each column of the incomplete factor L and retain only the largest entries. In this case, the dropping inclusion property is often satisfied, that is, by increasing the maximum number of entries, a higher quality preconditioner is obtained. This strategy appears to have been proposed first by Axelsson and Munksgaard [1983]. It enabled them to significantly simplify their right-looking implementation as it allowed simple bounds on the amount of memory required. Dropping based on a prescribed upper bound for the largest number of entries in a column of L combined with an efficient strategy to keep track of left-looking updates was implemented in a successful and influential IC code by Jones and Plassman [1995a; 1995b]. They retained the n_j largest entries in the strictly lower triangular part of the $j-$th column of L, where n_j is the number of entries in the $j-$th column of the strictly lower triangular part of A. The code has predictable memory demands and, in the event of factorization breakdown (that is, a zero or negative pivot is encountered), it uses a strategy similar to that proposed by Manteuffel [1980] of applying a global diagonal shift (so that $A + \alpha I$ is factorized for some positive α).

The combination of dropping by magnitude with bounding the number of entries in a column first appeared in [Freund and Nachtigal 1990] but a very popular concept that has predictable memory requirements is the dual threshold $ILUT(p, \tau)$ factorization of Saad [1994]. This is designed for nonsymmetric problems and combines the use of a

drop tolerance τ with the strategy of prescribed maximum column and row counts. All entries in the computed factors that are smaller in magnitude than τ_l are discarded, where τ_l is the product of τ times the l_2-norm of the $l-$th row of A. Additionally, only the p largest entries in each column of L and row of U are retained. This approach ignores symmetry in A and, if A is symmetric, the sparsity patterns of L and U^T are normally different.

A widely-used limited-memory IC factorization implementation is provided by the ICFS code of Lin and Moré [1999]. It aims to exploit the best features of both the Jones and Plassmann and the Saad factorizations, incorporating l_2-norm based scaling and adding a loop for efficiently changing the Manteuffel diagonal shift to prevent breakdown. Given a user-controlled parameter p, ICFS retains the n_j+p largest entries in the lower triangular part of the $j-$th column of L and uses only memory as the criterion for dropping entries (thus having both the advantage and disadvantage of not requiring a drop tolerance). Reported results for large-scale trust region subproblems indicate that using additional memory by selecting $p > 0$ can substantially improve performance on difficult problems. However, following [Scott and Tůma 2011], let us define the *efficiency* of an IC preconditioner to be

$$iter \times nz(L), \tag{1}$$

where $iter$ is the iteration count (number of iterations required by the Krylov subspace method using the preconditioner to achieve the requested accuracy) and $nz(L)$ is the number of entries in the incomplete factor L. Numerical experimentation reported in [Scott and Tůma 2013] has shown that, for a general set of problems arising from a range of applications (see Section 4.1 for details), increasing p improves reliability but the efficiency is not very sensitive to the choice of p (although, of course, the time to compute the factorization and the storage for L increase with p).

An alternative approach to incomplete factorizations focuses on avoiding breakdown by incorporating positive semidefinite modifications. The scheme introduced by Jennings and Malik [1977; 1978] in the late 1970s modifies the corresponding diagonal entries every time an off-diagonal entry is discarded. It can be shown that the sequence of these modifications leads to a breakdown-free factorization for which the error matrix E is a sum of positive semidefinite matrices with non-positive off-diagonal entries and is thus itself positive semidefinite. This approach has been adopted in some engineering applications (and can give good results on moderately ill-conditioned problems) but in the experiments reported on in [Scott and Tůma 2013] for a more general test set, it was observed that, in general, the quality of the computed preconditioner was less than was obtained using a global diagonal shift.

A more sophisticated modification scheme is one due to Tismenetsky [1991] and which was later significantly enhanced by Kaporin [1998]. This approach, which we discuss further in Section 2, introduces the use of intermediate memory that is employed during the construction of L but is then discarded. It has been shown to be very robust but, as Benzi [2002] remarks, it "has unfortunately attracted surprisingly little attention". This may be because it suffers from a serious drawback: its memory requirements can be prohibitively high (in some cases, it has been reported that it uses more than 70 per cent of the storage required for a complete Cholesky factorization, see [Benzi and Tůma 2003]). The dropping strategy of Kaporin may help to alleviate this but a breakdown-free factorization is no longer guaranteed.

This leads us to the two main motivations that lie behind our work. The first is that we would like to develop a generalization of the successful ICFS algorithm that is such that the efficiency of the IC preconditioner improves with the prescribed memory. Secondly, we would like a memory-efficient variant of the Tismenetsky-Kaporin approach, without diagonal compensation but using a global shift to avoid breakdown.

We want to combine both these aims within the development a single "black-box" IC factorization code that is demonstratively robust and efficient on a wide range of problems, while also being flexible in allowing experienced users to tune the parameters to potentially enhance performance for their particular problems of interest. The new package is called HSL_MI28 and is available within the HSL mathematical software library [HSL 2013].

The remainder of this paper is organised as follows. In Section 2, we present a brief overview of the Tismenetsky-Kaporin scheme. We also explain how we can limit the memory requirements and, in so doing, we obtain a generalization of the ICFS algorithm. An outline of the algorithm implemented by HSL_MI28 is given in Section 3; implementation details and user parameters are discussed. In Section 4, numerical results for HSL_MI28 are presented. In particular, we report on the effects of using intermediate memory and of scaling, ordering and dropping, and provide comparisons with a level-based preconditioner and a modern sparse direct solver. Concluding remarks are made in Section 5.

2. A LIMITED MEMORY TISMENETSKY-KAPORIN MODIFICATION SCHEME

2.1. The approach of Tismenetsky

The Tismenetsky scheme [Tismenetsky 1991] is based on a matrix decomposition of the form

$$A = (L + R)(L + R)^T - E, \tag{1}$$

where L is a lower triangular matrix with positive diagonal entries that is used for preconditioning, R is a strictly lower triangular matrix with small entries that is used to stabilise the factorization process, and we consider the error E given by

$$E = RR^T. \tag{2}$$

Although HSL_MI28 uses a left-looking implementation, for simplicity, consider for a moment an equivalent right-looking approach: at each step, the next column of L is computed and then the remaining Schur complement is modified. On the $j-$th step, the first column of the Schur complement can be decomposed into a sum of two vectors

$$l_j + r_j,$$

such that $|l_j|^T|r_j| = 0$ (with the first entry in l_j nonzero), where l_j (respectively, r_j) contains the entries that are retained in (respectively, discarded from) the incomplete factorization. On the next step of a complete decomposition, the Schur complement of order $n - j$ is updated by subtracting the outer product of the pivot row and column. That is, by subtracting

$$(l_j + r_j)(l_j + r_j)^T.$$

The Tismenetsky incomplete factorization does not compute the complete update as it does not subtract

$$E_j = r_j r_j^T. \tag{3}$$

Thus, the positive semidefinite modification E_j is implicitly added to A.

As proposed by Tismenetsky, the obvious choice for r_j is the smallest off-diagonal entries in the column (those that are smaller in magnitude than a chosen tolerance). Then at each step of the right-looking formulation, implicitly adding E_j is combined with the standard steps of the Cholesky factorization, with entries dropped from L *after* the updates have been applied to the Schur complement. The approach is naturally breakdown-free because the only modification of the Schur complement that is used in the later steps of the factorization is the addition of the positive semidefinite matrices E_j.

2.2. Kaporin's second order incomplete Cholesky factorization scheme

While the fill in L can be controlled through the use of a drop tolerance, this does not limit the memory required to compute L. A right-looking implementation of a sparse factorization is generally very demanding from the point of view of memory as it is necessary to store all the fill-in for column j until the modification is applied in the step j, as follows from (3). Hence, a left-looking implementation (or, as in [Kaporin 1998], an upward-looking implementation) might be thought preferable. But to compute column l_j and r_j in a left-looking implementation and to apply the modification (3) correctly, all the vectors l_k and r_k for $k = 1, \ldots, j-1$ have to be available. Therefore, the dropped entries have to be stored throughout the left-looking factorization and the r_k cannot be discarded until the factorization is finished (and similarly for an upward-looking implementation). These r_k vectors thus represent intermediate memory. Note the need for intermediate memory is caused not just by the fill in the factorization: it is required because of the structure of the positive semidefinite modification that forces the use of the r_k. Sparsity allows some of the r_k to be discarded before the factorization is complete but essentially the total memory is as for a complete factorization, without the other tools that direct methods offer. This memory problem was discussed by Kaporin [1998], who proposed using two drop tolerances τ_1, τ_2 with $\tau_1 > \tau_2$. Only entries of magnitude at least τ_1 are kept in L and entries smaller than τ_2 are dropped from R. In this case, the error matrix E has the structure

$$E = RR^T + F + F^T,$$

where F is a strictly lower triangular matrix that is not computed while R is used in the computation of L but is then discarded.

When non-zero drop tolerances are introduced, the factorization is no longer guaranteed to be breakdown-free. To avoid breakdown, diagonal compensation (as in the Jennings-Malik scheme discussed in Section 1) for the entries that are dropped from R may be used. Kaporin coined the term *second order incomplete Cholesky factorization* to denote this combined strategy.

In recent years, the Tismenetsky-Kaporin approach has been used to provide a robust preconditioner for some practical applications, see for example [Axelsson et al. 2000; Beirão da Veiga et al. 2009; Kaporin 1998; Lipnikov et al. 2007; Lipnikov et al. 2009]. We note, however, that there are no reported comparisons with other approaches that take into account not only iteration counts but also the size of the preconditioner; providing some comparisons is one of our aims (see Sections 4.7 and 4.8).

2.3. A limited memory Tismenetsky-Kaporin approach

The use of drop tolerances τ_1 and τ_2 can help to reduce the amount of intermediate memory needed. However, picking suitable tolerances can be highly problem dependent: too small and the dropping has little effect, too large and the resulting preconditioner is ineffective. Applying an appropriate scaling can help significantly. Yamazaki et al [2009] use the Tismenetsky-Kaporin approach without diagonal compensation; in the event of breakdown, they employ a global diagonal shift. By restricting their attention to a specific class of problems, they are able to determine an interval of useful drop tolerances that limit the size of the computed factor and to obtain good preconditioners for their examples.

In a recent study [Scott and Tůma 2013], we found that, in terms of efficiency (recall (1)), using unlimited intermediate memory leads to a good preconditioner. However, efficiency as a measure for assessing how well an algorithm works does not take into account the time required to compute the incomplete factorization and this can be prohibitive (or at least uncompetitive) for the Tismenetsky-Kaporin approach, unless the factorization time can be amortized over several problems. Furthermore, for a number

of the largest test problems used in [Scott and Tůma 2013], the incomplete factorization failed because of insufficient memory, that is, a memory allocation error was returned. Thus, the Tismenetsky-Kaporin approach can only be considered robust if the problems to be solved are not too large for the available memory. These key issues highlight that the Tismenetsky-Kaporin approach can be impractical for the very large problems that we want to solve using an iterative method and motivates us impose on it the use of limited memory.

The memory predictability of our approach depends on specifying a parameter $lsize$ that limits the maximum number of nonzero off-diagonal fill entries in a column of L. Thus we retain at most $nz(A)+(n-1)*lsize$ off-diagonal entries in the incomplete factorization (where $nz(A)$ is the number of entries in the lower triangular part of A and n is the order of A). We employ a second parameter $rsize \geq 0$ that controls the amount of intermediate memory that is used for R; it is limited to at most $(n-1)*rsize$ entries. At step j, the candidate entries for inclusion in the $j-$th column of L are held in a temporary array. The entries of this array are sorted and (at most) the $n_j + lsize$ entries of largest magnitude (plus the diagonal) are retained in L; the next $rsize$ largest entries form the $j-$th column of R and all other entries are discarded. Following Kaporin, our implementation offers the option of employing two drop tolerances; entries in L are retained only if they are at least τ_1 in magnitude while those in R must be at least τ_2. Numerical results (see Section 4.4) will illustrate the potential benefits of employing dropping according to the magnitude of the entries.

3. AN ENHANCED LIMITED-MEMORY IC FACTORIZATION

Based on the above discussion, we now summarise our limited-memory IC factorization algorithm and briefly describe its user interface and implementation; further details of how to use the code are supplied in the user documentation that is distributed with the software and is also available at `http://www.hsl.rl.ac.uk/catalogue/`. Given a symmetric sparse matrix A, `HSL_MI28` computes an IC factorization $(QL)(QL)^T$, where Q is a permutation matrix. The matrix A is optionally scaled and, if necessary, shifted to avoid breakdown of the factorization, so that the incomplete factorization of $\overline{A} = SAS + \alpha I$ is computed, where $S = \{s_i\}$ is a diagonal scaling matrix and α is a positive shift. The user supplies the lower triangular part of A in compressed sparse column format and the computed L is returned to the user in the same format; a separate entry performs the preconditioning operation $y = Pz$, where $P = (\overline{L}\overline{L}^T)^{-1}$, $\overline{L} = S^{-1}QL$, is the incomplete factorization preconditioner.

3.1. Algorithm outline

In Figure 1, we present a summary outline of our left-looking IC algorithm. Here $A = \{a_{ij}\}$ and a_j, l_j and r_j denote the $j-$th columns of the lower triangular parts of A, L and R, respectively; w is a work array of length n. For simplicity of explanation, we assume that each l_j can have at most $lsize$ fill entries (that is, we ignore the use of any spare space that has been passed from a previous column). The scalar $small$ is used to determine whether a diagonal entry is sufficiently large; if at any stage a diagonal entry is less than $small$, the factorization is considered to have broken down and in this case, the shift α is increased and the factorization restarted. Details of the shift strategy are given in Section 3.2.5 (note that in Figure 1 we omit the possibility of decreasing the chosen α but this is discussed in Section 3.2.5). $\tau_1 > \tau_2 \geq 0$ are chosen drop tolerances.

The ICFS algorithm of Lin and Moré [1999] is a special case in which ordering is not incorporated, $\tau_1 = \tau_2 = 0$ and $rsize = 0$ (so that there is no dropping of entries by size, $R = 0$ and only LL^T updates are applied).

ALGORITHM 1: Outline of the HSL_MI28 incomplete Cholesky factorization

! Reorder (see Section 3.2.4)
Compute an ordering Q for A
Permute the matrix: $A \leftarrow Q^T AQ$

! Scale (see Section 3.2.3)
Compute a diagonal scaling S
Scale the matrix: $A \leftarrow SAS$

! Diagonal shift (see Section 3.2.5)
Choose α such that $\min(a_{ii}) + \alpha > small$
Initialise breakdown = false and $\alpha_0 = 0$

! Loop over shifts
Set $w = 0$
do
  Set $A \leftarrow A + (\alpha - \alpha_0)I$ and $d(1:n) = (a_{11}, a_{22}, ..., a_{nn})$

  ! Factorization : loop over columns
  for $j = 1 : n$ **do**
    Copy a_j into w
    Apply $LL^T + RL^T + LR^T$ updates from columns $1 : j-1$ to w
    Apply $LL^T + RL^T + LR^T$ updates from columns $1 : j-1$ to $d(j+1:n)$
    Optionally apply RR^T updates from columns $1 : j-1$ that cause
      no additional fill-in in w *! User-controlled option (see Section 3.2.6)*
    if $(\min(d(j+1:n)) < small)$ **then**
      Set breakdown = true, $\alpha_0 = \alpha$ and increase α *! (see Section 3.2.5)*
      exit
    end if
    Sort entries in w by magnitude
    Keep at most $n_j + lsize$ entries of largest magnitude in l_j such that
      they are all at least τ_1
    Keep at most $rsize$ additional entries that are next largest in magnitude
      in r_j such that they are all at least τ_2
    Reset entries of w to zero
  end do
  if breakdown = false **exit**
end do

3.2. User interface and implementation details

3.2.1. Setting the memory limits. The user must supply the following parameters that determine the amount of memory and work involved in computing the incomplete factorization:

. `lsize` is the maximum number of fill entries within each column of the incomplete factor L. The number of entries in the computed factor is at most $nz(A) +$ `lsize` $\times (n-1)$.
. `rsize` is the maximum number of entries within each column of the strictly lower triangular matrix R. An integer array and a real array of size `rsize` $\times (n-1)$ are allocated internally to hold R thus the amount of intermediate memory used, as well as the amount of work involved in computing the preconditioner, depends on

rsize. If rsize $= 0$, R is not used (and the code is then our implementation of the ICFS algorithm [Lin and Moré 1999]).

3.2.2. Control parameters. A number of control parameters provide the experienced user with the means of experimenting with different settings and to tune the code for a particular application. These parameters have default values that we have chosen on the basis of numerical experimentation (see Section 4 for results that support our choices); for some problems, selecting different values can be beneficial but the default settings have been found to be generally robust. In particular, we have two parameters, tau1 ($= \tau_1$) and tau2 ($= \tau_2$), that control dropping of small entries from L and R, respectively. Note that, by allowing lsize and rsize to be sufficiently large, the user can force the code to simulate the dropping strategy of Suarjana and Law [1995] and Kaporin [1998]. But, as we will see from our experimental results, for our test examples this is unnecessary since limiting memory appears to afford a more practical approach. From the results of our experiments (see Section 4.4), the dropping parameters have default values 0.001 and 0.0001, respectively; dropping based on the size of entries is disabled by setting both to zero. Observe that these values, that were chosen on the basis of our general set of scaled matrices, are significantly smaller than those that are typically recommended (see, for example, [Benzi 2002; Kaporin 1998]).

3.2.3. Scaling options. For scaling and ordering A before the factorization begins, HSL_MI28 is able to offer a range of options by taking advantage of existing HSL routines. The default scaling is l_2 scaling, in which the entries in column j of A are normalised by the 2-norm of column j; this is used in ICFS [Lin and Moré 1999]. We also offer diagonal scaling, scaling based on maximum matching using a symmetrized version of the package MC64 [Duff and Koster 2001; Duff and Pralet 2005], and equilibration scaling using MC77 [Ruiz 2001; Ruiz and Uçar 2011]; in addition, there is a facility for the user to supply a scaling. The effects of the choice of scaling are illustrated in Section 4.3.

3.2.4. Ordering options. Based on our numerical experiments (see Section 4.5), a profile reduction ordering using a variant of the Sloan algorithm [Reid and Scott 1999; Sloan 1986; 1989] is the default ordering (this variant is implemented by the HSL package MC61). Other orderings that are currently offered are reverse Cuthill McKee (RCM) [Cuthill and McKee 1969] (again, implemented within MC61), approximate minimum degree [Amestoy et al. 2004] (HSL_MC68), nested dissection (currently, provided by routine METIS_NodeND from the METIS package [Karypis and Kumar 1997] but it is anticipated that this may be replaced by an HSL package in the future), and ordering of the rows by ascending degree. In addition, an option is available for the user to supply an ordering; this is convenient if a series of problems having the same sparsity pattern is to be solved.

3.2.5. Coping with breakdown. In the event of breakdown within the factorization, HSL_MI28 employs a global diagonal shift. It is important to try and use as small a shift as possible but also to limit the number of breakdowns. An option exists for the user to supply an initial (positive) shift α_0. Otherwise, α_0 is computed as in [Lin and Moré 1999] so that, if $\beta = \min(s_i^2 a_{ii}) > 0$ (where s_i is the ith diagonal entry of the scaling matrix S), then $\alpha_0 = 0.0$; otherwise, $\alpha_0 = -\beta +$ lowalpha, where lowalpha > 0 may be chosen by the user. The incomplete factorization algorithm is applied to $\overline{A}_0 = SAS + \alpha_0 I$. If breakdown occurs, a larger shift

$$\alpha_1 = \max(\texttt{lowalpha},\ \ \alpha_0 \times \texttt{shift\_factor}), \tag{1}$$

with shift_factor > 1, is tried. The process continues until an incomplete factorization of $\overline{A}_k = SAS + \alpha_k I$ is successful. If breakdown occurs at the same (or nearly the

same) stage of the factorization for two successive shifts, we try to limit the number of restarts by more rapidly increasing the shift. In this case, we increase α by a factor of 2 $\times$ shift_factor. Conversely, if $\alpha_k =$ lowalpha, to prevent an unnecessarily large shift from being used, we try decreasing α. We first take a copy of the successful factorization (if there is insufficient memory to do this, we accept the successful factorization), then set

$$\alpha_{k+1} = \alpha_k/\texttt{shift\_factor2}, \tag{2}$$

with shift_factor2 > 1, and apply the incomplete factorization algorithm to $\overline{A}_{k+1} = SAS + \alpha_{k+1}I$. If this factorization is also breakdown free, we repeat (up to maxshift times); otherwise we use the stored factorization. In all cases, the value of the final shift is returned to the user, along with the number of shifts tried and the number of restarts.

In summary, the parameters within HSL_MI28 that determine the initial and subsequent choice of the shift α and their default settings are as follows:

. alpha holds the initial shift α. It has default value zero.
. lowalpha controls the choice of the first non-zero shift. The default value is 0.001.
. maxshift determines the maximum number of times the shift can be decreased after a successful factorization with a positive shift. Limiting maxshift may reduce the factorization time but may result in a poorer quality preconditioner. It has default value 3.
. shift_factor controls how rapidly the shift is increased after a breakdown. Increasing shift_factor rapidly may reduce the factorization time but may result in a poorer quality preconditioner. It has default value 2.
. shift_factor2 controls how rapidly the shift is decreased after a successful factorization with $\alpha =$ lowalpha. It has default value 4.

In all the experiments reported on in Section 4, we use the default settings for these parameters (which were selected on the basis of experimentation).

3.2.6. Other control parameters. As well as the parameters already described and parameters that are used to control diagnostic printing, HSL_MI28 uses the following control parameters:

. rrt is used to control whether entries of RR^T that cause no additional fill-in are included (rsize $>$ 0 only). Experiments in [Scott and Tůma 2013] found that allowing such entries can improve the quality of the preconditioner but this is not guaranteed. Such entries are allowed if rrt = .true.. The default is rrt = .false.
. small is used to decide when factorization breakdown has occurred. Any pivot whose modulus is less than small is treated as zero and, if such a pivot is encountered, the factorization breaks down, the shift is increased and the factorization restarted (see Figure 1). The default value in the double precision version of the package is 10^{-20} and in the single version it is 10^{-12}.

4. NUMERICAL EXPERIMENTS

4.1. Test environment

All the numerical results reported on in this paper are performed (in serial) on our test machine that has two Intel Xeon E5620 processors with 24 GB of memory. The ifort Fortran compiler (version 12.0.0) with option -O3 is used. The implementation of the conjugate gradient (CG) algorithm offered by the HSL routine MI22 is employed, with starting vector $x_0 = 0$, the right-hand side vector b computed so that the exact solution

is $x = 1$, and stopping criteria

$$\|A\hat{x} - b\|_2 \leq 10^{-10}\|b\|_2, \tag{1}$$

where $\hat{x}$ is the computed solution. In addition, for each test we impose a limit of 2000 CG iterations.

In assessing the effectiveness of a preconditioner we use the definition of efficiency given by (1). A weakness of this measure is that it does not take into account the number of entries in R. If `lsize` is fixed, increasing `rsize` will generally lead to a more efficient preconditioner. However, this will be at the cost of additional work in the incomplete factorization. Thus we record the time to compute the preconditioner together with the time for convergence of the iterative method: the sum of these is referred to as the *total time* and is also used to assess the quality of the preconditioner. Note that in our tests, a simple matrix-vector product routine is used with the lower triangular part of A held in compressed sparse column format: we have not attempted to perform either the matrix-vector products or the application of the preconditioner in parallel and all times are serial times.

Our test problems are real positive-definite matrices of order at least 1000 taken from the University of Florida Sparse Matrix Collection [Davis and Hu 2011]. Many papers on preconditioning techniques and iterative solvers select a small set of test problems that are somehow felt to be representative of the applications of interest. However, as in [Scott and Tůma 2013], our interest is more general and we want to test our new software on as wide a range of problems as we can. Thus we took all such problems and then removed any that were diagonal matrices and, where there was more than one problem with the same sparsity pattern, we chose only one representative problem. This resulted in a test set of 153 problems of order up to 1.5 million. Following initial experiments, 8 problems were removed from this set as we were unable to achieve convergence to the required accuracy within our limit of 2000 iterations without allowing a large amount of fill. To assess performance on our test set and compare different settings and options, we use performance profiles [Dolan and Moré 2002], which in recent years have been widely used to compare the performance of linear solvers, both direct and iterative (see, for example, [George et al. 2011; 2012; Gould and Scott 2004; Gould et al. 2007]).

Performance profiles provide a way to compare several different algorithms on a set of problems with respect to a performance measure such as efficiency, iteration count or time. The performance ratio for an algorithm on a particular problem is the performance measure for that algorithm divided by the smallest performance measure for the same problem over all the algorithms (here we are assuming that the performance measure is one for which smaller is better). The performance profile is the set of functions $\{\rho_k(\tau) : \tau \in [1, \infty)\}$, where $\rho_k(\tau)$ is the proportion of problems where the performance ratio of the kth algorithm is at most τ. Thus $\rho_k(\tau)$ is a monotonically increasing function taking values in the interval $[0, 1]$. By plotting the curves $\rho_k(\tau)$ on a single plot, it is straightforward to compare them and deduce information about the relative performance of the respective algorithms.

For $\tau = 1$, $\rho_k(\tau)$ is how often the kth algorithm is the best (or joint best). If we assume failure to solve a problem (for example, through the maximum iteration count being exceeded) is signaled by a performance measure of ∞, then $lim_{\tau\to\infty}\rho_k(\tau) < 1$ if the kth algorithm failed to solve at least one problem that was solved by another algorithm. Thus for large values of τ, the performance profile reveals the reliability of the algorithm.

4.2. Results for lsize fixed and rsize varying

As reported in [Scott and Tůma 2013], increasing lsize with rsize = 0 does little to improve the preconditioner quality (in terms of efficiency), although it can improve reliability. We thus start by examining the effects of introducing intermediate memory. We set the drop tolerances to zero, employ l_2 scaling, no reordering, and fix lsize = 5. We run with no intermediate memory, rsize = 2, 5, and 10, and with unlimited intermediate memory (all entries in R are retained, which we denote by rsize = -1). We remark that the latter is not an option offered by HSL_MI28 but corresponds to the original Tismenetsky approach. Figure 1 presents the efficiency performance profile (on the right) and total time performance profile (on the left). Since lsize is the same for all runs, the fill in L is essentially the same in each case and thus comparing the efficiency here is equivalent to comparing the iteration counts. Note that the asymptotes of the performance profile provide a statistic on reliability (that is, the proportion of problems that are successfully solved) and as the curve for rsize = -1 lies below the others on the right-hand axis of the profiles in Figure 1, this indicates poorer reliability when using unlimited memory. This poor reliability is because, as anticipated, in this case there was insufficient memory to factorize a number of our large test problems. Moreover, in terms of time as well as memory, unlimited intermediate memory

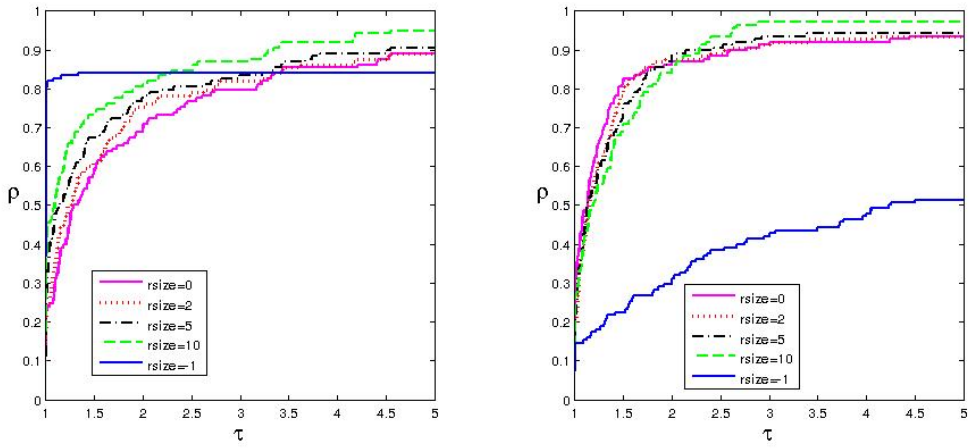

Fig. 1. Efficiency (left) and total time (right) performance profiles for rsize varying.

is the most expensive option. With the introduction of a limit on memory, we see that, as rsize is increased from 0 to 10, the efficiency and robustness of the preconditioner steadily increases, without significantly increasing the total time. Since a larger value of rsize reduces the number of iterations required, if more than one problem is to be solved with the same preconditioner, it may be worthwhile to increase rsize in this case.

Using intermediate memory can reduce the number of restarts required during the factorization and thus the size of the final shift. This is illustrated in Table I. In general, for a given problem, the smaller the shift the fewer the number of iterations. But this is not always the case. For example, for problem GHS_psdef/hood, for rsize = 10, $\alpha = 3.2 \times 10^{-2}$ and 783 iterations are required. However, for rsize= 5, a larger shift $\alpha = 6.4 \times 10^{-2}$ is used but the iteration count drops to 662.

Table I. The effects of increasing rsize having fixed lsize $= 5$. The iteration counts are reported and, in parentheses, the number of shifts used and the final shift.

Problem	rsize $= 0$	rsize $= 5$	rsize $= 10$
Boeing/msc01440	142 (3, 8.0×10^{-3})	119 (1, 1.0×10^{-3})	130 (0, 0.0)
HB/bcsstk17	227 (4, 3.2×10^{-2})	209 (3, 1.6×10^{-2})	172 (2, 8.0×10^{-3})
Overwolfach/gyro-m	230 (4, 3.2×10^{-2})	230 (3, 8.0×10^{-3})	193 (0, 0.0)
GHS_psdef/hood	762 (6, 6.4×10^{-2})	662 (6, 6.4×10^{-2})	783 (4, 3.2×10^{-2})

4.3. Effect of scaling

The importance of scaling in the solution of linear systems is well known. In Figure 2, we illustrate how scaling effects the performance of HSL_MI28. The code is run with lsize $=$ rsize $= 10$, drop tolerances tau1 $= 0.001$, tau2 $= 0$, and no ordering. We see that, considering the test set as a whole (a set that contains some problems that are initially well-scaled and others that are initially poorly scaled), scaling can significantly improve performance. Interestingly, each of the tested scalings gives a preconditioner of similar quality. For each, there was at most two problems for which we failed to achieve the requested accuracy within the iteration limit and the number of shifts used was at most four. If no scaling is used, the number of restarts following factorization breakdown can be large, the final shift can be large and the computed IC preconditioner of poor quality. There were ten failures to converge and up to 22 restarts were necessary using the default setting for the shift strategy. We remark that the studies by Hogg and Scott [2008; 2014] into the effects of scalings on the performance of direct solvers for the solution of sparse symmetric indefinite systems found MC64 to be expensive but it produced the highest quality scalings, particularly for some "tough" problems (problems that a direct solver can struggle to solve efficiently as well as accurately). MC64 has also been used to advantage in the non symmetric case (for example, Benzi, Haws and Tuma [2000] report on the beneficial effects of scalings to place large entries on the diagonal when computing incomplete factorization preconditioners for use with Krylov subspace methods). However, since it is expensive, does not significantly reduce the number of factorization breakdowns compared with the other scalings, and does not lead to higher quality IC preconditioners for our test problems, it is not the default scaling within HSL_MI28; the default is the much cheaper and simpler l_2 scaling.

4.4. Effect of dropping

Figure 3 presents an efficiency performance profile (on the left) and an iteration performance profile (on the right) for the drop tolerance tau1 in the range 0 (no threshold-based dropping) and 0.0005. Here we use lsize=rsize=5, l_2 scaling, no reordering and tau2 $= 0$ (so that small entries are dropped from L but dropping is not used on R). In terms of efficiency, it is clearly advantageous to use a small drop tolerance greater than zero but, if tau1 is increased too far, reliability of the computed preconditioner is reduced. Looking at the iteration performance profile, we see that, as expected, dropping entries reduces the quality of the preconditioner but for small tau1 the number of extra iterations is generally modest. We have selected tau1 $= 0.001$ as the default setting in HSL_MI28. Provided the problem has been scaled, we found that also using a non zero value for tau2 has a relatively small but beneficial effect on the efficiency. We have chosen the default value to be tau2 $= 0.0001$.

4.5. Effect of ordering

The effects of sparse matrix orderings on the convergence of preconditioned Krylov subspace methods have been widely reported on in the literature (the survey by Benzi [2002], for example, contains a large number of references and a brief sum-

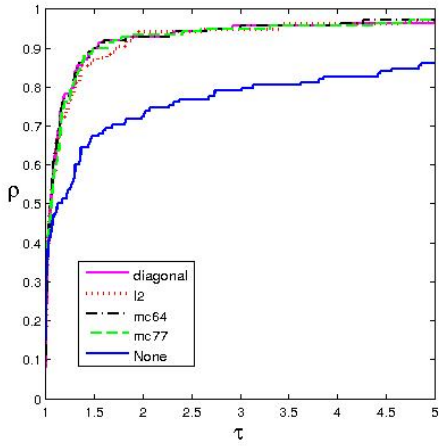

Fig. 2. Efficiency performance profile for different scalings.

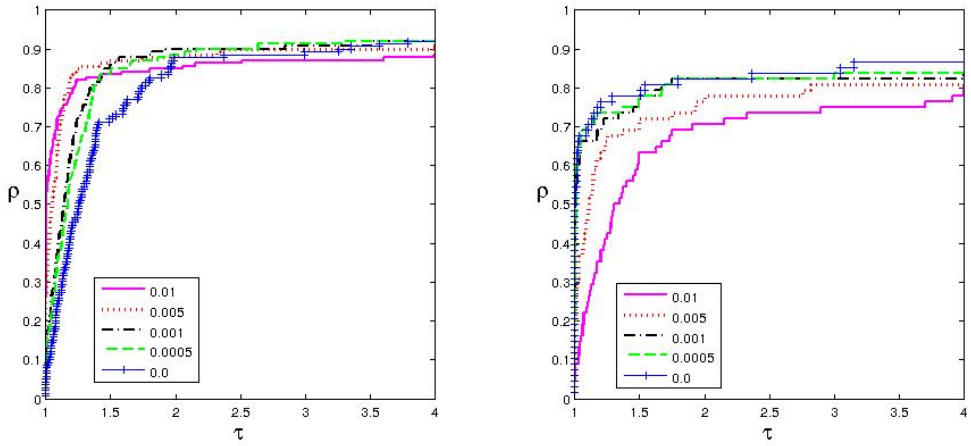

Fig. 3. Efficiency (left) and iteration count (right) performance profiles for `tau1` varying.

mary discussion). In Figure 4, we compare the performance of the ordering algorithms offered by HSL_MI28 (with the same settings as above and l_2 scaling). Our results agree with previously reported findings; in particular, minimum degree and nested dissection perform less well than the natural order and this, in turn, is outperformed by RCM and the Sloan algorithms. It is interesting to observe how well the Sloan algorithm does: it is significantly better (in terms of efficiency and reliability) than the more widely-used RCM ordering and, as its use adds very little to the time to compute the factorization, we have made it the default ordering within HSL_MI28. Closer examination of the results shows that the Sloan ordering gives the best results across the range of problems tested (in particular, it gives the same kind of overall performance improvement for the largest problems as for the smallest).

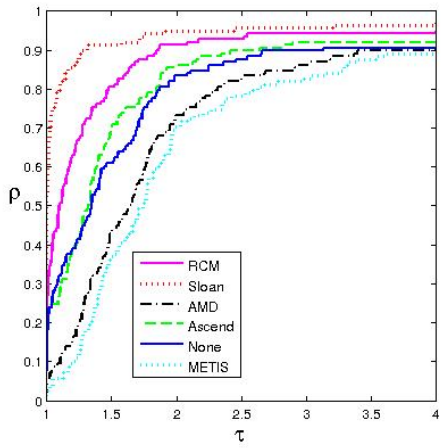

Fig. 4. Efficiency performance profile for different orderings. RCM is the reverse Cuthill-McKee bandwidth minimisation algorithm, Sloan is a profile reduction algorithm, AMD is an approximate minimum degree ordering, Ascend is ordering by ascending degree, METIS is a nested dissection ordering, and None indicates the supplied (natural) ordering.

4.6. The shift strategy

We now consider the effects of varying the parameter `lowalpha` that controls the choice of the initial shift. We set `lsize` $=$ `rsize` $=$ `10` and default settings are used for the control parameters. For these parameter choices and initial shift $\alpha_0 = 0$, convergence is achieved for all but one of our test problems and for 60 of the problems a non zero shift is needed. If $\alpha > 0$ is required, at least two shifts are tested. This is because $\alpha =$`lowalpha` is first tried and, as discussed in Section 3.2.5, if the factorization is breakdown free, by default we reduce α by a factor of `shift_factor` and restart the factorization. For 30 out of the 60 problems only two shifts are required; for 24 problems, three shifts are needed and for the remainder four shifts are used.

Table II. The effects of varying `lowalpha`. The iteration counts are reported and, in parentheses, the number of shifts used and the final shift.

Problem	`lowalpha` $=$ `0.01`	`lowalpha` $=$ `0.001`	`lowalpha` $=$ `0.0001`
Boeing/msc01440	40 (4, 1.56×10^{-4})	40 (3,2.50×10^{-4})	39 (2, 2.00×10^{-4})
FIDAP/ex33	427 (2, 1.00×10^{-2})	415 (3, 8.00×10^{-3})	460 (5, 1.28×10^{-2})
HB/bcsstk17	112 (3, 2.50×10^{-3})	151 (3, 8.00×10^{-3})	136 (5, 6.40×10^{-3})
GHS_psdef/oilpan	817 (3, 2.50×10^{-3})	698 (2, 1.00×10^{-3})	708 (3, 1.60×10^{-3})
Um/offshore	69 (2, 1.00×10^{-2})	71 (3, 1.60×10^{-2})	72 (4, 6.40×10^{-3})

A subset of the problems that require a non zero shift are reported on in Table II. The subset was chosen to illustrate different behaviours. It is clear that there is no single best choice for `lowalpha`: if it is small, it may be necessary to increase the shift several times to avoid breakdown (for example, FIDAP/ex33 with `lowalpha` $=$ `0.0001`), while if it is too large, a number of reductions may need to be made so that an unnecessarily large shift is avoided (for example, Boeing/msc01440 with `lowalpha` $=$ `0.01`). For some problems (including GHS_psdef/oilpan) the iteration count is sensitive to the choice of shift while for others (such as Um/offshore) this is not the case. Based on our experiments, we have selected `lowalpha` $=$ `0.001` as the default value (but as it is a control parameter, users can easily experiment with other values to try and limit the number

of shifts used for their problems). Note that this choice is made on the assumption that the problem has been scaled; if the problem is not well-scaled, we found larger shifts were needed and so a larger `lowalpha` could be advantageous.

As already noted, with our default settings, half of our test problems that required a non zero shift used $\alpha =$ `lowalpha` $=$ `0.001` and the time spent in trying a reduced value was wasted. It is thus of interest to consider whether allowing the possibility of reducing α below its initial non-zero value is beneficial. If the control parameter `maxshift` is set to zero, there is no attempt to reduce the shift if $\alpha =$ `lowalpha` yields a breakdown-free factorization. Table III presents results for `maxshift` $=$ `0` and `3`. We also ran `maxshift` $=$ `2` and `4` and found, for the problems tested, that these gave the same results as `maxshift` $=$ `3`. This is not guaranteed and depends on how rapidly α is decreased (controlled by `shift_factor2`). We see that for some examples (including ND/nd3k and GHS_psdef/shipsec8) the iteration count is significantly reduced by reducing α but others are not sensitive (GHS_psdef/bcsstk13)

Table III. The effects of varying `maxshift`. The iteration counts are reported and, in parentheses, the number of shifts used and the final shift.

Problem	`maxshift` $= 0$	`maxshift` $= 3$
Boeing/msc01440	47 (1, 1.00×10^{-3})	40 (3, 2.50×10^{-4})
GHS_psdef/bcsstk13	54 (1, 1.00×10^{-3})	54 (4, 1.56×10^{-4})
ND/nd3k	208 (1, 1.00×10^{-3})	135 (3, 2.50×10^{-4})
GHS_psdef/shipsec8	883 (1, 1.00×10^{-3})	661 (3, 2.50×10^{-4})

4.7. Comparison with a level-based approach

In an earlier paper, Scott and Tůma [2011] considered level-based preconditioning and presented an improved strategy that considers the individual entries of the system matrix and restricts small entries to contributing to fewer levels of fill than the largest entries. Their numerical results showed that preassigning levels of fill can be beneficial. In Figure 5, we present an efficiency performance profile (on the left) and an

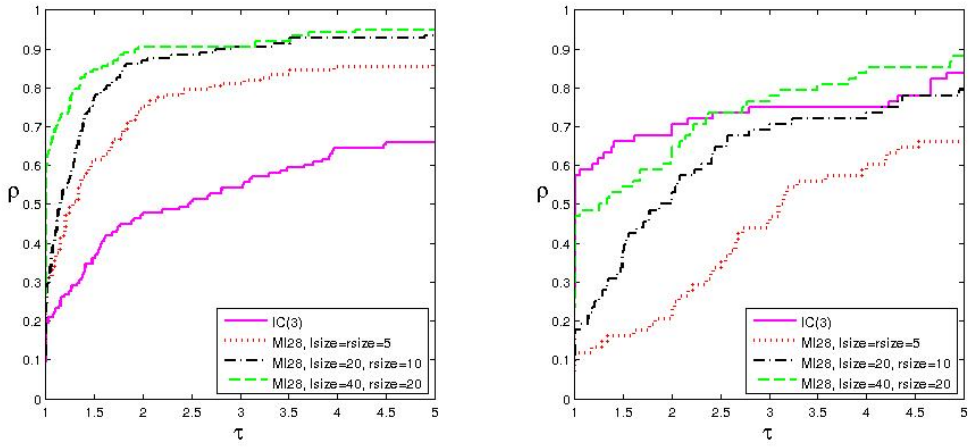

Fig. 5. Efficiency and iteration performance profiles for level-based and memory-based methods.

iteration performance profile (on the right) that compare the performance of the level-based preconditioner $IC(\ell)$ with $\ell = 3$ with that of HSL_MI28. The settings for computing $IC(3)$ were those found in [Scott and Tůma 2011] to give the best performance. For $IC(3)$ and HSL_MI28 we use l_2 scaling and no reordering. We see that the HSL_MI28 preconditioner is more efficient than $IC(3)$. However, while the $IC(3)$ incomplete factor is much denser than that produced by HSL_MI28, it requires fewer iterations. In terms of reliability, $IC(3)$ failed to give convergence for 19 problems, while with lsize $=$ rsize $=$ 5, the HSL_MI28 preconditioner failed on 8 problems and with lsize $=$ 20, rsize $=$ 10 and lsize $=$ 40, rsize $=$ 20 all the problems were solved. These findings suggest that the use of intermediate memory can have a positive effect compared with using the concept of levels. However, as level-based factors are denser and are more structured and they offer the possibility of being computed in parallel [Hysom and Pothen 2001], they remain a useful approach to incomplete factorization.

4.8. Comparisons with a direct solver

It is of interest to examine how well the preconditioned conjugate method performs in comparison with a modern direct solver. In Figure 6, we present total time performance profiles that compare HSL_MI28 with the ICFS code of Lin and Moré [1999] and the modern direct solver HSL_MA97 [Hogg and Scott 2011] (Version 2.1.0). Although

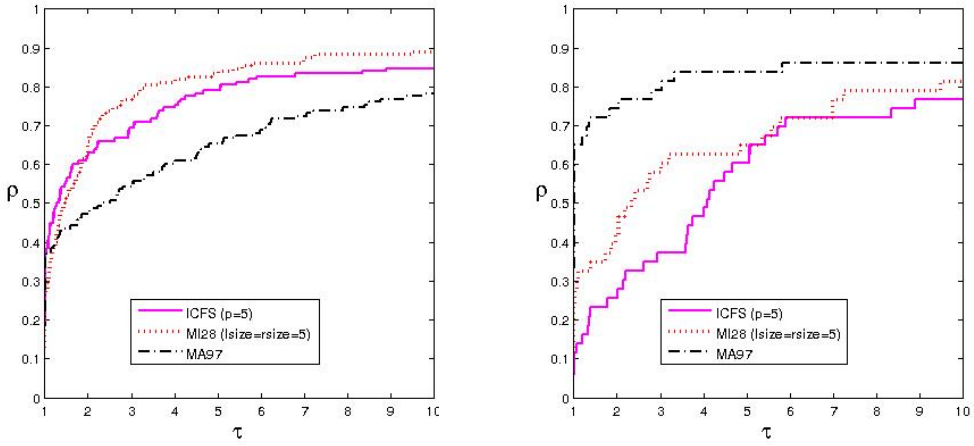

Fig. 6. Total time performance profile for direct and iterative methods (all problems on left, large problems only on right).

HSL_MA97 is a parallel OpenMP code, here we run it in serial (since our current incomplete factorization code is a serial code); all parameter settings within HSL_MA97 are the default settings and we use the Intel MKL BLAS (10.3.0). ICFS is run with $p = 5$ (a maximum of 5 fill entries in each column of L) and for HSL_MI28 we use lsize=rsize=5 plus default settings for the control parameters. The left-hand profile includes all problems in our test set while that on the right is restricted to the set of 43 problems for which the total ICFS time is at least 1 second. On small problems, ICFS performs well; it is a much simpler code than the others and includes no overhead for ordering or for the other options that are available within HSL_MI28. For larger problems, we see the benefits of incorporating reordering and using rsize $>$ 0, with HSL_MI28 performing significantly better than ICFS. For these problems, the direct solver is clearly the best

approach; it employs a block algorithm and is able to make extensive use of Level 3 BLAS to enhance performance. However, memory limitations mean HSL_MA97 was unable to solve two of the largest problems that were successfully solved by HSL_MI28 and ICFS. Thus the results demonstrate the potential of HSL_MI28 to be used to efficiently solve problems that are too large to be tackled by a direct solver but also suggests a need to incorporate techniques used by the direct solver into the incomplete factorization (see, for example, [Gupta and George 2010]).

5. CONCLUDING REMARKS

In this paper, we have discussed the design and development of a new software package HSL_MI28 for incomplete Cholesky factorizations. The algorithm implemented within HSL_MI28 is based on the robust Tismenetsky-Kaporin scheme, which we have made into a practical approach through the introduction of limited memory for both the incomplete factor L and the lower triangular matrix R that is used to stabilise the factorization but is subsequently discarded. Through the use of R and the option to drop entries by size, HSL_MI28 generalizes the ICFS code. Because our previous work [Scott and Tůma 2013] found that, in many instances, the use of a diagonal compensation scheme to prevent factorization breakdown can lead to a poor quality preconditioner, HSL_MI28 follows ICFS in using a global diagonal shift.

HSL_MI28 can be used as a black box package. However, through the inclusion of control parameters that have default settings but that can be reset by the user, the design is such that the experienced user can tune the code to optimise its performance on his or her problems of interest.

We have presented the results of extensive numerical experiments. In our opinion, using a large test set drawn from a range of practical applications allows us to make general and authoritative conclusions. The experiments emphasize the concept of the efficiency of a preconditioner (defined by (1)) as a measure that can be employed to help capture preconditioner usefulness, although we recognise that it can also be necessary to consider other statistics (such as the time and the number of iterations). The experiments have been used to isolate the effects of ordering, scaling and dropping. An interesting observation is that the ordering that gives the best results overall is the Sloan profile reduction algorithm, rather than the more widely-used RCM ordering.

Our results suggest that Tismenetsky-type updates are better able to stabilise the decomposition than the concept of levels (even in the enhanced form introduced recently in [Scott and Tůma 2011]). Thus we conclude that memory-limited decompositions are not only possible but are also highly effective and a potentially useful addition to the armoury of tools for solving sparse linear systems.

HSL_MI28 is available as part of the 2013 release of the HSL mathematical software library. All use of HSL requires a licence; licences are available to academics without charge for individual research and teaching purposes. Details of how to obtain a licence and the code are available at http://www.hsl.rl.ac.uk or by email to hsl@stfc.ac.uk.

ACKNOWLEDGMENTS

We would like to thank three anonymous reviewers and the associate editor for carefully reading our manuscript and making constructive suggestions for its improvement.

REFERENCES

Patrick R. Amestoy, Timothy A. Davis, and Iain S. Duff. 2004. Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Trans. Math. Software* (2004), 381–388.

Owe Axelsson, Igor Kaporin, Igor Konshin, Andrey Kucherov, Maya Neytcheva, Ben Polman, and Alex Yeremin. 2000. *Comparison of algebraic solution methods on a set of benchmark problems in linear*

elasticity. Final Report of the STW project NNS.4683. Department of Mathematics, University of Nijmegen.

Owe Axelsson and Niels Munksgaard. 1983. Analysis of incomplete factorizations with fixed storage allocation. In *Preconditioning Methods: Analysis and Applications*. Topics in Computational Mathematics, Vol. 1. Gordon & Breach, New York, 219–241.

Lourenco Beirão da Veiga, Vitaliy Gyrya, Konstantin Lipnikov, and Gianmarco Manzini. 2009. Mimetic finite difference method for the Stokes problem on polygonal meshes. *J. Comput. Phys.* 228, 19 (2009), 7215–7232.

Michele Benzi. 2002. Preconditioning techniques for large linear systems: a survey. *J. Comput. Phys.* 182, 2 (2002), 418–477.

Michele Benzi, John C. Haws, and Miroslav Tůma. 2000. Preconditioning highly indefinite and nonsymmetric matrices. *SIAM J. Sci. Comput.* 22, 4 (2000), 1333–1353.

Michele Benzi and M. Tůma. 2003. A robust incomplete factorization preconditioner for positive definite matrices. *Numer. Linear Algebra Appl.* 10, 5-6 (2003), 385–400.

Elisabeth H. Cuthill and James McKee. 1969. Reducing the bandwidth of sparse symmetric matrices. In *Proc.* 24^{th} *National Conference of the ACM*. ACM Press, 157–172.

Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Software* 38 (2011). Article 1, 25 pages.

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

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

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

Roland W. Freund and Nöel M. Nachtigal. 1990. *An implementation of the look-ahead Lanczos algorithm for non-Hermitian matrices, part II.* Technical Report TR 90-46. RIACS, NASA Ames Research Center.

Thomas George, Anshul Gupta, and Vivek Sarin. 2009, revised 2011. *An empirical analysis of the performance ofp reconditioners for SPD systems*. Research Report RC 24737. IBM.

Thomas George, Anshul Gupta, and Vivek Sarin. 2012. An empirical analysis of the performance of preconditioners for SPD systems. *ACM Trans. Math. Software* 38 (2012). Article 24, 30 pages.

Nicholas I. M. Gould, Yifan F. Hu, and Jennifer A. Scott. 2007. A numerical evaluation of sparse direct symmetric solvers for the solution of large sparse, symmetric linear systems of equations. *ACM Trans. Math. Software* 33 (2007). Article 10, 32 pages.

Nicholas I. M. Gould and Jennifer A. Scott. 2004. A numerical evaluation of HSL packages for the direct solution of large sparse, symmetric linear systems of equations. *ACM Trans. Math. Software* 30, 3 (2004), 300–325.

Anshul Gupta and Thomas George. 2010. Adaptive techniques for improving the performance of incomplete factorization preconditioning. *SIAM J. Sci. Comput.* 1 (2010), 84–110.

Jonathan D. Hogg and Jennifer A. Scott. 2008. *The effects of scalings on the performance of a sparse symmetric indefinite solver*. Technical Report RAL-TR-2008-007. Rutherford Appleton Laboratory.

Jonathan D. Hogg and Jennifer A. Scott. 2011. *HSL_MA97: a bit-compatible multifrontal code for sparse symmetric systems*. Technical Report RAL-TR-2011-024. Rutherford Appleton Laboratory.

Jonathan D. Hogg and Jennifer A. Scott. 2014. Pivoting strategies for tough sparse indefinite systems. *ACM Trans. Math. Software* (2014). To appear.

HSL. 2013. A collection of Fortran codes for large-scale scientific computation. (2013). http://www.hsl.rl.ac.uk.

David Hysom and Alex Pothen. 2001. A scalable parallel algorithm for incomplete factor preconditioning. *SIAM J. Sci. Comput.* 22 (2001), 2194–2215.

Alan Jennings and George M. Malik. 1977. Partial elimination. *J. Inst. Math. Appl.* 20, 3 (1977), 307–316.

Alan Jennings and George M. Malik. 1978. The solution of sparse linear equations by the conjugate gradient method. *Internat. J. Numer. Methods Engrg.* 12, 1 (1978), 141–158.

Mark T. Jones and Paul E. Plassmann. 1995a. Algorithm 740: Fortran subroutines to compute improved incomplete Cholesky factorizations. *ACM Trans. Math. Software* 21, 1 (1995), 18–19.

Mark T. Jones and Paul E. Plassmann. 1995b. An improved incomplete Cholesky factorization. *ACM Trans. Math. Software* 21, 1 (1995), 5–17.

Igor E. Kaporin. 1998. High quality preconditioning of a general symmetric positive definite matrix based on its $U^TU + U^TR + R^TU$ decomposition. *Numer. Linear Algebra Appl.* 5 (1998), 483–509.

George Karypis and Vipin Kumar. October 1997. *METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices (Version 3.0)*. Technical Report. University of Minnesota, Department of Computer Science and Army HPC Research Center.

Chih-Jen Lin and Jorge J. Moré. 1999. Incomplete Cholesky factorizations with limited memory. *SIAM J. Sci. Comput.* 21, 1 (1999), 24–45.

Konstantin Lipnikov, Mikhail Shashkov, Daniil Svyatskiy, and Yuri Vassilevski. 2007. Monotone finite volume schemes for diffusion equations on unstructured triangular and shape-regular polygonal meshes. *J. Comput. Phys.* 227, 1 (2007), 492–512.

Konstantin Lipnikov, Daniil Svyatskiy, and Yuri Vassilevski. 2009. Interpolation-free monotone finite volume method for diffusion equations on polygonal meshes. *J. Comput. Phys.* 228, 3 (2009), 703–716.

Thomas A. Manteuffel. 1980. An incomplete factorization technique for positive definite linear systems. *Math. Comp.* 34 (1980), 473–497.

John K. Reid and Jennifer A. Scott. 1999. Ordering symmetric sparse matrices for small profile and wavefront. *Internat. J. Numer. Methods Engrg.* 45 (1999), 1737–1755.

Daniel Ruiz. 2001. *A scaling algorithm to equilibrate both rows and columns norms in matrices*. Technical Report RAL-TR-2001-034. Rutherford Appleton Laboratory, Chilton, Oxfordshire, England.

Daniel Ruiz and Bora Uçar. 2011. *A symmetry preserving algorithm for matrix scaling*. Technical Report INRIA RR-7552. INRIA, Grenoble, France.

Yousef Saad. 1994. ILUT: a dual threshold incomplete LU factorization. *Numer. Linear Algebra Appl.* 1, 4 (1994), 387–402.

Yousef Saad. 1996. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Co., Boston.

Jennifer A. Scott and Miroslav Tůma. 2011. The importance of structure in incomplete factorization preconditioners. *BIT* 51 (2011), 385–404.

Jennifer A. Scott and Miroslav Tůma. 2013. *On positive semidefinite modification schemes for incomplete Cholesky factorization*. Technical Report RAL-TR-2013-P-005. Rutherford Appleton Laboratory.

Scott W. Sloan. 1986. An algorithm for profile and wavefront reduction of sparse matrices. *Internat. J. Numer. Methods Engrg.* 23 (1986), 239–251.

Scott W. Sloan. 1989. A Fortran program for profile and wavefront reduction. *Internat. J. Numer. Methods Engrg.* 28 (1989), 2651–2679.

Made Suarjana and Kincho H. Law. 1995. A robust incomplete factorization based on value and space constraints. *Internat. J. Numer. Methods Engrg.* 38 (1995), 1703–1719.

Miron Tismenetsky. 1991. A new preconditioning technique for solving large sparse linear systems. *Linear Algebra Appl.* 154–156 (1991), 331–353.

Ichitaro Yamazaki, Zhaojun Bai, Wenbin Chen, and Richard Scalettar. 2009. A high-quality preconditioning technique for multi-length-scale symmetric positive definite linear systems. *Numerical Mathematics: Theory, Methods and Applications* 2, 4 (2009), 469–484.