Scaling and Pivoting in an Out-of-Core Sparse Direct Solver

JENNIFER A. SCOTT
Rutherford Appleton Laboratory

Out-of-core sparse direct solvers reduce the amount of main memory needed to factorize and solve large sparse linear systems of equations by holding the matrix data, the computed factors, and some of the work arrays in files on disk. The efficiency of the factorization and solution phases is dependent upon the number of entries in the factors. For a given pivot sequence, the level of fill in the factors beyond that predicted on the basis of the sparsity pattern alone depends on the number of pivots that are delayed (i.e., the number of pivots that are used later than expected because of numerical stability considerations). Our aim is to limit the number of delayed pivots, while maintaining robustness and accuracy. In this article, we consider a new out-of-core multifrontal solver `HSL_MA78` from the HSL mathematical software library that is designed to solve the unsymmetric sparse linear systems that arise from finite element applications. We consider how equilibration can be built into the solver without requiring the system matrix to be held in main memory. We also examine the effects of different pivoting strategies, including threshold partial pivoting, threshold rook pivoting, and static pivoting. Numerical experiments on problems arising from a range of practical applications illustrate the importance of scaling and show that, in some cases, rook pivoting can be more efficient than partial pivoting in terms of both the factorization time and the sparsity of the computed factors.

Categories and Subject Descriptors: G.1.0 [**Numerical Analysis**]: General—*Numerical algorithms*; G.4 [**Mathematical Software**]

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Large sparse unsymmetric linear systems, element problems, out-of-core solver, multifrontal, rook pivoting, partial pivoting, scaling

ACM Reference Format:

Scott, J. A. 2010. Scaling and pivoting in an out-of-core sparse direct solver. ACM Trans. Math. Softw. 37, 2, Article 19 (April 2010), 23 pages.
DOI = 10.1145/1731022.1731029 http://doi.acm.org/10.1145/1731022.1731029

This work was funded by the EPSRC Grant EP/E053351/1.
Author's address: J. A. Scott, Computational Science and Engineering Department, Atlas Centre, Rutherford Appleton Laboratory, Oxon OX11 0QX, U.K.; email: `j.a.scott@stfc.ac.uk`.

DOI 10.1145/1731022.1731029 http://doi.acm.org/10.1145/1731022.1731029

1. INTRODUCTION

Most direct methods for solving large sparse linear systems of equations $AX = B$ are variants of Gaussian elimination, involving a factorization $PAQ = LDU$ of the system matrix A, where L is unit lower triangular, U is unit upper triangular, D is diagonal, and P and Q are permutation matrices. The solution process is completed by performing forward and then back substitutions (i.e., by first solving a lower triangular system and then an upper triangular system). Direct methods are popular because, when properly implemented, they are generally robust and so can be used as general-purpose black-box solvers for a wide range of problems. One of their major limitations is that the memory they require normally increases rapidly with problem size. In recent years, this has led to an interest in the development of out-of-core solvers, that is, solvers that hold the system matrix A and its factors (and possibly some of the auxillary data structures used by the solver) in files (see, e.g., Reid and Scott [2009b, 2009c]; Rothberg and Schreiber [1999]; Rotkin and Toledo [2004]). This allows much larger problems to be solved than would otherwise be possible. The main disadvantage is the cost involved in reading from and writing to files held on disk and it is essential that this be done efficiently; how we achieve this within our recent out-of-core solvers is described in detail in Reid and Scott [2009a].

As well as efficiently handling the input/output operations, it is important to limit the need for such operations by minimising the number of entries in the factors. The purpose of this article is to examine the effects of scaling and of different pivoting strategies on the fill-in of the factors computed using an unsymmetric out-of-core solver `HSL_MA78` [Reid and Scott 2009b] that we have developed for the mathematical software library HSL [HSL 2007].

`HSL_MA78` is a Fortran 95 package that is designed to solve efficiently the large sparse systems that arise from finite-element problems. These systems are unsymmetric but are symmetric in structure. In common with other direct solvers, `HSL_MA78` has a number of distinct phases. A pivot sequence must first be chosen, that is, the order in which the eliminations are to be carried out. For the chosen pivot sequence, the analyse phase (`MA78_analyse`) predicts the nonzero pattern of the factors using the symmetric sparsity pattern of A. `MA78_analyse` determines lower bounds on the number of entries in the matrix factors, the memory required by the factorization, and the number of operations needed to compute the factors. The factorize phase (`MA78_factor`) computes the numerical factorization using the data structures set up by the analyse phase. The forward and back substitutions are performed by the solve phase (`MA78_solve`), which may be called repeatedly for different right-hand sides B. There is also an option to solve transpose systems $A^T X = B$. In general, to maintain numerical stability, it is necessary to modify the pivot sequence during the factorize phase, delaying small pivots until alternatives are available or they are safer to use. These delayed pivots cause additional fill-in of the factors beyond the lower bound computed by the analyse phase and lead to extra work in both the factorize and solve phases. In the case of an out-of-core solver, the extra work is not just an increase in the flop count, but also an increase in the number of input/output operations. We are therefore interested in trying to limit the

number of delayed pivots, while maintaining the robustness and accuracy of our solver.

The first technique we use to try and limit delayed pivots is scaling. How to find a good scaling is an open question, but a number of scalings have been proposed and are widely used. In particular, there are a number of scaling routines available within the HSL library. Unfortunately, these require that the matrix A be held in main memory. For very large problems, A may not fit into main memory and so we want to consider how we can scale A while it is held out-of-core. In particular, we implement an out-of-core equilibration algorithm. Equilibration [Ruiz 2001] is a particular form of scaling, where the rows and columns of a matrix are modified so that they have approximately the same norm. In addition to equilibration, we look at the effects of different pivoting strategies. Within `HSL_MA78` we have included options for threshold partial pivoting, threshold rook pivoting, and static pivoting. In the LDU factors, partial pivoting tends to control the condition of L, while rook pivoting controls that of both L and U.

The remainder of this article is organized as follows. In Section 2, we give a brief introduction to the out-of-core multifrontal method that is implemented within our solver. We then discuss, in Section 3, how we can incorporate an equilibration algorithm within our multifrontal solver, avoiding the need to assemble the system matrix A in main memory. In Section 4, we consider the dense linear algebra kernels that lie at the heart of `HSL_MA78` and explain the pivoting options that are offered. Numerical results for a range of practical problems are presented in Section 5 and concluding remarks are made in Section 6.

2. INTRODUCTION TO AN OUT-OF-CORE MULTIFRONTAL METHOD

Our unsymmetric out-of-core solver `HSL_MA78` is designed for solving problems coming from finite-element analysis in which the $n \times n$ matrix A can be expressed in the form

$$A = \sum_{k=1}^{nelt} A^{(k)}. \tag{2.1}$$

Here *nelt* is the number of elements in the model and $A^{(k)}$ corresponds to the contribution from element k and has nonzeros in only a small number of rows and columns. In practice, each $A^{(k)}$ is held as a small square dense matrix, called an *element matrix*. A list of the global indices of the variables associated with element k, which identifies where the entries in $A^{(k)}$ belong in A, must also be held. Each $A^{(k)}$ is symmetrically structured (the list of indices is both a list of column indices and a list of row indices) but, in the general case, is numerically unsymmetric. The advantage of holding the matrix A as a sum of element matrices is that it is possible to avoid assembling (and thus the storage for) A. Instead, the assembly and elimination operations are interleaved and the main work is performed using a dense *frontal matrix* that is significantly smaller than A: this is the key idea behind the frontal method [Irons 1970] and, more generally, multifrontal methods.

At each stage of the frontal method, the frontal matrix is a square matrix of order $m \ll n$ that may be expressed in the form

$$F = \begin{pmatrix} F_1 & F_2 \\ F_3 & F_4 \end{pmatrix}, \tag{2.2}$$

where F_1, F_2, and F_3 are *fully summed*, that is, all the entries in the corresponding part of A have already been assembled, while is F_4 is not yet fully summed. If F_1 has order p and p pivots can be chosen stably from F_1, the *partial factorization* of F takes the form

$$F = \begin{pmatrix} P_1 & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} L_1 & 0 \\ L_2 & I \end{pmatrix} \begin{pmatrix} D_1 & 0 \\ 0 & F_S \end{pmatrix} \begin{pmatrix} U_1 & U_2 \\ 0 & I \end{pmatrix} \begin{pmatrix} Q_1 & 0 \\ 0 & I \end{pmatrix}, \tag{2.3}$$

where P_1 and Q_1 are permutation matrices, L_1 and U_1 are unit lower and unit upper triangular matrices, and D_1 is a diagonal matrix, all of order p. The Schur complement F_S is given by

$$F_S = F_4 - L_2 D_1 U_2. \tag{2.4}$$

At the next stage of the computation, the contributions from another element are assembled with the Schur complement to form a new frontal matrix; the process continues until all the element matrices have been assembled and the final elimination operations are performed. The matrices L_i, U_i $(i = 1, 2)$, D_1, and the permutation matrices P_1 and Q_1, are part of the factorization and are not needed again until the forward and back substitutions are performed. Thus, as they are generated, they can be transfered to a file. The data in these files is read into main memory as it is required (one record at a time) during the forward and back substitution phases. The original element data may also be held in files. In Fortran, it is convenient to use separate files for the real data and for the integer data.

In the frontal method, there is a single front (that is, there is a single set of variables that have not yet been eliminated but are involved in one or more of the elements that have been assembled). Duff and Reid [1983] extended the frontal concept to use more than one front and, reflecting the use of several fronts, their generalization is a *multifrontal* method. For each pivot in turn, all the elements that contain the pivot are assembled into the frontal matrix and a partial factorization is performed. The computed entries of the factors are stored and the Schur complement matrix F_S is treated as a new element, called a *generated element* (the term *contribution block* is also used in the literature). The generated element is added to the set of unassembled elements and the next uneliminated pivot then considered. The basic algorithm is summarised in Figure 1.

In the case we are interested in, the symmetrically structured A is unsymmetric and the key difference between the original elements and the generated elements is that, for the latter, it is necessary to hold an integer list of *both* the row indices and the column indices of the variables in the front. This is because the original symmetric structure is lost by choosing off-diagonal pivots during the partial factorizations of the frontal matrices and so the generated elements are held as square frontal matrices with an unsymmetric structure.

```
Basic Multifrontal Factorization
do for each pivot in the given pivot sequence
    if the pivot has not yet been eliminated
        assemble all unassembled elements and generated elements that
            contain the pivot into a square frontal matrix;
        perform a partial factorization of the frontal matrix;
        add the generated element to the set of elements
    end if
end do
```

Fig. 1. A basic multifrontal factorization.

The assemblies can be recorded as a tree, called an *assembly* tree. Each leaf node represents an original element and each non-leaf node v corresponds to an assembly and subsequent eliminations. The children of v are the original elements together with the generated elements that contain the variables eliminated at node v. If A is structurally irreducible there will be a single *root* node, that is, a node with no parent. Otherwise, the matrix may be permuted to block diagonal form, with one root for each block.

The partial factorization of the frontal matrix at a node v in the tree can be performed once the partial factorizations at all the nodes belonging to the subtree rooted at v are complete. If the nodes of the tree are ordered using a depth-first search, the generated elements required at each stage are the most recently generated ones of those so far unused. This makes it convenient to use a stack for temporary storage during the factorization. This, of course, alters the order in which the pivots are selected, but the arithmetic is identical apart from the round-off effects of reordering the summations into the frontal matrices and the effects that this can have on later computations. The stack can be held out of core in files.

In general, to obtain a numerically stable factorization, it is necessary to incorporate numerical pivoting. This may mean that $q \leq p$ pivots are chosen at non-root nodes and the matrices L_1, U_1, and D_1 in (2.3) are then of order q, while the permutation matrices P_1 and Q_1 remain of order p. In this case, $p-q$ pivots are *delayed* and the generated element that is passed to the parent node will be larger than anticipated on the basis of the sparsity pattern alone. This larger generated element may be expressed as

$$F_G = \begin{pmatrix} F_{G1} & F_{G2} \\ F_{G3} & F_{S1} \end{pmatrix}, \tag{2.5}$$

where the order of the leading square submatrix F_{G1} is $p-q$ and F_{S1} has the same order as the matrix F_S in the partial factorization (2.4). Extra rows and columns are prepended to the frontal matrix at the parent node to accommodate the delayed pivots from each of its children; the number of extra rows and columns is equal to the total number of delayed pivots from the children. At the parent, the delayed pivots are again tested as pivot candidates and, if necessary, are passed further up the tree until they become safe to use. Further details are given in Reid and Scott [2009b]. Since delayed pivots lead to the computed L and U factors containing more fill-in, our interest lies in reducing the number

of delayed pivots by scaling (Section 3) and by the choice of pivoting algorithm (Section 4) while maintaining accuracy.

We end this section by noting that HSL_MA78 provides an implementation of the classical multifrontal method that has been described here and, although this method can be extended to assembled systems, HSL_MA78 is designed exclusively for element problems. Other multifrontal algorithms for unsymmetric systems have been proposed, in particular, the well-known UMPACK package [Davis 2004] (and its predecessor MA38 [Davis and Duff 1997]) implements a combined unifrontal/multifrontal approach for assembled systmes. This does not require structural symmetry and uses rectangular frontal matrices. Furthermore, as in the frontal method for unsymmetric assembled systems (see, e.g., [Duff 1984] and the code MA42 [Duff and Scott 1996]), it avoids the need to delay pivots. Although the frontal code MA42 offers the option to hold the matrix factors in files, UMPACK has not currently been extended to the out-of-core case.

3. SCALING WITHIN A MULTIFRONTAL ALGORITHM

There has been much work on the effects of scaling and equilibration on the stability and accuracy of LU factorizations of general matrices. For example, Skeel [1979] concluded that no systematic scaling can be derived for general matrices that is always successful; others have also suggested that the benefits of scaling are limited so that it is often the practice to scale matrices on a case-by-case basis. To enable users to experiment with different scalings, the HSL mathematical software library [HSL 2007] offers a number of packages that are designed to compute scalings of large sparse matrices. From our point of view, the main limitation of these packages is that they all require the user to supply the matrix in assembled form, that is, the nonzero entries of A must be entered on a single call using either coordinate format or compressed column format; there are no facilities for working out-of-core or for matrices that are held in the form (2.1). Thus, as we wish to avoid assembly of A, the existing packages are not suitable for use with our solver HSL_MA78. In this section, we describe how we can incorporate an equilibration option into HSL_MA78, using only minimal additional memory.

The matrix

$$D_R^{-1} A D_C^{-1},$$

where D_R and D_C are diagonal matrices, is an equilibration of A if the norms of its rows and columns have approximately the same magnitude. Ruiz [2001] proposed defining the diagonal matrices to be

$$D_R = \mathrm{diag}\left(\sqrt{\max_j |A_{ij}|}\right) \quad \text{and} \quad D_C = \mathrm{diag}\left(\sqrt{\max_i |A_{ij}|}\right).$$

Ruiz also suggests applying the equilibration process iteratively, as summarized in Figure 2.

This algorithm computes the scaling diagonal matrices $D_1^{(k)}$ and $D_2^{(k)}$ such that the infinity norm of each row and column of $A^{(k)} = (D_1^{(k)})^{-1} A (D_2^{(k)})^{-1}$ tends

Equilibration algorithm

$A^{(1)} = A, \quad D_1^{(1)} = I, \quad \text{and } D_2^{(1)} = I$

for $k = 2, 3, \ldots,$ until termination do :

$$D_R = \text{diag}\left(\sqrt{\max_j |A_{ij}^{(k)}|}\right), \quad \text{and } D_C = \text{diag}\left(\sqrt{\max_i |A_{ij}^{(k)}|}\right)$$

$$A^{(k)} = D_R^{-1} A^{(k-1)} D_C^{-1}$$

$$D_1^{(k)} = D_1^{(k-1)} D_R \quad \text{and} \quad D_2^{(k)} = D_2^{(k-1)} D_C.$$

Fig. 2. Equilibration algorithm.

to 1 as $k \to +\infty$. The iteration is terminated when

$$\eta = \max_l \left(1 - \max_j |A_{lj}^{(k)}|, \;\; 1 - \max_i |A_{il}^{(k)}|\right) \leq \epsilon \tag{3.1}$$

for a given value of the tolerance $\epsilon \geq 0$.

The properties of this algorithm (for the infinity norm and for other norms) are discussed in Ruiz [2001]. The particular case when the infinity norm of each row and column of A is equal to 1 is clearly a fixed point for the equilibration algorithm. Furthermore, if, for each i, $|A_{ii}| \geq \max(\max_l |A_{il}|, \; \max_l |A_{li}|)$, then the algorithm converges in one iteration and the resulting scaled matrix has ones on the diagonal.

The equilibration algorithm is implemented for dense matrices and for sparse (assembled) matrices within the HSL package `MC77`. In `MC77`, the tolerance ϵ and the maximum number of iterations may be controlled by the user (with default settings of zero and 10, respectively).

We now look at how we can implement equilibration within `HSL_MA78`, avoiding the need to assemble A explicitly. We hold D_R and D_C as one-dimensional arrays of length n that we initialize to zero. Consider again the $m \times m$ frontal matrix $F = \{f_{ij}\}$ given by (2.2), in which the p rows of F_1 and F_2 and the p columns of F_1 and F_3 are fully summed. Together with the reals in F, an integer list *ind* of the global row and column indices of the variables in the front is held. Each row i of F is considered in turn. Let

$$D_R(ind_i) \leftarrow \max\left(D_R(ind_i), \;\; \max_{j \leq k} |f_{ij}|\right),$$

$$D_C(ind_j) \leftarrow \max\left(D_C(ind_j), \;\; \max_{i \leq k} |f_{ij}|\right),$$

where

$$k = \begin{cases} m, & \text{if } i \leq p, \\ p, & \text{otherwise.} \end{cases}$$

We refer to this searching and setting of the entries of the scaling matrices as *updating* D_R and D_C.

Once the search of the fully summed part of each row and each column is complete, F_1, F_2, and F_3 can be removed from the frontal matrix and what remains (F_4) can be stored in the same way as the generated element (2.4) is

```
Basic multifrontal equilibration algorithm
flag all pivots as uneliminated and initialise D_R and D_C to zero
do for each pivot in the chosen pivot sequence
    if the pivot is flagged as uneliminated
        assemble all unassembled elements and F_4-elements that contain
            the pivot into the frontal matrix F
            and let p be the number of fully summed rows and columns;
        update D_R and D_C with the largest entries in the fully summed part of
            each row and column of F;
        flag the variables corresponding to the first p rows and columns of F as
            eliminated and add the F_4-element to the set of elements
    end if
end do
```

Fig. 3. First iteration of an equilibration algorithm within a multifrontal algorithm.

stored during the multifrontal algorithm, that is, it can be placed on a stack. We call the $(m-p)\times(m-p)$ matrix that remains after the removal of the fully summed rows and columns an F_4-*element*.

Provided the elements (the original elements and the F_4-elements) are assembled in the order determined by the chosen pivot sequence, we can mimic what happens in the factorization phase of the multifrontal algorithm except that, at each stage, we perform no permutations and no actual elimination operations, but instead find the maximum entries in the fully summed part of each row and column. Note that, since no permutations are performed, the row and column indices of the variables in the front are the same and so a single list is needed. The first iteration of our multifrontal equilibration algorithm is outlined in Figure 3.

As in Figure 2, we can apply the algorithm iteratively and terminate when either we have satisfied the requested tolerance (3.1) or the maximum number of iterations has been reached. On the second and subsequent iterations, before they are searched for their largest entries, the entries in the fully summed part of F are scaled with the scaling factors computed so far, and the accumulated scaling factors are stored. Note that, although we have described the construction of the equilibration factors for the infinity norm, it is straightforward to extend the algorithm to other norms.

For an assembled matrix, the time taken to run an equilibration package such as `MC77` is, in general, small compared with the subsequent time needed for the numerical factorization. The main cost associated with implementing the equilibration algorithm within our multifrontal algorithm is that of accessing the element data. By default, the original element matrices supplied by the user are held in files and the F_4-elements are written to a temporary stack that is also held in a file; we read this data for each subsequent iteration. It is important, therefore, to limit the number of iterations and to perform only the minimum needed to obtain a sufficiently good equilibration. There is a tradeoff between the number of iterations and the quality of the equilibration, with the best choice being problem dependent. In our experiments (Section 5), we include results that illustrate this but remark that, in our tests, one iteration is often sufficient.

We observe that, on the first equilibration iteration, instead of discarding the fully summed parts of the frontal matrix (the matrices F_1, F_2, and F_3 in (2.2)) as soon as they have been searched and used to update D_R and D_C, we may write F_i $(i = 1, 2, 3)$ to a file. Once the iteration is complete, the whole of the assembled matrix will be held in the file. On subsequent iterations, it is then sufficient to pass through the assembly tree in the order determined by the pivot sequence, at each node reading the stored rows and columns of the assembled matrix back into the frontal matrix, scaling them with the scaling factors accumulated so far, and then updating D_R and D_C. Because this reduces the amount of data that must be read from file, it may reduce the cost of subsequent iterations but the overhead of writing the F_i at each node to file will increase the cost of the first iteration.

3.1 Equilibration Within HSL_MA78

The use of equilibration is optional within HSL_MA78. In Version 2.0.0, we have added an extra user-callable subroutine, MA78_scale, that may be called by the user after the analyse phase and before the factorize phase. The user can control the maximum number of iterations performed (the default is 1), the norm used (the one-norm or the infinity norm are offered with the default being the infinity norm), and the tolerance parameter ϵ. Based on our numerical experiments, we set the default value of ϵ to 0.5 (see Section 5.2). The scaling factors may be passed as an optional argument to the factorization subroutine MA78_factor. This allows the user to compute scaling factors using an alternative approach and then to pass them directly to MA78_factor.

MA78_scale includes an option to compute the infinity norm of the (unscaled) matrix A. At each stage of the first iteration of the equilibration algorithm, we accumulate the sum of the absolute values of the fully summed part of each row of F. Once the first iteration is complete (i.e., all pivots are flagged as eliminated), $\|A\|_\infty$ is computed to be the maximum norm of this work array.

4. PARTIAL FACTORIZATION OF THE FRONTAL MATRICES

The efficiency of MA78_factor is dependent upon the partial factorization of the frontal matrices. Since the frontal matrices are held as full matrices, dense linear algebra kernels may be used. We have developed a separate package, HSL_MA74, that is used by MA78_factor to perform the partial factorizations of the frontal matrices and by MA78_solve to perform the partial forward and back substitutions. In this section, we describe HSL_MA74 and discuss the pivoting options that it offers and that are available to users of the multifrontal solver HSL_MA78.

4.1 Overview of HSL_MA74

Given a dense unsymmetric $m \times m$ matrix F, HSL_MA74 performs a partial factorization, limiting eliminations to the leading $p \leq m$ rows and columns. Stability considerations may lead to $q \leq p$ eliminations being performed (i.e., fewer than p pivots are chosen). The factorization takes the form (2.3) where the matrices L_1, U_1, and D_1 are of order q and the permutation matrices P_1 and Q_1 are of

order p. Subroutines are provided for partial solutions, that is, solving systems of the form

$$\begin{pmatrix} L_1 & 0 \\ L_2 & I \end{pmatrix} X = B, \quad \begin{pmatrix} D_1 & 0 \\ 0 & I \end{pmatrix} X = B,$$

$$\begin{pmatrix} D_1 & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} U_1 & U_2 \\ 0 & I \end{pmatrix} X = B, \quad \text{and} \quad \begin{pmatrix} U_1 & U_2 \\ 0 & I \end{pmatrix} X = B,$$

and the corresponding equations for a single right-hand side b and solution x. Subroutines are also provided for partial solutions to transposed systems.

The user inputs the matrix F in a two-dimensional array which, on exit, is overwritten by the factorized matrix. Each diagonal entry holds either the inverse of a pivot or, if a zero pivot is chosen (because the matrix is singular), the corresponding diagonal entry is set to zero. A one-dimensional integer array `pperm` of length p is used to hold the row permutations P_1 so that, on exit, `pperm`(i) holds the index of the row of F that is permuted to row i, $i = 1, \ldots, p$. Similarly, `qperm` is used to hold the column permutations Q_1.

`HSL_MA74` uses a block algorithm. If the factorization were to proceed by choosing a single pivot at a time, the updates to the rest of F could only be performed using Level 2 BLAS [Dongarra et al. 1988]. To enhance performance by taking advantage of the more efficient Level 3 BLAS [Dongarra et al. 1990], the partial factorization is programmed as a sequence of block steps, with the block size nb under the user's control. If q is the number of pivots chosen so far, the code searches columns $q + 1$ to p of F in turn for a pivot. If the column to be searched has had $k < q$ updates, it is first updated with the $q - k$ most recently chosen pivots. Since a single column is being updated, this is performed using the Level 2 BLAS kernels `_trsv` and `_gemv`. Each time a pivot is chosen, q is incremented by 1 and the pivotal column is swapped with column q. The position m_1 of the right-most column of F that has been searched for a pivot is held and, whenever a pivot is chosen, columns $q + 1$ to m_1 are updated (using the Level 2 rank-1 update routine `_ger`) so that all the columns that have been tested and rejected are fully updated. This avoids the need to hold an array of updates. Once nb pivots have been chosen or $q = p$, columns $m_1 + 1$ to n are updated using the Level 3 BLAS kernels `_trsm` and `_gemm`. If $q < p$, m_1 is reset to $q + 1$ and the column search restarts from column $q + 1$. The remaining columns are searched cyclically to avoid repeatedly searching a previously rejected column.

4.2 Pivoting Options

For numerical stability, it is generally necessary to incorporate pivoting within Gaussian elimination. A balance needs to be achieved between stability and efficiency: we want to solve the sparse system fast, with as little fill-in as possible in the factors, but we also want the computed solution to be of the required accuracy. There are a number of pivoting strategies available, each of which places a different emphasis on stability and efficiency. The basic options available are threshold partial pivoting, threshold rook pivoting, and threshold complete pivoting.

Consider first the case where the matrix to be factorized is dense and set $V = DU$. At the kth stage of Gaussian elimination, partial pivoting selects as pivot the largest entry on or below the diagonal in the kth column. The growth factor ρ, which for partial pivoting can be defined by

$$\rho = \max |V_{ij}| / \max |A_{ij}|,$$

is used to describe the backward stability of Gaussian elimination. It can be shown that $\rho \leq 2^{m-1}$ and, although ρ usually behaves like m or less, examples can be found for which partial pivoting is unstable (see, e.g., Higham and Higham [1989]). Complete pivoting searches for the largest entry in the remaining matrix of order $m - k$. In this case, the growth factor satisfies $\rho \leq 2\sqrt{m}m^{\ln(m)/4}$ and, in practice, complete pivoting is considered to be numerically stable. The disadvantage of complete pivoting is the cost since it requires approximately $m^3/3$ comparisons, beyond the work required by Gaussian elimination with no pivoting, whereas partial pivoting requires only $m^2/2$ comparisons. Gaussian elimination with rook pivoting [Neal and Poole 1992] offers a strategy that is intermediate between partial and complete pivoting in terms of both efficiency and stability. To locate the kth pivot, rook pivoting performs a sequential search (column, row, column, etc.) of the remaining matrix until an entry is located whose absolute value is not exceeded by the absolute value of any other entry in the row or column in which it lies. In general, rook pivoting is observed to be more accurate than partial pivoting (Chang [2002]). Furthermore, Foster [1997] has shown that the growth factor satisfies $\rho \leq 1.5m^{3\ln(m)/4}$ and Poole and Neal [2000] reported in their numerical experiments that the expected cost of rook pivoting is about three times that of partial pivoting.

Our interest is in Gaussian elimination for sparse matrices. In this case, pivoting strategies that require the largest entry in a column and, possibly, a row are too restrictive. Instead, a threshold is introduced. `HSL_MA74` offers a number of threshold pivoting options that are controlled using the parameters `pivoting`, `small`, `static`, and `u`. We now discuss these options.

4.2.1 *Threshold Partial Pivoting.* The default strategy within `HSL_MA74` is threshold partial pivoting (`pivoting = 1`). In this case, an entry f_{ij} of the reduced matrix after q pivots have been selected is normally only chosen as a pivot if $i \leq p$ and $j \leq p$ and it satisfies

$$|f_{ij}| \geq \max\left(\mathtt{u} * \max_{l>q} |f_{lj}|, \mathtt{small}\right). \tag{4.1}$$

Here $0 \leq \mathtt{u} \leq 1$ is the pivoting *threshold* parameter. Values of `u` close to zero will generally result in a faster factorization with fewer entries in the factors (for `u` sufficiently small, no pivots will be delayed and so the number of entries in the factors will be equal to the number predicted by the analyse phase) but values close to 1 are more likely to result in a stable factorization (in particular, a value of 1 ensures all the entries of L satisfy $l_{ij} \leq 1$). The default of 0.01 is a compromise between stability and sparsity and is recommended in the user documentation for other direct solvers (see, e.g., Scott [2006]). `small` controls the size of the smallest pivot that is acceptable. The default value is `tiny(small)`, where `tiny()` is the Fortran numeric inquiry function that returns the smallest

positive number that is stored in full precision (in double precision, this is approximately $2.225*10^{-308}$). When column j of the reduced matrix is searched for a suitable pivot, the row index r corresponding to the largest entry in rows $q+1$ to m is sought using the BLAS kernel `i_amax`. If $r \leq p$, the pivot has been found, q is incremented by 1, and rows q and r are swapped. If $r > p$, the largest entry in rows q to p is found (again using `i_amax`) and, if this satisfies (4.1), it is chosen as the next pivot. Note that, if `u` $= 0.0$, the pivot is still chosen to be the largest entry in rows q to p (even though any entry in the column larger than `small` satisfies (4.1)).

4.2.2 *Diagonal Pivoting.* In some applications, it may be known that the pivots can be chosen stably from the diagonal (e.g., if A is close to a symmetric positive-definite matrix). For threshold diagonal pivoting (`pivoting = 2`) with threshold `u` > 0.0, pivots are initially chosen from the diagonal and must satisfy the threshold criteria (4.1) with $j = i$. If $p = m$ (so that a complete factorization of the matrix F is required) and only $q < m$ pivots can be chosen from the diagonal that satisfy (4.1), the code switches to choosing off-diagonal pivots (so that the final $m - q$ pivots may be off-diagonal entries). The number of pivots chosen from the diagonal is returned to the user. If the threshold parameter `u` is equal to zero, no search is made for the largest entry in the column of the candidate pivot. In this case, there is no pivoting and (4.1) simplifies to checking the candidate pivot is at least `small`.

4.2.3 *Threshold Rook Pivoting.* Threshold rook pivoting may be selected in `HSL_MA74` by setting `pivoting = 3`. A candidate f_{ij} may be chosen as a pivot if $i \leq p$ and $j \leq p$ and it satisfies (4.1) and, additionally, with the same `u`,

$$|f_{ij}| \geq \mathtt{u} * \max_{l>q} |f_{il}|. \tag{4.2}$$

In other words, for rook pivoting the pivot candidate must satisfy the threshold test in both its column and its row. Suppose q pivots have been chosen. Having found a candidate with row index i in the column that is currently being searched, row i is swapped with row $q+1$. It must then be updated so that all q pivots have been applied to it, before it can be searched for its largest entry and then tested. Thus, rook pivoting involves more Level 2 BLAS updates (and hence fewer Level 3 BLAS operations) and the additional overheads of row swaps and row searches. Because of this extra cost, it is not the default pivoting strategy within `HSL_MA74`.

If column j is searched but rejected because it fails the test (4.1), provided $j < p$, we next update and search column $j+1$, and continue to search the columns cyclically. However, if the largest entry in column j is in row $i \leq p$ and (4.1) is satisfied, then we update and search row i. Suppose the largest entry in row i lies in column l and that f_{ij} does not satisfy (4.2). If $l > p$ we update and search column $j+1$ but if $l \leq p$ (i.e., column l is fully summed), we swap columns $j+1$ and l so that we next update and search column l, and continue in a like manner until a pivot is found. Our experience has been that, compared with searching in a strictly cyclic fashion, this reduces the total number of rows

and columns searched during the factorization. Note that a count of the number of rows and columns searched is returned to the user.

If rook pivoting is used with large enough u, the condition of A is likely to be reflected in the condition of D, so that rook pivoting is likely to be rank revealing. Indeed, Gill et al. [2005] reported that, provided u is chosen to be sufficiently close to 1, the rank-revealing properties of rook pivoting are essentially as good as for threshold complete pivoting (see also O'Sullivan and Saunders [2002]) and they include rook pivoting as an option within the sparse direct solver LUSOL. Since threshold rook pivoting with a sufficiently large u appears to offer the same advantages as threshold complete pivoting but at less expense, we have chosen not to offer an option for threshold complete pivoting within HSL_MA74.

4.2.4 *Static Pivoting.* In some applications, using a value of u equal to 0.1 or 0.01 can lead to a large number of delayed (rejected) pivots. In this case, the size of the frontal matrices as the factorization moves up the tree can grow significantly beyond that which was anticipated by the analyse phase. This results in a more expensive factorization, both in terms of the number of flops required to perform the factorization and the number of entries in the matrix factors (the factorization is also more complicated since the data structures set up by the analyse phase must be modified to accommodate the delayed pivots); this, in turn, leads to a more expensive solve phase. Furthermore, more storage will be required for the frontal matrix (which is held in main memory) and more data has to be written to and read from the stack during the factorization. In recent years, this has led to a number of direct solvers offering options for *static* pivoting (see, e.g., Duff and Pralet [2005] and Li and Demmel [1998]). To ensure the pivot selection closely follows that provided by the user to the analyse phase, static pivoting may allow pivots to be selected that do not satisfy condition (4.1). The danger is that there may be a potential loss of accuracy in the factorization and it may be necessary to perform refinement steps after the solve phase to try to recover the required accuracy, that is, to use the computed factorization as a preconditioner for an iterative method. If the pivots are selected from the diagonal, it is advantageous to first permute large entries on to the diagonal. For assembled matrices that can be held in memory, the HSL software package MC64 is widely used to do this (details and results to illustrate the benefits are given in Duff and Koster [2001]). Static pivoting remains a subject of research (see, e.g., Arioli et al. [2007]).

Different variants of static pivoting have been proposed: the strategy we have adopted aims to use the best available pivot and to modify pivots only when they become very small. Within HSL_MA74 (and HSL_MA78), static pivoting is controlled by the parameter static. If static is positive and fewer than p pivots can be selected that satisfy (4.1), the pivot that came closest to satisfying this condition is chosen, that is, the pivot for which the ratio

$$\max_{q<i\le p} |f_{ij}| \,/\, \max_{q<l\le m} |f_{lj}|, \quad q \le j \le p, \tag{4.3}$$

is the largest. If its absolute value is greater than static, the information parameter usmall (which is initialized to the user-supplied threshold u) is set to

Table I. Test Problems
(n and *nelt* denote the number of variables and elements, respectively, $nz(L)$ is the predicted number entries in L, and *max_front* is the predicted maximum frontsize.)

Identifier	n	*nelt*	$\|A\|_\infty$	$nz(L)$	max_front
1. ship_001	34920	3431	$2.18*10^{12}$	$1.56*10^{7}$	1596
2. thread	29736	2176	$1.80*10^{19}$	$2.47*10^{7}$	3099
3. x104	108384	26019	$4.32*10^{5}$	$2.71*10^{7}$	2076
4. mt1	97578	5328	$1.83*10^{12}$	$3.27*10^{7}$	1941
5. shipsec8	114919	32580	$3.19*10^{12}$	$3.63*10^{7}$	2624
6. shipsec1	140874	41037	$3.15*10^{13}$	$3.87*10^{7}$	2142
7. shipsec5	179860	52272	$4.89*10^{12}$	$5.42*10^{7}$	3243
8. ship_003	121728	45464	$3.48*10^{18}$	$6.03*10^{7}$	3204
9. raju_001	151656	46980	$4.34*10^{8}$	$1.40*10^{8}$	5232

the minimum of usmall and (4.3). The computation continues using the reduced threshold u ← usmall. If the absolute value of (4.3) is less than static, the pivot is given the value that has the same sign but absolute value static and u is unchanged. On exit, usmall holds the threshold parameter that was used or is set to zero if one or more pivots have been replaced by static, num_thresh holds the number of pivots that did not satisfy the threshold criteria based on the user-supplied value of u, and num_perturbed holds the number of pivots that were replaced by static. Note that on a single call to HSL_MA74, u may be reduced a number of times and, within HSL_MA78, once u has been reduced during a call to HSL_MA74, the factorization continues using this smaller threshold.

5. NUMERICAL EXPERIMENTS

In this section, we report on the effects of the choice of pivot strategy and scaling when using our multifrontal code HSL_MA78 to solve a number of problems from practical applications. The test problems are listed in Table I in order of the predicted number of entries in the factors (i.e., the number of entries if no pivots are delayed) when the analysis phase of the HSL solver MA57 [Duff 2004] is used to compute the pivot order. The predicted maximum frontsize is also given (i.e., the maximum size of the frontal matrix if no pivots are delayed). With the exception of the last problem (which came from a user of HSL_MA78), the problems are all taken from an online Web site.[1] The right-hand side for each problem is selected so that the required solution is the vector of 1s. Note that, in the partial factorization (2.3), the lower triangular part of L_1 and the upper triangular part of U_1, and the rectangular matrices L_2 and U_2, are stored as dense matrices (explicit zeros within the front are ignored). Thus the number of entries $nz(L)$ in the L factor is equal to the number $nz(U)$ in the U factor.

The numerical results were obtained using double precision (64-bit) reals on a 3.6-GHz Intel Xeon dual processor Dell Precision 670 with 4 GB of RAM and a 146.8-GB hard disk. The operating system was Red Hat Enterprise Linux Server release 5.3 with the ext3 file system (default settings). The NAG Fortran

[1] http://www.parallab.uib.no/projects/parasol/data.

f95 compiler with the optimization flag `-O` was used together with the ATLAS BLAS and LAPACK.[2] The reported times are elapsed (wall clock) times in seconds and, unless stated otherwise, are the total solution times (i.e., the time for the analyze, factorize and solve phases with a single right-hand side and, where used, for scaling). We computed the scaled residual

$$\frac{\|Ax - b\|_\infty}{\|A\|_\infty \|x\|_\infty + \|b\|_\infty}. \tag{5.1}$$

In our experiments, we also monitored $\max_i |1 - x_i|$. Note that in all tests the original unscaled matrix A was used when computing the scaled residual.

5.1 Comparison of Partial and Rook Pivoting

We first compare the performance of partial threshold pivoting and rook threshold pivoting (without scaling). Using the default threshold parameter $u = 0.01$, in Table II we report the total solution time, the maximum frontsize, the number of flops required to compute the factors, the number $nz(L)$ of entries in the L factor, the number `delay` of delayed eliminations, and the number of rows and columns searched during the factorization. If p_i and q_i are, respectively, the numbers of candidate and actual pivots chosen at node i then

$$\texttt{delay} = \sum_i (p_i - q_i).$$

Note that a pivot may be delayed (and hence counted) more than once. The number of columns searched was at least n and, for rook pivoting, the minimum number of rows searched was n. For partial pivoting, no rows were searched and so no row search count was included in this case.

A standard technique to recover from inaccuracies in the factorization is to use the computed matrix factorization as a preconditioner for an iterative method, such as iterative refinement. In Table III, the residuals are given, both after `MA78_solve` and after a single step of iterative refinement.

If there are only a small number of delayed eliminations, we see that rook pivoting adds an overhead; this is because both rows and columns must be searched. But this overhead is small compared with the total solution time (see problems 1 and 2). Furthermore, the more stringent test used by rook pivoting can result in less growth and smaller residuals (see columns 2 and 3 of Table III). Smaller growth can also mean that, although rook pivoting initially rejects more pivots than partial pivoting, eventually rook pivoting rejects fewer pivots, leading to the total number of delayed eliminations as well as the maximum frontsize being less for rook pivoting. This in turn gives sparser factors that are computed using fewer flops. Because looking for each pivot is more expensive than for partial pivoting, the time can still increase (as illustrated by problem 8) but, in some cases, the total time using rook pivoting is significantly less than for partial pivoting (notably for problems 4, 6, and 7). This observation is somewhat unexpected. Problem 9 illustrates what we would perhaps anticipate happening: rook pivoting is more cautious and here it leads to more delayed

[2] `http://math-atlas.sourceforge.net`.

Table II. Comparison Between Rook and Partial Threshold Pivoting with $u = 0.01$
(For the first five pairs of results, 1 is in bold if it is significantly better than the other.)

	Time		*max_front*		flops/10^9		$nz(L)/10^6$		delay/10^3		Searched/10^3	
											Rows/cols.	Cols.
Problem	Rook	Partial	Rook	Partial	Rook	Partial	Rook	Partial	Rook	Partial	Rook	Partial
1. ship_001	15.0	**13.4**	1598	1598	22	22	15.6	15.6	0	0	35/35	35
2. thread	37.8	**35.4**	3099	3099	72	72	24.7	24.7	0	0	30/30	30
3. x104	**34.0**	37.8	**2351**	3078	**36**	59	**30.3**	34.5	**20**	33	137/200	202
4. m_t1	**55.7**	94.9	**2298**	3449	**69**	159	**40.2**	56.2	**34**	67	117/231	358
5. shipsec8	91.6	92.8	**3579**	4268	**130**	175	**49.0**	55.6	**61**	78	139/324	329
6. shipsec1	**110**	174	**3450**	5182	**150**	305	**58.6**	78.2	**97**	135	175/467	554
7. shipsec5	**175**	275	**4197**	5872	**246**	492	**80.4**	105	**121**	169	225/589	687
8. ship_003	146	**118**	**3788**	3982	**206**	228	**70.8**	74.0	**50**	610	138/285	281
9. raju_001	335	**226**	5232	5232	701	**579**	177	**154**	123	**79**	414/512	345

Table III. Comparison Between the Scaled Residuals for Factorization with Rook and Partial Threshold Pivoting Before and After One Step of Iterative Refinement ($u = 0.01$).
(For each pair of results, one is in bold if it is significantly better than the other)

	Before		After	
Problem	Rook	Partial	Rook	Partial
1. ship_001	$5.7 * 10^{-16}$	$3.1 * 10^{-16}$	$5.6 * 10^{-17}$	$9.5 * 10^{-17}$
2. thread	$3.1 * 10^{-16}$	$4.0 * 10^{-16}$	$7.4 * 10^{-17}$	$6.8 * 10^{-17}$
3. x104	$\mathbf{6.2 * 10^{-16}}$	$9.9 * 10^{-14}$	$5.4 * 10^{-17}$	$7.1 * 10^{-17}$
4. m_t1	$\mathbf{4.7 * 10^{-16}}$	$8.5 * 10^{-14}$	$3.7 * 10^{-16}$	$2.7 * 10^{-16}$
5. shipsec8	$\mathbf{5.3 * 10^{-16}}$	$2.4 * 10^{-14}$	$7.9 * 10^{-17}$	$9.1 * 10^{-17}$
6. shipsec1	$\mathbf{4.0 * 10^{-16}}$	$7.6 * 10^{-14}$	$9.0 * 10^{-17}$	$1.4 * 10^{-16}$
7. shipsec5	$\mathbf{1.8 * 10^{-15}}$	$6.8 * 10^{-13}$	$1.5 * 10^{-16}$	$1.9 * 10^{-16}$
8. ship_003	$\mathbf{7.9 * 10^{-16}}$	$1.5 * 10^{-13}$	$8.7 * 10^{-17}$	$7.9 * 10^{-17}$
9. raju_001	$1.8 * 10^{-16}$	$5.3 * 10^{-16}$	$3.3 * 10^{-17}$	$2.1 * 10^{-17}$

pivots, more searching, and an increase in the fill-in and in the flop count and hence a greater computational time. We note that, in our tests, after one step of iterative refinement, there was no appreciable difference in the quality of the residuals for partial and rook pivoting.

We have also run the same tests with a larger threshold parameter of 0.1. Our results are given in Table IV. For some of our test problems, the difference between the performance of rook and partial pivoting is more extreme. For example, for problems 4 and 7, it is much better to use rook pivoting while, for problem 9, rook pivoting leads to so many delayed pivots that the code is unable to allocate a sufficiently large frontal matrix on our 32-bit test machine. We do not give full details of the residuals but, again, rook pivoting results in smaller residuals and, as expected, the scaled residuals using $u = 0.1$ are generally smaller before iterative refinement than when using $u = 0.01$ (but following one step of refinement they are comparable).

5.2 Effects of Equilibration

We now present results for HSL_MA78 run with both rook and partial pivoting following a single iteration of the equilibration algorithm (using the infinity norm). The reported times in Table V include the time taken by one iteration of the equilibration algorithm; the threshold parameter used by HSL_MA78 was 0.01. The times with and without equilibration for both rook and partial pivoting are summarized in Table VI. We see that, with the exception of problems 1, 2, and 9, equilibration significantly enhances the performance of the solver. In particular, there are now (almost) no delayed pivots for the ship problems and this results in substantial reductions in the total time (the savings in the factorization times more than offset the costs of the equilibration). The scaled residuals for rook pivoting are $\mathcal{O}(10^{-16})$; they are typically an order of magnitude larger for partial pivoting.

Problems 1 and 2, which did not suffer from delayed pivots when not scaled, do not benefit from one iteration of the equilibration algorithm. In fact, for problem thread, the performance of HSL_MA78 is significantly worse with scaling.

Table IV. Comparison Between Rook and Partial Threshold Pivoting with $u = 0.1$
(NS indicates not solved. For the first five pairs of results, one is in bold if it is significantly better than the other.)

	Time		*max-front*		flops/10^9		$nz(L)/10^6$		delay/10^3		Searched/10^3	
											Rows/cols.	Cols.
Problem	Rook	Partial	Rook	Partial	Rook	Partial	Rook	Partial	Rook	Partial	Rook	Partial
1. ship_001	14.9	**12.9**	1608	1605	22	22	15.7	15.7	0	0	36/37	36
2. thread	35.9	**30.7**	3135	3122	73	73	24.7	24.7	0	0	30/30	30
3. x104	**46.3**	90.8	**3125**	5399	**65**	191	**38.5**	56.1	**55**	87	135/319	451
4. m_t1	**74.6**	273	**2920**	7380	**113**	539	**50.9**	96.6	**68**	137	114/348	695
5. shipsec8	**161**	251	**4912**	6861	**267**	507	**71.0**	92.9	**134**	165	136/550	615
6. shipsec1	**264**	776	**5648**	10147	**413**	1410	**97.5**	163	**203**	259	174/809	1056
7. shipsec5	**502**	1629	**6857**	13468	**768**	3187	**143**	261	**295**	395	218/1161	1569
8. ship_003	**227**	303	**5502**	6927	**395**	623	**96.6**	116	**140**	170	137/551	604
9. raju_001	NS	**300**	NS	**5232**	NS	**718**	NS	**176**	NS	**96**	NS	428

Table V. Comparison Between Rook and Partial Threshold Pivoting with equilibration and $u = 0.01$
(For the first four pairs of results, one is in bold if it is significantly better than the other)

	Time		flops/10^9		$nz(L)/10^6$		delay/10^3		Searched/10^3 Rows/Cols	Cols
Problem	Rook	Partial	Rook	Partial	Rook	Partial	Rook	Partial	rook	partial
1. ship_001	16.7	16.3	22	22	15.6	15.6	0	0	35/35	35
2. thread	**55.0**	64.6	**93**	154	**27.4**	32.9	**4**	8	33/46	60
3. x104	24.9	**23.0**	29	29	27.2	27.2	1	1	111/114	111
4. m_t1	**45.0**	63.0	**60**	104	**37.6**	46.7	**24**	45	109/182	273
5. shipsec8	51.6	**45.3**	74	74	36.3	36.3	0	0	117/117	115
6. shipsec1	49.3	**44.4**	68	68	38.7	38.7	0	0	144/144	141
7. shipsec5	78.0	**69.8**	114	114	54.2	54.2	0	0	183/183	180
8. ship_003	98.4	**89.1**	156	156	60.3	60.3	0	0	123/123	122
9. raju_001	344	**255**	635	**567**	163	**149**	81	**54**	208/420	297

Table VI. Comparison Between Times for Rook and Partial Pivoting with and Without Equilibration ($u = 0.01$)
(For each problem, the fastest time is in bold)

	Rook		Partial	
Problem	No scaling	Scaling	No scaling	Scaling
1. ship_001	15.0	16.7	**13.4**	16.3
2. thread	37.8	55.0	**35.4**	64.6
3. x104	34.0	24.9	37.8	**23.0**
4. m_t1	55.7	**45.0**	94.9	63.0
5. shipsec8	91.6	51.6	92.8	**45.3**
6. shipsec1	110	49.3	174	**44.4**
7. shipsec5	175	78.0	275	**69.8**
8. ship_003	146	98.4	118	**89.1**
9. raju_001	335	344	**226**	255

Similar results were observed when using the one-norm. This illustrates that scaling will not help all problems and the user is advised of the need to experiment with using it.

After one iteration of equilibration, only problems 2, 4, and 9 have a significant number of delayed pivots. These problems may benefit from using more than one iteration. We have experimented with up to 10 iterations, setting ϵ in the stopping criterion (3.1) to 0.0 so that termination occurs when either the maximum number of iterations is reached or $\max_j |A^{(k)}_{lj}| = \max_i |A^{(k)}_{il}| = 1$. For problem 2 (thread), termination happened after four iterations and did not reduce the number of delayed pivots or entries in the factor. In Table VII, we report detailed results for problems 4 and 9 (zero iterations corresponds to no scaling). Before the first iteration, η (given by (3.1)) is approximately equal to $(nz(A) * \|A\|_\infty)/n$; before the second iteration, it is close to 1, and then decreases steadily, with an asymptotic linear rate of 1/2 (see Ruiz [2001]). For problem mt_1, it is beneficial to use two iterations; with three or four iterations, the factors are sparser but the increased cost of the scaling leads to an increase in the total time. For problem raju_001, the number of iterations has little effect on the performance of HSL_MA78 but each extra iteration adds a significant cost.

Table VII. Effect of Number of Iterations Used by Equilibration Algorithm on Performance of HSL_MA78 for Problems m_t1 and raju_001 with Partial Pivoting and $u = 0.01$ (η is defined in (3.1).)

	Num.		Time					Searched
Problem	Its	η	Scaling	Total	flops/10^9	$nz(L)/10^6$	delay	cols./10^3
m_t1	0	—	0.00	94.9	158	56.2	67467	358
	1	$2.32*10^{11}$	5.73	63.0	104	46.7	45127	273
	2	0.99	10.1	56.8	86	43.3	37384	227
	3	0.92	13.7	57.8	81	42.3	35124	218
	4	0.63	17.4	60.8	78	41.6	34279	212
	5	0.40	20.8	65.8	82	42.3	34945	220
	10	0.14	37.8	80.8	78	41.7	33748	212
raju_001	0	—	0.00	227	579	147.3	78669	345
	1	$1.22*10^{7}$	30.8	255	568	149.5	53235	297
	2	1.00	59.1	284	564	147.0	34279	250
	3	0.98	81.6	302	564	147.0	34945	247
	4	0.85	99.5	320	564	147.0	30073	248
	5	0.61	122	342	564	147.0	30341	247
	10	0.03	219	439	564	147.0	30206	247

5.3 Static Pivoting Results

Finally, we present results for static pivoting. We observe that, because the matrix A is in the unassembled form (2.1) and we are assuming that we have insufficient memory to hold the assembled matrix in memory, we are unable to use MC64 [Duff and Koster 2001] to prepermute large entries on to the diagonal (MC64 requires the sparse system matrix to be input in standard compressed sparse column format and to be in memory). As far as we are aware, no analogous routine exists for the unassembled case (although it would be possible to permute the entries of each element matrix individually) or for the case when the assembled matrix cannot be held in memory. We consider only those problems for which our earlier experiments reported a significant number of delayed pivots (if there are no delayed pivots, there is no need for our static pivoting). We start the factorization with the default threshold $u = 0.01$ and set the control parameter static $= 10^{-12}$ (see Section 4.2.4). With this choice of static, in our tests it was not necessary to replace any small pivots. In Table VIII, we report the elapsed times, flop count, the value of the resulting threshold (usmall), and the residuals before and after one step of iterative refinement. The number of entries in the factor is equal to the predicted number of entries (see Table I). Comparing the results with those for partial pivoting, we see that static pivoting can lead to significant savings and, for our test problems, a single step of iterative refinement is generally able to recover accuracy.

We also performed experiments with static $= 10^{-8}$. In this case, for problems x104 and raju_001 with scaling, a few pivots (7 and 13, respectively) were replaced by static. This resulted in a loss of accuracy. For x104, performing further iterations of iterative refinement improved the accuracy of the solution but for raju, the computed solution had infinity norm $2.5 * 10^4$ (recall the exact solution is the vector of ones), and iterative refinement was unable to improve this. This illustrates the importance of using static pivoting with care and the need to experiment with different values of u and static.

Table VIII. Elapsed Times, Flops, Value of Threshold Used (usmall), and Residuals Before and After Iterative Refinement with Static Pivoting ($u = 0.01$) (Figures in parentheses are for partial pivoting (taken from Tables II and V))

					Residual	
	Problem	Time	Flops/10^9	usmall	Before	After
No scaling	3. x104	18.4 (37.8)	28 (59)	$3.84 * 10^{-5}$	$2.2 * 10^{-13}$	$5.2 * 10^{-17}$
	4. m_t1	25.3 (94.9)	47 (159)	$5.46 * 10^{-4}$	$4.6 * 10^{-15}$	$3.1 * 10^{-16}$
	5. shipsec8	39.1 (92.8)	74 (175)	$1.95 * 10^{-4}$	$7.0 * 10^{-15}$	$1.2 * 10^{-16}$
	6. shipsec1	35.5 (174)	68 (303)	$2.17 * 10^{-4}$	$1.8 * 10^{-14}$	$9.1 * 10^{-17}$
	7. shipsec5	60.1 (275)	114 (492)	$1.30 * 10^{-4}$	$8.4 * 10^{-14}$	$1.6 * 10^{-16}$
	8. ship_003	73.4 (118)	156 (228)	$8.23 * 10^{-5}$	$2.8 * 10^{-14}$	$9.4 * 10^{-17}$
	9. raju_001	211 (226)	542 (579)	$7.54 * 10^{-5}$	$2.9 * 10^{-15}$	$2.8 * 10^{-16}$
With scaling	2. thread	36.7 (64.6)	72 (154)	$1.94 * 10^{-3}$	$4.1 * 10^{-16}$	$8.0 * 10^{-17}$
	3. x104	22.6 (23.0)	28 (29)	$8.73 * 10^{-4}$	$6.2 * 10^{-16}$	$8.0 * 10^{-17}$
	4. m_t1	30.0 (63.0)	47 (104)	$9.66 * 10^{-4}$	$2.3 * 10^{-15}$	$2.4 * 10^{-16}$
	9. raju_001	241 (255)	542 (567)	$9.93 * 10^{-5}$	$2.6 * 10^{-15}$	$3.3 * 10^{-17}$

6. CONCLUDING REMARKS AND CODE AVAILABILITY

We have described the pivoting and scaling options that are available within our new out-of-core multifrontal solver HSL_MA78. The default strategy within HSL_MA78 is threshold partial pivoting but our numerical results have illustrated the potential benefits of rook pivoting. Although selecting a single pivot is more expensive for rook than for partial pivoting, for some applications the total number of delayed pivots is less for rook pivoting and this leads to faster total factorization times as well as sparser factors.

Based on a multifrontal algorithm, we have proposed a novel implementation of an equilibration algorithm that does not require the system matrix to be assembled. Our numerical experiments have illustrated the importance for the efficiency of HSL_MA78 of prescaling A. However, our out-of-core equilibration algorithm can add a significant overhead and so, if possible, we recommend that a problem be scaled before reaching the solver stage.

In the future, we would like to investigate the performance of our pivoting strategies when used on singular matrices and, following Gill et al. [2005], to examine how large the rook pivoting threshold parameter needs to be to give reliable rank detection. However, for this we need a wider range of test problems; we would welcome being given access to more large-scale problems from practical applications that are held in unassembled form.

HSL_MA78 is available as part of the mathematical software library HSL. All use of HSL requires a license. Individual HSL packages (together with their dependencies and accompanying documentation) are available without charge to individual academic users for their personal (noncommercial) research and for teaching; licenses for other uses involve a fee. Details of all HSL packages and how to obtain a licence plus conditions of use are online.[3]

[3]http://www.cse.stfc.ac.uk/nag/hsl.

ACKNOWLEDGMENTS

I am very grateful to my colleague John Reid, who collaborated with me on the development of HSL_MA78 and commented on a draft of this article. I would also like to thank Michael Saunders of Stanford University, both for interesting discussions on scaling and rook pivoting at the Householder Symposium XVII and for his careful reading of and comments on this article. Finally, I am indebted to the referees for their constructive comments and suggestions that helped improve this article.

REFERENCES

Arioli, M., Duff, I. S., Gratton, S., and Pralet, S. 2007. A note on GMRES preconditioned by a perturbed LDL^T decomposition with static pivoting. *SIAM J. Sci. Comput. 29*, 5, 2024–2044.

Chang, X.-W. 2002. Some features of Gaussian elimination with rook pivoting. *BIT 42*, 66–83.

Davis, T. 2004. Algorithm 832: UMFPACK, an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw. 30*, 2, 196–199.

Davis, T. and Duff, I. 1997. An unsymmetric-pattern multifrontal method for sparse LU factorization. *SIAM J. Matrix Anal. Appl. 18*, 140–158.

Dongarra, J., Du Croz, J., Duff, I. S., and Hammarling, S. 1990. A set of Level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw. 16*, 1, 1–17.

Dongarra, J., Du Croz, J., Duff, I. S., and Hammarling, S. 1998. An extended set of Fortran basic linear algebra subprograms. *ACM Trans. Math. Softw. 14*, 1, 1–17.

Duff, I. 1984. Design features of a frontal code for solving sparse unsymmetric linear systems out-of-core. *SIAM J. Sci. Comput. 5*, 270–280.

Duff, I. 2004. MA57—a new code for the solution of sparse symmetric definite and indefinite systems. *ACM Trans. Math. Softw. 30*, 118–144.

Duff, I. and Koster, J. 2001. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM J. Matrix Anal. Appl. 22*, 4, 973–996.

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

Duff, I. and Reid, J. 1983. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Trans. Math. Softw. 9*, 302–325.

Duff, I. and Scott, J. 1996. The design of a new frontal code for solving sparse unsymmetric systems. *ACM Trans. Math. Softw. 22*, 1, 30–45.

Foster, L. 1997. The growth factor and efficiency of Gaussian elimination with rook pivoting. *J. Comput. Appl. Math. 86*, 177–194.

Gill, P., Murray, W., and Saunders, M. 2005. SNOPT: An SQP algorithm for large-scale constrained optimization. *SIAM Rev. 47*, 99–131.

Higham, N. and Higham, D. 1989. Large growth factors in Gaussian elimination with pivoting. *SIAM J. Matrix Anal. Appl. 10*, 155–164.

HSL. 2007. A collection of Fortran codes for large-scale scientific computation. http://www.cse.stfc.ac.uk/nag/hsl/.

Irons, B. 1970. A frontal solution program for finite-element analysis. *Int. J. Numer. Meth. Eng. 2*, 5–32.

Li, X. and Demmel, J. 1998. Making sparse Gaussian elimination scalable by static pivoting. In *Proceedings of the Conference on Supercomputing*.

Neal, L. and Poole, G. 1992. A geometric analysis of Gaussian elimination, II. *Lin. Alg. Appl. 173*, 239–264.

O'Sullivan, M. and Saunders, M. 2002. Sparse rank-revealing LU factorization (via threshold complete pivoting and threshold rook pivoting). *Householder Symposium XV on Numerical Linear Algebra*. http://www.stanford.edu/group/SOL/talks.html.

Poole, G. and Neal, L. 2000. The rook's pivoting strategy. *J. Comp. Appl. Math. 123*, 353–369.

Reid, J. and Scott, J. 2009a. Algorithm 891: A Fortran virtual memory system. *ACM Trans. Math. Softw. 36*, 1, Article 5.

REID, J. AND SCOTT, J. 2009b. An efficient out-of-core multifrontal solver for large-scale unsymmetric element problems. *Int. J. Numer. Meth. Eng. 77*, 7, 901–921.

REID, J. AND SCOTT, J. 2009c. An out-of-core sparse Cholesky solver. *ACM Trans. Math. Softw. 36*, 2, Article 9.

ROTHBERG, E. AND SCHREIBER, R. 1999. Efficient methods for out-of-core sparse Cholesky factorization. *SIAM J. Sci. Comput. 21*, 129–144.

ROTKIN, V. AND TOLEDO, S. 2004. The design and implementation of a new out-of-core sparse Cholesky factorization method. *ACM Trans. Math. Softw. 30*, 1, 19–46.

RUIZ, D. 2001. A scaling algorithm to equilibrate both rows and columns norms in matrices. Tech. rep. RAL-TR-2001-034. Rutherford Appleton Laboratory, Chilton, U.K.

SCOTT, J. 2006. A frontal solver for the 21st century. *Comm. Numer. Meth. Eng. 22*, 1015–1029.

SKEEL, R. 1979. Scaling for numerical stability in Gaussian elimination. *J. ACM 26*, 494–526.

Received July 2008; revised February 2009, July 2009; accepted November 2009