Science & Technology Facilities Council

Technical Report

Partial factorization of a dense symmetric indefinite matrix

J. K. Reid and J. A. Scott

RAL-TR-2009-015

Library and Information Services
SFTC Rutherford Appleton Laboratory
Harwell Science and Innovation Campus
Didcot
OX11 0QX
UK
Tel: +44 (0)1235 445384
Fax: +44(0)1235 446403
Email: library@rl.ac.uk

ISSN 1358-6254

Partial factorization of a dense symmetric indefinite matrix [1]

by

J. K. Reid and J. A. Scott

Abstract

At the heart of a frontal or multifrontal solver for the solution of sparse symmetric sets of linear equations, there is the need to partially factorize dense matrices (the frontal matrices) and to be able to use their factorizations in subsequent forward and backward substitutions. For a large problem, packing (holding only the lower or upper triangular part) is important to save memory. It has long been recognized that blocking is the key to efficiency and this has become particularly relevant on modern hardware. For stability in the indefinite case, the use of interchanges and 2×2 pivots as well as 1×1 pivots is equally well established. It is shown here that it is possible to use these three ideas together to achieve stable factorizations of large real-world problems with good execution speed.

The ideas are not restricted to frontal and multifrontal solvers and are applicable whenever partial or complete factorizations of dense symmetric indefinite matrices are needed.

Keywords: sparse symmetric linear systems, LDL^T factorization, 2×2 pivots, interchanges, frontal, multifrontal.

---

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

Computational Science and Engineering Department,
Atlas Centre, Rutherford Appleton Laboratory,
Oxon OX11 0QX, England.

August 26, 2009.

Contents

1 Introduction

In this paper, we consider the factorization of a dense symmetric indefinite matrix A of order n whose lower-triangular part is packed by columns and has the form

$$A = \begin{pmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{pmatrix}, \tag{1.1}$$

where A_{11} is a square matrix of order p and pivots are restricted to being within A_{11}. Sometimes, it is not possible to choose pivots for the whole of A_{11}. For example, a column of A may be zero in A_{11} and nonzero in A_{21}. Our factorization therefore takes the form

$$P^T A P = L D T^T \tag{1.2}$$

where P is a permutation matrix of the form

$$P = \begin{pmatrix} P_{11} & \\ & I \end{pmatrix} \tag{1.3}$$

with P_{11} of order p, L is the unit lower-triangular matrix

$$L = \begin{pmatrix} L_{11} & \\ L_{21} & I \end{pmatrix}, \tag{1.4}$$

with L_{11} of order $q \leq p$, and D is the matrix

$$D = \begin{pmatrix} D_{11} & \\ & S_{22} \end{pmatrix}, \tag{1.5}$$

where D_{11} is a block diagonal matrix of order q with blocks of order 1 or 2 and S_{22} is a dense matrix of order $n-q$. We refer to this as a *partial* factorization, but allow the case of a complete factorization ($p = n$), in which case q has the value n.

Once the factorization is available, it may be used for the following partial solutions:

$$Lx = b, \quad \begin{pmatrix} D_{11} & \\ & I \end{pmatrix} x = b, \quad \text{and} \quad L^T x = b. \tag{1.6}$$

and the corresponding equations for rectangular matrices X and B of the same shape.

For speed of execution when n is large, we rearrange the matrix into a block form that enables most of the operations to be performed using Level-3 BLAS (Dongarra, Du Croz, Duff and Hammarling, 1990*b*). Details are given in Section 5.

For stability, we follow the recommendation of Duff, Gould, Reid, Scott and Turner (1991) to use symmetric permutations to ensure that 1×1 and 2×2 pivots satisfy a relative pivot threshold. This ensures that the entries of L are bounded in size. We explain our reasons for this choice in Section 3.

Combining blocking with symmetric permutations that are chosen dynamically during the factorization is a substantial challenge and is the main achievement that we report in this paper.

The code is collected into the module `HSL_MA64` of the mathematical software library HSL (HSL 2007). We expect the principal application to be within the multifrontal method and our data formats have been designed to take this into account, as we explain in Section 2. However, this work is also available for other applications, including the case where a complete factorization is needed ($p = n$).

The rest of the paper is organized as follows. Section 3 explains our pivoting strategy. Section 4 describes the implementation in the simplest case and is followed by Section 5 which explains how blocking and other details are handled. Parallel working is considered in Section 6 and the results of some numerical experiments are reported in Section 7.

2 The multifrontal method

The multifrontal method (see, for example, Duff and Reid, 1983) separates the factorization of a large sparse symmetric matrix into steps, each involving a set of rows and corresponding columns. The entries of these rows and columns are compressed into a dense symmetric matrix of the form (1.1), called the frontal matrix, and the operations are performed within it. The partial factorization of the frontal matrix is the task that HSL_MA64 performs and is the focus of this paper. The multifrontal application has strongly influenced the design of HSL_MA64.

For economy of storage, it is desirable to take advantage of symmetry and hold only the upper or lower triangular part of each frontal matrix. The HSL multifrontal solver HSL_MA77 (Reid and Scott, 2008, 2009*b*) holds the lower triangular part by columns, so we have adopted this format for input of A to HSL_MA64 and output of S_{22} (see (1.5)).

Only the first p rows and columns of the frontal matrix A are available for pivots because the entries in A_{22} will have additional values added later from other frontal matrices. The 'generated elements' S_{22} from two or more frontal matrices are added into a later frontal matrix, whose rows and columns include all those of the generated elements that contribute to it. This merging operation would be awkward with blocking in place, which is why HSL_MA77 does not use it. We refer to this later frontal matrix as the 'parent'.

If $q < p$ pivots are selected, the generated element S_{22} has $p - q$ more rows and columns than it would have had if pivots could have been chosen for all of A_{11}. The enlarged matrix is passed to the parent and the extra rows and columns will be eliminated there or at an ancestor, where there is more choice for the pivot and the entries may have been modified by other eliminations. The extra rows and columns and the pivots within them are called 'delayed'. The delayed rows and columns will be unchanged when processing starts within the parent matrix and are unlikely to contain entries that are suitable as pivots until other elimination operations have been performed. There is therefore an option in HSL_MA64 to defer looking for pivots in these rows and columns (see Section 5.5) until all the others have been tried.

If there are many delayed pivots, the multifrontal factorization becomes substantially slower and needs much more storage. Once the factorization is available, the speed of solution of a set of equations is approximately proportional to the number of entries in its factorization, so this is affected by delayed pivots. HSL_MA64 therefore has options for a relaxed pivot threshold and for static pivoting (see, for example, Li and Demmel, 1998), that is, forcing pivots that do not satisfy the stabililty test to be chosen, perhaps after modification. The intention is to reduce fill-in, probably at the expense of an iterative procedure (such as iterative refinement) for each solution to obtain the required accuracy.

3 Pivoting strategy

We choose the pivots one by one and suppose that q denotes the number of rows and columns of D found so far (that is, the number of 1×1 pivots plus twice the number 2×2 pivots). We use the notation a_{ij}, with $i > q$ and $j > q$, to denote an entry of the matrix after it has been updated by all the permutations and pivot operations so far. Our stability test for a 1×1 pivot in column m, $q < m \leq p$, is the usual threshold test

$$|a_{mm}| > u \max_{i \neq m,\, i > q} |a_{im}|, \tag{3.1}$$

where the relative pivot tolerance u is a user-set value in the range $0 \leq u \leq 1.0$. This is equivalent to the test

$$|a_{mm}|^{-1} \max_{i \neq m,\, i > q} |a_{im}| < u^{-1}. \tag{3.2}$$

In the case where u is zero, this is interpreted as requiring that the pivot be nonzero. This was generalized by Duff et al. (1991) to the test

$$\left| \begin{pmatrix} a_{mm} & a_{ml} \\ a_{ml} & a_{ll} \end{pmatrix}^{-1} \right| \begin{pmatrix} \max_{i \neq m,l,\, i > q} |a_{im}| \\ \max_{i \neq m,l,\, i > q} |a_{il}| \end{pmatrix} < \begin{pmatrix} u^{-1} \\ u^{-1} \end{pmatrix}, \tag{3.3}$$

for a 2×2 pivot in rows and columns l and m, where the absolute value notation for a matrix refers to the matrix of corresponding absolute values and u is the same user-set threshold value. In the case where u is zero, this is interpreted as requiring that the pivot be nonsingular. We use this test with the default value 0.1 for u. Another value that is often used is 0.01.

If a_{mm} is accepted as a 1×1 pivot, it becomes the next diagonal block of D and row and column m are permuted (if necessary) to the next pivotal position, q+1. The corresponding diagonal entry of L is 1 and inequality (3.2) tells us that the off-diagonal entries of this column of L are bounded in modulus by u^{-1}. If $\begin{pmatrix} a_{mm} & a_{ml} \\ a_{ml} & a_{ll} \end{pmatrix}$ is accepted as a 2×2 pivot, it becomes the next diagonal block of D and rows and columns l and m are permuted (if necessary) to the next two pivotal positions, $q+1$ and $q+2$. The corresponding diagonal block of L is the identity matrix of order 2 and inequality (3.3) tells us that the off-diagonal entries of these columns of L are bounded in modulus by u^{-1}. Thus, all the diagonal entries of L are 1 and all the off-diagonal entries of L are bounded in modulus by u^{-1}. This limits the growth of the size of the entries of the reduced matrix.

Our strategy for relaxed and static pivoting is based on the work of Duff and Pralet (2007). Relaxed pivoting may be requested by providing a lower bound `umin` for u. If no 1×1 or 2×2 candidate pivot satisfies the test (3.1) or (3.3) but the pivot that is nearest to satisfying the test would satisfy it with $u = v \geq$ `umin`, the pivot is accepted and u is reduced to v. The new value of u is employed thereafter. Our default value for `umin` is 0.1. If $p = n$, we do not allow its value to exceed 0.5 in order to ensure that a complete set of pivots is chosen.

If static pivoting is requested (see final paragraph of Section 2) and no 1×1 or 2×2 candidate pivot satisfies the test (3.1) or (3.3) even after relaxing the value of u, the 1×1 pivot that is nearest to satisfying the test is accepted. If its absolute value is less than another user-set threshold `static`, it is given the value that has the same sign but absolute value `static`.

If no pivot is modified by static pivoting and $q = p$, the largest value of u that would have resulted in all pivots satisfying the stability tests is returned. If no pivot is modified by static pivoting and $q < p$, the largest value of u that would have resulted in more pivots satisfying the stability tests is returned.

Stability was considered by Ashcraft, Grimes and Lewis (1999), who showed that bounding the size of the entries of L, together with a backward stable scheme for solving 2×2 linear systems, suffices to show backward stability for the entire process. They found that the widely used strategy of Bunch and Kaufman (1977) does not have this property.

Our default pivoting strategy is the symmetric equivalent of rook pivoting. For the unsymmetric problem, Gill, Murray and Saunders (2005) report 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 and they include rook pivoting as an option within the sparse direct solver LUSOL. Our experience of rook pivoting (see Reid and Scott, 2009*a* and Scott, 2008) has been satisfactory, too, for the unsymmetric problem; the factors usually had less entries, the residuals were usually smaller, while the factorization was sometimes significantly faster. Our code is arranged to handle singular matrices and return a low-norm solution if the problem is consistent.

4 Implementation in the simplest case

We begin by describing our implementation for the case where n is less than the block size so there is no blocking, the matrix is not found to be singular or nearly singular and the options described in Subsections 5.4 and 5.5 are not requested. To allow the use of Level-2 BLAS (Dongarra, Du Croz, Hammarling and Hanson, 1988) and Level-3 BLAS (Dongarra, Du Croz, Duff and Hammarling, 1990*a*), the matrix is held in full storage with valid entries only in the lower-triangular part. In this simple case, the implementation is primarily left-looking, that is, each column of the lower-triangular part of the matrix is not altered until just before the column is first searched for a pivot. We have chosen a left-looking algorithm because caching is more efficient for referencing variables than for altering their values. Suppose that $q < p$ pivots have

been chosen and their rows and columns have been permuted to the leading positions. Suppose further that column m is about to be searched and that all the columns between q and m have been updated. For column m, the update required is

$$A_{m:n,m} \Leftarrow A_{m:n,m} - L_{m:n,1:q} D_{1:q,1:q} (L_{m,1:q})^T. \tag{4.1}$$

Here and later, we use section notation in subscripts to refer to submatrices. Note that the submatrices sometimes have size zero. After forming the vector

$$U_{1:q,m} = D_{1:q,1:q} (L_{m,1:q})^T, \tag{4.2}$$

we can express the update as

$$A_{m:n,m} \Leftarrow A_{m:n,m} - L_{m:n,1:q} U_{1:q,m}, \tag{4.3}$$

which can be performed by the BLAS-2 subroutine `_gemv`.

Note that to test for a 1×1 pivot in column m, we need the vector of entries to the left of the diagonal in row m, that is, $A_{m,q+1:m-1}$, which will be fully updated. If the 1×1 pivot is accepted, we increment q by one, interchange rows and columns m and q, calculate column q of D and L, then perform the right-looking update

$$A_{q+1:n,q+1:m} \Leftarrow A_{q+1:n,q+1:m} - L_{q+1:n,q} D_{q,q} (L_{q+1:m,q})^T. \tag{4.4}$$

After forming the vector

$$U_{q,q+1:m} = D_{q,q} (L_{q+1:m,q})^T, \tag{4.5}$$

we can express the update as

$$A_{q+1:n,q+1:m} \Leftarrow A_{q+1:n,q+1:m} - L_{q+1:n,q} U_{q,q+1:m}, \tag{4.6}$$

which could be performed by the BLAS-2 subroutine `_ger`, but we have chosen to use a sequence of calls to the BLAS-1 (Lawson, Hanson, Kincaid and Krogh, 1979) subroutine `_axpy` because this avoids wasted operations in the upper-triangular part and there are no data-movement advantages in using `_ger`.

If $m > q+1$, we can look for a 2×2 pivot in the pairs of rows and columns (j, m), $j = q+1, \ldots, m-1$. If an acceptable 2×2 pivot is found, we increment q by two, interchange rows and columns to move the selected two forward, calculate columns $q-1$ and q of D and L, and form the 2-rowed matrix

$$U_{q-1:q,q+1:m} = D_{q-1:q,q-1:q} (L_{q+1:m,q-1:q})^T. \tag{4.7}$$

We can then perform the double update

$$A_{q+1:n,q+1:m} \Leftarrow A_{q+1:n,q+1:m} - L_{q+1:n,q-1:q} U_{q-1:q,q+1:m}, \tag{4.8}$$

with the BLAS-3 subroutine `_gemm`. The performance might be as much as twice that for (4.6), but will not be as good as it is for full-sized blocks.

If both a 1×1 and a 2×2 pivot are available, we choose the 2×2 pivot because it reduces by two the number, $m - q$, of columns that are being kept updated.

If we cannot choose a 1×1 or 2×2 pivot, the column is simply retained for updating in later steps and possible choice as the first half of a 2×2 pivot. Note that this results in every pair of fully-summed columns that do not have an acceptable 1×1 pivot being considered for defining a 2×2 pivot.

Unless $m = p$, we now increment m by one and continue. If $m = p$ and $q = p$, the calculation of L and D is complete. If $m = p$ and $q \neq p$, we restart the search from column $q+1$. We continue in this way until $q = p$ or $q - p$ successive columns fail to provide a pivot. The algorithm is summarized in pseudo-Fortran in Algorithm 1. After each pivot has been accepted and the consequent operations applied, we could have retested all the columns that are up to date but do not have a pivot, but do not do so because it is likely to lead to repeated unsuccessful testing.

Algorithm 1 Basic algorithm summary. Here `_axpy`, `_gemv` and `_gemm` are BLAS subroutines. At each stage, q is the number of pivots chosen so far and m is the index of the column to be searched for a pivot.

Input: matrix A in the form (1.1) and the order p of A_{11}
Initialise: $q = 0$; $m = 0$
do while $(q < p)$
  do test $= 1, p - q$
    $m = m + 1$
    if $(m > p)$ **then**
      $m = q + 1$
      $up = q$! Number of updates applied to column p
    end if
    use `_gemv` to apply rank-$(q - up)$ updates to column m
    if (columns j, m have an acceptable 2×2 pivot) **then**
      $q = q + 2$
      interchange columns $(q - 1, q)$ with columns (j, m)
      calculate columns $q - 1, q$ of D and L
      use `_gemm` to apply rank-2 updates to columns $q + 1$ to m
      exit
    else if (column m has an acceptable 1×1 pivot) **then**
      $q = q + 1$
      interchange column q with column m
      calculate column q of D and L
      use `_axpy` to apply rank-1 updates to columns $q + 1$ to m
      exit
    end if
  end do
end do

It is conceivable that many of the pivots are found only after a large number of unsuccessful tests of other columns, so that the total number of column searches is $O(p^2)$. We have not been able to show that this is impossible, but when calling `HSL_MA64` more than a million times during the multifrontal factorization of 56 large sparse matrices with $u = 0.01$, we found that the total number of searches in a call exceeded $4p$ only twice and never exceeded $7p$. With $u = 0.1$, the total number of searches in a call exceeded $5p$ only 4 times and never exceeded $8p$.

Once L and D are known, the first $p-q$ rows and columns of S_{22} (see (1.5)) are known, but the trailing part still needs to be calculated. Note that no interchanges are performed in this part of the matrix, see equations (1.2) and (1.3). The calculation may be performed by forming

$$U_{1:q,p+1:n} = D_{1:q,1:q}(L_{p+1:n,1:q})^T \tag{4.9}$$

and performing the update

$$A_{p+1:n,p+1:n} \Leftarrow A_{p+1:n,p+1:n} - L_{p+1:n,1:q}U_{1:q,p+1:n} \tag{4.10}$$

with the BLAS-3 subroutine `_gemm`.

5 The block form

For the Cholesky factorization of a positive-definite dense symmetric matrix, Andersen, Gunnels, Gustavson, Reid and Wasniewski (2005) recommend a 'lower blocked hybrid' format that is as economical of storage as packing the lower-triangular part by columns but is able to take advantage of Level-3 BLAS. They divide the lower-triangular part of the matrix into blocks, most of which are square and of the same order nb.

Unfortunately, the interchanges that are needed in the indefinite case make the blocked hybrid format very awkward. Furthermore, there is intense activity in columns $q+1$ to m (see Algorithm 1) which makes it desirable to hold the columns contiguously in memory. We therefore use a format that we call the 'block column' format. This holds the lower triangular part of the matrix by block columns, with each block having nb columns (except possibly the final block) and being stored by columns. The numbers of rows in the blocks are n, $n-nb$, etc. This format is illustrated on the left of Figure 5.1. We chose not to pack the lower triangular parts of the blocks on the diagonal in order to simplify the code and allow us to use more efficient BLAS without any further rearrangements. It may readily be verified that the total waste is limited to $n(nb-1)/2$.

1	#	#							
2	12	#							
3	13	23							
4	14	24	31	#	#				
5	15	25	32	39	#				
6	16	26	33	40	47				
7	17	27	34	41	48	52	#	#	
8	18	28	35	42	49	53	57	#	
9	19	29	36	43	50	54	58	62	
10	20	30	37	44	51	55	59	63	64

1							
2	4						
3	5	6					
7	14	21	28				
8	15	22	29	31			
9	16	23	30	32	33		
10	17	24	34	38	42	46	
11	18	25	35	39	43	47	48
12	19	26	36	40	44	49	51
13	20	27	37	41	45	50	52

Figure 5.1: Block column format when $n = 10$ and $nb = 3$, and after rearrangement when $q = 8$.

Prior to performing any eliminations, we rearrange A to block column format.

We begin the factorization as in Section 4 and summarized in Algorithm 1, except that account must be taken of the blocking once m exceeds nb. We continue until a 1×1 pivot is chosen in column nb or a 2×2 pivot is chosen that includes column nb. Now we apply the first nb pivot operations to columns

$m+1$ to n, block column by block column (and in Algorithm 1 increment up by nb). By employing the temporary matrix

$$U_{1:nb,m+1:n} = D_{1:nb,1:nb}(L_{m+1:n,1:nb})^T, \tag{5.1}$$

we can apply these operations to the remaining part of the block column that contains column $m+1$ and to subsequent block columns. For a block column that spans columns j to k, we apply the BLAS-3 subroutine `_gemm` thus

$$A_{j:n,j:k} \Leftarrow A_{j:n,j:k} - L_{j:n,1:nb}U_{1:nb,j:k}. \tag{5.2}$$

This is much more efficient than waiting for the updates to be done as in equation (4.3) using the BLAS-2 subroutine `_gemv`. The array that holds U really needs only nb rows, but we find it convenient to give it an extra row for the case of a 2×2 pivot spanning two block columns. Here, row $nb+1$ of U is moved to row 1 of the array and rows found thereafter are placed in the array from row 2.

We do not retain the block column format for the matrix $\begin{pmatrix} L_{11} \\ L_{21} \end{pmatrix}$ on return because that would significantly increase the memory needed. Our experience is that quite large values of the block size nb are desirable, for example, 96 (see Section 7). Once the partial factorization is complete, this matrix is rearranged; it is still held by block columns, but each consists of the diagonal block packed by columns followed by the off-diagonal part held by columns. This form is as economical of storage as that of Andersen et al. (2005) and is illustrated on the right of Figure 5.1. It allows the partial solution operations for a single right-hand side to be performed with the BLAS-2 subroutines `_gemv` and `_tpsv` and, for multiple right-hand sides, to be performed with the BLAS-3 subroutine `_gemm` and repeated calls of `_tpsv`.

Before return, columns $q+1$ to n need to be rearranged to packed lower triangular format.

5.1 Rearranging only the first p columns

If we know that there will be only one block step (that is, if $p \leq nb$), the trailing $n-p$ columns are updated only once and caching considerations make it better to rearrange this part of the matrix only when it is updated and rearrange it back immediately afterwards. In the case, we therefore use the block column format only for the first p columns.

5.2 Inner blocking

We have found that a large block size is desirable for a large matrix. Unfortunately, with a large block size, many operations are performed with `_gemv`. Indeed, if $p \leq nb$, no rank-nb `_gemm` calls are made while calculating L and D. In this case, the code would have been faster with a smaller block size. The same argument can be made when p is large for the calculation of the first nb columns of L and for each subsequent block of nb columns. We have therefore adopted the concept of an inner block size nbi for use solely within the computation of each block of nb columns of L. For simplicity of coding, we require nb/nbi to be an integer so that all the inner blocks have size exactly nbi. We continue to store the block column in a rectangular array with nb columns, which differs from the usual practice with nested blocking (see, for example, Elmroth, Gustavson, Jonsson and Kågström, 2004) since the data format is not affected. We keep the columns in contiguous memory to facilitate checking pivots for stability and making row and column interchanges. Whenever an integer multiple of nbi columns of L have been found, we use `_gemm` to update the columns of the block column from column $m+1$ onwards.

5.3 Factorization in the singular case

So far, we have assumed that the matrix is nonsingular, but consistent systems of linear equations with a singular matrix occur quite frequently in practice and we wish to accommodate them. Therefore, when column m is searched, if its largest entry is found to be less than a user-set tolerance `small`, the row and column are set to zero, the diagonal entry is accepted as a zero 1×1 pivot, and no corresponding pivotal operations are applied to the rest of the matrix. This is equivalent to perturbing the corresponding entries

of A by at most `small` to make our factorization be of a nearby singular matrix. To allow for this case, we hold the inverse of D_{11}, and set the entry corresponding to the zero pivot to zero. This avoids the need for special action in BLAS-2 and BLAS-3 calls later in the factorization and during the solution. It leads to a correct result with a reasonable norm when the given set of equations is consistent and avoids the solution having a large norm if the equations are not consistent.

5.4 Requesting particular 2×2 pivots

`HSL_MA64` allows its user to specify that particular adjacent pairs in the pivot sequence should be used as 2×2 pivots provided they satisfy the stabililty test (3.3) when first encountered. If m indexes the first half of such a pair, the largest entry in the column is determined, but no pivot is accepted immediately. Processing continues to the next column and now the recommended pivot can be tested for acceptance as a 2×2 pivot. Either it is accepted or the recommendation is cancelled and normal pivot tests are performed.

5.5 Delaying the testing of early columns

`HSL_MA64` has an optional argument `s` that allows the user to specify that the first `s` columns be avoided initially when searching for pivots. This is useful for delayed pivots in the multifrontal method, see Section 2. If `s` is present, swaps are made between columns `j` and `p+1-j` for `j=1,min(s,p-s)`, unless this would involve splitting a recommended 2×2 pivot which may happen if $\texttt{s} < \texttt{p} - \texttt{s}$. In the case that would cause a split, the swaps are made only for `j = 1` to `s-1`. Following this, the leading `s` columns (`s-1` in the exceptional case) will have been moved to the end of the set of columns `1:p`.

5.6 Long integers

Because large memories and 64-bit architectures are becoming increasing commonplace, we have designed `HSL_MA64` to handle very large matrices. We use default integers (assumed to be stored in 32 bits) for row and column indices, but use long integers that hold at least 18 decimals to index the one-dimensional array holding the packed matrix A.

6 Parallel working

In common with all other packages in HSL 2007, `HSL_MA64` is thread-safe. All its data is supplied through its arguments, so it may be called at the same time, for example, on two threads working on independent parts of the multifrontal tree.

This form of parallelism is not sufficient near the root of the tree where n and p are large. Here, most of the work is done within `_gemm` when updating the trailing submatrix, see equation (5.2). We therefore provide an option to use an OpenMP `parallel do` for this computation. It is simplest to loop over the block columns so that each block column update is treated as a separate task, but such tasks are unbalanced and there may be too few of them for the available threads; instead, we break each block column into blocks of nb rows and loop over those.

If p is small there may be insufficient work to justify working in parallel. `HSL_MA64` therefore has a user-set parameter, `p_thresh` and parallel work takes place only if p is greater than this. If parallel working has been chosen and $p \leq nb$, we use the block column format for the whole matrix; were we to rearrange only the first p columns (see Section 5.1), there would be no opportunity for parallel working. We have found that 32 is a suitable value for `p_thresh` on our test platform (see Section 7 for details) and use this as the default value in `HSL_MA64`.

To allow the possibility of parallel processing of more than one execution of `HSL_MA64`, there is an optional argument to specify the number of threads for the `parallel do` loop. The argument is accessed before each execution of the `parallel do` loop to allow the user to alter its value during execution.

7 Numerical experiments

The numerical experiments reported in this paper were performed on our multicore test machine `fox`, details of which are given in Table 7.1. For timing, we used wall-clock times in seconds on a lightly loaded machine.

Table 7.1: Specifications of our test machine `fox`.

	2-way quad Harpertown (`fox`)
Architecture	Intel(R) Xeon(R) CPU E5420
Clock	2.50 GHz
Cores	2×4
Theoretical peak (1/8 cores)	10 / 80 Gflop/s
`dgemm` peak (1/8 cores[1])	9.3 / 72.8 Gflop/s
Memory	8 GB
Compiler	Intel 11.0 with option -fast
BLAS and LAPACK	Intel MKL 10.1

[1] Measured by using MPI to run independent matrix-matrix multiplies on each core

7.1 Choice of block sizes

We begin by illustrating the relevance to performance of the choices that are available by considering the partial factorization of randomly generated matrices of order 4000 and p ranging between 16 and 2048. Table 7.2 shows that the performance does not vary greatly as the block size ranges over the interval (48, 120). We have chosen 96 for our default block size. Table 7.3 shows that parallelizing by blocks gives a worthwhile gain over parallelizing by block columns. Table 7.4 shows the effect of the inner block size nbi. We have chosen 16 for our default inner block size and show the speed-ups with this choice in Table 7.5. Note that the speed on one thread approaches the `dgemm` peak of 9.3 Gflop/s for the larger values of p.

Table 7.2: The effect of varying the block size, nb, with $nbi = nb/6$ for a range of values of p. Speeds in wall-clock Gflop/s are reported.

	One thread				Four threads				Eight threads			
nb	48	72	96	120	48	72	96	120	48	72	96	120
p												
16	2.7	2.7	2.7	2.6	3.5	3.5	3.4	3.4	3.5	3.5	3.4	3.4
32	4.1	4.2	4.1	4.1	6.4	6.4	6.3	6.2	6.9	6.8	6.7	6.5
64	5.2	5.5	5.5	5.5	9.8	10.3	10.1	10.1	10.9	12.0	11.8	11.7
128	6.3	6.5	6.5	6.2	14.2	14.7	14.5	13.7	17.1	18.2	17.5	16.6
256	6.9	7.2	7.2	7.1	17.6	18.4	18.4	17.9	22.8	24.3	24.7	23.6
512	7.3	7.4	7.6	7.6	20.3	20.8	21.0	20.8	27.8	28.5	29.7	28.8
1024	7.4	7.7	7.8	7.8	21.4	22.4	22.3	22.1	30.1	32.0	32.8	32.4
2048	7.4	7.7	7.8	7.8	21.5	22.4	22.4	22.1	30.3	32.0	32.8	32.0

7.2 Experiments with a multifrontal solver

In Table 7.6, we show some data for the execution of the multifrontal solver `HSL_MA77` (Reid and Scott, 2008, 2009*b*) on problems from practical applications that have varying front sizes. We scale using the

Table 7.3: Parallelizing by block columns or by blocks, with $nb = 96$ and $nbi = nb/6$. Speeds in wall-clock Gflop/s are reported.

	Four threads		Eight threads	
p	columns	blocks	columns	blocks
64	10.3	10.1	11.9	11.8
128	13.9	14.5	16.6	17.5
256	17.3	18.4	22.0	24.7
512	19.8	21.0	25.5	29.7
1024	20.7	22.3	28.3	32.8
2048	20.6	22.4	28.3	32.8

Table 7.4: The effect of varying the inner block size, nbi, with $nb = 96$ for a range of values of p. Speeds in wall-clock Gflop/s are reported.

	One thread				Four threads				Eight threads			
nbi	96	24	16	12	96	24	16	12	96	24	16	12
p												
64	5.4	5.5	5.5	5.5	9.8	10.4	10.2	10.1	11.4	11.9	11.9	11.8
128	6.4	6.5	6.5	6.5	14.0	14.1	14.5	14.5	16.8	16.7	17.6	17.5
256	7.1	7.2	7.2	7.2	17.5	17.7	18.4	18.4	23.1	22.4	24.9	24.7
512	7.5	7.6	7.6	7.6	19.7	20.1	21.1	21.0	27.1	25.7	29.8	29.7
1024	7.6	7.8	7.8	7.8	20.7	21.2	22.4	22.3	29.1	28.2	32.9	32.8
2048	7.6	7.8	7.8	7.8	20.5	21.0	22.3	22.4	28.9	27.6	32.7	32.8

Table 7.5: Speed (wall-clock Gflop/s) and speed-up with $nb = 96$ and $nbi = 16$.

	One thread		Four threads		Eight threads	
p	Speed	Speed-up	Speed	Speed-up	Speed	Speed-up
64	5.5	1.0	10.1	1.8	11.8	2.1
128	6.5	1.0	14.5	2.2	17.5	2.7
256	7.2	1.0	18.4	2.5	24.7	3.4
512	7.6	1.0	21.0	2.6	29.7	3.9
1024	7.8	1.0	22.3	2.9	32.8	4.2
2048	7.8	1.0	22.4	2.9	32.8	4.2

HSL package HSL_MC64 (Duff and Koster 2001)). With the exception of NICE20MC, all the problems in this and subsequent tables are taken from the University of Florida Sparse Matrix Collection (Davis 2007) (NICE20MC is available from www.gridtlse.org). We use threshold $u = 0.01$, which is the default setting within HSL_MA77. It may be seen that a significant proportion of the total factorization time is spent within HSL_MA64 and the execution rate of HSL_MA64 is good on problems with large front sizes.

Table 7.6: Wall-clock times and speeds (Gflop/s) for the factorization phase of the multifrontal solver HSL_MA77 on one and eight threads with $nb = 96$ and $nbi = 16$.

Problem	n	Max front	One thread MA77 Time	One thread MA64 Time	One thread MA64 Gflop/s	Eight threads MA77 Time	Eight threads MA64 Time	Eight threads MA64 Gflop/s
helm2d03	392257	1024	2.7	1.6	3.0	2.7	1.6	3.0
bratu3d	27792	1521	4.2	3.4	3.7	3.7	2.9	4.2
qa8fk	66127	2075	4.4	3.5	6.2	3.0	2.0	10.4
halfb	224617	3240	16.3	11.1	6.3	13.6	5.9	12.0
Si5H12	19896	5551	43.7	32.8	4.7	38.1	26.5	5.8
NICE20MC	715923	11688	945	690	7.6	460	160	32.7
SiO2	155331	21406	2760	2298	5.8	1498	1033	12.8
Ga19As19H42	133123	18675	2018	1673	5.4	1186	834	10.8

We have performed some static pivoting experiments with HSL_MA77. Results are given in Table 7.7 for some problems of augmented type, that is the system matrix is of the form

$$\mathbf{A} = \begin{pmatrix} \mathbf{H} & \mathbf{B}^T \\ \mathbf{B} & \mathbf{0} \end{pmatrix}, \tag{7.1}$$

where $\mathbf{H}$ and $\mathbf{B}$ are large and sparse. In our tests, we compare using static = 0 (no static pivoting, which is the default) and static $= 10^{-6} * norm$, where $norm$ is the norm of the scaled system matrix. In Table 7.8 we report the scaled residuals

$$\frac{\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_\infty}{\|\mathbf{A}\|_\infty \|\mathbf{x}\|_\infty + \|\mathbf{b}\|_\infty}$$

where $\mathbf{x}$ is the computed solution and $\mathbf{b}$ the right-hand side. In our experiments, the right-hand side is selected so that the required solution is the vector of ones and we monitor $\max_i |1 - \mathbf{x}_i|$. Iterative refinement is performed until the scaled residual is less than 10^{-14}. We see that, for the chosen examples,

Table 7.7: Number of entries in $\mathbf{L}$ (in thousands) and times (in seconds) for the factorization and solve phases of HSL_MA77 on a single thread without and with static pivoting.

Problem	n	$nz(\mathbf{L})$ without	$nz(\mathbf{L})$ with	Factor without	Factor with	Solve without	Solve with
cvxqp3	17500	4884	3131	1.6	0.7	0.1	0.07
dtoc	24993	6701	227	1.3	0.1	0.2	0.02
mario001	38434	746	665	0.1	0.1	0.04	0.03
ncvxqp7	87500	39192	24707	43	13	0.8	0.6

static pivoting can significantly reduce the number of delayed pivots, resulting in much sparser factors and faster factorization and solution. The penalty is that several steps of iterative refinement are needed to restore accuracy. During testing of other examples using static pivoting, we found that iterative refinement was sometimes unable to restore the accuracy. Furthermore, the success of static pivoting in some cases

Table 7.8: Scaled residuals for HSL_MA77 without and with static pivoting, both before and after iterative refinement. The numbers in parentheses are the number of steps of iterative refinement.

	without		with	
Problem	before	after	before	after
cvxqp3	$1.3 * 10^{-10}$	$2.0 * 10^{-16}$ (1)	$1.7 * 10^{-06}$	$1.9 * 10^{-15}$ (3)
dtoc	$1.1 * 10^{-14}$	$7.4 * 10^{-17}$ (1)	$6.7 * 10^{-13}$	$1.1 * 10^{-16}$ (1)
mario001	$6.1 * 10^{-15}$	$6.1 * 10^{-15}$ (0)	$1.3 * 10^{-10}$	$3.2 * 10^{-16}$ (4)
ncvxqp7	$2.1 * 10^{-09}$	$1.9 * 10^{-16}$ (1)	$1.5 * 10^{-07}$	$3.1 * 10^{-16}$ (5)

was sensitive to the choice of `static`. This illustrates the importance of using static pivoting with care. Sometimes, it may be necessary to employ a more powerful refinement process (for a discussion, see Arioli, Duff, Gratton and Pralet, 2007).

We have also experimented with using relaxed pivoting (`umin` $< u$). For many problems of the form (7.1) we found that this did not help, that is, the number of delayed pivots (and hence the fill in $\mathbf{L}$) was not reduced (see also Duff and Pralet, 2007). Closer investigation showed that this was because, on many of the calls to HSL_MA64, the (1,1) block A_{11} of A (see (1.1)) comprised a matrix of all zeros so that, independently of u and `umin`, pivots could not be chosen. However, if the system matrix is of the form

$$\mathbf{A} = \begin{pmatrix} \mathbf{H} & \mathbf{B}^T \\ \mathbf{B} & -\delta\mathbf{I} \end{pmatrix}, \tag{7.2}$$

where δ is small, we found relaxed pivoting can be useful. In Table 7.9 results are presented for a subset of the matrices of the form (7.2) that are reported on by Schenk and Gartner (2006). For relaxed pivoting `umin` was set to 10^{-10}; we report the final value u_{final} of the relative pivot tolerance. For these problems, a single step of iterative refinement was needed to obtain scaled residuals of less than 10^{-14}. We see that the savings from relaxed pivoting are modest for our test examples; the main potential benefit is that the pivot sequence and data structures set up by the analyse phase of the solver do not need to be modified during the factorization.

Table 7.9: Number of entries in $\mathbf{L}$ (in thousands) and factorization times (in seconds) for HSL_MA77 on a single thread without and with relaxed pivoting. u_{final} denotes the final value of the relative pivot tolerance.

		$nz(L)$		Factor		
Problem	n	without	with	without	with	u_{final}
c-60	43640	1736	1650	0.34	0.32	$1.9 * 10^{-09}$
c-65	48066	1549	1495	0.30	0.28	$1.0 * 10^{-10}$
c-68	64810	2082	1962	4.04	3.85	$2.1 * 10^{-09}$
c-73	169422	5276	5124	1.23	1.18	$4.4 * 10^{-10}$

7.3 Comparison with LAPACK and preference for 2×2 pivots

LAPACK (Anderson, Bai, Bischof, Blackford, Demmel, Dongarra, Du Croz, Greenbaum, Hammarling, McKenney and Sorensen, 1999) contains two subroutines for Bunch-Kaufman factorization of indefinite symmetric matrices. For the packed form, `_sptrf` is available but is slow since it does not use blocking. For the full form, wasting about half the storage, there is `_sytrf`. For best performance, `_sytrf` needs a work array of size $n \times nb$, where nb is its block size (64 in our implementation), so its total storage requirement is for $n^2 + n \times nb$ reals. This should be compared with $n(n+1)/2$ reals for `_sptrf` and $n^2/2 + 3n(nb+1)/2$ reals for HSL_MA64 when its work array is included.

Table 7.10: Comparison between HSL_MA64 and the LAPACK routine dsytrf.

	Wall-clock Gflop/s					No. 2×2 pivots		
	dsytrf	HSL_MA64				dsytrf	HSL_MA64	
	1 thread	1 thread		8 threads				
favour:		2×2	1×1	2×2	1×1		2×2	1×1
n								
500	4.4	4.9	4.6	5.5	4.6	164	189	31
1000	5.6	6.1	5.9	10.5	9.1	312	369	87
2000	6.4	6.9	6.8	19.5	17.8	666	769	182
4000	7.0	7.6	7.5	29.9	28.4	1403	1473	366
8000	7.3	7.8	7.8	38.0	36.1	2902	2910	758
16000	7.2	7.9	7.8	43.2	39.9	6050	5830	1611

In Table 7.10, we show the performance of dsytrf and HSL_MA64 for 6 values of n within the range of front sizes that we have encountered. For HSL_MA64, we also show results for a version that uses a 1×1 pivot when both a 1×1 and a 2×2 pivot are available. It may be seen that this chooses far less 2×2 pivots and is slower because more single-column updates are performed. It is interesting that the version that favours 2×2 pivots chooses about as many 2×2 pivots as dsytrf. On a single thread, HSL_MA64 is slightly faster than dsytrf and of course it has the merit of using about half as much memory. We tried varying the block size nb in the range $[48, 120]$ and observed little difference in the performance. We found that dsptrf was some ten times slower than dsytrf on these problems.

Finally, Table 7.11 presents a comparison between favouring 2×2 over 1×1 pivots within HSL_MA77. In this table, we report the number of 2×2 pivots used during the factorization, the number of gigaflops performed using vector and block operations, and the wall-clock times for the factorization phases of HSL_MA64 and HSL_MA77. Note that the vector count includes flops performed by _gemm with internal matrix dimension 2. We see that, as expected, favouring 2×2 pivots leads to significantly more 2×2 pivots being used and this results in a modest reduction in the HSL_MA64 time. The block count is similar for both pivoting strategies while there is a reduction in the (much smaller) vector count if 2×2 pivots are preferred.

Table 7.11: Comparison between favouring 2×2 over 1×1 pivots within HSL_MA77 on eight threads.

	Favour 2×2					Favour 1×1				
	2×2	gigaflops		Time		2×2	gigaflops		Time	
	pivots	vector	block	HSL_MA64	HSL_MA77	pivots	vector	block	HSL_MA64	HSL_MA77
ncvxqp3	38176	17.7	54.5	23.5	35.9	5526	17.9	54.3	26.6	39.1
kkt	791462	16.2	582	75.2	187	180291	20.5	575	78.4	191
af_shell10	732433	4.6	410	28.5	102	0	5.8	410	29.3	108
NICE20MC	349734	14.8	5284	160	470	0	19.4	5282	167	476

8 Concluding remarks

We have shown that it is possible to achieve good execution speed and good accuracy for the partial or complete factorization and solution of sparse symmetric indefinite sets of linear equations by carefully combining blocking with threshold partial pivoting and the use of both 1×1 and 2×2 pivots. We have also considered parallelization with OpenMP. For large dense systems, the execution speed of HSL_MA64 on one thread is quite close to optimal and the speed-up on eight threads exceeds four.

For better parallelization of the sparse symmetric indefinite problem, we are planning to follow our experience in the definite case (see Hogg, Reid and Scott, 2009) of dividing the computation into a set of individually scheduled tasks, each of which involves updating a single block of the matrix. It is our belief that these ideas, too, can be combined with threshold pivoting and the use of 1×1 and 2×2 pivots.

All the HSL codes referred to in this paper are part of HSL 2007. Use of HSL requires a licence. Licences are available without charge to individual academic users for their personal (non-commercial) research and for teaching; for other users, a fee is normally charged. Details of how to obtain a licence together with information on all HSL packages are available at `www.cse.clrc.ac.uk/nag/hsl/`.

9 Acknowledgements

We would like to express our appreciation to our colleagues Iain Duff and Jonathan Hogg for many helpful discussions and to Tim Davis for his amazing collection of matrices.

References

B.S. Andersen, J.A. Gunnels, F.G. Gustavson, J.K. Reid, and J. Wasniewski. A fully portable high performance minimal storage hybrid format Cholesky algorithm. *ACM Trans. Math. Softw.*, **31**, 201–207, 2005.

E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edn, 1999.

M. Arioli, I. S. Duff, S. Gratton, and S. Pralet. A note on GMRES preconditioned by a perturbed LDLT decomposition with static pivoting. *SIAM J. Sci. Comput.*, **25**(5), 2024–2044, 2007.

Cleve Ashcraft, Roger G. Grimes, and John G. Lewis. Accurate symmetric indefinite linear equation solvers. *SIAM J. Matrix Anal. Appl.*, **20**(2), 513–561, 1999.

J. R. Bunch and L. Kaufman. Some stable methods for calculating inertia and solving symmetric linear systems. *Math. Comp.*, **31**, 163–179, 1977.

Tim Davis. The University of Florida Sparse Matrix Collection. Technical Report, University of Florida, 2007. `http://www.cise.ufl.edu/~davis/techreports/matrices.pdf`.

J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, **16**(1), 18–28, March 1990*a*.

J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, **16**(1), 1–17, March 1990*b*.

J. Dongarra, J. Du Croz, S. Hammarling, and Richard J. Hanson. Algorithm 656: An extended set of Fortran Basic Linear Algebra Subroutines. *ACM Trans. Math. Soft.*, **14**(1), 18–32, March 1988.

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

I.S. Duff and S. Pralet. Towards a stable static pivoting strategy for the sequential and parallel solution of sparse symmetric indefinite systems. *SIAM J. Matrix Anal. Appl.*, **29**, 1007–1024, 2007.

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

I.S. Duff, N.I.M. Gould, J.K. Reid, J.A. Scott, and K. Turner. Factorization of sparse symmetric indefinite matrices. *IMA J. Numer. Anal.*, **11**, 181–2044, 1991.

E. Elmroth, F. Gustavson, I. Jonsson, and B. Kågström. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Review*, **46**(1), 3–45, 2004.

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

J.D. Hogg, J.K. Reid, and J.A. Scott. A DAG-based sparse Cholesky solver for multicore architectures. Technical Report RAL-TR-2009-004, Rutherford Appleton Laboratory, 2009.

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

C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Soft.*, **5**, 308–323, 1979.

X.S. Li and J.W. Demmel. Making sparse Gaussian elimination scalable by static pivoting. *in* 'Proceedings of Supercomputing', Orlando, Florida, 1998.

J.K. Reid and J.A. Scott. An efficient out-of-core sparse symmetric indefinite direct solver. Technical Report RAL-TR-2008-024, Rutherford Appleton Laboratory, 2008.

J.K. Reid and J.A. Scott. An efficient out-of-core multifrontal solver for large-scale unsymmetric element problems. *Intl. J. Numer. Methods Engrng.*, **77**(7), 901–921, 2009*a*.

J.K. Reid and J.A. Scott. An out-of-core sparse Cholesky solver. *ACM Trans. Math. Softw.*, **36**(2), 2009*b*. Article 9, 33 pages.

O. Schenk and K. Gartner. On fast factorization pivoting methods for sparse symmetric indefinite systems. *Electronic Trans. Numer. Anal.*, **23**, 158–179, 2006.

J.A. Scott. Scaling and pivoting in an out-of-core sparse direct solver. Technical Report RAL-TR-2008-016, Rutherford Appleton Laboratory, 2008.