Partial Factorization of a Dense Symmetric Indefinite Matrix

JOHN K. REID and JENNIFER A. SCOTT, Rutherford Appleton Laboratory

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 $2 \times 2$ pivots as well as $1 \times 1$ pivots is equally well established. In this article, the challenge of using these three ideas (packing, blocking, and pivoting) together is addressed to achieve stable factorizations of large real-world symmetric indefinite 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.

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

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Symmetric linear systems, indefinite matrices, $LDL^T$ factorization, frontal, multifrontal

ACM Reference Format:


1. INTRODUCTION

In this article, we consider the efficient and stable factorization of a dense symmetric indefinite matrix $A$ of order $n$ that has the form

$$
\begin{pmatrix}
A_{11} & A_{21}^T \\
A_{21} & A_{22}
\end{pmatrix},
$$

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

$$
P^T A P = LDL^T
$$

where $P$ is a permutation matrix of the form

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

The work of J. A. Scott was funded by the EPSRC Grant EP/E053351/1.

Authors’ address: J. K. Reid and J. A. Scott, Computational Science and Engineering Department, Atlas Centre, Rutherford Appleton Laboratory, Oxon OX11 0QX, England; email: [John.Reid, Jennifer.Scott]@stfc.ac.uk.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2011 ACM 0098-3500/2011/12-ART10 $10.00

DOI 10.1145/2049673.2049674 http://doi.acm.org/10.1145/2049673.2049674
with $P_{11}$ of order $p$, $L$ is a unit lower-triangular matrix

$$
\begin{pmatrix}
L_{11} & 1 \\
L_{21}
\end{pmatrix},
$$

(1.4)

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

$$
D = \begin{pmatrix}
D_{11} & 0 \\
0 & S_{22}
\end{pmatrix},
$$

(1.5)

where $D_{11}$ is a block diagonal matrix of order $q$ with blocks of order 1 or 2 corresponding to $1 \times 1$ and $2 \times 2$ pivots 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:

$$
Lx = b, \quad \begin{pmatrix}
D_{11} & 0 \\
0 & I
\end{pmatrix}x = b, \quad \text{and} \quad L^T x = b
$$

(1.6)

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

Since the early 1990s, the use of block-based algorithms have become standard within state-of-the-art software for dense linear algebra computations (see, e.g., LAPACK [Anderson et al. 1999] and FLAME [Van Zee et al. 2009; Van Zee 2010]). Many modern dense linear algebra algorithms are designed to perform most of their computation using BLAS-3 subroutines [Dongarra et al. 1990], thereby guaranteeing high performance as well as portability. Some researchers have aimed to improve performance further through the introduction of new BLAS subroutines (e.g., Gustavson et al. [1998] employ recursive blocked BLAS). However, we wish to design our partial factorization and solve algorithms to exploit the standard BLAS-3 subroutines. We also want to limit memory requirements and, since $A$ is symmetric, to store only half the matrix. Unfortunately, there are no BLAS-3 subroutines for $A$ held in packed lower triangular form. Thus, a key challenge that we face is organising the computation to use (approximately) $\frac{1}{2}n^2$ real storage whilst also incorporating blocking and making maximum use of high-level BLAS. We explore two approaches. The first is a block approach that builds on the work of Andersen et al. [2005] for the factorization of positive-definite matrices while the second is based on a recursive formulation of the factorization.

Another important issue we face is that of pivoting for numerical stability. To ensure robustness of our indefinite factorization algorithm while maintaining symmetry, we need to incorporate threshold partial pivoting using $1 \times 1$ and $2 \times 2$ pivots [Duff et al. 1991]. Accommodating pivoting and efficiently performing the symmetric permutations that are needed once a pivot has been chosen represents a substantial challenge when $A$ is held in packed form. Designing and implementing algorithms that succeed in this is the main achievement that we report in this article. We note that the HSL multifrontal solver MA57 [Duff 2004] uses the same pivotal strategy but holds its matrices in full form.

The code that we have designed and developed for the partial factorization and solution of symmetric indefinite systems is collected into the module HSL MA64 within the HSL mathematical software library [HSL 2011]. We expect the principal application to be within the multifrontal method for factorizing sparse symmetric indefinite matrices and our data formats have been designed to take this into account. However, HSL MA64 is also available for other applications, including the case where a complete factorization is needed ($p = n$). LAPACK contains two subroutines for Bunch-Kaufman factorization of indefinite symmetric matrices; these are for the case $p = n$ only. The Bunch-Kaufman factorization has stability limitations, see Section 3.3. Furthermore,
Table I. Specifications of Our Test Machine, a 2-Way Quadcore Harpertown Architecture

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Intel(R) Xeon(R) CPU E5420</td>
</tr>
<tr>
<td>Clock</td>
<td>2.50 GHz</td>
</tr>
<tr>
<td>Cores</td>
<td>2 × 4</td>
</tr>
<tr>
<td>Theoretical peak (1/8 cores)</td>
<td>10 / 80 Gflop/s</td>
</tr>
<tr>
<td>dgemm peak (1/8 cores)</td>
<td>9.3 / 72.8 Gflop/s</td>
</tr>
<tr>
<td>Memory</td>
<td>8 GB</td>
</tr>
<tr>
<td>Compiler</td>
<td>Intel 11.0 with option -fast</td>
</tr>
<tr>
<td>BLAS and LAPACK</td>
<td>Intel MKL 10.1</td>
</tr>
</tbody>
</table>

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

one LAPACK subroutine is slow because it does not use BLAS-3 subroutines and the other stores the whole matrix. Therefore, one of the roles of HSL_MA64 is to fill a gap that currently exists in LAPACK.

The numerical experiments reported in this article were performed using 64-bit real arithmetic on our multicore test machine, details of which are given in Table I. For timing, we used wall-clock times in seconds on a lightly loaded machine. When working with blocks, redundant floating-point operations in the upper-triangular part of the matrix are excluded when computing speeds in Gflop/s.

The rest of this article is organized as follows. In Section 2, we discuss the requirement within a multifrontal sparse direct solver to perform the partial factorization and partial solution of dense systems. In Section 3, the pivoting strategy that we employ is explained. Section 4 describes a simple implementation of the factorization without blocking. Section 5 then presents and compares two block approaches, a block column approach and a recursive approach. The former is chosen for implementation within HSL_MA64. Further implementation details, including parallel working, are considered in Section 6. Numerical experiments using HSL_MA64 are reported in Section 7 and concluding remarks are made in Section 8.

2. THE MULTIFRONTAL METHOD

The multifrontal method (see, e.g., 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 operations needed for the frontal matrix are the partial factorization (1.2) and the partial solutions (1.6). These are the tasks that are the focus of this article.

In recent years, our main interest has been the solution of large-scale problems. In such cases, even with a good elimination order, the frontal matrices can become very large and, for economy of storage, it is essential to take advantage of symmetry and hold only the lower (or upper) triangular part of each frontal matrix. For example, the HSL multifrontal solver HSL_MA77 [Reid and Scott 2008, 2009b] holds the lower triangular part by columns.

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” \( S_{22} \) (see (1.5)) 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. We refer to this later frontal matrix as the “parent.” It is convenient for the merging of the generated elements to hold the lower triangular part of \( S_{22} \) by columns.

If \( q < p \) pivots are selected, \( S_{22} \) 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. 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. It is therefore advantageous to defer looking for pivots in these rows and columns until all the other candidates 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 also affected by delayed pivots. The number of delayed pivots may be limited by using a relaxed pivot threshold; failing this, static pivoting may be employed, that is, forcing pivots that do not satisfy the stability test to be chosen, perhaps after modification (see, e.g., Li and Demmel [1998]). These possibilities are discussed further in Section 3.2. The intention is to reduce the fill in the factors, probably at the expense of an iterative procedure (such as iterative refinement) for each solution to obtain the required accuracy.

3. PIVOTING AND STABILITY
In this section, we review the pivoting strategies we use within our code.

3.1. Threshold Pivoting
We choose the pivots one-by-one, with the aim of limiting the size of the entries in $L$:

$$|l_{ij}| < u^{-1},$$

(3.1)

where the threshold $u$ is a user-set value in the range $0 \leq u \leq 1.0$. In the case where $u$ is zero, this is interpreted as requiring that the entries be finite. Suppose that $q$ denotes the number of rows and columns of $D$ found so far (i.e., the number of $1 \times 1$ pivots plus twice the number of $2 \times 2$ pivots). We use the notation $a_{i,j}$, 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. For a $1 \times 1$ pivot in column $j = q + 1$, the requirement for inequality (3.1) corresponds to the usual threshold test

$$|a_{q+1,q+1}| > u \max_{i=q+1} |a_{i,q+1}|.$$  

(3.2)

Following Duff et al. [1991], the corresponding test for a $2 \times 2$ pivot is

$$\begin{vmatrix}
  a_{q+1,q+1} & a_{q+1,q+2} \\
  a_{q+1,q+2} & a_{q+2,q+2}
\end{vmatrix}^{-1} \left( \begin{vmatrix}
  \max_{i>q+2} |a_{i,q+1}| \\
  \max_{i>q+2} |a_{i,q+2}|
\end{vmatrix} < \left( \begin{vmatrix}
  u^{-1} \\
  u^{-1}
\end{vmatrix} \right) \right),$$

(3.3)

where the absolute value notation for a matrix refers to the matrix of corresponding absolute values. 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 $u$.

If necessary, one or two symmetric permutations are used to bring a suitable pair of rows and columns to positions $q + 1$ and $q + 2$ or a symmetric permutation is used to bring a suitable row and column to position $q + 1$. Of course, the presence of larger entries beyond row $p$ may mean that it is not possible to choose $p$ stable pivots, so that the final value of $q$ may be less than $p$.

3.2. Relaxed and Static Pivoting
The strategy we use 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 \( 1 \times 1 \) or \( 2 \times 2 \) candidate pivot satisfies the test (3.2) or (3.3) but the pivot that is nearest to satisfying the test would satisfy it with \( u = v \geq \text{umin} \), the pivot is accepted and \( u \) is reduced to \( v \). The new value of \( u \) is employed thereafter. We set the default value for \( \text{umin} \) to 1, which avoids relaxed pivoting. If \( p = n \), we treat a value that exceeds 0.5 as 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 \( 1 \times 1 \) or \( 2 \times 2 \) candidate pivot satisfies the test (3.2) or (3.3) even after relaxing the value of \( u \), the \( 1 \times 1 \) pivot that is nearest to satisfying the test is accepted. If its absolute value is less than another user-set threshold \( \text{static} \), it is given the value that has the same sign but absolute value \( \text{static} \).

### 3.3. Stability

Stability of the factorization of symmetric indefinite systems was considered in detail by Ashcraft et al. [1999], who showed that bounding the size of the entries of \( L \), together with a backward stable scheme for solving \( 2 \times 2 \) linear systems, suffices to show backward stability for the entire process. They found that the widely used strategy of Bunch and Kaufmann [1977] does not have this property.

For unsymmetric matrices, rook pivoting [Neal and Poole 1992; Foster 1997; Poole and Neal 2000] offers a strategy that is intermediate between partial and complete pivoting in terms of both efficiency and stability. To locate a pivot, threshold 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 threshold \( u \) times the absolute value of any other entry in the row or column in which it lies. Gill et al. [2005] report that, provided \( u \) is chosen to be sufficiently close to 1, the rank revealing properties of threshold 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 [2009a] and Scott [2010]) has been satisfactory, too; we found the factors usually had fewer entries, the residuals were usually smaller, while the factorization was sometimes significantly faster. Our default pivoting strategy is the symmetric equivalent of threshold rook pivoting in the sense that the pivot is large compared with other entries in its rows and columns as measured by inequality (3.2) or inequality (3.3).

### 4. SIMPLE APPROACH WITHOUT BLOCKING

Having discussed the pivoting strategy to be used, we now consider implementing the symmetric indefinite factorization in the simple case where \( A \) is not singular or nearly so and \( n \) is not large enough for blocking to be beneficial. Rearranging \( A \) to be in full storage, as an \( n \times n \) matrix with valid entries only in its upper or lower triangular part, allows the use of Level-2 BLAS and Level-3 BLAS. In this simple case, there is an advantage in a left-looking implementation that works on the lower-triangular part, where each column \( m \) of this part of \( A \) is not altered until just before column \( m \) is first searched for a pivot. Performing all the alterations together reduces the movement of cache lines between memory levels.

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. We refer to these as active columns. For column \( m \), the update required is

\[
a_{mn,m} \leftarrow a_{mn,m} - l_{mcn,1:q}d_{1:q,1:q}(l_{mc,1:q})^T.
\]
Here and elsewhere, we use section notation in subscripts to refer to submatrices. Note that the submatrices sometimes have size zero. After forming the vector
\[ u_{1q,m} = d_{1q,1q}(l_{m,1q})^T, \]
we can express the update as
\[ a_{m,n,m} \leftarrow a_{m,n,m} - l_{m,1q}u_{1q,m}, \quad (4.1) \]
which can be performed using the BLAS-2 subroutine \_gemv [Dongarra et al. 1988].

Note that to test for a 1 × 1 pivot in row and 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 already be fully updated. If the 1 × 1 pivot is accepted, we interchange rows and columns \( m \) and \( q+1 \), calculate column \( q+1 \) of \( D \) and \( L \), then perform the right-looking update
\[ a_{q+2n,q+2m} \leftarrow a_{q+2n,q+2m} - l_{q+2n,q+1}d_{q+1,q+1}(l_{q+2m,q+1})^T. \]
of the active columns. After forming the vector
\[ u_{q+1,q+2m} = d_{q+1,q+1}(l_{q+2m,q+1})^T, \]
we can express the update as
\[ a_{q+2n,q+2m} \leftarrow a_{q+2n,q+2m} - l_{q+2n,q+1}u_{q+1,q+2m}. \quad (4.2) \]
This could be performed by the BLAS-2 subroutine \_ger, but using a sequence of calls to the BLAS-1 subroutine \_axpy avoids wasted operations in the upper-triangular part and there are no data-movement disadvantages.

If there are two or more active columns \( (m > q + 1) \), 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 interchange rows and columns to move the selected two forward, calculate columns \( q+1 \) and \( q+2 \) of \( D \) and \( L \), and form the 2-rowed matrix
\[ u_{q+1,q+2,q+3m} = d_{q+1,q+2,q+1}d_{q+1,q+2}(l_{q+3m,q+1})^T. \]
We can then perform the double update
\[ a_{q+3n,q+3m} \leftarrow a_{q+3n,q+3m} - l_{q+3n,q+1}u_{q+1,q+2}u_{q+1,q+2} \]
using the BLAS-3 subroutine \_gemm. The performance might be as much as twice that for (4.2), but will not be as good as it is for full-sized blocks and we will regard it as a BLAS-2 update.

If a pivot is chosen, we increment \( q \) by one or two once the updates are complete. If no pivot is chosen, column \( m \) is retained as an active column.

Unless \( m = p \), we increment \( m \) by one and continue. If \( m = p \) and \( q = p \), the calculation of \( L \) and \( D \) is complete. If \( m = p \) and \( q < p \), we restart the search from column \( q+1 \) with this as the only active column \( (m \) is reset to \( q+1) \). We continue in this way until either \( q = p \) or \( q = p \) successive columns fail to provide a pivot. After each pivot has been accepted and the consequent update operations applied, we could have retested all the active columns, but do not do so because it is likely to lead to repeated unsuccessful testing. They will be retested once \( m \) reaches \( p \) and we restart the search.

When searching for a pivot in column \( m \), there is an advantage in testing for a 2 × 2 pivot first because choosing a 2 × 2 pivot reduces the number, \( m - q \), of active columns by two. Our order of tests is illustrated in Figure 1 for a case where \( m \) began with the value 4 but no stable pivot was found in row/column 4, \( m \) was then incremented to 5, and the pivot (5, 4) was found to be acceptable. Note that our strategy results in all possible pivots eventually being considered. Numerical results that demonstrate the advantage of favoring 2 × 2 pivots over 1 × 1 pivots are given in Section 7.3 (Table XII).
Partial Factorization of a Dense Symmetric Indefinite Matrix

### Fig. 1

An illustration of the order in which potential $2 \times 2$ pivots are tested. Each potential $2 \times 2$ pivot is represented by its off-diagonal entry.

---

**ALGORITHM 1:** Find the pivot. $L$, $D$, $A$, $p$, $q$, and $m$ are accessed from the environment of the call.

```fortran
subroutine find_pivot(piv_size)
piv_size = 0
do test = 1, p - q      
m = m + 1; if (m > p) m = q + 1 ! left-looking
if (there is a $j$ in $[q + 1:m - 1]$ such that $(a_{ij}, a_{jm})$ is OK as a $2 \times 2$ pivot) then
    piv_size = 2; call swap ($q + 1$, $j$); call swap ($q + 2$, $m$)
calculate columns $q + 1$, $q + 2$ of $D$ and $L$
    return
else if ($a_{im}$ is OK as a $1 \times 1$ pivot) then
    piv_size = 1; call swap ($q + 1$, $m$)
calculate column $q + 1$ of $D$ and $L$
    return
end if
end do
end subroutine find_pivot

subroutine swap(j, m)
Interchange rows and columns $j$ and $m$
end subroutine swap
```

**ALGORITHM 2:** Simple factorization algorithm (no blocking); $q$ 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 $A_{11}$
Output: $q$, $P$, $L$ and $D$ (see (1.3)-(1.5))
Initialise: $q = 0$; $m = 0$
do while ($q < p$)
call find_pivot(piv_size) ! See Algorithm 1
if (piv_size == 0) exit ! Failed to find a pivot
$q = q + piv_size$
update the active columns $q + 1$ to $p$ ! right-looking
end do
apply outstanding updates to columns $p + 1$ to $n$ ! left-looking
```

Our procedure for selecting pivots is summarized using pseudo-Fortran in Algorithm 1 and the factorization algorithm is summarized in Algorithm 2.

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 our code more than a million times during the multifrontal factorization of 56 large sparse indefinite matrices with the threshold parameter $u = 0.01$, we found that the total number of searches in a call to our code exceeded $4p$ only twice and never exceeded...
Fig. 2. Block column format when \( n = 10 \) and \( nb = 3 \), and after rearrangement when \( q = 8 \).

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 also known, but the trailing part still needs to be calculated. Note that no interchanges are performed in this part of the matrix (see Eq. (1.2) and (1.3)). The calculation may be performed by forming

\[
U_{1:q,p+1:n} = D_{1:q,1:q}(l_{p+1:n,1:q})^T
\]

and then performing the update

\[ a_{p+1:n,p+1:n} \leftarrow a_{p+1:n,p+1:n} - l_{p+1:n,1:q}U_{1:q,p+1:n} \]

using the BLAS-3 subroutine \texttt{gemm}.

5. FACTORIZATION INCORPORATING BLOCKING

In this section, we consider two possible approaches to incorporating blocking. We will refer to these as the \textit{block column} approach and the \textit{recursive} approach.

5.1. Block Column Approach

For the Cholesky factorization of a positive-definite dense symmetric matrix, Andersen et al. [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 \( nb \).

In the indefinite case, there is intense activity in columns \( q + 1 \) to \( m \) (see Algorithm 2) which makes it desirable to hold the columns contiguously in memory. Furthermore, the row and column interchanges that are needed in the indefinite case are easier to program and more efficient in execution if each column is held contiguously in memory. We therefore propose using 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 block columns are \( n, n - nb \), etc. This format is illustrated on the left of Figure 2. We choose not to pack the lower triangular parts of the blocks on the diagonal in order to simplify the code and allow the use of more efficient BLAS without any further rearrangements. It may readily be verified that the total waste is limited to \( n(nb - 1)/2 \).

Prior to performing any eliminations, we rearrange \( A \) to block column format. We begin the factorization as in Section 4 and summarized in Algorithm 2, except that account must be taken of the blocking once \( m \) exceeds \( nb \). We can either update at once all the later columns (right-looking) or delay the updates on each block column until it is needed for pivoting and then perform them together (left-looking). The right-looking alternative provides more scope for parallelization (Section 6.4), so that is our choice.
Whenever a $1 \times 1$ pivot is chosen in the last column of a block or a $2 \times 2$ pivot is chosen that includes the last column of a block, we apply the $nb$ pivot operations associated with the block to columns $m + 1 : n$. If the block starts at column $c$, we apply the temporary matrix

$$u_{1:nb,m+1:n} = \text{d}_{c+1x+nb,c+1x+nb}(l_{m+1n,c+1x+nb})^T.$$ 

For a block column that spans columns $j$ to $k$, we apply the BLAS-3 subroutine $\text{gemm}$ thus

$$a_{j:n,j:k} \leftarrow a_{j:n,j:k} - l_{j:n,c+1x+nb}u_{1:nb,j:k}. \quad (5.1)$$

This is much more efficient than waiting for the updates to be done as in Eq. (4.1) using the BLAS-2 subroutine $\text{gemv}$. 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 $2 \times 2$ pivot spanning two block columns. Once the BLAS-3 calls are complete, 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.

Our experience is that quite large values of the block size $nb$ are desirable, for example, 96 (see Section 7). Retaining the block column format for the matrix $\begin{pmatrix} L_{11} & L_{12} \\ L_{21} & L_{22} \end{pmatrix}$ on return would therefore significantly increase the memory needed. We therefore rearrange this matrix once the partial factorization is complete. For a block column with $mb$ columns, where $mb = nb$ except for the trailing block which is usually smaller, it is desirable to hold rows $mb + 1$ onwards in a rectangular form. This allows the partial solution operations to be performed with the help of the BLAS-2 subroutine $\text{gemv}$ for a single right-hand side and the BLAS-3 subroutine $\text{gemm}$ for multiple right-hand sides. We therefore rearrange each block column to consist of the rows 1 to $mb$ packed by columns followed by the rest held by columns. This form is as economical of storage as that in Andersen et al. [2005] and is illustrated on the right of Figure 2. During partial solutions, the BLAS-2 subroutine $\text{tpsv}$ may be used for the packed part.

We rearrange columns of $S_{22}$ to packed lower triangular format because, in the multifrontal method, this is a “generated element” that must be merged with other generated elements. This merging operation would be awkward with blocking in place.

5.1.1. 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 $\text{gemv}$. Indeed, if $p \leq nb$, no rank-$nb \text{gemm}$ calls are made while calculating $L$ and $D$ and it would have been faster to use 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 within 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 outer 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, e.g., Elmroth et al. [2004]) since the data format is not affected. Whenever an integer multiple of $nbi$ columns of $L$ have been found, we use the BLAS-3 subroutine $\text{gemm}$ to update the columns of the block column from column $m + 1$ onwards.

We summarize the block column factorization algorithm incorporating both outer and inner blocking in Algorithm 3.

5.2. Recursive Factorization

Elmroth et al. [2004] discuss a wide variety of algorithms for dense matrices in which the matrix is recursively split into submatrices, usually by bisection. For our application, this has the potential for using BLAS-3 calls for very large submatrices and avoiding the need for detailed coding of addressing within blocks. We have used this
ALGORITHM 3: Summary of the block column factorization algorithm; \( q \) is the number of pivots chosen so far.

**Input:** matrix \( A \) in the form (1.1) and the order \( p \) of \( A_{11} \)
**Output:** \( q, P, L \) and \( D \) (see (1.3)–(1.5))
**Initialise:** \( q = 0; m = 0 \)

**do while** \( (q < p) \)
  **call** find pivot(\( piv \_size \)) ! See Algorithm 1
  **if** \( (piv \_size == 0) \) **exit** ! Failed to find a pivot
  \( q = q + piv \_size \)
  update the active columns \( q + 1 \) to end of inner block ! right-looking
  **if** (outer block complete) **then**
    update outer block columns (Section 5.1) ! right-looking
  **else if** (inner block complete) **then**
    update inner block columns (Section 5.1.1) ! right-looking
  **end if**
**end do**

apply outstanding updates to columns \( p + 1 \) to \( n \) ! left-looking

ALGORITHM 4: Factorization algorithm using recursion.

**Input:** \( A, p, nr \)
**Output:** \( q, P, L \) and \( D \) (see (1.3)–(1.5))
**Initialise:** \( q = 0; m = 0 \)
**call** rec_factor (1, \( p \))

**contains**
**recursive subroutine** rec_factor (\( r_1, r \))
  **Accessed from host and not changed:** :: \( nr \)
  **Accessed from host and changed:** :: \( q, m, l_{1:n,1:q}, d_{1:q,1:q}, a_{1:n,q+1:x} \)
  **if** \( (r - r_1 > nr) \) **then**
    \( q_0 = q + 1 \)
    \( r_0 = (r_1 + r)/2 \)
    **call** rec_factor (\( r_1, r_0 - 1 \))
    update columns \( r_0 \) to \( r \):
    \( a_{0:n,r_0:x} \leftarrow a_{0:n,r_0:x} - l_{0:n,q_0:q_0} d_{q_0:q_0:q_0} (l_{r_0:r_0:q})^T \)
    **call** rec_factor (\( r_0, r \))
  **else**
    **do while** \( (q < r) \)
      **call** find pivot(\( piv \_size \)) ! See Algorithm 1
      **if** \( (piv \_size == 0) \) **exit** ! Failed to find a pivot
      \( q = q + piv \_size \)
      update the active columns \( q + 1 \) to \( m \) ! right-looking
    **end do**
    \( m = r \) ! Ensure that rejected columns are not retested next time
  **end if**
**end subroutine** rec_factor

idea and incorporated pivoting to yield the algorithm summarized in Algorithm 4. It is given a range of columns \( (r_1, r) \) from \( (1, p) \) and accesses data from the environment of its call, including \( q \), the number of pivots found so far. If \( q_0 \) is the value of \( q + 1 \) on entry, the subroutine acts on columns \( q_0 \) to \( r \), looking for as many pivots as possible here, and updating \( q \) as it goes. On exit, columns 1 to \( q \) of \( L \) and \( D \) have been calculated, and columns \( q + 1 \) to \( r \) are up-to-date.
Partial Factorization of a Dense Symmetric Indefinite Matrix

Fig. 3. The updates for a case with $n = 4000$, $p = 2048$.

If $r - r_1 \leq nr$ (a suitable value for $nr$ is the inner block size $nbi$ introduced in Section 5.1.1), we proceed as in Algorithm 2. Otherwise, we bisect $(r_1, r)$ to find $r_0 = (r_1 + r)/2$, apply the recursive algorithm to the first half, $(r_1, r_0 - 1)$, update the second half, $(r_0, r)$,

$$a_{n,n,r:x} \leftarrow a_{n,n,r:x} - l_{n,q,q}d_{q,q}(l_{x,q,q}q)^T,$$  \hspace{1cm} (5.2)

then apply the recursive algorithm to the second half $(r_0, r)$.

It is convenient to use the notation

$$u_{q,q,r:x} = d_{q,q}(l_{x,q,q}q)^T$$  \hspace{1cm} (5.3)

for the upper-triangular submatrix in Eq. (5.2). In Figure 3, we picture these submatrices for a matrix with $n = 4000$, $p = 2048$. One of them is shown shaded differently, together with the corresponding submatrices $l_{n,q,q}$ and $a_{n,n,r:x}$. We note that all the updates involving $u_{q,q,r:x}$ are done together, so there is no overlap between the submatrices in the upper triangular part of Figure 3.

Note that the do while loop at the lowest level of the recursion will be visited successively for $r$ at the bisection points of the interval $(1, p)$ in increasing order. Hence, setting $m$ to $r$ at the end of the loop ensures that the next time it is visited, time will not be wasted searching columns within which no pivots are available.

If $p$ is large, most of the work will be within the update steps (5.2) and having many large submatrices appears very desirable for the application of BLAS-3. Unfortunately, no BLAS-3 subroutines are available for this update when the matrix is in packed form, so it is necessary to subdivide $a_{n,n,r:x}$ and $l_{n,q,q}$ into block columns of size up to $nb$ and $u_{q,q,r:x}$ into blocks of size up to $nb \times nb$. For each block column of $l_{n,q,q}$ and block of $u_{q,q,r:x}$, we make a copy in rectangular form in a buffer, then apply _gemm, accumulating the update in another rectangular buffer. Finally, the accumulated matrix is added into
Table II. Comparison of the Number of \texttt{gemm} Calls and Wall-Clock Times (seconds) for the Recursive and Block Column Approaches

<table>
<thead>
<tr>
<th></th>
<th>Recursive packed</th>
<th>Recursive blocked</th>
<th>Block column</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{gemm} calls</td>
<td>617</td>
<td>637</td>
<td>544</td>
</tr>
<tr>
<td>\texttt{gemm} time</td>
<td>2.280</td>
<td>2.367</td>
<td>2.235</td>
</tr>
<tr>
<td>Total time</td>
<td>2.926</td>
<td>2.636</td>
<td>2.452</td>
</tr>
</tbody>
</table>

the packed matrix. Unfortunately, this copying to and from buffers makes the recursive version slower than the block column approach. We illustrate this in the second and fourth columns of Table II for the partial factorization of randomly generated matrices of order 4000 with $p = 2048$, $nb = 120$, and $nbi = nr = 20$.

Much of the extra time taken by the recursive code is due to the copying of blocks of packed columns of $L$ to a buffer (0.383 secs for the case in Table II). We therefore tried rearranging the new columns of $L$ to block column format after each recursive call with $r - r_1 \leq nr$. This avoids the copying of $l_{p:n, q:q}$, but its block columns now have to be aligned with those of block column format, so that in general the first as well as the last blocks have less than $nb$ columns. For the case shown in Table II, the total number of \texttt{gemm} calls increased from 617 to 637. Nevertheless, the total time improved significantly, while still being greater than that of the block column approach, see Table II.

In the block column approach of Algorithm 3, updates are delayed until a complete block column of $L$ is available, so there are less calls of \texttt{gemm} (see Table II) and they are more efficient. The problem of having undersized blocks can be mitigated slightly by choosing the recursion points at integer multiples of $nb$, but this will not help if $q < r$ on completion of a recursion. In our test case, this reduced the number of calls of \texttt{gemm} to 601, but had no noticeable affect on the time.

Based on our experiments, we have chosen to implement the block column approach (using inner and outer block sizes) within \texttt{HSL MA64}. Were optimized BLAS-3 subroutines available for the update (5.2) on a packed matrix, the recursive approach would be attractive. It would also be attractive were we to store the full matrix so that no rearrangements are needed to use BLAS-3. Note that, in this case, we would still need to subdivide the updates into blocks to avoid a large number of redundant updates in the upper triangular part.

It is interesting that there are block column variants of the recursive approach. If $r_0$ in Algorithm 4 is set to $r_1 + nr$, the result is a right-looking algorithm with the updates in blocks of size about $nr$ (the exact size is affected by pivoting). If $r_0$ is set to $r - nr + 1$, the result is a left-looking algorithm, again with the updates in blocks of size about $nr$. We have not tried these variants since they will involve more rearrangements than the approach of Algorithm 3 when the matrix is packed.

6. FURTHER IMPLEMENTATION DETAILS WITHIN \texttt{HSL MA64}

Before we present our numerical experiments using \texttt{HSL MA64}, we discuss some further implementation details that are important for efficient performance.

6.1. Rearranging Only the First $p$ Columns

If we know that there will be only one block step (i.e., 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 to block column format only when it is updated and rearrange it back immediately afterwards. In this case, we therefore use the block column format only for the first $p$ columns.
6.2. Leading Rows and Columns with a Large Number of Small Entries

Because of the order in which we search for pivots (see Figure 1), it can happen that the number of columns that have been searched unsuccessfully for a pivot and are being kept up to date (the active columns) is large. These columns are updated using BLAS-2 after each pivot is chosen, which will be slow. In particular, this is certain to happen if the leading submatrix is zero. To avoid this possibility, we check (in the loop in the subroutine find pivot of Algorithm 1) for the number of active columns exceeding a limit \( na \). If this happens and there are at least \( na \) untouched columns (columns that have never been searched), we apply all the outstanding updates to these untouched columns, swap the first \( na \) rows and columns with the \( na \) rows and columns that correspond to the last \( na \) untouched columns, and restart with a single active column. Note that the swapping will be with columns \( n - na + 1 : n, n - 2 * na + 1 : n - na, n - 3 * na + 1 : n - 2 * na, \ldots \). This procedure will involve a very small overhead when all is well and avoid very slow execution in the cases that need it. A suitable value for \( na \) is \( nbi \).

6.3. 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 within HSL MA64. Therefore, when column \( m \) is searched, if its largest entry is found to have magnitude less than a user-set tolerance \( small \), the row and column are set to zero, the diagonal entry is accepted as a zero \( 1 \times 1 \) 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, HSL MA64 holds the inverse of \( D_{11} \) (see (1.5)), and sets 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.

6.4. Parallel Working

If all the data that a code needs are supplied through its arguments, it may be called at the same time on threads working on independent parts of the multifrontal tree. In common with all other packages in HSL 2011, HSL MA64 is written in this way.

This form of parallelism is not sufficient towards the end of the multifrontal process where \( n \) and \( p \) are large. Here, most of the work is done by \texttt{gemm} calls when updating the trailing submatrix by block columns, see (5.1). We therefore provide an option to use an OpenMP \texttt{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 all of these in a single loop.

If \( p \) is small, there may be insufficient work to justify working in parallel. HSL MA64 therefore has a user-set parameter, \( p\text{thresh} \) 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 6.1), there would be no opportunity for parallel working. We have found that 32 is a suitable value for \( p\text{thresh} \) on our test platform (see Section 1 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 \texttt{parallel do}
7. NUMERICAL EXPERIMENTS

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 III 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 IV shows that parallelizing by blocks gives a worthwhile gain over parallelizing by block columns. Table V 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 VI. Note that the speed on one thread approaches the \texttt{dgemm} peak of 9.3 Gflop/s for the larger values of \( p \).
Partial Factorization of a Dense Symmetric Indefinite Matrix

Table VI. Speed (Wall-Clock Gflop/s) and Speed-up with \( nb = 96 \) and \( nbi = 16 \)

<table>
<thead>
<tr>
<th>( p )</th>
<th>One thread</th>
<th>Four threads</th>
<th>Eight threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>5.5</td>
<td>1.0</td>
<td>10.1</td>
</tr>
<tr>
<td>128</td>
<td>6.5</td>
<td>1.0</td>
<td>14.5</td>
</tr>
<tr>
<td>256</td>
<td>7.2</td>
<td>1.0</td>
<td>18.4</td>
</tr>
<tr>
<td>512</td>
<td>7.6</td>
<td>1.0</td>
<td>21.0</td>
</tr>
<tr>
<td>1024</td>
<td>7.8</td>
<td>1.0</td>
<td>22.3</td>
</tr>
<tr>
<td>2048</td>
<td>7.8</td>
<td>1.0</td>
<td>22.4</td>
</tr>
</tbody>
</table>

Table VII. Wall-Clock Times and Speeds (Gflop/s) for the Factorization Phase of the Multifrontal Solver \( \text{HSL} \text{MA77} \) on One and Eight Threads with \( nb = 96 \) and \( nbi = 16 \). \( N \) Denotes the Order of the System Matrix and \( f_{\text{max}} \) Is the Maximum Front Size

<table>
<thead>
<tr>
<th>Problem</th>
<th>( N )</th>
<th>( f_{\text{max}} )</th>
<th>One thread</th>
<th>Eight threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>helm2d03</td>
<td>392257</td>
<td>1024</td>
<td>2.7</td>
<td></td>
</tr>
<tr>
<td>bratu3d</td>
<td>27792</td>
<td>1521</td>
<td>4.2</td>
<td></td>
</tr>
<tr>
<td>qa89k</td>
<td>66127</td>
<td>2075</td>
<td>4.4</td>
<td></td>
</tr>
<tr>
<td>halfb</td>
<td>224617</td>
<td>3240</td>
<td>16.3</td>
<td></td>
</tr>
<tr>
<td>5i5h12</td>
<td>19896</td>
<td>5551</td>
<td>43.7</td>
<td></td>
</tr>
<tr>
<td>Nice20Mc</td>
<td>715923</td>
<td>11688</td>
<td>945</td>
<td></td>
</tr>
<tr>
<td>Ga19As19h42</td>
<td>133123</td>
<td>18675</td>
<td>2018</td>
<td></td>
</tr>
<tr>
<td>SiO2</td>
<td>155331</td>
<td>21406</td>
<td>2760</td>
<td></td>
</tr>
</tbody>
</table>

Table VIII. Number of Entries in \( L \) (in Thousands) and Times (in Seconds) for the Factorization and Solve Phases of \( \text{HSL} \text{MA77} \) on a Single Thread without and with Static Pivoting. \( N \) Denotes the Order of the System Matrix

<table>
<thead>
<tr>
<th>Problem</th>
<th>( n_{\text{L}(L)} )</th>
<th>Factor</th>
<th>Solve</th>
</tr>
</thead>
<tbody>
<tr>
<td>cvxqp3</td>
<td>17500</td>
<td>4884</td>
<td>3131</td>
</tr>
<tr>
<td>ddoc</td>
<td>24993</td>
<td>6701</td>
<td>227</td>
</tr>
<tr>
<td>mario001</td>
<td>38434</td>
<td>746</td>
<td>665</td>
</tr>
<tr>
<td>nevaxp7</td>
<td>87500</td>
<td>39192</td>
<td>24707</td>
</tr>
</tbody>
</table>

7.2. Experiments with a Multifrontal Solver
The HSL multifrontal solver \( \text{HSL} \text{MA77} \) [Reid and Scott 2008, 2009b] employs \( \text{HSL} \text{MA64} \) for the partial factorization of the frontal matrices. In Table VII, we show some data for the execution of \( \text{HSL} \text{MA77} \) on problems from practical applications that have varying front sizes. We scale each problem using the HSL package \( \text{HSL} \text{MC64} \) [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 and Hu 2011] (Nice20MC is available from \text{www.gridtlse.org}). We use the pivot threshold \( u = 0.01 \), which is the default setting within \( \text{HSL} \text{MA77} \). It may be seen that a significant proportion of the total factorization time is spent within \( \text{HSL} \text{MA64} \) and the execution rate of \( \text{HSL} \text{MA64} \) is good on problems with large front sizes.

We have performed some static pivoting experiments with \( \text{HSL} \text{MA77} \). Results are given in Table VIII for some problems of augmented type, that is, the system matrix is of the form

\[
A = \begin{pmatrix} H & B^T \\ B & 0 \end{pmatrix}.
\]

where \( H \) and \( B \) are large and sparse. In our tests, we compare using \( \text{static} = 0 \) (no static pivoting, which is the default) and \( \text{static} = 10^{-6} \ast \text{norm} \), where \( \text{norm} \) is the norm...
Table IX. 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

<table>
<thead>
<tr>
<th>Problem</th>
<th>before</th>
<th>after</th>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>cvxqp3</td>
<td>1.3 × 10^{-10}</td>
<td>2.0 × 10^{-16}</td>
<td>(1)</td>
<td>1.7 × 10^{-9}</td>
</tr>
<tr>
<td>dtoc</td>
<td>1.1 × 10^{-14}</td>
<td>7.4 × 10^{-17}</td>
<td>(1)</td>
<td>6.7 × 10^{-13}</td>
</tr>
<tr>
<td>mario001</td>
<td>6.1 × 10^{-15}</td>
<td>6.1 × 10^{-15}</td>
<td>(0)</td>
<td>1.3 × 10^{-10}</td>
</tr>
<tr>
<td>ncvxp7</td>
<td>2.1 × 10^{-9}</td>
<td>1.9 × 10^{-16}</td>
<td>(1)</td>
<td>1.5 × 10^{-7}</td>
</tr>
</tbody>
</table>

Table X. Number of Entries in 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. $N$ Denotes the Order of the System Matrix

<table>
<thead>
<tr>
<th>Problem</th>
<th>N</th>
<th>nz(L) without</th>
<th>Factor without</th>
<th>Factor with</th>
<th>$u_{final}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>c-60</td>
<td>43640</td>
<td>1736</td>
<td>1650</td>
<td>0.34</td>
<td>0.32</td>
</tr>
<tr>
<td>c-65</td>
<td>48066</td>
<td>1549</td>
<td>1495</td>
<td>0.30</td>
<td>0.28</td>
</tr>
<tr>
<td>c-68</td>
<td>64810</td>
<td>2082</td>
<td>1962</td>
<td>4.04</td>
<td>3.85</td>
</tr>
<tr>
<td>c-73</td>
<td>169422</td>
<td>5276</td>
<td>5124</td>
<td>1.23</td>
<td>1.18</td>
</tr>
</tbody>
</table>

of the scaled system matrix. In Table IX we report the scaled residuals

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

where $x$ is the computed solution and $b$ 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 $\max_i |1 - x_i|$. Iterative refinement is performed until the scaled residual is less than $10^{-14}$. We see that, for the chosen examples, static pivoting can significantly reduce the number of delayed pivots, resulting in much sparser factors and faster factor and solve times. 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 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 et al. [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 $L$) 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

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

(7.2)

where $\delta$ is small, we found relaxed pivoting can be useful. In Table X 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.

### 7.3. Comparison with LAPACK and Preference for $2 \times 2$ Pivots

LAPACK 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. At the time of writing, Intel does not offer a multi-threaded version of \_sytrf.

In Table XI, we show the performance of \_dasytrf and HSL\_MA64 for six 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 $1 \times 1$ pivot when both a $1 \times 1$ and a $2 \times 2$ pivot are available. It may be seen that this chooses far fewer $2 \times 2$ pivots and is slower because more single-column updates are performed. It is interesting that the version that favors $2 \times 2$ pivots chooses about as many $2 \times 2$ pivots as \_dasytrf. On a single thread, HSL\_MA64 is slightly faster than \_dasytrf 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 \_dasytrf on these problems.

Finally, Table XII presents a comparison between favoring $2 \times 2$ over $1 \times 1$ pivots within HSL\_MA77. In this table, we report the number of $2 \times 2$ pivots used during the factorization, the number of flops 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, favoring $2 \times 2$ pivots leads to significantly more $2 \times 2$ pivots.

---

**Table XI. Comparison between HSL\_MA64 and the LAPACK Subroutine \_dasytrf**

<table>
<thead>
<tr>
<th>$n$</th>
<th>_dasytrf 1 thread</th>
<th>HSL_MA64 1 thread</th>
<th>HSL_MA64 8 threads</th>
<th>No. $2 \times 2$ pivots HSL_MA64</th>
<th>No. $1 \times 1$ pivots HSL_MA64</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>4.4 4.9 4.6 5.5 4.6</td>
<td>164 189 31</td>
<td>1 thread</td>
<td>2 thread</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>5.6 6.1 5.9 10.5 9.1</td>
<td>312 369 87</td>
<td>2 thread</td>
<td>4 thread</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>6.4 6.9 6.8 19.5 17.8</td>
<td>666 769 182</td>
<td>4 thread</td>
<td>8 thread</td>
<td></td>
</tr>
<tr>
<td>4000</td>
<td>7.0 7.6 7.5 29.9 28.4</td>
<td>1403 1473 366</td>
<td>8 thread</td>
<td>16 thread</td>
<td></td>
</tr>
<tr>
<td>8000</td>
<td>7.3 7.8 7.8 38.0 36.1</td>
<td>2902 2910 758</td>
<td>16 thread</td>
<td>32 thread</td>
<td></td>
</tr>
<tr>
<td>16000</td>
<td>7.2 7.9 7.8 43.2 39.9</td>
<td>6050 5830 1611</td>
<td>32 thread</td>
<td>64 thread</td>
<td></td>
</tr>
</tbody>
</table>

**Table XII. Comparison between Favoring $2 \times 2$ over $1 \times 1$ Pivots within HSL\_MA77 on Eight Threads**

<table>
<thead>
<tr>
<th>$n$</th>
<th>_dasytrf 1 thread</th>
<th>HSL_MA64 1 thread</th>
<th>HSL_MA64 8 threads</th>
<th>Time HSL_MA64</th>
<th>Time HSL_MA77</th>
</tr>
</thead>
<tbody>
<tr>
<td>ncvxp3</td>
<td>38176 17.7 54.5</td>
<td>23.5 35.9</td>
<td>_dasytrf vector block HSL_MA64 HSL_MA77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>kkt</td>
<td>791462 16.2 582</td>
<td>75.2 187</td>
<td>_dasytrf vector block HSL_MA64 HSL_MA77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>af_shell10</td>
<td>732433 4.6 410</td>
<td>28.5 102</td>
<td>_dasytrf vector block HSL_MA64 HSL_MA77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NICE20MC</td>
<td>349734 14.8 5284</td>
<td>160 470</td>
<td>_dasytrf vector block HSL_MA64 HSL_MA77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ncvxp3</td>
<td>5526 17.9 54.3</td>
<td>26.8 39.1</td>
<td>_dasytrf vector block HSL_MA64 HSL_MA77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>kkt</td>
<td>180291 20.5 575</td>
<td>78.4 191</td>
<td>_dasytrf vector block HSL_MA64 HSL_MA77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>af_shell10</td>
<td>0 5.8 410</td>
<td>29.3 108</td>
<td>_dasytrf vector block HSL_MA64 HSL_MA77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NICE20MC</td>
<td>0 19.4 5282</td>
<td>167 476</td>
<td>_dasytrf vector block HSL_MA64 HSL_MA77</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
being used and this results in a modest reduction in the \texttt{HSL MA64} time. The block count is similar for both pivoting strategies while there is a reduction in the (much smaller) vector count if $2 \times 2$ pivots are preferred.

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 while minimizing storage requirements by carefully combining blocking and the use of standard BLAS-3 subroutines with threshold partial pivoting and the use of both $1 \times 1$ and $2 \times 2$ pivots. We have also considered parallelization with OpenMP. For large dense systems, the execution speed of our code \texttt{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 et al. [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 $1 \times 1$ and $2 \times 2$ pivots.

All the HSL codes referred to in this article are part of HSL 2011. Use of HSL requires a license. Licenses are available without charge to individual academic users for their personal (noncommercial) research and for teaching; for other users, a fee is normally charged. Details of how to obtain a license together with information on all HSL packages are available at \url{www.cse.clrc.ac.uk/nag/hsl/}.

ACKNOWLEDGMENTS
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. We are grateful to Cleve Ashcraft for suggesting the recursive approach and for other helpful remarks.

REFERENCES

Partial Factorization of a Dense Symmetric Indefinite Matrix


Received November 2010; revised April 2011; accepted May 2011