Facilities Council

An efficient out-of-core multifrontal solver for large-scale unsymmetric element problems

J. K. Reid and J. A. Scott

December 2007

RAL-TR-2007-014

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

An efficient out-of-core multifrontal solver for large-scale unsymmetric element problems[1,2]

by

J. K. Reid and J. A. Scott

Abstract

In many applications where the efficient solution of large sparse linear systems of equations is required, a direct method is frequently the method of choice. Unfortunately, direct methods have a potentially severe limitation: as the problem size grows, the memory needed generally increases rapidly. However, the in-core memory requirements can be limited by storing the matrix and its factors externally, allowing the solver to be used for very large problems. We have designed a new out-of-core package for the large sparse unsymmetric systems that arise from finite-element problems. The code, which is called `HSL_MA78`, implements a multifrontal algorithm and achieves efficiency through the use of specially designed code for handling the input/output operations and efficient dense linear algebra kernels. These kernels, which are available as a separate package called `HSL_MA74`, use high level BLAS to perform the partial factorization of the frontal matrices and offer both threshold partial and rook pivoting. In this paper, we describe the design of `HSL_MA78`, explain its user interface and the options it offers. We also describe the algorithms used by `HSL_MA74` and illustrate the performance of our new codes using problems from a range of practical applications.

Keywords: large sparse unsymmetric linear systems, element problems, out-of-core solver, multifrontal, rook pivoting, partial pivoting, Fortran 95.

---

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

[2] The work of this author was supported by the EPSRC grants GR/S42170 and EP/E053351/1.

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

December 17, 2007.

Contents

1 Introduction

Many areas of computational science and engineering rely on finite-element analysis of large and complex structures. In general, most of the computation time for such analyses is spent on solving the systems of linear equations associated with the finite-element model. These systems take the form

$$AX = B, \tag{1.1}$$

where the system matrix A is of order $n \times n$, B is an $n \times nrhs$ ($nrhs \geq 1$) matrix of right-hand sides and X is the $n \times nrhs$ solution matrix. The matrix A can be written as the sum

$$A = \sum_{k=1}^{nelt} A^{(k)}, \tag{1.2}$$

where $nelt$ is the number of elements in the model and $A^{(k)}$ corresponds to the contribution from element k and has nonzeros in only a small number of rows and columns. In practice, each $A^{(k)}$ is held as a small dense matrix, called an element matrix, of order equal to the number of nodes in element k times the number of degrees of freedom per node. A list of the global indices of the variables associated with element k, which identifies where the entries in $A^{(k)}$ belong in A, must also be held. Each $A^{(k)}$ is symmetrically structured (the list of indices is both a list of column indices and a list of row indices) but, in the general case, is numerically unsymmetric.

To minimise the overall computational cost of the finite-element analysis, it is essential that an efficient and robust solver is used for (1.1). Over the last forty years or so, significant effort has been expended on the development of suitable solvers. In broad terms, these solvers can be divided into two main classes: iterative solvers and direct solvers (in recent years, hybrid methods that combine iterative and direct techniques have begun to emerge). The main advantages of iterative solvers are that they require little memory (typically, a small number of vectors of size the order of the system) and are straightforward to implement. Their main weakness is that, in general, without a good preconditioner they are slow to converge or may fail to converge altogether. By contrast, direct solvers are much harder to write and, even when well implemented, the memory they require increases rapidly with problem size. However, direct solvers are generally much more robust for a wide range of problems and they are often preferred for systems with multiple right-hand sides. They can also be used (sometimes in modified form) to provide preconditioners for iterative methods.

One way of extending the size of problem that can be solved using a direct method is to work out of core, that is, to hold the system matrix A and its factors (and possibly some of the work arrays used by the solver) in files. We recently developed an out-of-core frontal solver for unsymmetric element problems called `HSL_MA42_ELEMENT` [18]. This package is part of the HSL mathematical software library [8]. In general, for large-scale problems, multifrontal methods are more efficient than frontal methods. Thus, in this paper, we extend our work on multifrontal solvers and report on the design and development of an out-of-core multifrontal solver for unsymmetric element problems. The new solver is included in HSL as package `HSL_MA78`. This paper is organised as follows. Section 2 provides a brief overview of the multifrontal method for element problems. In Section 3, we discuss the design of `HSL_MA78` and then, in Section 4, we describe the user interface. Two important aspects of the new package are discussed in the next two sections, namely, the dense linear algebra kernels that are at the heart of `HSL_MA78` and are implemented in a separate HSL package called `HSL_MA74`, and the efficient handling of input/output operations, which is essential for the success of any out-of-core solver. In Section 7, we present numerical experiments and compare the performance of `HSL_MA78` with that of `HSL_MA42_ELEMENT` for a range of practical problems.

We note that HSL is a Fortran library and the names `HSL_MA74` and `HSL_MA78` follow the HSL naming convention that routines written in Fortran 95 have the prefix `HSL_` (which distinguishes them from the Fortran 77 codes).

2 The out-of-core multifrontal method

The first out-of-core solvers were based on the original work on the frontal algorithm by Irons [9]. The frontal method is a variant of Gaussian elimination and involves the matrix factorization

$$A = PLDUQ,$$

where P and Q are permutation matrices, D is a diagonal matrix, and L and U are unit lower and upper triangular matrices, respectively. The solution process is completed by performing the forward substitution

$$PLY = B, \tag{2.1}$$

then the diagonal solve

$$DZ = Y, \tag{2.2}$$

followed by the back substitution

$$UQX = Z. \tag{2.3}$$

Clearly, the diagonal solve and one of the substitution phases may be combined and very often D and U are stored together as the upper triangular matrix $\tilde{U} = DU$; (2.2) and (2.3) are then replaced by $\tilde{U}QX = Y$. The frontal method aims to limit in-core memory requirements by assembling the contributions from each element one at a time and, by interleaving assembly and elimination operations, avoids storing the whole coefficient matrix A. This allows the computation to be performed using a small (dense) *frontal matrix* that at each stage may be expressed in the form

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

where the p rows and columns of F_1 are *fully summed*, that is, all the entries in these rows and columns of the overall matrix have already been assembled, while the rows and columns of F_4 are not yet fully summed. Provided p pivots can be chosen stably from F_1, the *partial factorization* of F takes the form

$$F = \begin{pmatrix} P_1 & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} L_1 & 0 \\ L_2 & I \end{pmatrix} \begin{pmatrix} D_1 & 0 \\ 0 & F_S \end{pmatrix} \begin{pmatrix} U_1 & U_2 \\ 0 & I \end{pmatrix} \begin{pmatrix} Q_1 & 0 \\ 0 & I \end{pmatrix}, \tag{2.4}$$

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

$$F_S = F_4 - L_2 D_1 U_2.$$

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

In the frontal method, there is a single front at each stage of the computation (that is, there is a single set of variables that have not yet been eliminated but are involved in one or more of the elements that have been assembled) and, provided the elements are assembled in a suitable order, this front gradually moves across the whole finite-element domain as elements are assembled and fully summed variables are eliminated. Duff and Reid [7] extended the frontal concept to use more than one front and, reflecting the use of several fronts, their generalisation is called the *multifrontal* method. If we assume that a pivot

order (that is, a tentative order in which the eliminations are to be performed) has been chosen then, for each pivot in turn, the multifrontal method first assembles all the elements that contain the pivot into the frontal matrix and performs a partial factorization. The computed entries of the factors are stored and the Schur complement matrix F_S is treated as a new element, called a *generated element* (the term *contribution block* is also used in the literature). The generated element is added to the set of unassembled elements and the next uneliminated pivot then considered. The basic algorithm is summarised in Figure 2.1.

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

Figure 2.1: Basic multifrontal factorization

The assemblies can be recorded as a tree, called an *assembly* tree. Each leaf node represents an original element and each non-leaf node represents a set of eliminations and the corresponding generated element. The children of a non-leaf node represent the elements and generated elements that contain the pivot. If A is structurally irreducible there will be a single *root* node, that is, a node with no parent. Otherwise, there will be one root for each independent subtree.

The partial factorization of the frontal matrix at a node v in the tree can be performed once the partial factorizations at all the nodes belonging to the subtree rooted at v are complete. If the nodes of the tree are ordered using a depth-first search, the generated elements required at each stage are the most recently generated ones of those so far unused. This makes it convenient to use a stack for temporary storage during the factorization. This, of course, alters the pivot sequence, but the arithmetic is identical apart from the round-off effects of reordering the assemblies and the knock-on effects of this.

In general, numerical stability considerations mean that $q \leq p$ pivots can be chosen at non-root nodes and the matrices L_i, U_i and D_i $(i = 1, 2)$ in (2.4) are then of order q, while the permutation matrices P_1 and Q_1 remain of order p. In this case, $p - q$ pivots must be *delayed* and the generated element will be larger than anticipated on the basis of the sparsity pattern alone. It may be expressed as

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

where the order of the leading submatrix F_{G1} is $p-q$ and F_{S1} has the same order as the matrix F_S in the partial factorization (2.4).

3 Design of HSL_MA78

In this section, we briefly discuss the overall design of HSL_MA78 and highlight some of the key features of the package.

3.1 Overview of the structure of HSL_MA78

One of the important initial design decisions for HSL_MA78 was to use a reverse communication interface, with control being returned to the calling program for each element. This is explained further in Section 4. Reverse communication keeps the memory requirements for the initial matrix to a minimum and gives the user maximum freedom as to how the original matrix data is held.

The effectiveness of the multifrontal method is dependent upon having a suitable pivot order. For the first release of HSL_MA78, we require the user to supply a suitable pivot order for his or her problem. The main reasons for making this design decision were that research in this area is still active and no single algorithm produces the best pivot sequence for all problems. By not incorporating ordering into the package, the user can use whatever approach works well for his or her problem. A number of stand-alone ordering packages already exist. For example, the code METIS_NodeND [11], [12]. can be used to compute a nested dissection ordering while the HSL package HSL_MC68 offers efficient implementations of the minimum degree algorithm [19] and the approximate minimum degree algorithm [1], [2]. As far as we are aware, no satisfactory ordering code that holds the matrix data out of core is currently available; instead, the sparsity pattern plus some additional integer arrays of size related to the order and density of A must be held in main memory. Users of HSL may use MC57 to assemble the sparsity pattern of A, which can be discarded as soon as the pivot order has been chosen.

Given the pivot sequence, the multifrontal method can be split into a number of phases:

- An analyse phase that uses the index lists for the elements and the pivot sequence to construct the assembly tree. It also calculates lower bounds on the work and storage required for the subsequent numerical factorization (the bounds become equalities if numerical considerations do not cause any pivots to be delayed).
- A factorize phase that uses the assembly tree to factorize the matrix (incorporating numerical pivoting as necessary) and (optionally) solves $AX = B$.
- A solve phase that performs forward substitution followed by back substitution using the stored matrix factors.

The HSL_MA78 package has separate routines for each of these phases; this is discussed further in Section 4.

3.2 Language

HSL is a Fortran library and many of the older solvers in the library are written in Fortran 77. However, with high quality Fortran 95 compilers becoming more widely available (including the freely available g95 compiler at g95.sourceforge.net), in recent years a number of Fortran 95 solvers have been included within HSL. For HSL_MA78, Fortran 95 offers us a number of advantages, including allocatable arrays, enabling a more friendly user-interface, derived types (see Section 4.2), and recursion. In a serial implementation of a multifrontal algorithm, recursion is a convenient and efficient way to visit the nodes of the assembly tree. When called for a node of the tree, the factorize subroutine calls itself for each of the node's children, assembles their elements (original or generated), and performs the partial factorization of the resulting frontal matrix. A direct call for the root node performs a complete factorization. Similarly, subroutines that recursively perform forward substitution and back substitution are called during the solve phase.

To allow the package to solve very large problems, we selectively make use of *long* (64-bit) integers, declared in Fortran 95 with the syntax `selected_int_kind(18)` and supported by all the Fortran 95 compilers to which we have access. These long integers are used for addresses within files and for operation counts. We assume that the order of A is less than 2^{31}, so that long integers are not needed for the variable indices.

We have adhered to the Fortran 95 standard except that we use allocatable structure components and dummy arguments. These are part of the official extension that is defined by Technical Report TR 15581(E) [10] and is included in Fortran 2003. It allows arrays to be of dynamic size without the computing overheads and memory-leakage dangers of pointers.

Addressing is less efficient in code that implements pointer arrays since it has to allow for the possibility that the array is associated with a array section, such as `a(i,:)`, that is not a contiguous part of its parent. Furthermore, optimization of a loop that involves a pointer may be inhibited by the possibility that its target is also accessed in another way in the loop.

3.3 Data structures and files

The multifrontal method needs data structures for the original matrix A, the frontal matrix, the stack of generated elements, and the matrix factor. If the stack and frontal matrix are held in main memory and the factors are written to disk as they are generated, the method performs the minimum possible input/output for an out-of-core method: it writes the factor data to disk once and reads DU once during back substitution or L and then DU when solving for further right-hand sides. However, for very large problems, it may be necessary to hold further data on disk. The first release of `HSL_MA78` holds the frontal matrix in main memory and allows the multifrontal stack and the original matrix data on be held disk but, by using a system for virtual memory management (which we explain further in Section 6), avoids much of the actual input/output.

The virtual memory management system used by `HSL_MA78` includes the facility of grouping a set of files into a *superfile* that is treated as an entity. `HSL_MA78` uses four superfiles: one holds integer information, one holds real information, one provides real workspace, and the last one holds real data associated with delayed pivots. We refer to these as the *main integer*, *main real*, *main work* and *delay* superfiles, respectively.

The main real superfile holds the reals of the original element matrices $A^{(k)}$ followed by the columns of the factor PL and the rows of the factor DUQ, which are in the order that they were calculated.

When supplying the list of indices associated with an element matrix $A^{(k)}$, the user may include duplicated and/or out-of-range entries. We check the user-supplied lists and flag this case, then we store in the main integer superfile the list of indices left after the duplicates and/or out-of-range entries have been squeezed out, the number of entries in the original user-supplied index list, and a mapping from the original list into the compressed list.

During the analyse phase, for each non-leaf node of the tree, we store the list of original indices of the variables in the front. At the end of the analyse phase, these lists are rewritten in the elimination order that this phase has chosen. This facilitates the merging of generated elements during the factorization. During the numerical factorization, after each partial factorization of a frontal matrix, the permuted row and column indices are stored in the main integer superfile. The lists of these indices are stored after the analyse data; we do not overwrite the analyse data so that the user can factorize more than one matrix with the same sparsity pattern but different numerical values without recalling the analyse phase. By default, the factorization uses unsymmetric pivoting to maintain numerical stability (we discuss the pivoting options in detail in Section 5.2) and thus, at non-leaf nodes, it is necessary to hold both a list of the row indices and and a list of the column indices. Each list of row (respectively, column) indices starts with a list of the row (respectively, column) indices of the chosen pivots, and is followed by a list of the row (respectively, column) indices of the delayed pivots (pivots that could not be chosen for stability reasons).

The principal role of the main work superfile is to hold the stack of intermediate results that are generated during the depth-first search. When the partial factorization of a frontal matrix is processed, a generated element is stacked, which increases the stack size; when this is later merged into the frontal matrix at the parent node, it is taken from the top of the stack and the stack size decreases. A long integer is used to store the position of the top of the stack.

During the factorization phase, after the partial factorization of a frontal matrix, the delay superfile is used to hold (in a stack) the remaining rows and columns that are fully summed but, for numerical reasons, could not be pivoted on. Refering back to (2.5), F_{S1} is stacked in the main workspace superfile while F_{G1}, F_{G2}, and F_{G3} are stacked in the delay superfile. Once all the contributions F_{S1} at the parent node have been assembled into its frontal matrix F_p, the contributions corresponding to the delayed pivots for each of its child nodes are assembled into the leading rows and columns of F_p. Thus, at each non-leaf node v, the frontal matrix takes the form

$$F_p = \begin{pmatrix} F_{p1} & F_{p2} \\ F_{p3} & F \end{pmatrix}, \tag{3.1}$$

where the order of F is the *anticipated* order of the frontal matrix (that is, the order if no pivots were delayed) and F_{p1}, F_{p2}, F_{p3} correspond to the delayed pivots passed to v from its children. We note that it is not necessary to hold integer information on the delayed rows and columns in a stack since this data can be retrieved from the row and column lists stored in the main integer superfile.

4 User interface

In this section, we give brief details of the routines within the `HSL_MA78` package that may be called by the user and of the derived types used by the package. Full details are provided in the user documentation.

4.1 User-callable routines

`HSL_MA78` uses a reverse communication interface. The main routines that comprise this interface are as follows:

`MA78_open`: must be called once for a problem to initialize the data structures and open the superfiles, which were described in Section 3.3.

`MA78_input_vars`: must be called once for each element to specify the variables associated with it. The index lists are written to the main integer superfile.

`MA78_analyse`: must be called after all calls to `MA78_input_vars` are complete. As already noted, the pivot order must be supplied by the user. `MA78_analyse` uses the sparsity pattern of the matrix to construct the assembly tree. The index lists for each node of the tree are written to the main integer superfile. The analyse phase is as for the symmetric case; full details, including the use of supervariables, node amalgamation, and the ordering of the child nodes, are given by Reid and Scott in [17].

`MA78_input_reals`: must be called for each element to specify the entries. The index list must have already been specified by a call of `MA78_input_vars`. Each element matrix must be input by columns. For large problems, the data may be provided in more than one adjacent call. The data is written to the main real superfile. If data is entered for an element that has already been entered, the original data is overwritten.

`MA78_factor`: may be called after all the calls to `MA78_input_reals` are complete and after the call to `MA78_analyse`. The matrix A is factorized using the assembly tree constructed by `MA78_analyse` and the factor entries are written to the main real superfile as they are generated. It may be called afresh after one or more calls of `MA78_input_reals` have specified changed real values.

`MA78_factor_solve`: may be called in place of `MA78_factor` if the user wishes to solve the system $AX = B$ at the same time as the matrix A is factorized.

`MA78_solve`: uses the computed factors generated by `MA78_factor` for solving the system $AX = B$ or the transpose system $A^TX = B$. A partial solution may be computed by setting the parameter `job` to have one of the following values:

`1` for solving $PLX = B$ (or $Q^TU^TX = B$)

`2` for solving $DUQX = B$ (or $DL^TP^TX = B$)

In addition to the above routines, `MA78_resid` may be called after a call to `MA78_factor_solve` or `MA78_solve` to compute the residual $R = B - AX$ (or $R = B - A^TX$). This involves reading the original user-supplied matrix data from the main integer and real superfiles. There is an option to compute an upper bound on the infinity norm of the system matrix (it is an upper bound because the absolute values

of the element entries are taken before they are summed). Denoting this bound by $\|A\|_{b,\infty}$, the user can compute the scaled residuals

$$\frac{\|res_j\|_\infty}{\|A\|_{b,\infty}\|x_j\|_\infty + \|b_j\|_\infty} \tag{4.1}$$

where b_j is the jth right-hand side and x_j and res_j are the corresponding solution and residual vector, respectively. If the user decides that the computed residual is unacceptably large, iterative refinement can be performed by recalling `MA78_solve` with the right-hand side set to R.

Once all other calls are complete for a given problem, `MA78_finalise` should be called. This routine deallocates the components of the derived data types used by `HSL_MA78` and closes the superfiles associated with the problem. `MA78_finalise` includes an option to keep the superfiles and store all the in-core data for the problem (in another file) so that the computation can be restarted later. This is done by calling `MA78_restart`. The main use of `MA78_restart` is to solve for further right-hand sides using a previously computed factorization, but it also allows the reuse of the analysis data to factorize a matrix of the same structure but different real values.

4.2 Derived types

`HSL_MA78` uses derived types to pass data between the different routines within the package. The following derived types are available to the user:

`MA78_keep` has private components. The user must declare an object of this type for each problem and pass it on each subroutine call. An important use of `MA78_keep` is for holding copies of some of the user-defined parameters so that checks can be made. Checking the user's data is particularly important, we believe, in a package such as this with a reverse communication interface because reverse communication increases the opportunities for errors to be made.

`MA78_control` has components that control the action within the package. They are given default values when a variable of this type is declared. The controls include parameters that determine the level of diagnostic printing, the virtual memory management (Section 6), the block size for full-matrix operations on the frontal matrix (Section 5) and the choice of numerical pivoting (Section 5.2). The defaults have been chosen on the basis of our numerical experiments and are likely to be appropriate for most users. However, for maximum flexibility, these parameters may be reset by the user.

`MA78_info` has components that return information from each subroutine call. At the end of the analyse phase, information is returned on the expected maximum front size, the number of entries in the factors, and the number of floating-point operations, based on the assumption that no pivots are delayed for numerical reasons. At the end of the factorization phase, the actual statistics are returned, as well as the computed determinant of A (its sign and the logarithm of its absolute value). Information is also available on the number of integers and reals that have been written to files and on the maximum stack size, as well as information on the pivots chosen during the factorization. We have attempted to make the information available to the user at the end of the computation as comprehensive as possible so that he or she can assess how well the code has performed on his or her problem and, if necessary, can use the information to reset one or more of the control parameters for a subsequent problem.

5 Dense linear algebra kernels

The efficiency of the factorization phase is dependent upon the partial factorization of the frontal matrices. Since the frontal matrices are held as full matrices, dense linear algebra kernels may be used. In particular, we can make use of high level BLAS [4] when performing the factorizations and the forward and back substitutions. We have chosen to write a separate package, `HSL_MA74`, that is called by `MA78_factor` to perform the partial factorizations of the frontal matrices and by `MA78_solve` to perform the partial forward

and back substitutions. In this section, we describe HSL_MA74 and discuss the pivoting options that it offers and that may be used by users of the multifrontal solver HSL_MA78.

5.1 Overview of HSL_MA74

Given a dense unsymmetric $m \times m$ matrix F, HSL_MA74 performs a partial factorization, limiting eliminations to the leading $p \leq m$ rows and columns. Stability considerations may lead to $q \leq p$ eliminations being performed (that is, fewer than p pivots are chosen). The factorization takes the form (2.4) where the matrices L_1, U_1 and D_1 are of order q and the permutation matrices P_1 and Q_1 are of order p. Subroutines are provided for partial solutions, that is, solving systems of the form

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

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

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

The user inputs the matrix F in a rank-2 array which, on exit, is overwritten by the factorized matrix. Each diagonal entry holds either the inverse of a pivot or, if a zero pivot is chosen, the corresponding diagonal entry is set to zero (see Section 5.3). A rank-1 array pperm of length p is used to hold the row permutations P_1 so that, on exit, pperm(i) holds the index of the row of F that is permuted to row i, $i = 1, ..., p$. Similarly, a rank-1 array qperm is used to hold the column permutations Q_1.

HSL_MA74 uses a block algorithm. If the factorization was to proceed by choosing a single pivot at a time, the updates to the rest of F could only be performed using Level 2 BLAS. To take advantage of the more efficient Level 3 BLAS, the partial factorization is programmed as a sequence of block steps. The block size nb is a parameter under the user's control (we will discuss the choice of this parameter further in Section 7.1). If q is the number of pivots chosen so far, the code searches columns $q+1$ to p of F in turn for a pivot. If the column to be searched has had $k < q$ updates, it is first updated with the $q-k$ most recently chosen pivots. Since a single column is being updated, this is performed using the Level 2 BLAS kernels _trsv and _gemv. Each time a pivot is chosen, q is incremented by one and the pivotal column is swapped with column q. The position m_1 of the right-most column of F that has been searched for a pivot is held and, whenever a pivot is chosen, columns $q+1$ to m_1 are updated (using the Level 2 rank-1 update routine _ger) so that all the columns that have been tested and rejected are fully updated. In this way, we avoid holding an array of updates. Once nb pivots have been chosen or $q = p$, columns $m_1 + 1$ to n are updated using the Level 3 BLAS kernels _trsm and _gemm. If $q < p$, m_1 is reset to $q+1$ and the column search restarts from column $q+1$. We search the remaining columns cyclically to avoid repeatedly searching a previously rejected column.

In MA78_factor, the rows and columns corresponding to delayed pivots are assembled into the leading rows and columns of the frontal matrix (see (2.5)). Since these columns have already been rejected as potential pivot columns, we do not want to search them again until the remaining fully summed columns have been searched and pivotal operations performed. To facilitate this, MA74_factor has an optional parameter s which, if present, specifies the column of F from which the search is to start. If s is present and less than p, on entry to MA74_factor each column i is swapped with column $p-i+1$ ($1 \leq i \leq min(\mathtt{s}, p-\mathtt{s})$) before the computation begins.

5.2 Pivoting options

During the partial factorization, a potential pivot is selected only if it satisfies a numerical stability test. HSL_MA74 offers a number of options that are controlled using the parameters pivoting, small, static,

and u. The default strategy is *threshold partial pivoting* (pivoting = 1). In this case, an entry f_{ij} of the reduced matrix is normally only chosen as a pivot if it satisfies

$$|f_{ij}| \geq \max(\mathtt{u} * \max_{l>q} |f_{lj}|, \mathtt{small}). \tag{5.1}$$

Here u is the pivoting *threshold* parameter. Values of u close to zero will generally result in a faster factorization with fewer entries in the factors but values close to 1 are more likely to result in a stable factorization; the default of 0.01 is a compromise between stability and sparsity and is recommended in the user documentation for other direct solvers (for example, HSL_MA42_ELEMENT). Values that are less than 0.0 are treated as 0.0 and values greater than 1.0 are treated as 1.0. small controls the size of the smallest pivot that is acceptable. The default value is tiny(small), where tiny() is the Fortran numeric inquiry function that returns the smallest positive number that is stored in full precision. In HSL_MA74, the row index r corresponding to the largest entry in rows $q+1$ to m of the reduced matrix is found using the BLAS kernel i_amax. If $r \leq p$, the pivot has been found, q is incremented by 1 and rows q and r are swapped. If $r > p$, the largest entry in rows q to p of column i is found (again using i_amax) and, if this satisfies (5.1), it is chosen as the next pivot.

In some applications, using a value of u equal to 0.1 or 0.01 can lead to a large number of delayed (rejected) pivots. In this case, the size of the frontal matrices as the factorization moves up the tree can grow significantly beyond that which was anticipated by the analyse phase. This results in a more expensive factorization, both in terms of the number of flops required to perform the factorization and the number of entries in the matrix factors; this, in turn, leads to a more expensive solve phase. Furthermore, more memory will be required for the frontal matrix (which is held in main memory). In recent years, this has led to a number of direct solvers offering options for *static* pivoting (see, for example, [6], [13]). Here, the essential idea is not to allow pivots to be delayed but, instead, if necessary to choose pivots that do not satisfy condition (5.1) so that the pivot selection closely follows that provided by the user to the analyse phase. The danger is that there will be a potential loss of accuracy in the factorization and it may be necessary to perform refinement steps after the solve phase to try to recover the required accuracy. How this should be done is still a subject of research (see, for example, [3]).

Within HSL_MA74, static pivoting is controlled by the parameter static. If static is positive and fewer than p pivots can be chosen that satisfy (5.1), the pivot that came closest to satisfying this condition is chosen, that is, the pivot for which the ratio

$$\max_{q<i\leq p} |f_{ij}| \;/\; \max_{q<l\leq m} |f_{lj}|, \quad q \leq j \leq p, \tag{5.2}$$

is the largest. If its absolute value is greater than static, the information parameter usmall (which is initialised to u) is set to the minimum of usmall and (5.2). Otherwise, the pivot is given the value that has the same sign but absolute value static and usmall is set to zero. On exit, usmall holds the threshold parameter that was used or is zero if any pivots are replaced by static, num_thresh holds the number of pivots that did not satisfy the threshold criteria based on the user-supplied value of u, and num_perturbed holds the number of pivots that were replaced by static.

For *threshold diagonal pivoting* (pivoting = 2), pivots may only be chosen from the diagonal and should satisfy the threshold criteria (5.1) (with $i = j$). Threshold diagonal pivoting may be combined with static pivoting by setting static to be positive. If $p = m$ (which occurs at a root of the tree) and fewer than p pivots can be chosen from the diagonal, the code issues a warning and switches to choosing off-diagonal pivots. The number of pivots that are chosen from the diagonal is returned to the user.

The partial pivoting threshold test (5.1) controls the size of the entries in L. In general, this works well but examples can be found for which the computed factors are not sufficiently accurate. Thus, in the 1990s, the more stringent (but more costly) *rook pivoting* strategy was introduced (see, for example, [14], [15]). Rook pivoting controls the size of the entries in both L and U by checking a potential pivot candidate against the entries in both its row and column. Threshold rook pivoting may be selected in HSL_MA74 by setting pivoting = 3. A pivot candidate f_{ij} is chosen as a pivot if it satisfies (5.1) and,

additionally,

$$|f_{ij}| \geq \mathtt{u} * \max_{l>q} |f_{il}|. \tag{5.3}$$

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

5.3 Singular matrices

`HSL_MA74` is designed to factorize singular matrices and to (partially) solve consistent singular systems. If all the entries in the column currently being searched for a pivot are less than `small`, all the entries in the column are set to zero and the column is swapped with column p_1, where $p_1 \leq p$ is the largest index of a candidate column that is not all zeros. To prevent a zero column from being searched again, a flag is set. Once all possible pivots have been chosen that satisfy the threshold test (5.1), a check is made to see if any of the rows $p_1 + 1$ to p are zero rows (or, at least, have all entries less than `small`). For each such row of zeros, a zero pivot is chosen (the corresponding entry of D is set to zero). The number of zero pivots is returned to the user.

6 Virtual memory management

A key part of the design of `HSL_MA78` is that all input and output to disk is performed through a set of Fortran subroutines that manage a virtual memory system so that actual input/output occurs only when really necessary. This set of subroutines is available within HSL as the Fortran 95 package `HSL_OF01` [16].

Fortran 95 offers two forms of file access: sequential and direct. Sequential access is not suitable for us because the original matrix data must be accessed non-sequentially and other data has to be accessed backwards as well as forwards and our experience has been that backwards access is slow; thus we use direct-access files. A disadvantage of direct-access files is that they use fixed-length records but we need to be able to read and write different amounts of data at each stage of the multifrontal computation. To get around this, the data is buffered and, as we explain below, this is done for us by `HSL_OF01`. We note that Fortran 2003 offers a third form of file access: stream. At the time of writing, no compilers are available that fully support Fortran 2003, although the Nag compiler does support stream access. Our experience with this compiler has been that using stream access within the multifrontal solver is currently less efficient than using direct-access files within `HSL_OF01` (see [16]) and so do not use stream access in the current release of our solvers.

6.1 The virtual memory package `HSL_OF01`

`HSL_OF01` provides facilities for reading from and writing to direct-access files. There is a version for reading and writing real data and a separate version for integer data, both of which are used by `HSL_MA78`. Each version has its own buffer, which is used to avoid actual input/output operations whenever possible. One buffer may be associated with more than one direct-access file. We take advantage of this within `HSL_MA78` to enable the available memory to be dynamically shared between the main real superfile, the main work superfile and the delay superfile, according to their needs at each stage of the computation. It would be desirable to have a single buffer (and a single version of the package) for both the real and the integer data, but this is not possible in standard Fortran 95 without some copying overheads.

Each `HSL_OF01` buffer is divided into *pages* that are all of the same size, which is also the size of each file record. All actual input/output is performed by transfers of whole pages between the buffer and records of

the file. The size and number of pages are parameters that may be set by the user. Numerical experiments that we report in [16] were used to choose default settings of for these parameters within HSL_MA78.

The data in a file is addressed as a virtual array of rank one. Because it may be very large, long integers are used to address it. The most active pages of the virtual array are held in the buffer. Any contiguous section of the virtual array may be read or written, without regard to page boundaries. HSL_OF01 does this by first looking for parts of the section that are in the buffer and performing a direct transfer for these. For any remaining parts, there may have to be actual input and/or output of pages of the buffer. If room for a new page is needed in the buffer, by default the page that was least recently accessed is written to its file (if necessary) and is overwritten by the new page. Note that, because HSL_OF01 reads and writes contiguous sections of the virtual array, when HSL_MA78 needs to write the computed rows of the *DU* factor to file, it is necessary first to copy each row to a temporary column vector and then write the column vector to file.

A file is often limited in size to less than 2^{32} bytes, so the virtual array may be too large to be accommodated on a single file. In this case, secondary files are used; a primary file and its secondaries are refered to as a *superfile*. The files of a superfile may reside of different devices.

There are situations where is it known that data accessed in the virtual array is unlikely to be needed again soon and do not deserve to be given priority in the buffer. During the factorization phase of HSL_MA78, once the rows and columns of the factors have been written to the main superfile it is known that most of them will not be needed for some time and so it is more efficient to use the buffer for the stack. We therefore make use of the HSL_OF01 option for 'inactive' access, which has the effect that the relevant pages do not stay long in the buffer unless they contain other data that makes them do so. HSL_OF01 also has an option to specify that data read need not be retained thereafter. If no part of a page in the buffer is required to be retained, the page may be overwritten without writing its data to an actual file. This is used when reading data from the multifrontal stack and from the delay superfile since it is known that it will not be needed again. Further details of these options are included in [16].

HSL_OF01 also offers an option to add a section of the virtual array into an array under the control of a map. If the optional array argument `map` is present and the section starts at position `loc` in the virtual array, `OF01_read` behaves as if the virtual array were the array `virtual_array` and the statement

```
read_array(map(1:k)) = read_array(map(1:k)) + virtual_array(loc:loc+k-1)
```

were executed. Without this, a temporary array would be needed, the call would behave as if the following statement were executed:

```
temp_array(1:k) = virtual_array(loc:loc+k-1)
```

and the calling code would need to execute the statement

```
read_array(map(1:k)) = read_array(map(1:k)) + temp_array(1:k)
```

We use this option in the factorization phase of HSL_MA78 to efficiently assemble elements into the frontal matrix.

6.2 Option for in-core working within HSL_MA78

If its buffer is big enough, HSL_OF01 will avoid any actual input/output, but there remain the overheads associated with copying data to and from the buffer. For HSL_MA78, this is particularly serious during the solve phase for a single right-hand side since each datum read during the forward or back substitution is used only once. We have therefore included within HSL_MA78 an option that allows the superfiles to be replaced by arrays. The user can specify the initial sizes of these arrays and an overall limit on their total size. If an array is found to be too small, the code attempts to reallocate it with a larger size. If this breaches the overall limit or if the allocation fails because of insufficient available memory on the computer being used, the code automatically switches to out-of-core working by writing the contents of

the array to a superfile and then freeing the memory that had been used by the array. This may result in a combination of superfiles and arrays being used. To ensure the automatic switch can be made, we always require path and superfile names to be provided on the call of `MA78_open`. If a user specifies the total size of the arrays without specifying the initial sizes of the individual arrays, the code automatically chooses suitable sizes.

In some applications, a user may need to factorize a series of matrices of the same size and the same (or similar) sparsity pattern. We envisage that the user may choose to run the first problem using the out-of-core facilities and may then want to use the output from that problem to determine whether it would be possible to solve the remaining problems in-core (that is, using arrays in place of superfiles). On successful completion of the factorization, `HSL_MA78` returns the number of integers and reals stored for the matrix and its factor, and the maximum size of the multifrontal stack, together with the maximum size of the stack used for holding delayed pivots. This information can be used to set the array sizes for subsequent runs. Note, however, that additional in-core memory is required during the computation for the frontal matrix and other local arrays. If the allocation of the frontal matrix fails at the start of the factorization phase, the arrays being used in place of superfiles are discarded one-by-one and a switch to superfiles is made in the hope of achieving a successful allocation.

7 Numerical experiments

In this section, we report on using `HSL_MA78` to solve a number of problems from practical applications. The test problems are listed in Table 7.1 in order of the predicted number of entries in the factors (that is, the number of entries if no pivots are delayed) when the analysis phase of the HSL solver `MA57` [5] is used to compute the pivot order. The problems range in size from fewer than 1000 elements to more than 70,000 elements with almost 225,000 degrees of freedom. If only the sparsity pattern is available, numerical values for the matrix entries are generated using the HSL pseudo-random number generator `FA14`. The right-hand side for each problem is selected so that the required solution is the vector of ones. The numerical results were obtained using `double precision` (64-bit) reals on a 3.6 GHz Intel Xeon dual processor Dell Precision 670 with 4 Gbytes of RAM. The Nag Fortran f95 compiler with the optimization flag `-O` was used together with the ATLAS BLAS and LAPACK (`math-atlas.sourceforge.net`). In all our tests, the scaled residual (4.1) was computed; in each case, this was found to be less than 10^{-12}.

7.1 Effect of the block size

The efficiency of the kernel code `HSL_MA74` that performs the partial factorization of the frontal matrices is dependent upon the choice of the block size nb. As explained in Section 5.1, the partial factorization proceeds by finding a block of pivots and then (for $nb > 1$) uses Level 3 BLAS kernels to update the remaining rows and columns. Factorization timings (CPU times in seconds) for a subset of our test problems using a range of block sizes are presented in Tables 7.2 and 7.3. These timings are for the default pivoting strategy (threshold partial pivoting). At the root node, a complete factorization of the final frontal matrix must be performed and, if partial pivoting is in use, this can be done by using either `MA74_factor` or the LAPACK subroutine `_getrf`. Table 7.2 reports the factorization time at the root node and Table 7.3 reports the total time for the factorization phase and the Mflop rates. The last column in Table 7.3 is for using `MA74_factor` with $nb = 80$ at non-root nodes and `_getrf` at the root node. The results illustrate the importance of using a block algorithm and show that, for some examples, the root node factorization can account for more than 20 per cent of the total factorization time. For our test computer and our test set, the blocksize nb =80 appears to be a good choice; this is the default within `HSL_MA78`. Using `_getrf` at the root node gives some modest gains.

Table 7.1: The test problems. n and $nelt$ denote the number of variables and elements, respectively, and $nz(L)$ is the predicted number entries in L, in millions. * indicates only pattern available.

Identifier	n	$nelt$	$nz(L)$	Description/discipline
1. trdheim*	22098	813	1.74	CFD simulation; mesh of Trondheim fjord
2. cham*	12834	11070	2.58	Part of an engine cylinder
3. crplat2*	18010	3152	2.94	Corrugated plate field
4. tubu*	26573	23446	4.87	Engine cylinder model
5. opt1*	15449	977	5.27	Part of condeep cylinder
6. tsyl201*	20685	960	6.43	Part of condeep cylinder
7. srb1*	54924	9240	10.01	Space shuttle rocket booster
8. ship_001	34920	3431	15.61	Ship structure - predesign
9. thread	29736	2176	24.66	Threaded connector
10. x104	108384	26019	27.15	Beam joint
11. mt1	97578	5328	32.68	Tubular joint
12. shipsec8	114919	32580	36.30	Section of a ship
13. shipsec1	140874	41037	38.70	Section of a ship
14. shipsec5	179860	52272	54.22	Section of a ship
15. fcondp2*	201822	35836	55.16	Oil production platform
16. ship_003	121728	45464	60.28	Ship structure - production
17. troll*	213453	41084	63.68	Structural analysis
18. halfb*	224617	70211	66.20	Half-breadth barge
19. fullb*	199187	59738	75.02	Full-breadth barge
20. inv-ext-2*	78142	7193	126.05	Fluid flow

7.2 Times for each phase

In Section 4, we discussed the different phases of the HSL_MA78 package. In Table 7.4, we report the elapsed times for each phase for our five largest test problems (this includes the time taken to perform the input/output operations). The input time is the time taken by the calls to MA78_input_vars and MA78_input_reals, and the ordering time is the time for MA57 to compute the pivot sequence. MA78_factor(0) and MA78_factor(1) are, respectively, the times for MA78_factor and for MA78_factor_solve when called with a single right-hand side and the L factor is stored for future solves. MA78_factor(1) (no L) is the time for MA78_factor_solve with a single right-hand side but without storing the L factor. MA78_solve(k) is the time for MA78_solve with k right-hand sides.

We see that the time for inputting the matrix, ordering, and performing the analyse phase is very small comapred with the factorization time which, for these examples, dominates the total solution time. As expected, because of the better use of high level BLAS and because the amount of data to be read from

Table 7.2: Comparison of the CPU time (in seconds) required to factorize the root node with different blocksizes (nb) and _getrf. n_{root} is the order of the root node.

		HSL_MA74					
	n_{root}	nb =1	nb =10	40	80	120	_getrf
9. thread	3099	88.1	9.42	10.6	5.75	6.94	4.93
10. x104	2350	68.5	9.37	12.3	6.78	5.67	5.01
17. troll	2313	36.7	4.51	4.87	2.91	3.26	2.30
19. fullb	3053	89.1	11.0	11.4	7.69	6.96	5.00

Table 7.3: Comparison of the factorization phase CPU times (in seconds) and Mflop rates using HSL_MA74 with different blocksizes (nb) and _getrf.

	HSL_MA74					nb =80
	nb =1	10	40	80	120	_getrf
9. thread	323/ 224	42.2/ 1720	45.3/ 1599	29.8/ 2434	32.2/ 2250	28.5/ 2546
10. x104	225/ 227	37.7/ 1415	42.6/ 1347	30.9/ 1768	32.2/ 1863	29.9/ 1832
17. troll	460/ 228	71.2/ 1472	76.0/ 1379	55.6/ 1885	58.0/ 1806	54.8/ 1913
19. fullb	865/ 230	133/ 1494	136/ 1468	96.0/ 2079	102/ 1947	94.2/ 2117

disk is independent of the number of right-hand sides, solving for multiple right-hand sides is significantly more efficient than solving repeatedly for a single right-hand side.

Table 7.4: Elapsed times (in seconds) for the different phases of HSL_MA78.

Problem / Phase	16	17	18	19	20
Input	0.73	0.75	1.12	1.06	2.36
Ordering	1.74	2.92	2.52	2.44	1.54
MA78_analyse	0.41	0.35	0.47	0.45	0.39
MA78_factor(0)	114	58.6	71.8	94.0	263
MA78_factor(1)	116	59.2	74.8	98.5	269
MA78_factor(1)(no L)	108	53.8	67.4	90.7	245
MA78_solve(1)	3.95	3.38	3.77	4.12	17.2
MA78_solve(8)	6.19	5.33	5.80	6.42	20.9
MA78_solve(64)	21.7	20.9	22.5	23.1	35.5

Unfortunately, we found that the elapsed times can be very dependent on the other activity on our machine, as may be seen by the time for MA78_factor(0) plus MA78_solve(1) sometimes being less than the MA78_factor(1) time (problem 19). Another way to judge the performance is to look at the number of records actually read or written using HSL_OF01, see Table 7.5. By comparing the sum of the number of records read and written for MA78_factor(0) and MA78_solve(1) with the number read and written for MA78_factor(1), we see that there are worthwhile i/o savings if the solve is performed at the same time as the factorization. There are further significant savings if the L factor is not stored.

Table 7.5: Records read and written (in thousands) for the factorization and solve phases of HSL_MA78.

— Problem / Phase		16	17	18	19	20
MA78_factor(0)	read	18.88	9.79	20.32	31.77	90.46
	write	50.27	38.52	46.99	56.83	112.1
	both	79.15	48.31	77.31	88.60	202.2
MA78_factor(1)	read	39.82	29.94	41.20	54.22	122.9
	write	50.27	38.53	47.00	56.83	112.1
	both	90.09	68.47	88.20	111.0	235.0
MA78_factor(1)	read	18.85	9.70	20.23	31.70	90.44
(no L)	write	32.12	22.86	30.68	38.37	81.23
	both	50.97	32.56	50.71	70.07	171.7
MA78_solve(1)	read	41.87	40.32	41.89	44.9	64.78

7.3 Comparison of partial and rook pivoting

As we discussed in Section 5.2, HSL_MA78 offers partial threshold pivoting and rook threshold pivoting. In Table 7.6, we compare the performance of both options for the test problems that are supplied complete with numerical values. For the default threshold tolerance of 0.01, the total solution time, the number of flops required to compute the factors, the number $nz(L)$ of entries in the L factor, the number delay of delayed eliminations, and the scaled residuals (see (4.1)) are given. If p_i and q_i are, respectively, the numbers of candidate and actual pivots chosen at node i then

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

Table 7.6: Comparison between the elapsed times (in seconds), flops, the number of entries in L, the number of delayed eliminations, and the scaled residuals for factorization with rook and partial threshold pivoting.

Problem	Time		flops$*10^8$		$nz(L)*10^6$		delay		Residual	
	rook	partial	rook	partial	rook	partial	rook	partial	rook	partial
8. ship_001	15.0	13.4	224	224	15.6	15.6	33	33	$5.7*10^{-16}$	$3.1*10^{-16}$
9. thread	37.8	35.4	725	725	24.7	24.7	0	0	$3.1*10^{-16}$	$4.0*10^{-16}$
10. x104	34.0	37.8	365	548	30.3	34.5	19918	31596	$6.2*10^{-16}$	$9.9*10^{-14}$
11. m_t1	55.7	94.9	688	1584	40.2	56.2	34161	67467	$4.7*10^{-16}$	$8.5*10^{-14}$
12. shipsec8	91.6	92.8	1297	1753	49.0	55.6	61004	78033	$5.3*10^{-16}$	$2.4*10^{-14}$
13. shipsec1	110	174	1505	3047	58.6	78.2	96811	135258	$4.0*10^{-16}$	$7.6*10^{-14}$
14. shipsec5	175	275	2460	4919	80.4	105	120881	168479	$1.8*10^{-15}$	$6.8*10^{-13}$
16. ship_003	146	118	2065	2281	70.8	74.0	49985	61262	$7.9*10^{-16}$	$1.5*10^{-13}$

In our tests, if there are only a few delayed eliminations, rook pivoting adds a small overhead (problems 8 and 9). However, the more stringent test used by rook pivoting can result in less growth and smaller residuals. Smaller growth can also lead to fewer delayed eliminations for rook pivoting, which in turn gives sparser factors that are computed using fewer flops. Because looking for each pivot is more expensive than for partial pivoting, the time can still increase (as illustrated by problem 16) but in some cases, the total time using rook pivoting is significantly faster than for partial pivoting (notably for problems 11, 13 and 14).

7.4 Comparison with HSL_MA42_ELEMENT

We now compare the performance of HSL_MA78 with that of our out-of-core (uni)frontal solver HSL_MA42_ELEMENT [18]. In our tests, for both solvers the default settings are used for all control parameters. Note that both use partial threshold pivoting with a threshold of 0.01. Table 7.7 reports the total solution time (that is, the time for analyse, factorize and solve for a single right-hand side), the number of flops required to compute the factors, and the number $nz(L)$ of entries in the L factor. For two of the smaller problems in the top half of the table, HSL_MA42_ELEMENT is faster than HSL_MA78 but, with the exception of problems thread and ship_001, the later performed fewer flops and produced the sparser factors. For the larger problems, HSL_MA78 significantly outperforms HSL_MA42_ELEMENT. Since both codes are written to exploit Level 3 BLAS and to perform input/output efficiently, and for both codes the elements are appropriately preordered, these results clearly illustrate the benefits of using a multifrontal algorithm in preference to a (uni)frontal method.

Table 7.7: Comparison between the elapsed times (in seconds), flops and the number of entries in L for HSL_MA78 and HSL_MA42_ELEMENT.

Problem	Time		flops$*10^8$		$nz(L)*10^6$	
	MA42_ELEMENT	MA78	MA42_ELEMENT	MA78	MA42_ELEMENT	MA78
1. trdheim	0.82	0.92	5.87	3.38	2.22	1.74
2. cham	2.45	2.03	29.4	18.2	4.23	2.58
3. crplat2	1.81	1.43	21.7	17.7	4.36	2.94
4. tubu2	8.03	4.01	107	29.9	11.3	4.88
5. opt1	7.31	4.23	94.4	59.5	7.51	5.27
6. tsyl201	8.83	5.08	108	72.3	10.4	6.44
7. srb1	9.46	6.13	119	64.3	17.5	10.1
8. ship_001	10.6	13.4	146	224	15.6	15.6
9. thread	46.8	35.4	720	725	31.0	24.7
10. x104	1289	37.8	5206	548	137.4	34.5
11. m_t1	556	94.9	4164	1584	138.5	56.2
12. shipsec8	993	92.8	7265	1753	186.1	55.6
13. shipsec1	4132	174	14673	3047	300.0	78.2
14. shipsec5	1988	275	9974	4919	196.3	104.9
15. fcondp2	683	61.4	11547	1106	304.8	55.2
16. ship_003	577	118	4706	2281	161.3	74.0
17. troll	3102	67.9	55184	1048	671.1	63.7
18. halfb	547	79.6	8256	1404	291.3	66.2
19. fullb	786	102	13304	1995	356.2	75.1
20. inv-ext-2	442	286	7588	6863	153.1	126.1

8 Concluding remarks

We have designed and developed a new multifrontal solver for unsymmetric finite-element problems. The new Fortran 95 solver, HSL_MA78, optionally holds the matrix data, the matrix factors and the multifrontal stack out of core, allowing much larger systems to be solved than is possible with a conventional solver. The efficiency of HSL_MA78 is dependent upon the partial factorization and solution of the dense frontal matrices and upon the writing to and reading from direct access files. We have developed separate HSL packages to perform these operations.

HSL_MA78, together with the subsidiary packages HSL_MA74 and HSL_OF01, are available as part of the 2007 release of the mathematical software library HSL. All use of HSL requires a licence; details of how to obtain a licence and the packages are available at www.cse.clrc.ac.uk/nag/hsl/. For symmetric problems (both positive definite and indefinite problems), the HSL solver HSL_MA77 [17] should be used.

References

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

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

[3] M. Arioli, I.S. Duff, S. Gratton, and S. Pralet. A note on GMRES preconditioned by a perturbed ldlt decomposition with static pivoting. Technical Report RAL-TR-2006-07, Rutherford Appleton Laboratory, 2006.

[4] 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.

[5] I.S. Duff. MA57– a new code for the solution of sparse symmetric definite and indefinite systems. *ACM Transactions on Mathematical Software*, 30:118–144, 2004.

[6] I.S. Duff and S. Pralet. Towards a stable static pivoting strategy for the sequential and parallel solution of sparse symmetric indefinite systems. Technical Report RAL-TR-2005-07, Rutherford Appleton Laboratory, 2005.

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

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

[9] B.M. Irons. A frontal solution program for finite-element analysis. *Inter. Journal on Numerical Methods in Engineering*, 2:5–32, 1970.

[10] ISO/IEC. TR 15581(E): Information technology - Programming languages - Fortran - Enhanced data type facilities (second edition), edited by M. Cohen. Technical Report, ISO/IEC, 2001. ISO, Geneva.

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

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

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

[14] L. Neal and G. Poole. A geometric analysis of Gaussian elimination, II. *Linear Algebra and Applications*, 173:239–264, 1992.

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

[16] J.K. Reid and J.A. Scott. HSL_OF01, a virtual memory system in Fortran. Technical Report RAL-TR-2006-026, Rutherford Appleton Laboratory, 2006.

[17] J.K. Reid and J.A. Scott. An out-of-core sparse Cholesky solver. Technical Report RAL-TR-2006-013, Rutherford Appleton Laboratory, 2006.

[18] J. A. Scott. A frontal solver for the 21st century. *Communications in Numerical Methods in Engineering*, 22:1015–1029, 2006.

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