Science & Technology Facilities Council

Technical Report

An efficient out-of-core sparse symmetric indefinite direct solver

J. K. Reid and J. A. Scott

October 2008

RAL-TR-2008-024

ISSN 1358-6254

An efficient out-of-core sparse symmetric indefinite direct solver [1,2]

by

J. K. Reid and J. A. Scott

Abstract

The fast and accurate solution of large sparse linear systems of equations is important in many problems from computational science and engineering. `HSL_MA77` is a state-of-the-art high-performance, robust, Fortran 95 software package that implements a multifrontal algorithm. It can be used to solve both positive-definite and indefinite linear systems and, by holding the system matrix and its factors out-of-core, it is able to solve very large problems. In this paper, we focus on features of the code that are specifically designed for the more challenging indefinite case and present numerical results for indefinite problems that arise from a range of practical applications.

Keywords: sparse linear systems, indefinite, symmetric, out-of-core solver, multifrontal.

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

[2] The work of the second 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.

October 1, 2008.

Contents

1 Introduction

Systems of sparse, symmetric linear equations arise naturally in a number of important engineering and science applications. The size of the systems users want to solve is constantly increasing as mathematical models become ever more complex and the requirement to solve three-dimensional problems becomes commonplace. Our interest is in efficiently solving linear systems of the form

$$Ax = b \tag{1.1}$$

where the $n \times n$ matrix A is sparse, symmetric and indefinite. Direct methods generally split the solution process into three separate phases: analyse, factorize and solve. The analyse phase takes only the pattern of the matrix and uses it to construct data structures in preparation for the numerical factorization. The factorization phase forms the matrix decomposition

$$A = (PL)D(PL)^T, \tag{1.2}$$

where P is a permutation matrix, L is a unit lower triangular matrix, and D is a block diagonal matrix with blocks of size 1×1 and 2×2. Finally, the solution phase uses the matrix factor to perform forward eliminations where

$$PLy = b \tag{1.3}$$

is solved for y, then the block diagonal system

$$Dz = y \tag{1.4}$$

is solved for z, followed backward substitution where

$$(PL)^T x = z \tag{1.5}$$

is solved for x.

For sparse indefinite systems, direct methods are frequently the method of choice because, provided numerical pivoting is properly incorporated, they are robust for a wide range of problems and so can be used as "black-box" solvers. Furthermore, they are often preferred for systems with multiple right-hand sides since the additional work required to solve for more than one right-hand side is normally much less than is required to form the initial factorization. They can also be used (sometimes in modified form) to provide preconditioners for iterative methods. However, one of the main limitations of direct methods is that the amount of memory they require generally increases rapidly with problem size. Our interest is in the case where the matrix A and its factor L are too large for the factorization to be performed and held in main memory. Instead, A, D, L and some of the main work arrays are held in files. Such a solver is termed an out-of-core solver.

The aim of this paper is to report on a new sparse symmetric out-of-core solver that has been developed for the HSL mathematical software library HSL [16]. The code is designed to efficiently solve both positive-definite and indefinite systems. In a previous paper [22], we described the first release of `HSL_MA77`, which was only for positive-definite problems; in this paper, we focus on the features of the solver that are specifically for the indefinite case and present numerical results that illustrate its performance on a range of indefinite systems that arise from practical applications.

2 Overview of `HSL_MA77`

`HSL_MA77` is a Fortran 95 package that implements an out-of-core multifrontal algorithm. In this section, we give a brief introduction to the out-of-core multifrontal algorithm implemented within `HSL_MA77` and then discuss the design and user interface of `HSL_MA77`.

2.1 Introduction to the out-of-core multifrontal algorithm

The multifrontal method [8] is an important generalisation of the frontal method [17] that has been popular since the mid 1980s. The factorization of A proceeds using a succession of assembly operations into small dense matrices (the so-called *frontal matrices*), interleaved with partial factorizations of these matrices. Each frontal matrix can be expressed in the form

$$F = \begin{pmatrix} F_{11} & F_{21}^T \\ F_{21} & F_{22} \end{pmatrix}, \tag{2.1}$$

where F_{11} and F_{21} are *fully summed*, that is, all the entries in the corresponding part of the overall matrix have been assembled, while F_{22} is not yet fully summed. If F_{11} has order p and q pivots can be chosen stably from F_{11}, the partial factorization of F takes the form

$$F = Q \begin{pmatrix} L_1 & 0 \\ L_2 & I \end{pmatrix} \begin{pmatrix} D_1 & 0 \\ 0 & F_S \end{pmatrix} \begin{pmatrix} L_1^T & L_2^T \\ 0 & I \end{pmatrix} Q^T. \tag{2.2}$$

where Q is a permutation matrix of the form

$$Q = \begin{pmatrix} Q_1 & 0 \\ 0 & I \end{pmatrix}$$

with Q_1 having order p, L_1 is a unit lower triangular matrix of order q, and D_1 is a block diagonal matrix of order q. The matrices Q_1, L_1, and D_1 are not required again until the forward elimination and back substitution phases and so may be stored elsewhere, while the Schur complement F_S is treated as a new element, called a *generated element* (or *contribution block*).

The assembly operations can be recorded as a tree, termed an *assembly* tree. 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 (the so-called *multifrontal 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.

During the depth-first search, the storage required by the factors always grows while the size of the stack varies depending on the operations performed: when the partial factorization of a frontal matrix is processed, the resulting generated element is stacked, increasing the stack size; on the other hand, when the frontal matrix at node v is formed and assembled, the generated elements corresponding to children of v are popped from the stack and its size decreases. The stack memory is thus very dependent on the assembly tree topology, which in turn depends on the chosen elimination order. As the stack size can become large, it is clear that, even if it is possible to hold the matrix A and its factor in memory, there may be insufficient memory for the stack as well. As we have already observed, after a partial factorization of a frontal matrix, the computed partial factor is not required again until the solve phase. Thus, the simplest out-of-core multifrontal approach writes the partial factors as they are computed to file, and reads them back as needed during the forward elimination and again during the back substitution. This minimizes the number of input-output operations but, if the stack becomes very large, there may still be insufficient memory and it is then necessary to write the stack to a file.

2.2 Input/output operations within `HSL_MA77`

One of our main design goals for `HSL_MA77` was that it should be modular, that is, a number of the key steps should be implemented by independent, standalone packages. Modularity not only aids us in the development and maintenance of a complex code but also allows other software developers to exploit the separate packages in other contexts.

One of the standalone packages that we have developed is HSL_OF01 [21]. This Fortran 95 package performs all the input/output operations required by HSL_MA77. The files used by HSL_MA77 need to be direct-access files to allow data to be read in an order that differs from the order in which they were written. Records in a direct-access file are all the same length but, in the multifrontal algorithm, it is necessary to read and write varying amounts of data at different stages of the computation. To achieve this, HSL_OF01 provides facilities for reading from and writing to direct-access files using a buffer that is a work array held in main memory; careful handling of the buffer aims to minimise the actual input/output operations. Each version of HSL_OF01 has its own buffer (there are separate versions for real, complex and integer data), and each buffer can be associated with more than one direct-access file. HSL_MA77 uses one integer superfile. In addition, when solving an indefinite system, three real files are used: the *main* file is used to hold the original matrix and the computed factor, the *stack* file holds the multifrontal stack and the *delay* file holds further temporary data during factorization (see Section 3.2). One buffer is associated with the three files enabling the available memory to be dynamically shared amongst them according to their needs at each stage of the computation.

Each HSL_OF01 buffer is divided into *pages* that are all of the same size, which is also the size of each record in the associated files. 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 in the buffer are parameters that may be set by the user. Numerical experiments that we reported in [21] were used to choose default values for these parameters.

The data in a file are addressed as a virtual array of rank one. 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.

HSL_OF01 has an option when writing data 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. We use this during the factorization phase of HSL_MA77 when writing the columns of the factor since it is known that most of them will not be needed for some time and it is more efficient to use the buffer for the stack. There is also 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 stack file since it is known that it will not be needed again. Further details of HSL_OF01 and the options it offers are included in [21] and in the user documentation.

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. If a file becomes full, HSL_OF01 opens secondary files and treats the primary file and all its secondaries as a single *superfile*. This is all done automatically within the code, without the user needing to take any action, but to enable the secondary files to reside on different devices, the user may provide an array of path names; the full name of a file is the concatenation of a path name with the user-supplied file name.

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. To avoid unnecessary copying in such a case, we have included within HSL_MA77 an option that allows the superfiles to be replaced by arrays. If this is used, we refer to HSL_MA77 as running *in-core*. This option is discussed further in [22] (see also Section 4.4).

2.3 Kernel code, HSL_MA64

The partial factorization of the frontal matrix (equation (2.2)) and the corresponding partial solutions needed later are performed by the kernel code HSL_MA64. We use nf to denote the order of the frontal matrix F.

To perform assembly operations efficiently, HSL_MA77 packs the lower triangular part of each generated element by columns and constructs each frontal matrix to have this form. It follows that this is the desirable form for F on input to HSL_MA64 and for the generated element F_S on return. Unfortunately, this is not a good form for the partial factorization since it does not allow the BLAS-3 subroutine _gemm (and other BLAS) to be used.

We therefore require the user of HSL_MA64 to choose a block size nb and the code begins the factorization by rearranging the first p columns of the lower triangular part of F, $\begin{pmatrix} F_{11} \\ F_{21} \end{pmatrix}$, so that it is held by block columns, with each block having nb columns (except the final block, which usually has fewer). The first block is rectangular with nf rows, the next block is rectangular with $nf - nb$ rows, etc. This format is a little wasteful since the first $j-1$ entries of the jth column of each block are not used, but the total waste is no more than $p(nb-1)/2$. Our reason for choosing this form rather than one in which the blocks on the diagonal are packed is that it makes the code for interchanging rows and columns, needed for pivoting, faster and less complicated.

We do not retain this format for $\begin{pmatrix} L_1 \\ L_2 \end{pmatrix}$ on return since that would significantly increase the file space needed; our experience is that quite large values of nb are desirable, for example, 120 (see [20]). Once the partial factorization is complete, this matrix is rearranged; it is still held by block columns, but each consists of the diagonal block packed by columns followed by the off-diagonal part held by columns. This form allows the partial solution operations for a single right-hand side to be performed with the BLAS-2 subroutines _gemv and _tpsv and for multiple right-hand sides to be performed with the BLAS-3 subroutine _gemm and repeated calls of _tpsv.

If the number of chosen pivots q is less than p, we need to rearrange columns $p+1$ to q back to be packed by columns, ready for assembly operations in HSL_MA77.

We found that there is also a need to subdivide the blocks, but defer explaining this until Section 2.3.2.

2.3.1 Factorization in the simple case

We begin by describing the case where the matrix is not found to be singular or nearly singular, and none of the options described in subsections 2.3.4, 2.3.5, and 2.3.6 are requested.

We choose the pivots one by one by searching the first p columns in sequence, 1, 2, ... We use the notation f_{ij} to denote an element of the frontal matrix after it has been updated by all the pivot operations so far. Our test for a 1×1 pivot is

$$|f_{kk}| \geq \mathtt{u} \max_{j \neq k} |f_{kk}|, \tag{2.3}$$

where $\mathtt{u}$ is a user-set threshold in the range $0 \leq \mathtt{u} \leq 0.5$. Our test for a 2×2 pivot is

$$\left| \begin{pmatrix} f_{kk} & f_{kl} \\ f_{kl} & f_{ll} \end{pmatrix} \right| \begin{pmatrix} \max_{j \neq k,l} |f_{kj}| \\ \max_{j \neq k,l} |f_{lj}| \end{pmatrix} \leq \begin{pmatrix} \mathtt{u}^{-1} \\ \mathtt{u}^{-1} \end{pmatrix} \tag{2.4}$$

where $\mathtt{u}$ is the same user-set threshold.

Let us use m to denote the index of the column being searched and q to denote the number of pivotal columns found so far. If we fail to find a pivot in column m, we leave the column in place, but keep it updated as further pivots are found. If we find a stable 1×1 pivot in column m, we swap rows and columns q+1 and m (unless $m = q$+1) then increment q by one. Otherwise, we look for a stable 2×2 pivot in columns t and m with $t < m$ (which have been kept updated); if one is found, we swap rows and columns q+1 and t and rows and columns q+2 and m, then increment q by two.

Having chosen a pivot, we apply the pivotal operations to any previously rejected columns, that is, columns q+1 to m. At this point, we have a factorization of the form (2.2) with L_1 and D_1 of order q but only the first $m-q$ columns of F_S available.

We now increment m by one and apply all the pivot operations to column m, which becomes available for the next pivot search. This continues until q has the value nb or $nb+1$ (or p).

Now we apply the first *nb* pivot operations to columns $m+1$ to p. In order that these operations can be performed with the BLAS-3 subroutine `_gemm`, we employ an array `buf` of size $(nb+1)\times nf$ to hold

$$U = D_1(\ L_1^T \quad L_2^T\ ).$$

If a 2×2 pivot spans two block columns, U will have $nb+1$ rows. In this case, we use only the first *nb* entries of a column of U to update the corresponding column of F and use the remaining entry later when doing the next block update.

U is calculated row by row with each pivot operation, so that `_gemv` (or `_gemm` for a 2×2 pivot) can be employed to update the rejected columns and to apply all the outstanding operations at once to the next column to be searched.

Once the first *nb* pivot operations have been applied, we perform a similar set of operations on the reduced matrix, except that we may start after the second half of a 2×2 pivot. This continues until column p has been searched, that is, $m = p$, after which a fresh search of the rejected columns is made, that is, m is reset to $q+1$.

The whole process continues until $q = p$ or all the remaining columns have failed to provide a pivot. At this point, we update columns $p+1$ to nf. In order to use `_gemm` here, we group the columns into blocks of size $nb/2$ (or less for the final block). This allows us to rearrange the block column to the format used for the first p columns into free space within the array, apply `_gemm`, and then rearrange back to packed format. The choice of $nb/2$ here is based on requiring the array to have size at least $nf(nf+nb+1)/2$, which is needed in the case $p = nf$.

Note that if nf and p are large, most of the work is done within `_gemm` and so the execution should be fast; and almost all the rest is done within `_gemv`. If p is small, almost all the work is done within `_gemv`.

2.3.2 The need for inner blocks

A disadvantage of the scheme described in the previous subsection is that a large block size *nb* is needed for rapid `_gemm` updating of the trailing blocks, but this results in a large number of `_gemv` updates of columns within the pivotal block column. Jonathan Hogg (private communication) suggested to us that we should have an inner block size *nbi* and perform `_gemm` updates on the trailing part of the pivotal block column as soon as *nbi* pivots have been chosen. For efficiency and to make the code simpler, we require that $\texttt{mod}(nb, nbi) = 0$, so that all the inner blocks are of size exactly *nbi*. We found that choosing $nbi = nb/3$, which leads to about 2/3 of the `_gemv` updates being grouped into `_gemm` updates, gave a worthwhile gain.

2.3.3 Factorization in the singular case

When a column is searched, if its largest element is found to be less than a user-set tolerance `cntl%small`, the diagonal entry is accepted as a zero 1×1 pivot and no corresponding pivotal operations are applied to the rest of the matrix. To accommodate this, we hold the inverse of D_1, and set the element corresponding to the zero pivot to zero. This avoids the need for special action in subsequent BLAS-2 and BLAS-3 calls later in the factorization and during the solution. It leads to the correct result when the given set of equations is consistent and avoids the solution having a large norm if the equations are not consistent. The number of zero pivots is returned to the user.

2.3.4 Factorization with delayed columns present

If one or more of the generated elements contributing to F have delayed pivots (Section 3.2), the corresponding columns will be leading columns of F and will not have changed since their prior rejection. It is therefore desirable to delay looking at these since they are likely to be rejected again. `HSL_MA64` has an optional argument `s` whose presence indicates that the first `s` columns are of this nature.

If `s` is present, swaps are made between columns j and $p+1-j$ for $j = 1$ to $\min(\texttt{s}, p-\texttt{s})$ before the search for pivots starts, unless this would involve splitting a recommended 2×2 pivot which may happen if s $< p -$ s. In the case that would cause a split, the swaps are made only for $j = 1$ to s $- 1$. Following this, the leading s columns (s $-$ 1 in the exceptional case) will have been moved to the back of the set of columns $1 : p$.

2.3.5 Requesting particular 2×2 pivots

HSL_MA64 allows its user to specify that particular pairs in the pivot sequence should be used as 2×2 pivots. If m indexes the first half of such a pair, the largest entry in column m is determined, but the diagonal entry of the column is not accepted as a 1×1 pivot. Processing continues to column $m + 1$ and now the recommended pivot can be tested for acceptance as a 2×2 pivot. Either it is accepted or the recommendation is cancelled and normal processing is resumed.

2.3.6 Static pivoting

HSL_MA64 has an option for static pivoting, that is, forcing pivots that do not satisfy the stability test to be chosen, perhaps after modification. The intention is to reduce fill-in, probably at the expense of iterative refinement of each solution to get good accuracy.

If static pivoting is requested, the following procedure is followed whenever no 1×1 or 2×2 candidate pivot satisfies the relative threshold test (2.3) or (2.4). The 1×1 pivot that is nearest to satisfying (2.3) is chosen. If its absolute value is less than `cntl%static`, it is given the value that has the same sign but absolute value `cntl%static`. If no diagonal entry is changed in this way, the value of `u` that would have resulted in all pivots satisfying the test is returned to the user, together with a count of the number of pivots that did not satisfy (2.3) with the user-set value of `u`.

2.3.7 Partial solutions

HSL_MA64 provides subroutines for the following partial solutions

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

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

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

$$\begin{pmatrix} L_1^T & L_2^T \\ 0 & I \end{pmatrix} X = B.$$

where X and B have the same shape. There are separate subroutines for the special case where X and B have just one column.

2.4 Selective use of 64-bit integers

To allow HSL_MA77 to be used 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 the frontal matrix, within the 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 its row and column indices. We have chosen not to make all integers within the code long integers since to do so would increase the amount of data stored and moved around within the solver, and there is no long integer version of the BLAS kernels that are used within HSL_MA64 to efficiently compute the partial factorizations of the dense frontal matrices.

2.4.1 32-bit and 64-bit architectures

As 64-bit architectures are becoming increasing commonplace, we have designed HSL_MA77 so that it can be run on both 32-bit and 64-bit architectures. In particular, HSL_MA77 has a control parameter control%bits that the user should set to 64 if running on a 64-bit architecture. On a 32-bit architecture, the maximum size of a rank-1 real array that can be allocated is taken to be huge(0_short)/4 in the single precision version and huge(0_short)/8 in the double precision version, where huge is the Fortran inquiry function. On a 64-bit architecture, it is taken to be huge(0_long)/4 and huge(0_long)/8, respectively. Since the frontal matrices are held in-core, we have to allocate a real array of size maxfront*(maxfront+nb+1)/2, where maxfront is the largest frontsize and nb is the block size. Thus, using a 64-bit architecture, allows us to factorize much larger problems than is possible on a 32-bit machine.

2.5 The elimination order

One of the early design decisions made for HSL_MA77 was that the user should supply the elimination order on the call to MA77_analyse. The performance of the multifrontal method is highly dependent upon the elimination order but, unfortunately, there is no single approach that always produces a good ordering. Variants of the minimum degree algorithm generally work well on problems that are either not too large (typically, those of order less than 50,000) or are highly sparse, and recently work has gone into improving performance of these orderings for matrices that have some (almost) dense rows [1, 3]. However, for larger problems, algorithms based on nested dissection, although more costly, usually yield higher quality orderings. In the HSL library, there are a number of packages available that can be used to compute minimum degree based orderings. These have been brought together within HSL_MC68. If a nested dissection ordering is wanted, the generalized multilevel nested-dissection routine METIS_NodeND from the METIS graph partitioning package of Karypis and Kumar [18, 19]) may be employed.

The analyse phase of HSL_MA77 returns statistics on the predicted number of entries in the factors, the maximum size of a frontal matrix, and the number of flops required to perform the factorization. The user may experiment with passing different orderings to MA77_analyse and then select the best on the basis of these statistics before calling the factorization phase. Since the factorization phase is significantly more expensive than the analyse phase (this is illustrated in Section 4), it can be worthwhile to try out different orderings. In particular, if the user wants to solve a series of linear systems in which the system matrices have the same (or similar) sparsity patterns, the savings from having a good ordering will more than offset the cost of several calls to MA77_analyse.

2.6 User interface

HSL_MA77 is designed to solve systems where the system matrix A is either an assembled matrix or is a sum of square symmetric elements (such as in a finite-element calculation). In both cases, A may be expressed in the form

$$A = \sum_{k=1}^{m} A^{(k)}.$$

In the element case, the summation is over elements and $A^{(k)}$ is nonzero only in those rows and columns that correspond to variables in the kth element, while in the assembled case, the summation is over rows and $A^{(k)}$ is nonzero only in row k. Many widely used direct solvers (including, for example, the HSL solvers MA57 [4] and MA48 [9]) require the matrix to be assembled and to be passed to the code using a single call. Since one of the main goals for HSL_MA77 is to limit the amount of main memory needed, we want to avoid having to hold the the whole of A in main memory and, instead, we follow the design used by, for example, the out-of-core frontal solver MA42 [11] and employ a reverse communication interface, with control being returned to the calling program for the user to supply each element or row.

The main routines that may be called by the user are:

- `MA77_input_vars` must be called once for each element or row to specify which variables are associated with it. The elements (or rows) may be entered in any order. `HSL_OF01` is called to write the index lists to the integer superfile.
- `MA77_analyse` must be called after all calls to `MA77_input_vars` are complete. The integer data and user-supplied elimination order is used to construct the assembly tree and other data structures needed for the factorization. These data structures are held in the integer superfile.
- `MA77_input_reals` must be called for each element or row to specify the entries of $A^{(k)}$. `HSL_OF01` is called to write the read data to the main real superfile.
- `MA77_factor` may be called after all the reals of A have been input and after the call to `MA77_analyse`. A is factorized using the assembly tree and data structures constructed on the call to `MA77_analyse`. The factor is stored in the main real superfile.
- `MA77_solve` uses the stored factor data to solve systems $AX = B$. Any number of calls to `MA77_solve` with different numbers of right-hand sides may follow a call to `MA77_factor`.

In addition, there is a routine `MA77_resid` that may be used to compute the residual matrix $B - AX$. There is also a restart facility that, after a successful factorization, allows the user to save the factor and the data structures used by `HSL_MA77` to solve for further right-hand sides at a later time, or to factorize another matrix with the same sparsity pattern without repeating the calls to `MA77_input_vars` and `MA77_analyse`.

Derived types are used to pass data between the different routines. In particular, `MA77_control` has components that control the action within the package and `MA77_info` has components that return information from subroutine calls. The control components are given default values when a variable of type `MA77_control` is declared and may be altered thereafter. The information returned to the user includes a flag to indicate error and warning conditions, the determinant of A (its sign and the logarithm of its absolute value), the maximum frontsize, the number of entries in the factor L, the computed rank, information on the pivot sequence, and the number of floating-point operations. In addition, data is returned on the number of integer and real records that are read from and written to the superfiles using `HSL_OF01`. Note that the use of derived types to hold data for a problem makes it possible to have more than one problem active at the same time.

If the buffer used by `HSL_OF01` is big enough, actual input/output will be avoided, but there remain the overheads associated with copying data to and from the buffer. Within `HSL_MA77`, we are particularly anxious to avoid this during the solve phase for a single right-hand side. `HSL_MA77` therefore offers an option that allows the superfiles to be replaced by arrays in memory.

Full details of the user interface and the options available to the user are provided in the documentation that accompanies the code.

3 `HSL_MA77` for indefinite systems

In this section, we discuss features of the different phases of `HSL_MA77` that are specific to the solution of indefinite systems.

3.1 The analyse phase

In the indefinite case, the elimination order computed by a minimum degree or nested dissection algorithm based on the sparsity pattern of the matrix provides only a tentative pivot order and the statistics returned by `MA77_analyse` are lower bounds on the factor size, flop count, etc. This is because numerical stability considerations during the factorization may lead to the elimination order being modified and pivots may be delayed (that is, used later than expected in the elimination order). These changes to the elimination ordering usually resulted in more fill in the factor, more flops, and (occasionally) a larger maximum front size. Results in the literature illustrate that the difference between the predicted and actual statistics can be significant (see, for example, [5]). This is particularly likely to happen if the matrix A has zeros on the diagonal and no allowance is made for this when choosing the elimination ordering. Most ordering algorithms implicitly assume that the diagonal is present. In the early 1990s, Duff, Gould, Reid, Scott and Turner [5] investigated taking the zeros on the diagonal of A into account when computing the elimination order and they proposed allowing the tentative pivot sequence to include 1×1 and 2×2 pivots. This study involved a special way of storing the frontal matrices and generated elements. It led to the development of the HSL package `MA47` [10] for the solution of indefinite problems in which the matrix A has a significant number of zero diagonal entries (note that `MA47` is now available from the HSL Archive, see `http://hsl.rl.ac.uk/archive/hslarchive.html`). The `MA47` ordering is one of the options available within the package `HSL_MC68`.

The pivot sequence (including any suggested 2×2 pivots) must be passed to `MA77_analyse`. If the input pivot sequence contains any 2×2 pivots, care has to be taken to ensure both entries are kept together so that a variable that is part of a 2×2 pivot is only marked as fully summed if both it and its partner are fully summed.

3.2 The numerical factorization

Consider again the frontal matrix (2.1) and assume that F_{11} is of order p and that $q < p$ pivots are chosen by `MA64_factor`. In this case, $p-q$ pivots are *delayed* and the generated element that is passed to the parent node will be larger than anticipated on the basis of the sparsity pattern alone. This larger generated element may be expressed as

$$F_G = \begin{pmatrix} F_{G1} & F_{G2}^T \\ F_{G2} & F_{S1} \end{pmatrix}, \tag{3.1}$$

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 (2.2).

After each partial factorization of a frontal matrix, `HSL_OF01` is used to store the permuted indices in the integer superfile. The analyse data is preserved so that the user can factorize more than one matrix with the same sparsity pattern but different numerical values without recalling `MA77_analyse`. Each list starts with the indices of the chosen pivots, and is followed by the indices of the delayed pivots. The delay superfile is used to hold (in a stack) the remaining columns that are fully summed but, for numerical reasons, could not be pivoted on. That is, the lower triangular part of F_{S1} is stacked in the stack superfile while the lower triangular part of F_{G1} together with F_{G2} are placed on a separate stack that is held 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 columns of F_p. Thus, at each non-leaf node v, the frontal matrix takes the form

$$F_p = \begin{pmatrix} F_{p1} & F_{p2}^T \\ F_{p2} & F \end{pmatrix},$$

where the order of F is the *anticipated* order of the frontal matrix and F_{p1}, F_{p2} 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 pivots in a stack since this data can be retrieved from the integer lists stored in the main integer superfile. Moreover, we only hold the lower triangular part of F_p. It is held in packed form within a rank-1 array that is allocated at the start of the factorization. If the number of delayed pivots increases so that size of this array is insufficient, the contents are temporarily written to the stack superfile, the array is reallocated with a larger size and the data read back in. This reallocation process may have to be repeated more than once because the new array may still be too small if there are many more delayed pivots later in the factorization. A control parameter allows the user to determine the amount of extra space that is allocated to accommodate delayed pivots. This is particularly useful if a series of similar problems are to factorized.

3.3 The solve phase

The forward eliminations and back substitutions are performed by calling `MA77_solve`. This in turn calls subroutines from the package `HSL_MA64`. The user may solve for any number of right-hand sides in a single call and repeated calls may be made. On each call, the factor data must be read from file twice: once for the forward elimination and once for the back substitution. As can be seen from the numerical experiments reported on in Section 4, this is expensive. It is more efficient to perform the forward elimination operations at the same time as the factorization so that the factor data only has to be read once for the back substitution. The user-callable routine `MA77_factor_solve` is provided to allow for this. Note that the amount of data that must be read is independent of the number of right-hand sides and so it is significantly more efficient to solve for several right-hand sides at once than to make repeated calls to `MA77_solve` with one right-hand side.

The argument list for `MA77_solve` includes an optional integer argument `job`, that may be used to indicate whether the complete solution is required (that is, $AX = B$ is to be solved) or only a partial solution (see (1.3), (1.4), and (1.5)). An example where partial solutions are required is the HSL package `HSL_VF06`. Given a real symmetric matrix H, a real vector c and a positive radius δ, `HSL_VF06` computes a global minimizer of the quadratic objective function $\frac{1}{2}x^T Hx + c^T x$, where x is required to satisfy the norm constraint $||x||_M \leq \delta$ with $||x||_M = \sqrt{x^T Mx}$. Such problems commonly occur as trust-region subproblems in nonlinear optimization calculations. Details may be found in [14].

3.4 Additional features for indefinite problems

In some applications, it is important to have access to the entries of the block diagonal matrix D and to be able to alter one or more of these entries. For example, it may be desirable to modify D to $\hat{D}$ so that the matrix $\hat{A} = (PL)\hat{D}(PL)^T$ is positive definite. This is discussed in [12, 15]. To facilitate this, `HSL_MA77` includes two routines, `MA77_enquire_indef` and `MA77_alter`. A call to `MA77_enquire_indef` returns the actual pivot sequence used by the numerical factorization phase (with 2×2 pivots identified using negative flags on both entries) together with the entries in D, which are returned in a rank-2 array `d`. The diagonal entries and the off-diagonal entries of D are held in `d(1,1:n)` and `d(2,1:n-1)`, respectively. Note that, if the user wishes to solve a series of problems having the same sparsity pattern but different numerical values, it may be advantageous to recall `MA77_analyse` using the pivot sequence from `MA77_enquire_indef` because, if the numerical values have not changed significantly, this will reduce the amount of searching for suitable pivots that is needed within the subsequent factorization.

If the user wishes to modify the entries of D, `MA77_alter` may be called with the new diagonal entries placed in `d(1,1:n)` and the off-diagonal entries in `d(2,1:n-1)`. Note that the new values will overwrite the original values so that the factorization of A is no longer available.

4 Numerical experiments

To illustrate the performance of `HSL_MA77`, we have selected all the large real square symmetric matrices from the University of Florida Sparse Matrix Collection [2] for which the values of the matrix entries are supplied and that are not flagged as positive definite. Our definition of large is that the matrix is of order at least 10^4 and the analyse phase of `HSL_MA77` predicts at least $1.5 * 10^6$ entries in the matrix factor L. Table 7.1 in the Appendix lists our set of 59 indefinite test problems in increasingly order of the predicted number $nz(L)$ of entries in L (this gives the problem indices used in the figues in this section). Here $nz(A)$ denotes the millions of entries in A (upper and lower triangular parts).

All reported experiments were performed using double precision reals on a Dell Precision T5400 with two Intel E5420 quad core processors running at 2.5GHz backed by 8GB of RAM. We used the Goto BLAS [13] and ifort compiler with the -fast option.

Unless stated otherwise, all control parameters are used with their default settings. The only exceptions are that `control%bits` is set to 64 since our test machine has 64-bit architecture and, in addition, we use a file size of 2^{25} scalars (in tests on very large problems, the default of 2^{21} was insufficient). Version 4.0.0 of HSL_MA77 is used, together with version 3.0.0 of HSL_OF01 and version 3.0.0 of HSL_MA64. In our experiments, the test matrices are scaled using a symmetrized version of the HSL package MC64 [6, 7].

Throughout this section, the *complete solution time* for HSL_MA77 refers to the total time for inputting the matrix data, computing the pivot sequence (the analyse phase of the MA57 package [4] is used for this), and calling the analyse, factorize and solve phases. Where appropriate, timings for HSL_MA77 include all input/output costs involved in holding the data in superfiles. All reported times are wall clock times in seconds.

4.1 The effects of node amalgamation

During the construction of the assembly tree by the analyse phase of HSL_MA77, a node is amalgamated with its parent if both involve less than a given number, nemin, of eliminations. This is explained in more detail in [22]. We show, in Tables 4.1 and 4.2, a few of our results on the effect of varying nemin. The subset of problems used here was chosen to illustrate the range of results we observed for the whole set. For nemin greater than 1 and less than about 32, we found that for many of our problems (including 6 and 37), the factor and solve times are not very sensitive to the precise choice of nemin (although the number of entries in L always increases with nemin, with a larger proportion of explicit zeros being stored). We also show in Table 4.1 the number of nodes that the HSL multifrontal solver MA57 finds from the same pivot sequence with its default nemin value of 16 as well as the resulting number of entries in L. We note that MA57 uses a different node amalgamation algorithm that generally performs far fewer amalgamations (see [22]).

Table 4.1: Comparison of the number of non-leaf nodes and the number of entries in L for different values of the node amalgamation parameter nemin.

	Number of nodes ($*10^3$)							Number of entries in L ($*10^6$)						
	MA57	HSL_MA77						MA57	HSL_MA77					
nemin	16	1	4	8	12	16	24	16	1	4	8	12	16	24
6. c-73	83	143	81	77	75	72	68	2.23	1.76	2.69	3.24	3.72	4.22	5.24
20. darcy003	157	312	104	46	26	19	13	9.61	6.94	7.77	9.28	10.8	12.3	14.9
35. Si10H16	1.5	6.2	2.0	1.1	0.7	0.6	0.4	31.2	30.7	31.0	31.4	31.7	32.0	32.6
37. t3dh	4.9	12	0.5	0.3	0.2	0.2	0.1	47.9	47.2	47.6	48.1	48.4	48.9	50.2
47. sparsine	4.3	17	6.0	3.3	2.2	1.8	1.2	201	200	200	202	202	204	206

Table 4.2: Comparison of the factorization phase and solve phase times (single right-hand side) for different values of the node amalgamation parameter nemin.

	Factorize times						Solve times					
nemin	1	4	8	12	16	24	1	4	8	12	16	24
6. c-73	0.80	0.73	0.75	0.76	0.77	0.80	0.23	0.18	0.18	0.18	0.19	0.20
20. darcy003	2.19	1.46	1.25	1.91	1.84	1.25	0.52	0.39	0.35	0.36	0.36	0.39
35. Si10H16	50.1	32.3	26.8	24.1	22.5	20.8	0.80	0.71	0.71	0.70	0.70	0.71
37. t3dh	14.4	13.8	13.8	13.7	13.6	13.8	2.23	1.07	1.08	1.08	1.09	1.10
46. sparsine	781	519	417	372	349	319	5.53	5.07	4.97	4.97	4.96	5.03

In Figure 4.1, we compare the HSL_MA77 factorize time for nemin set to 8 and to 16. We see that, for many of the smaller problems 8 slightly outperforms 16. Since for some examples (including problems 6 and 20) $nz(L)$ grows rapidly with nemin, we have chosen the default to be 8. However, for the larger

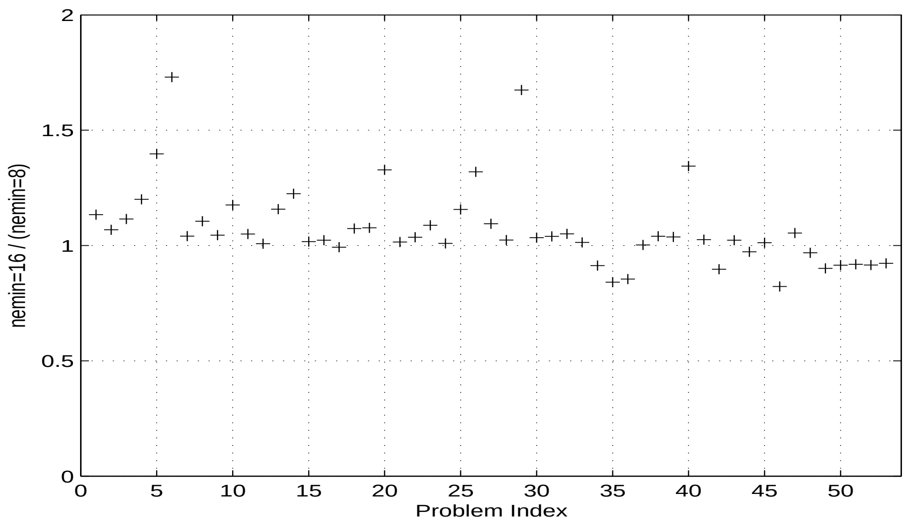

Figure 4.1: The ratios of the factorize times for `nemin` = `16` to the factorize times for `nemin` = `8`.

test examples, on our test machine it can be advantageous in terms of time to use `nemin` = `16` (or, as seen in Table 4.2, `nemin` = `24`). It should be noted that the largest problems in our test set come from the same subset (`PARSEC`) of the University of Florida Sparse Matrix Collection and so they may share characteristics that result in `nemin` = `16` outperforming `nemin` = `8`. This illustrates the importance of experimenting with different `nemin` if the factorization speed is the user's prime concern and a number of problems with the same (or similar) sparsity patterns are to be factorized. We remark that, if the number of entries in L increases slowly with `nemin`, it can be better to use an even larger `nemin`. For instance, we ran problem `sparsine` with `nemin` = `64` and found the factorization time reduced from 349 seconds with `nemin` = `16` to 276 seconds, while the number of entries in L increased only slightly from $204 * 10^6$ to $213 * 10^6$ (so that there was almost no increase in the solve time).

In Figure 4.2, we compare the number of entries in the matrix factor computed using `HSL_MA77` with the number for `MA57`, each with its default value of `nemin`. We see that the `MA57` factor is sparser than the `HSL_MA77` factor in about a quarter of the problems and is sometimes significantly so, notably for problems 2, 6 and 21. Problem 6 was included in Tables 4.1 and 4.2. We looked at problem 2 in more detail and found that for `MA57` the maximum frontsize is 7 whereas for `HSL_MA77` it is 14. This is much smaller than for the other problems in our test set (see Table 7.1), which suggests that a smaller value of `nemin` should be used in this case, illustrating the potential advantage of experimenting with different settings.

4.2 Times for each phase

In Table 4.3, we report the times for each phase of `HSL_MA77` for some of our largest test problems. The input time is the time taken to input the matrix data (using calls to `MA77_input_vars` and `MA77_input_reals`), and the ordering time is the time for `MA57` to compute the pivot sequence. The solve time is for a single right-hand side. We see that the solution time is, as expected, dominated by the factorization time.

Another way to judge the performance is to look at the number of records actually read from or written to files using `HSL_OF01`; this is reported for the same test problems in Table 4.4. We see that the largest problems have to perform significantly more input/output operations.

We can also assess the overall performance using gigaflop rates. On our test machine, a typical speed for `dgemm` multiplying two square matrices of order 1 is about 9 Gflops. In Table 4.5 the gigaflop rates corresponding to the results in Table 4.3 are presented (these are computed using the wall clock times).

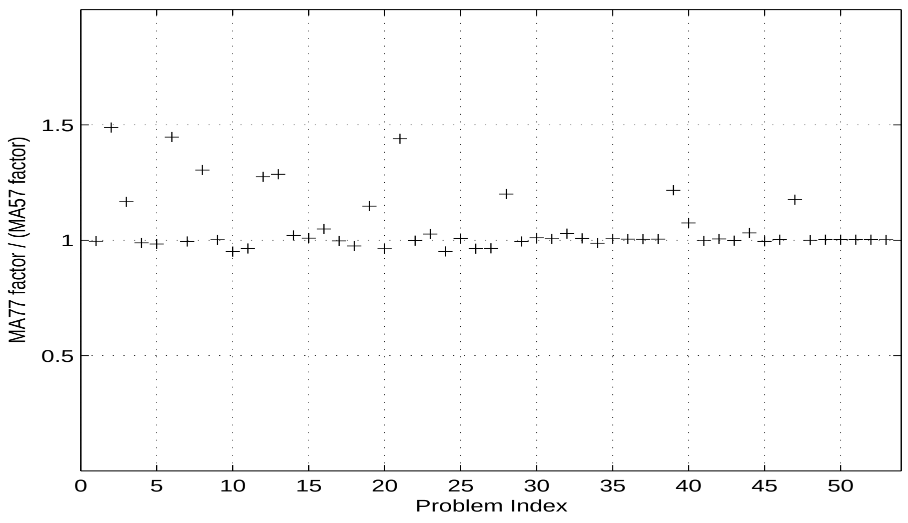

Figure 4.2: The ratio of the number of entries in L for `HSL_MA77` to the number for `MA57`, each with its default value of `nemin`.

Table 4.3: Times for the different phases of `HSL_MA77`.

Problem / Phase	46	47	48	54	55	56	57	58
Input	0.06	1.09	1.75	0.50	0.69	0.35	0.50	1.02
Ordering	3.61	28.0	9.38	12.9	18.0	20.2	23.0	29.9
`MA77_analyse`	5.28	11.1	2.21	19.3	27.3	33.0	36.8	48.9
`MA77_factor`	347	255	104	2891	4551	6666	7435	10478
`MA77_solve`	5.00	7.11	15.1	122	198	254	308	389

Table 4.4: Records read from and written to files (in thousands) for the factorization and solve phases of `HSL_MA77`.

Problem / Phase		46	47	48	54	55	56	57	58
`MA77_factor`	read	99.3	977	31.4	411	564	738	881	1048
	write	111	439	110	507	684	887	1021	1222
`MA77_solve`	read	99.4	134	239	510	694	906	1006	1231

Table 4.5: Gflop rates for the factorization and solve phases of `HSL_MA77`.

Problem / Phase	46	47	48	54	55	56	57	58
`MA77_factor`	3.96	2.60	3.79	4.59	4.43	4.54	4.68	4.51
`MA77_solve`	0.204	0.197	0.128	0.43	0.036	0.036	0.033	0.032

During the factorization rate, we achieve approximately half the peak rate for problems 54 to 59. The low rates for the solve phase indicates the cost of reading in the factor data dominates the total cost of the solve phase and this is particularly true for the largest problems that perform the most input/output.

4.3 Performance on positive-definite examples

`HSL_MA77` is designed to efficiently solve both positive-definite and indefinite systems. If the user knows that the matrix is positive definite, the parameter `pos_def` should be set to `.true.` on the call to `MA77_factor`. The partial factorization of the dense frontal matrices is then performed by the HSL module `HSL_MA54`; no numerical pivoting is performed by `HSL_MA54`, allowing the pivot sequence from `MA77_analyse` to be used without modification. Details of `HSL_MA54` and the positive-definite version of `HSL_MA77` are given in [22]. Since there is no searching and checking of pivots and no delayed pivots and thus no unexpected fill in, we anticipate that, as well as producing sparser factors, the positive-definite version will be faster than the indefinite version when run on a positive-definite system. One way of assessing how well the indefinite version of `HSL_MA77` is performing is to run it on positive-definite problems and compare the times and the factor sizes when run with `pos_def` = `.true.` and `pos_def` = `.false.` (with the default threshold parameter 0.01). Our set of positive-definite problems is taken from [22] and are listed in Table 7.2 in the Appendix. Our tests use Version 1.3.0 of `HSL_MA54`. Comparing the size of the factor, when run with `pos_def` = `.false.`, problem 1 was the only one for which the number of entries in L was greater than the number predicted by the analyse phase. In this case, $nz(L)$ increased from $24 * 10^6$ to $27 * 10^6$. In Figure 4.3, we show the ratios of the positive-definite and indefinite factorization times. These are CPU times for running in core (in-core working is discussed further in the next section). For all but problem 1, the indefinite time is within 20% of the positive-definite time. In all cases, the size of the residual was comparable.

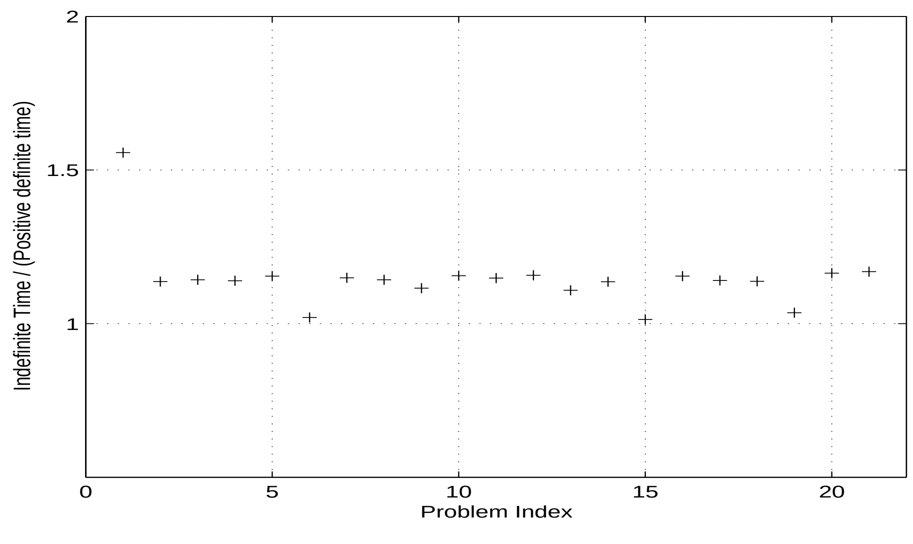

Figure 4.3: The factorization CPU time with `pos_def` = `.false.` divided by the CPU time with `pos_def` = `.true.` when the matrix is positive definite.

4.4 Comparisons with in-core working and with `MA57`

Finally, we compare the performance of `HSL_MA77` out of core with its performance in core and with the well-known HSL package `MA57` [4]. This is also a multifrontal code but it does not offer out-of-core options.

There is currently no 64-bit version of MA57 and so it is restricted to problems for which the factor (which is held as a rank-one array) can be addressed using a 32-bit integer. We have run Version 3.2.0 of MA57 on our test set using the default settings for all control parameters. In particular, as for HSL_MA77, the threshold parameter for partial pivoting is `u = 0.01`. If the size of the arrays for holding the factors that are passed to the factorization phase of MA57 are found to be insufficient, we use the subroutine MA57ED to move the computed factor data into larger arrays and then continue the factorization from this point. The final 5 problems in Table 7.1 could not be solved using MA57 and so are omitted.

When running HSL_MA77 in core, the code attempts to use arrays in place of superfiles. To run in core, the user has to set the control parameter control%maxstore to hold the maximum number of Fortran storage units (4 bytes) to be used for the arrays. Initial sizes for the superfiles are provided by the user or selected by the code from the value of control%maxstore. Should an array subsequently be found to be too small, an attempt is made to reallocate it with a larger size. If this either fails or violates control%maxstore, a file is used instead (thus a mixture of superfiles and arrays may be used). In our experiments, we first ran HSL_MA77 out-of-core and used the information returned on the maximum number of integers and reals stored in the superfiles together with the reported total amount of storage used by the superfiles to set the controls for the in-core run. However, on our test machine, we also imposed the restriction that control%maxstore should not exceed 10^9 (about half the actual memory). Experimentation showed that, if much larger values were permitted, the performance deteriorated because of page swapping. For example, when problem 53 was run in core without the above limit on control%maxstore, the elapsed time for the solve phase was more than 50% greater than for the out-of-core run. In our experiments with control%maxstore $= 10^9$, we found that problems 49 to 53 used a combination of arrays and superfiles. In particular, problem 49 used stack and delay superfiles (and held the integer data and the factor data in arrays), while problems 50 to 53 used a superfile for the factor data. We include these cases in our Figures and Tables for in-core runs, since at least some of their data are held in memory.

In Figures 4.4 to 4.6, we compare the factorize, solve (single right hand side) and total solution times for MA57 and for HSL_MA77 in-core (using arrays in place of superfiles) with those for HSL_MA77 out-of-core (using default settings). The complete solution time for MA57 is the total time for calling the analyse, factorize and solve phases of MA57. The figures show the ratios of the MA57 and HSL_MA77 in-core times to the HSL_MA77 out-of-core times. To ensure we are comparing like with like, the HSL_MA77 factorize and total solution times include the time for prescaling using MC64 (scaling is performed within the factorization phase of MA57 and hence any timings for the this phase include the MC64 time).

From Figure 4.4, we see that the HSL_MA77 out-of-core factorization time is less than the MA57 factorization time for about 70% of the problems. For 10 of the problems, it is more than twice as fast as MA57. For those problems that can be run using arrays in place of files, in-core working usually increases the speed of HSL_MA77 by between 5 and 25%. The only problem for which the MA57 factorization is faster than HSL_MA77 run in core is problem 2, where the maximum front size is very small (see end of Section 4.1). We believe that the main reason for the strong performance of HSL_MA77 is its use of the carefully designed kernel code HSL_MA64 (see Section 2.3).

For the solution phase with a single right-hand side, the penalty for working out-of-core is much greater than for the factorization phase because the ratio of data movement to arithmetic operations is significantly higher. This is evident in Figure 4.5. We also observe that the solve phase of MA57 is faster than the solve phase of HSL_MA77 run in core. This is partly because the MA57 factors are sparser than those for HSL_MA77. A comparison with the results in Figure 4.2 shows that, if we exclude problems 50 to 53 that hold the factor data in files, the problems for which the difference between the MA57 and HSL_MA77 in-core solve times is most significant correspond to those for which the ratio of the number entries in L for HSL_MA77 to those for MA57 is largest (see, for example, problems 2, 6 and 21).

The ratios of the total solution times are presented in Figure 4.6. Here we see that for many of the problems, HSL_MA77 out of core is less than 20% slower than HSL_MA77 in core. Furthermore, for more than 75% of the problems, HSL_MA77 in core is as fast or faster than MA57. We note that HSL_MA77 includes

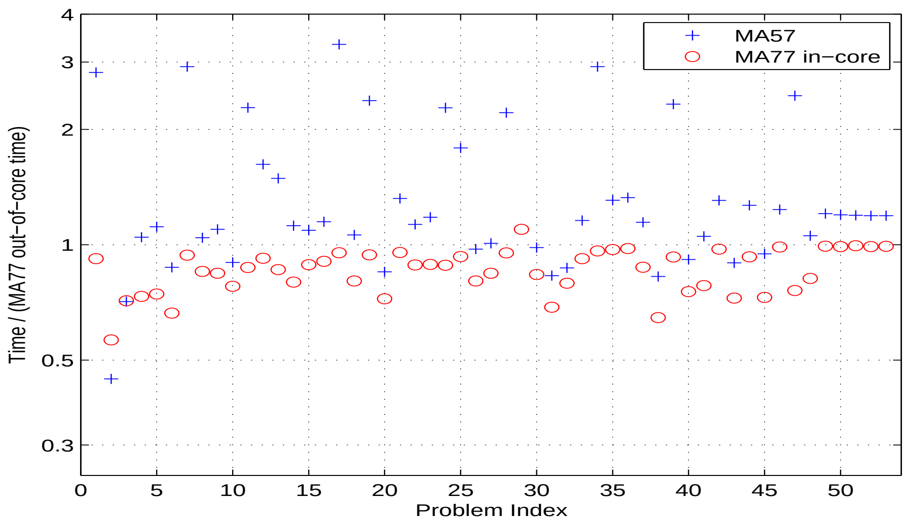

Figure 4.4: The ratios of the MA57 and HSL_MA77 in-core factorize times to the HSL_MA77 out-of-core factorize times.

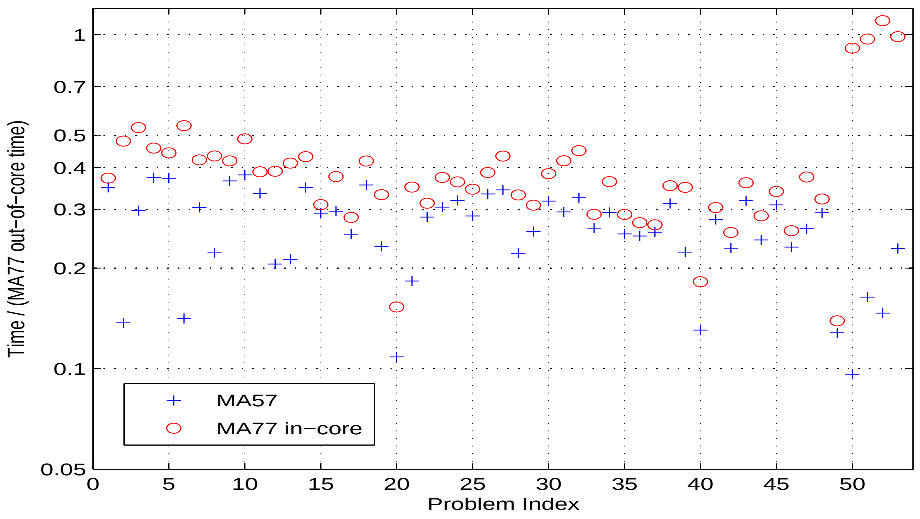

Figure 4.5: The ratios of the MA57 and HSL_MA77 in-core solve times to the HSL_MA77 out-of-core solve times (single right-hand side).

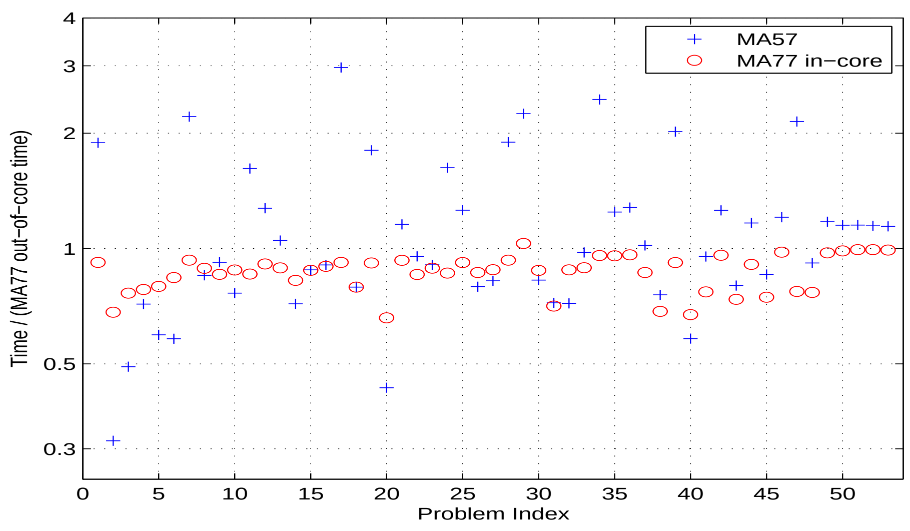

Figure 4.6: The ratios of the MA57 and HSL_MA77 in-core complete solution times to the HSL_MA77 out-of-core complete solution times (single right-hand side).

an option to solve at the same time as the factorization. If used, the factor data has to be read in only once, reducing the total solution time. However, as MA57 does not offer this option, our reported complete solution times for both packages are the total time for separate calls to the analyse, factorize and solve phases.

5 Concluding remarks

In this paper, we have discussed the design of a new multifrontal code HSL_MA77 for solving large-scale sparse symmetric linear systems. A key feature of the code is that it can hold the matrix and its factor, as well as its main work array, in files. The use of files is handled by a separate packge HSL_OF01 [21]. Numerical results have shown that, while working out-of-core adds an overhead, in our tests this was usually not prohibitive (although it may be if a large number of repeated calls to the solve phase are required). Moreover, it enabled us to solve problems that would otherwise be too large for a conventional sparse direct solver working only in main memory.

Another important aspect of HSL_MA77 is its use of a specially designed dense linear algebra kernel HSL_MA64. In the indefinite case, this performs the partial factorization of the frontal matrices and offers users a number of pivoting options, including threshold pivoting (using 1×1 and 2×2 pivots) and static pivoting.

HSL_MA77, together with the subsidiary packages HSL_MA64 and HSL_OF01, are included in the 2007 release of HSL. All use of HSL requires a licence. Licences are available without charge to individual academic users for their personal (non-commercial) research and for teaching; for other users, a fee is normally charged. Details of how to obtain a licence and further details of all HSL packages are available at www.cse.clrc.ac.uk/nag/hsl/.

6 Acknowledgements

We would like to express our appreciation to our colleagues Iain Duff and Jonathan Hogg for many helpful discussions.

References

[1] P. Amestoy, H.S. Dollar, J.K. Reid, and J.A. Scott. An approximate minimum degree algorithm for matrices with dense rows. Technical Report RAL-TR-2007-020, Rutherford Appleton Laboratory, 2007.

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

[3] H.S. Dollar and J.A. Scott. A note on fast approximate minimum degree orderings for matrices with some dense rows. Technical Report RAL-TR-2007-022, Rutherford Appleton Laboratory, 2007.

[4] 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.

[5] I.S. Duff, N.I.M. Gould, J.R. Reid, J.A. Scott, and K. Turner. Factorization of sparse symmetric indefinite matrices. *IMA Journal of Numerical Analysis*, 11:181–2044, 1991.

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

[7] I.S. Duff and S. Pralet. Strategies for scaling and pivoting for sparse symmetric indefinite problems. *SIAM Journal on Matrix Analysis and Applications*, 27:313 – 340, 2005.

[8] 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.

[9] I.S. Duff and J.K. Reid. The design of ma48, a code for the direct solution of sparse unsymmetris systems. *ACM Trans. Mathematical Software*, 22(1):30–45, 1996.

[10] I.S. Duff and J.K. Reid. Exploiting zeros on the diagonal in the direct solution of indefinite sparse symmetric linear systems. *ACM Trans. Mathematical Software*, 22(2):227–257, 1996.

[11] I.S. Duff and J.A. Scott. The design of a new frontal code for solving sparse unsymmetric systems. *ACM Trans. Mathematical Software*, 22(1):30–45, 1996.

[12] P.E. Gill, W. Murray, D.B. Ponceleón, and M.A. Saunders. Preconditioners for indefinite systems arising in optimization. *SIAM J. Matrix Analysis and Applications*, 13(1):292–311, 1992.

[13] Kazushige Goto and Robert van de Geijn. High performance implementation of the level-3 BLAS. *ACM Trans. Mathematical Software*, 34, 2008. To appear.

[14] N.I.M. Gould and J. Nocedal. The modified absolute-value factorization norm for trust-region minimization. In R. De Leone, A. Murli, P. M. Pardalos, and G. Toraldo, editors, *High Performance Algorithms and Software in Nonlinear Optimization*, pages 225–241. Kluwer Academic Publishers, 1998.

[15] N.J. Higham and S. Cheng. Modifying the inertia of matrices arising in optimization. *Linear Algebra and its Applications*, 275–276:261–279, 1998.

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

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

[18] G. Karypis and V. Kumar. METIS - family of multilevel partitioning algorithms, 1998. See http://glaros.dtc.umn.edu/gkhome/views/metis.

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

[20] J.K. Reid. Partial factorizations of dense symmetric matrices. Technical Report RAL-TR-2008-xxx, Rutherford Appleton Laboratory, 2008. To appear.

[21] 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. Revised Nov. 2007. To appear in *ACM Transactions on Mathematical Software.*

[22] J.K. Reid and J.A. Scott. An out-of-core sparse Cholesky solver. Technical Report RAL-TR-2006-013, Rutherford Appleton Laboratory, 2006. Revised Nov. 2007. To appear in *ACM Transactions on Mathematical Software.*

7 Appendix

Table 7.1: Indefinite test matrices and their characteristics. $nz(A)$ and $nz(L)$ denote the number of entries in A and the predicted number of entries in L, respectively, in millions. The problems are listed in order of increasing $nz(L)$. $front$ denotes the predicted maximum order of the frontal matrix. * indicates problem cannot be solved by `MA57` on our test machine.

Identifier	n	$nz(A)$	$nz(L)$	$front$
1. `ncvxqp1`	12111	0.07	1.68	912
2. `boyd2`	466316	1.50	2.59	14
3. `rail_79841`	79841	0.55	2.62	389
4. `bcsstk35`	30237	1.45	2.92	390
5. `stokes128`	49666	0.56	2.93	424
6. `c-73`	169422	1.28	2.94	604
7. `cvxqp3`	17500	0.12	3.13	1165
8. `c-63`	44234	0.43	3.28	823
9. `crystk02`	13965	0.97	4.39	777
10. `olesnik0`	88263	0.74	4.58	615
11. `cont-201`	80595	0.44	4.64	628
12. `c-59`	41282	0.48	5.11	1606
13. `c-72`	84064	0.71	5.12	1294
14. `dawson5`	51537	1.01	5.13	816
15. `t3dl`	20360	0.51	5.28	1092
16. `helm3d01`	32226	0.43	5.60	1192
17. `bratu3d`	27792	0.17	6.28	1521
18. `bcsstk39`	46772	2.09	6.88	515
19. `c-62`	41731	0.56	8.21	1750
20. `darcy003`	389874	2.10	8.31	618
21. `c-68`	64810	0.57	8.88	1984
22. `crystk03`	24696	1.75	9.82	1296
23. `copter2`	55476	0.76	10.44	1246
24. `cont-300`	180895	0.99	11.75	940
25. `ncvxqp5`	62500	0.42	12.05	1858
26. `turon_m`	189924	1.69	13.71	1045
27. `d_pretok`	182730	1.64	14.56	1139
28. `c-71`	76638	0.86	17.18	2542
29. `ncvxqp3`	75000	0.50	18.99	2443
30. `filter3D`	106437	2.71	20.10	945

Table 7.1: (continued)

Identifier	n	$nz(A)$	$nz(L)$	$front$
31. F2	71505	5.29	21.50	1236
32. helm2d03	392257	2.74	22.91	1024
33. qa8fk	66127	1.66	24.26	2075
34. ncvxqp7	87500	0.57	24.71	2734
35. Si10H16	17077	0.88	31.34	4383
36. Si5H12	19896	0.74	45.00	5551
37. t3dh	79171	4.35	48.13	3250
38. bmw3_2	227362	11.29	48.63	2090
39. c-big	345241	2.34	51.93	4277
40. ecology1	1000000	5.00	56.78	1649
41. halfb	224617	12.39	67.67	3240
42. SiO	33401	1.32	88.21	6914
43. af_shell9	504855	17.59	97.71	2205
44. Lin	256000	1.77	113.61	4761
45. schroeder_k	478788	23.62	114.32	3341
46. sparsine	50000	1.55	201.56	11459
47. kkt_power	2063494	14.61	213.69	6325
48. af_shell10	1508065	52.67	363.96	4355
49. Si34H36	97569	5.16	485.54	14490
50. Ge87H76	112985	7.89	642.64	16810
51. Ge99H100	112985	8.45	654.20	19250
52. Ga10As10H30	113081	6.12	674.25	16967
53. Ga19As19H42	133123	8.88	806.08	18675
54. SiO2*	155331	11.28	1037.07	21406
55. Si41Ge41H72*	185639	15.01	1410.90	23982
56. CO*	221119	7.67	1843.63	26380
57. Si87H76*	240369	10.66	2047.70	28326
58. Ga41As41H72*	268096	18.49	2507.06	30571

Table 7.2: Positive definite test matrices and their characteristics. $nz(A)$ and $nz(L)$ denote the number of entries in A and the number of entries in the Cholesky factor L, respectively, in millions. The problems are listed in order of increasing $nz(L)$. $front$ denotes the maximum order of the frontal matrix.

Identifier	n	$nz(A)$	$nz(L)$	$front$
1. `thread`	29,736	2.250	23.731	2994
2. `pkustk11`	87,804	2.653	28.517	2064
3. `pkustk13`	94,893	3.356	30.573	2145
4. `crankseg_1`	52,804	5.334	33.714	2124
5. `gearbox`	153,746	4.617	39.253	2215
6. `nd6k`	18,000	6.897	40.737	4430
7. `cfd2`	123,440	1.606	40.863	2522
8. `crankseg_2`	63,838	7.106	43.195	2205
9. `pwtk`	217,918	5.926	50.449	1128
10. `ship_003`	121,728	4.104	62.228	3336
11. `thermal2`	1,228,045	4.904	63.036	1413
12. `bmwcra_1`	148,770	5.396	71.230	2238
13. `af_shell3`	504,855	17.562	97.715	2205
14. `pkustk14`	151,926	7.494	108.931	3066
15. `g3_circuit`	1,585,478	4.623	118.476	2890
16. `nd12k`	36,000	14.221	118.492	7685
17. `ldoor`	952,203	23.737	154.742	2436
18. `inline_1`	503,712	18.660	179.269	3261
19. `bones10`	914,898	28.192	287.557	4695
20. `nd24k`	72,000	28.716	321.334	11363
21. `bone010`	986,703	36.326	1089.104	10722
22. `audikw_1`	943,695	39.298	1264.854	11223