An Out-of-Core Sparse Cholesky Solver

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

Direct methods for solving large sparse linear systems of equations are popular because of their generality and robustness. Their main weakness is that the memory they require usually increases rapidly with problem size. We discuss the design and development of the first release of a new symmetric direct solver that aims to circumvent this limitation by allowing the system matrix, intermediate data, and the matrix factors to be stored externally. The code, which is written in Fortran and called `HSL_MA77`, implements a multifrontal algorithm. The first release is for positive-definite systems and performs a Cholesky factorization. Special attention is paid to the use of efficient dense linear algebra kernel codes that handle the full-matrix operations on the frontal matrix and to the input/output operations. The input/output operations are performed using a separate package that provides a virtual-memory system and allows the data to be spread over many files; for very large problems these may be held on more than one device.

Numerical results are presented for a collection of 30 large real-world problems, all of which were solved successfully.

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

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Cholesky, sparse symmetric linear systems, out-of-core solver, multifrontal

ACM Reference Format:

Reid, J. K. and Scott, J. A. 2009. An out-of-core sparse Cholesky solver. ACM Trans. Math. Softw. 36, 2, Article 9 (March 2009), 33 pages. DOI = 10.1145/1499096.1499098
http://doi.acm.org/10.1145/1499096.1499098

1. INTRODUCTION

Direct methods for solving large sparse linear systems of equations are widely used because of their generality and robustness. Indeed, as a recent study of state-of-the-art direct symmetric solvers has demonstrated [Gould et al. 2007], the main reason for failure is a lack of memory. As the requirements

This work was partly funded by the EPSRC Grants GR/S42170 and EP/E053351/1.
Authors' addresses: Computational Science and Engineering Department, Atlas Centre, Rutherford Appleton Laboratory, Oxon OX11 0QX, U.K.; email: {john.reid Jennifer.scott}@stfc.ac.uk.

DOI 10.1145/1499096.1499098 http://doi.acm.org/10.1145/1499096.1499098

of computational scientists for more accurate models increases, so inevitably do the sizes of the systems that must be solved and thus the memory needed by direct solvers.

The amount of main memory available on computers has increased enormously in recent years and this has allowed direct solvers to be used to solve many more problems than was previously possible using only main memory. However, the memory required by direct solvers generally increases much more rapidly than the problem size so that they can quickly run out of memory, particularly when the linear systems arise from discretizations of three-dimensional (3D) problems. One solution has been to use parallel computing, for example, by using the MUMPS package [MUMPS 2007]. For many users, the option of using such a computer is either not available or is too expensive. An obvious alternative is to use an iterative method in place of a direct one. A carefully chosen and tuned preconditioned iterative method will often run significantly faster than a direct solver and will require far less memory. However, for many of the "tough" systems that arise from practical applications, the difficulties involved in finding and computing a good preconditioner can make iterative methods infeasible. An alternative is to use a direct solver that is able to hold its data structures on disk, that is, an out-of-core solver.

The idea of out-of-core linear solvers is not new. Indeed, the first author of this article wrote an out-of-core multifrontal solver for finite-element systems more than 20 years ago [Reid 1984] and the HSL mathematical software library [HSL 2007] has included out-of-core frontal solvers since about that time. The HSL package MA42 [Duff and Scott 1996] is particularly widely used, both by academics and as the linear solver within a number of commercial packages. The Boeing library [BCSLIB-EXT 2003] also includes multifrontal solvers with out-of-core facilities. More recently, a number of researchers [Dobrian and Pothen 2003; Rothberg and Schreiber 1999; Rotkin and Toledo 2004] have proposed out-of-core sparse symmetric solvers.

In this article, we discuss the design and development of the first release of a new HSL sparse symmetric out-of-core solver. The system matrix A, intermediate data, and the factors may be stored externally. The code, which is written in Fortran and called HSL_MA77, implements a multifrontal algorithm. The first release is for positive-definite systems and performs a Cholesky factorization. The second release has an option that incorporates numerical pivoting using 1×1 and 2×2 pivots, which extends the package to indefinite problems.

An alternative to the multifrontal algorithm is a left-looking strategy, where the column updates are delayed until the column is about to be eliminated. During the factorization, less data needs to be stored, but it has to be read many times. Our decision to use the multifrontal algorithm is based on our having extensive experience with this method and on not having seen evidence for its being consistently inferior.

This article describes the design of HSL_MA77, explaining many of the design decisions and highlighting key features of the package. Section 2 provides a brief overview of the multifrontal method. In Section 3, we describe the

structure of the new solver and, in particular, we explain the user interface. To minimize the storage needed for the system matrix A, a reverse communication interface is used. We note that designing a user-friendly interface while still offering a range options has been an important part of the development of HSL_MA77. A notable feature of our package 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 system is described elsewhere [Reid and Scott 2009a], but we include a brief overview in Section 4. Another key feature of HSL_MA77 is its use of efficient dense linear algebra kernels, which is discussed in Section 5. In Sections 6–8 we describe the different phases of the solver and, in particular, we look at the computation of supervariables and the construction of the assembly tree, node amalgamation, and the assembly order of the child nodes. Numerical experiments are reported in Section 9. These justify our choices of default settings for our control parameters and illustrate the performance of HSL_MA77; we also compare its performance with the well-known HSL solver MA57 [Duff 2004] on problems arising from a range of application areas. Finally, we look at future developments and make some concluding remarks.

We note that the name HSL_MA77 follows the HSL naming convention that routines written in Fortran 95 have the prefix HSL_ (which distinguishes them from the Fortran 77 codes).

2. OVERVIEW OF THE MULTIFRONTAL METHOD

HSL_MA77 implements an out-of-core multifrontal algorithm. The multifrontal method, which was first implemented by Duff and Reid [1982, 1983], is a variant of sparse Gaussian elimination. When A is positive definite, it involves a factorization

$$A = (PL)(PL)^T, \tag{2.1}$$

where P is a permutation matrix and the factor L is a lower triangular matrix with positive diagonal entries. Solving the linear system

$$AX = B$$

is completed by performing forward substitution

$$PLY = B, \tag{2.2}$$

followed by back substitution

$$(PL)^T X = Y. \tag{2.3}$$

If the right-hand side B is available when the factorization (2.1) is calculated, the forward substitution (2.2) may be performed at the same time, saving input/output operations when the factor is held out of core.

2.1 The Multifrontal Method for Element Problems

The multifrontal method is a generalisation of the frontal method [Irons 1970]. The frontal method was originally designed for finite-element problems. Here,

$A = \{a_{ij}\}$ is the sum of element matrices

$$A = \sum_{k=1}^{m} A^{(k)}, \tag{2.4}$$

where each element matrix $A^{(k)}$ has nonzeros in a small number of rows and columns and corresponds to the matrix from element k. The key idea behind frontal methods is to interleave assembly and elimination operations. As soon as pivot column p is *fully summed*, that is, involved in no more sums of the form

$$a_{ij} \leftarrow a_{ij} + a_{ij}^{(k)}, \tag{2.5}$$

the corresponding column of the Cholesky factor may be calculated:

$$l_{pp} \leftarrow \sqrt{a_{pp}}, \quad l_{ip} \leftarrow a_{ip}/l_{pp}, \ \ i > p,$$

and the basic Gaussian elimination operation

$$a_{ij} \leftarrow a_{ij} - l_{ip}\, l_{jp} \tag{2.6}$$

may be performed despite not all assembly operations (2.5) being complete for these entries. It is therefore possible to intermix the assembly and elimination operations.

Clearly, the rows and columns of any variables that are involved in only one element are fully summed before the element is assembled. These variables are called *fully summed*, too, and can be eliminated before the element is assembled, that is, the operations (2.6) can be applied to the entries of the element itself:

$$a_{ij}^{(k)} \leftarrow a_{ij}^{(k)} - l_{ip}\, l_{jp}.$$

This is called *static condensation*. The concept of static condensation can be extended to a submatrix that is the sum of a number of element matrices and this is the basis of the multifrontal method.

Assume that a pivot order (i.e., an order in which the eliminations are to be performed) has been chosen. For each pivot in turn, the multifrontal method first assembles all the elements that contain the pivot. This involves merging the index lists for these elements (That is, the lists of rows and columns involved) into a new list, setting up a full matrix (called the *frontal* matrix) of order the size of the new list, and then adding the elements into this frontal matrix. Static condensation is performed on the frontal matrix (i.e., the pivot and any other fully summed variables are eliminated). The computed columns of the matrix factor L are stored and the reduced matrix 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 summarized in Figure 1.

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

```
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 static condensation;
        add the generated element to the set of elements
    end if
end do
```

Fig. 1. Basic multifrontal factorization.

pivot. If A is 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.

2.2 The Multifrontal Method for Nonelement Problems

Duff [1984] extended the multifrontal method to nonelement problems (and assembled element problems). In this case, we can regard row i of A as a packed representation of a 1×1 element (the diagonal a_{ii}) and a set of 2×2 elements of the form

$$A^{(ij)} = \begin{pmatrix} 0 & a_{ij} \\ a_{ij} & 0 \end{pmatrix},$$

where a_{ij} is nonzero.

When i is chosen as pivot, the 1×1 element plus the subset of 2×2 elements $A^{(ij)}$ for which j has not yet been selected as a pivot must be assembled. Since they are all needed at the same time, a single leaf node can be used to represent them. To allow freedom to alter the pivot sequence, we hold the whole row. The non-leaf nodes represent generated elements, as before.

2.3 Partial Factorization at a Node

We now briefly consider the work associated with the static condensation that is performed at an individual node of the assembly tree. Static condensation performs a partial factorization of the frontal matrix. The frontal matrix is a dense matrix that may be expressed in the form

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

where the fully summed variables correspond to the rows and columns of F_{11}. The operations can be blocked as the Cholesky factorization

$$F_{11} = L_{11} L_{11}^T,$$

the update operation

$$L_{21} = F_{21} L_{11}^{-T},$$

and the calculation of the generated element

$$S_{22} = F_{22} - L_{21} L_{21}^{T}.$$

2.4 The Pivot Order

The performance of the multifrontal method is highly dependent upon the pivot sequence. During the past 20 years or so, considerable research has gone into the development of algorithms that generate good pivot sequences. The original HSL multifrontal code `MA27` [Duff and Reid 1983] used the minimum degree ordering [Tinney and Walker 1967]. Minimum degree and variants including approximate minimum degree [Amestoy et al. 1996, 2004] and multiple minimum degree [Liu 1985], have been found to perform well on many small and medium-sized problems (typically, those of order less than 50,000). However, nested dissection has been found to work better for very large problems, including those from 3D discretizations (see, for example, the results of Gould and Scott [2004]). Many direct solvers now offer users a choice of orderings including either their own implementation of nested dissection or, more commonly, an explicit interface to the generalized multilevel nested-dissection routine `METIS_NodeND` from the METIS graph partitioning package [Karypis and Kumar 1998, 1999].

2.5 Multifrontal Data Structures

The multifrontal method needs data structures for the original matrix A, the frontal matrix, the stack of generated elements, and the matrix factor. An out-of-core method writes the columns of the factor to disk as they are computed. If the stack and frontal matrix are held in main memory and only the factors written to disk, the method performs the minimum possible input/output for an out-of-core method: it writes the factor data to disk once and reads it once during back substitution or twice when solving for further right-hand sides (once for the forward substitution and once for the back substitution). However, for very large problems, it may be necessary to hold further data on disk. We hold the stack and the original matrix data on disk, but have a system for virtual memory management (see Section 4) that avoids much of the actual input/output. In the current version of our solver, we hold the frontal matrix in main memory.

3. STRUCTURE OF THE NEW SOLVER

Having outlined the multifrontal method, in this section we discuss the overall structure of our multifrontal solver `HSL_MA77`.

3.1 Overview of the Structure of `HSL_MA77`

`HSL_MA77` is designed to solve one or more sets of sparse symmetric equations $AX = B$. A may be input in either of the following ways:

(1) by square symmetric elements, such as in a finite-element calculation, or
(2) by rows.

In each case, the coefficient matrix is of the form (2.4). In (1), the summation is over elements and $A^{(k)}$ is nonzero only in those rows and columns that correspond to variables in the kth element. In (2), the summation is over rows and $A^{(k)}$ is nonzero only in row k. An important initial design decision was that the HSL_MA77 user interface should be through reverse communication, with control being returned to the calling program for each element or row. This is explained further in Section 3.4. 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; if convenient, the user may choose to generate the elements or rows without ever holding the whole matrix. There is no required input ordering for the elements or rows. In the future, a simple interface that avoids reverse communication will be offered.

We have chosen to require the right-hand sides B to be supplied in full format, that is, B must be held in an $n \times nrhs$ array, where $nrhs$ is the number of right-hand sides. The solution X is returned in the same array, again in full format. This is convenient for the user with a nonelement (or an assembled element) problem and the user who needs to perform some calculation on the solution and call the code again, such as for iterative refinement or an eigenvalue problem. During forward and back substitution, it is clearly advantageous to hold the right-hand sides in memory.

Another key design decision was that the package would not include options for choosing the pivot order. Instead, a pivot order must be supplied by the user. This is because 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 packages already exist that can be used. For example, METIS_NodeND can be used to compute a nested dissection ordering while the HSL package HSL_MC68 offers efficient implementations of the minimum degree and the approximate minimum degree algorithms As far as we are aware, no satisfactory ordering code that holds the matrix pattern out of core is currently available; instead, the pattern plus some additional integer arrays of size related to the order and density of A must be held in main memory.

Given the pivot sequence, the multifrontal method can be split into these phases:

—An analyze phase that uses the the pivot sequence and the index lists for the elements or rows to construct the assembly tree. It also calculates the work and storage required for the subsequent numerical factorization.
—A factorize phase that uses the assembly tree to factorize the matrix and (optionally) solve systems of equations.
—A solve phase that performs forward substitution followed by back substitution using the stored matrix factors.

The HSL_MA77 package has separate routines for each of these phases; this is discussed further in Section 3.4.

3.2 Language

HSL_MA77 is written in Fortran 95. 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) [ISO/IEC 2001] 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.

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 its row and column indices.

Fortran 95 also offers recursion and initially we used it in HSL_MA77 since it provides a convenient and efficient way to visit the nodes of the assembly tree. The disadvantage is that each recursion requires temporary memory and a deep tree involves a large number of recursions (one for each tree level). If there is insufficient memory to support this, we obtain a "segmentation fault," from which no recovery is possible. Our nonrecursive implemention requires two additional integer arrays of size the tree depth, which are allocated within HSL_MA77 before the factorization is performed.

3.3 The Files Used by HSL_MA77

HSL_MA77 holds the frontal matrix in main memory, but allows the matrix factor and the multifrontal stack, as well as the original matrix data, to be held out-of-core, in direct-access files. In this section, we discuss the files that are used by HSL_MA77. It accesses these through the package HSL_OF01 [Reid and Scott 2009a]), which is briefly described in Section 4 and includes the facility of grouping a set of files into a *superfile* that is treated as an entity.

We use three superfiles: one holds integer information, one holds real information, and one provides real workspace. We refer to these as the *main integer*, *main real*, and *main work* superfiles, respectively.

The main real superfile holds the reals of the original rows or elements of A followed by the columns of the factor L, which are in the order that they were calculated. The main integer superfile is used to hold corresponding integer information. If input is by rows, for each row we store the list of indices of the variables that correspond to the nonzero entries. If input is by elements, for each element we store the list of indices of its variables.

Duplicated and/or out-of-range entries are allowed in a row or element index list. We flag this case and store 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 analyze 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 analyze phase, these lists are rewritten in the elimination order that this phase has chosen. This facilitates the merging of generated elements during factorization (see Section 7). Note that the variables at the start of the list are those that are eliminated at the node. If input is by elements, we also rewrite the lists for the elements in the new order, but add the mapping from the user's order. This allows the user to provide the reals for each element without performing a reordering; instead `HSL_MA77` reorders the element so that when it is later merged with other elements and with generated elements it does not have to be treated specially.

The principal role of the main workspace superfile is to hold the stack of intermediate results that are generated during the depth-first search. As the computation proceeds, the space required to hold the factors always grows but the space required to hold the stack varies.

3.4 The User Interface

The following procedures are available to the user:

—`MA77_open` must be called once for a problem to initialize the data structures and open the superfiles.

—`MA77_input_vars` must be called once for each element or row to specify the variables associated with it. The index lists are written to the main integer superfile.

—`MA77_analyse` must be called after all calls to `MA77_input_vars` are complete. A pivot order must be supplied by the user. `MA77_analyse` constructs the assembly tree. The index lists for each node of the tree are written to the main integer superfile.

—`MA77_input_reals` must be called for each element or row to specify the entries. The index list must have already been specified by a call of `MA77_input_vars`. For element entry, the lower triangular part of the element matrix must be input by columns in packed form. For row entry, the user must input all the nonzeros in the row (upper and lower triangular entries). 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 or row that has already been entered, the original data is overwritten.

—`MA77_factor`: may be called after all the calls to `MA77_input_reals` are complete and after the call to `MA77_analyse`. The matrix A is factorized using the assembly tree constructed by `MA77_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 MA77_input_reals have specified changed real values.

—MA77_factor_solve: may be called in place of MA77_factor if the user wishes to solve the system $AX = B$ at the same time as the matrix A is factorized.

—MA77_solve uses the computed factors generated by MA77_factor for solving the system $AX = B$. Options exist to perform only forward substitution or only back substitution.

—MA77_resid computes the residual $R = B - AX$.

—MA77_finalise should be called after all other calls are complete for a problem. It deallocates the components of the derived data types and closes the superfiles associated with the problem. It has an option for storing all the in-core data for the problem to allow the calculation to be restarted later.

—MA77_restart restarts the calculation. Its main use is to solve further systems using a calculated factorization, but it also allows the reuse of analysis data for factorizing a matrix of the same structure but different real values.

—MA77_enquire_posdef may be called after a successful factorization to obtain the pivots used.

Derived types are used to pass data between the different routines within the package. 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 for detailed control over printing, virtual memory management (Section 4), node amalgamation (Section 6.3), and the block size for full-matrix operations on the frontal matrix (Section 5). The information available to the user includes a flag to indicate error conditions, the determinant (its sign and the logarithm of its absolute value), the maximum front size, the number of entries in the factor L, and the number of floating-point operations.

Full details of the user interface and the derived types are provided in the user documentation.

4. VIRTUAL MEMORY MANAGEMENT

A key part of the design of HSL_MA77 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 [Reid and Scott 2009a]. Handling input/output through a separate package was actually part of the out-of-core solver of Reid [1984] and our approach is a refinement of that used by the earlier code.

Fortran 95 offers two forms of file access: sequential and direct. We have chosen not to use sequential access because the data of the original matrix needs to be accessed nonsequentially and other data has to be accessed backwards as well as forwards and our experience has been that backwards access is slow. We use direct access, but it has the disadvantage that each file has fixed-length records. We need to be able to read and write different amounts of data at each

stage of the computation and thus, to enable the use of direct-access files, we need to buffer the data. This is done for us by HSL_OF01. Fortran 2003 offers a third form of file access: stream. At the time of writing, no compilers fully support Fortran 2003, but the Nag compiler supports stream access, so we have tried this but found that the factorization time is always increased and, in one example, we observed an increase of 55%. As a result, we do not currently plan to use stream access. More details and numerical results are given by Reid and Scott [2009a].

4.1 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. 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_MA77 to enable the available memory to be dynamically shared between the main real and main work superfiles 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 reported in Reid and Scott [2009a] were used to choose default settings for HSL_MA77.

The data in a file are addressed as a virtual array of rank one. Because it may be very large, long integers (64 bits) are used to address it. 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.

A file is often limited in size to less that 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 referred to as a *superfile*. The files of a superfile may reside of different devices.

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 factors 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 multifrontal stack since it is known that it will not be needed again. Further details of these options are given by Reid and Scott [2009a].

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 and the call would behave as if the statements

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

were executed. We use this option in HSL_MA77 for the efficient assembly of elements into the frontal matrix.

4.2 Option for In-Core Working Within HSL_MA77

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_MA77, this is particularly serious during the solve phase for a single right-hand side since each datum read during the forward substitution or back substitution is used only once. We have therefore included within HSL_MA77 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. Note that, because it is desirable to keep the multifrontal stack in memory, HSL_MA77 first switches the main integer data to a file, then the main real data, and only finally switches the stack to a file if there is still insufficient memory. To ensure the automatic switch can be made, we always require path and superfile names to be provided on the call of MA77_open. If a user specifies the total size of the arrays without specifying the initial sizes of the individual arrays, the code automatically choose 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 (i.e., using arrays in place of superfiles). On successful completion of the factorization, HSL_MA77 returns the number of integers and reals stored for the matrix and its factor, and the

maximum size of the multifrontal stack. 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.

5. KERNEL CODE FOR HANDLING FULL-MATRIX OPERATIONS

For the real operations within the frontal matrix and the corresponding forward and back substitution operations, we rely on a modification of the work of Andersen et al. [2005] for Cholesky factorization of a positive-definite full symmetric matrix. They pack the upper or lower triangular part of the matrix into a "block hybrid" format that is as economical of storage as packing by columns but is able to take advantage of Level-3 BLAS [Dongarra et al. 1990]. It divides the matrix into blocks, all of which are square and of the same size nb (except for the blocks at the bottom which may have fewer rows). Each block is ordered by rows and the blocks are ordered by block columns.

The factorization is programmed as a sequence of block steps, each of which involves the factorization of a block on the diagonal, the solution of a triangular set of equations with a block as its right-hand side, or the multiplication of two blocks. Andersen et al. [2005] have written a special kernel for the factorization of a block on the diagonal that uses blocks of size 2 to reduce traffic to the registers. The Level-3 BLAS `DTRSM` and `DGEMM` are available for the other two steps. If the memory needed for a block is comparable with the size of the cache, execution of each of these tasks should be fast. Andersen et al. [2005] reported good speeds on a variety of processors.

We have chosen to work with the lower triangular part of the matrix because this makes it is easy to separate the pivoted columns that hold part of the factor from the other columns that hold the generated element. The modification we need for the multifrontal method involves limiting the eliminations to the fully summed columns, the first p, say. The partial factorization (see Section 2.3) takes the form

$$F = \begin{pmatrix} F_{11} & F_{21}^T \\ F_{21} & F_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & \\ L_{21} & I \end{pmatrix} \begin{pmatrix} I & \\ & S_{22} \end{pmatrix} \begin{pmatrix} L_{11}^T & L_{21}^T \\ & I \end{pmatrix}, \tag{5.1}$$

where L_{11} is lower triangular and both F_{11} and L_{11} have order p. We use the lower blocked hybrid format for the lower triangular part of both F_{11} and F_{22}. The rectangular matrix F_{21} is held as a block matrix with matching block sizes. During factorization, these matrices are overwritten by the lower triangular parts of L_{11} and S_{22} and by L_{21}. The modified code is collected into the module `HSL_MA54`.

`HSL_MA77` retains the matrix $\begin{pmatrix} L_{11} & \\ L_{21} & I \end{pmatrix}$ in block hybrid format for forward and back substitution since this is efficient, but it reorders the matrix S_{22} back to lower packed format for the assembly operations at the parent node since the block structure at the parent is very unlikely to be the same.

If p is small, there is insufficient gain from the use of Level-3 BLAS to compensate for rearranging F_{22} to block hybrid form and S_{22} back to lower packed format. We have found it better in this case to rearrange only F_{11} and F_{21} and rely on Level-2 BLAS for updating the columns of F_{22} one by one. On our test platform, this was better for p less than about 30.

An alternative would be to apply BLAS and LAPACK subroutines directly to the blocks of factorization (5.1). For efficiency, it would be necessary to hold both F_{11} and F_{22} in full (unpacked) storage, so much more memory would be needed. We simulated the effect of this by running `HSL_MA77` with nb equal to size largest front size and with the call of its kernel subroutine for Cholesky factorization of F_{11} replaced by calls of the LAPACK subroutine `DPOTRF`. Some times are given in Table II (see Section 9.1), which show that on our platform `HSL_MA54` offers a modest speed advantage in addition to the substantial storage advantage.

For solving systems once the matrix has been factorized, there is an advantage in keeping the computed factor in block hybrid form. For a single right-hand side, `HSL_MA54` makes a sequence of calls of the Level-2 BLAS `DTPSV` and `DGEMV` each with matrices that are matched to the cache. For many right-hand sides, `HSL_MA54` makes a sequence of calls of the Level-3 BLAS `DTRSM` and `DGEMM`.

6. DATA INPUT AND THE ANALYSIS PHASE

6.1 Supervariables

It is well known that working with supervariables (groups of variables that belong to the same set of elements in the element-entry case or are involved in the same set of rows in the row-entry case) leads to significantly faster execution of the analyze phase. As was explained in Section 2.5 of Duff and Reid [1996], they can be identified with an amount of work that is proportional to the total length of all the lists. This is done by first putting all the variables into a single supervariable. This is split into two supervariables by moving those variables that are in the first list into a new supervariable. Continuing, we split into two each of the supervariables containing a variable of the ith list by moving the variables of the list that are in the supervariable into a new supervariable. The whole algorithm is illustrated in Figure 2. We have implemented this with four arrays of length n. For efficiency, this work is performed during the calls of `MA77_input_vars`.

In an early version of the code, we merged supervariables that became identical following eliminations, but found that the overheads involved were much greater than the savings made.

We need to interpret the pivot sequence that the user provides to `MA77_analyse` as a supervariable ordering. We expect all the variables of a supervariable to be adjacent, but in case they are not, we treat the supervariables as being ordered according to their first variables in the pivot sequence. This is justified by the fact that, after pivoting on one variable of a supervariable, no further fill is caused by pivoting on the others.

```
Put all the variables in one supervariable
do for each list
    do for each variable v in the list
        sv = the supervariable to which v belongs
        if this is the first occurrence in the list of sv then
            establish a new supervariable nsv
            record that nsv is associated with sv
        else
            nsv = the new supervariable associated with sv
        end if
        move v to nsv
    end do
end do
```

Fig. 2. Identifying supervariables.

We note that because the supervariables are found during calls to MA77_input_vars, we do not allow the user to change any of the index lists without first calling MA77_finalise to terminate the computation and then restarting by calling MA77_open.

6.2 Constructing the Tree

A key strategy for the speed of MA77_analyse is that we label each element and generated element according to which of its supervariables occurs earliest in the pivot sequence. Linking the elements and generated elements with the same label allows us to identify at each pivotal step exactly which elements are involved without a search. In the element-entry case, the first action is to read all the index lists and establish this linked list. We use an integer array of size the number of supervariables for the leading entries and an integer array of size the largest possible number of elements and generated elements for the links.

The tree is stored by holding, for each non-leaf node, a structure that contains an integer allocatable array for the indices of the children. It is convenient also to hold here the number of eliminations performed at the node. We use the derived type MA77_node for this purpose and allocate at the start of MA77_analyse an array of this type. The number of non-leaf nodes is bounded by the number of supervariables since at least one supervariable is eliminated at each node. In the element-entry case, it is also bounded by twice the number of elements since each original element could be an only child if static condensation occurs there and thereafter every node has at least two children. In this case, we use the lesser bound for the size for the array.

In the row-entry case, the leaf nodes represent elements of order 1 or 2 (see Section 2.2). Assembly of an element of order 1 can be delayed until its variable is eliminated and assembly of an element of order 2 can be delayed until one of its variables is eliminated. Therefore, the list of variables eliminated at a node can be used to indicate which leaf nodes are needed and there is no need to include them explicitly in the list of children.

For each variable in the given pivot sequence, we first check if its supervariable has already been eliminated. If it has, no action is needed.

Otherwise, we add a new node to the tree and construct its array of child indices from the linked list of the elements and generated elements for the supervariable. Next, we construct the index list for the new node by merging the index lists of the children. In the row-entry case, we also read the index list for the variable from the main integer superfile and merge it in, excluding any variables that have already been eliminated.

In the element-entry case, we identify other variables that can be eliminated at this time by keeping track of the number of elements and generated elements that each variable touches. Any variable that is involved in no elements may be eliminated. Unfortunately, this cannot be applied in the row-entry case without reading the row list from the main integer superfile and checking that all its variables are already in the list. Instead, we rely on the new node being amalgamated with its parent (see next subsection) when it is later considered as a child. We tried relying on this in the element-entry case but found that the analyse speed was increased by a factor of about three and the quality of the factorization was slightly reduced.

6.3 Node Amalgamation

Next, we check each of the child nodes to see if it can be amalgamated with the new parent. We do this if the list of uneliminated variables at the child is the same as the list of variables at the parent, since in that case the amalgamation involves no additional operations. We also do it if both involve less than a given number of eliminations, which the user may specify. By default, the number is 8 (see our experimental results in Tables III and IV). The rationale for this amalgamation is that it is uneconomic to handle small nodes. Note that these tests do not need to be applied recursively since if a child fails the test for amalgamation with its parent, it will also fail if it is retested after a sibling has been amalgamated with its parent or its parent has been amalgamated with its grandparent.

The strategy used by the HSL code `MA57` [Duff 2004], which is essentially the same as that used by the earlier HSL code `MA27` [Duff and Reid 1983], causes fewer node amalgamations since it applies the first test (no additional operations) only if there is just one child and applies the second test only with the child that is visited last in the depth-first search. The difference in the number of nodes in the tree will be illustrated in Table III.

Suppose node i has a child c_i with k children. If c_i is amalgamated with its parent, the children of c_i become children of node i. Thus if $k > 1$, the number of children of node i increases. For this reason, we use a temporary array to hold the children and delay allocation of the actual list until after all the children have been tested and the length is known.

Amalgamating c_i with i means that c_i is no longer needed. We therefore deallocate the array for its children and make the node available for reuse. We keep a linked list of all such available nodes and always look there first when creating a new node.

We now choose the assembly order for each set of children (see next subsection) and finally perform a depth-first search to determine the new pivot

order. We record this and use it to sort all the index lists of the generated elements, to ease element merging in `MA77_factor`.

6.4 The Assembly Order of Child Nodes

At each node of the assembly tree, all the children must be processed and their generated elements computed before the frontal matrix at the parent can be factorized. However, the children can be processed in any order and this can significantly affect the maximum size of the stack, which in turn is likely to affect the amount of input/output.

The simplest strategy (which is sometimes referred to as the *classical multifrontal approach*) is to wait until all the children of a node have been processed and then allocate the frontal matrix and assemble all the generated elements from its children into it. If node i has children c_j, $j = 1, 2, ..., n_i$ and the size of the generated element at node k is g_k, the stack storage needed to generate the element at node i is

$$s_i = \max_{j=1,n_i} \left(\sum_{k=1}^{j-1} g_{c_k} + s_{c_j} \right) = \max_{j=1,n_i} \left(\sum_{k=1}^{j} g_{c_k} + s_{c_j} - g_{c_j} \right).$$

This is minimized if the children are ordered so that $s_{c_j} - g_{c_j}$ decreases monotonically [Liu 1986].

The main disadvantage of the classical approach is that it requires the generated element from each child to be stacked. For very wide trees, a node may have many children so that $\sum_{k=1}^{n_i} g_{c_k}$ is large. The classical approach is also poor if the index lists have significant overlaps. Thus Liu [1986] also considered choosing the first child to be the one that requires the most memory and allocating the frontal matrix after it has been processed. The generated element from each child can then be assembled directly into the frontal matrix for the parent, which avoids the potentially large number of stacked generated elements. However, numerical experiments have shown that this can also perform poorly because a chain of frontal matrices (at the active tree levels) must be stored. This led Guermouche and L'Excellent [2006] to propose computing, for each node, the optimal point at which to allocate the frontal matrix and start the assembly.

Suppose the frontal matrix for node i has size f_i. If it is allocated and the assembly started after p_i children have been processed, Guermouche and L'Excellent [2006] showed that the total storage (including the active fronts) needed to process node i is

$$t_i = \max \left(\max_{j=1,p_i} \left(\sum_{k=1}^{j-1} g_{c_k} + t_{c_j} \right), f_i + \sum_{k=1}^{p_i} g_{c_k}, f_i + \max_{j>p_i} t_{c_j} \right). \qquad (6.1)$$

Their algorithm for finding the *split point*, that is, the p_i that gives the smallest t_i, then proceeds as follows: for each p_i ($1 \le p_i \le n_i$), order the children in decreasing order of t_{c_j}, then reorder the first p_i children in decreasing order of $t_{c_j} - g_{c_j}$. Finally, compute the resulting t_i and take the split point to be the p_i that gives the smallest t_i. Guermouche and L'Excellent [2006] proved they obtain the optimal t_i.

```
forall (i in the set of root nodes)
    call order_children(i)
end forall
subroutine order_children(i)
do
    if (i has an unvisited non-leaf child node) then
        i = next_non-leaf_child(i)
    else
        using the values of s_{v_j} for the children of i in (6.3), compute v_i and p_i
        if (i == root) exit
        i = parent (i)
    end if
end do
end subroutine order_children
```

Fig. 3. Depth-first tree search to order the children of each node.

Duff and Reid [1983] suggested that the generated element of the final child be expanded in-place to the storage of its parent, following which the generated elements of the other children are merged in. Guermouche and L'Excellent [2004] suggested that this be done at the split point, which reduces the total storage needed to process node i to

$$u_i = \max\left(\max_{j=1,p_i}\left(\sum_{k=1}^{j-1} g_{c_k} + u_{c_j}\right),\ f_i + \sum_{k=1}^{p_i-1} g_{c_k},\ f_i + \max_{j>p_i} u_{c_j}\right). \tag{6.2}$$

In our code, we use the same array to hold the front at each node of the tree. We use the array only for the front that is being assembled or factorized. The other fronts are stored temporarily on the stack. The formula for the stack memory at node i becomes

$$v_i = \begin{cases} \max\left(\max_{j=1,p_i}\left(\sum_{k=1}^{j-1} g_{c_k} + v_{c_j}\right),\ f_i + \max_{j>p_i} v_{c_j}\right), & p_i < n_i, \\ \max_{j=1,p_i}\left(\sum_{k=1}^{j-1} g_{c_k} + v_{c_j}\right), & p_i = n_i, \end{cases} \tag{6.3}$$

which is very similar to the formula (6.2) for u_i. Because we work with a separate array for the active frontal matrix, we have nothing comparable to the second term of (6.2). The third term of (6.2) should really be omitted if splitting is not advantageous at node i ($p_i = n_i$), but it can remain since it is less than the second term in this case. The change from (6.1) to (6.2) or (6.3) does not invalidate the algorithm of Guermouche and L'Excellent [2006] for finding the split point that minimizes the stack memory.

The code to order the children and find the split point is outlined in Figure 3.

This algorithm is implemented within `MA77_analyse`. When computing a split point, we ignore any children that are leaf nodes; any such children are ordered after the non-leaf children. This choice was made since the leaf nodes can be assembled directly into the front without going into the stack. The sorting is performed using the HSL heap-sort package `HSL_KB22`.

```
allocate the array F big enough for the largest front
do for each root node i
    call factorize(i)
end do
subroutine factorize(i)
do
    if (i has an unvisited non-leaf child node) then
        i = next_non-leaf_child(i) ! in the order determined by the analyse phase
    else
        if (solving an element problem) then
            assemble the original elements for all the leaf children into F
        else
            assemble into F the original entries in the columns to be eliminated
        end if
        perform partial factorization of F using HSL_MA54
        store the computed columns of L, leaving the generated element in F
        if (i is ahead of the split point for its siblings) then
            put the generated element in F onto the top of the stack
        else if (i is the split point for its siblings) then
            expand the generated element in F to the front of parent(i)
            assemble the generated elements for children i − 1 to 1 from the top of
                the stack into F, popping the stack
            if (i is not the final non-leaf sibling) then
                put the frontal matrix in F onto the top of the stack
            end if
        else
            if (i is not the final non-leaf sibling) then
                assemble the generated element in F into the stacked frontal matrix
            else
                expand the generated element in F to the front for parent (i)
                add the stacked frontal matrix into F
                pop the stack
            end if
        end if
        if (i == root) exit
        i = parent(i)
    end if
end do
end subroutine factorize
```

Fig. 4. The factorization phase of the multifrontal algorithm implemented within `MA77_factor`.

7. THE FACTORIZATION PHASE

In Figure 4 we outline how the factorization phase of `HSL_MA77` is performed by a depth-first search of the assembly tree using the ordering of the children that was determined during the analyze phase. The assembly steps are performed column by column to avoid the need to hold two frontal matrices in memory at the same time. This also means that when the generated element in `F` is assembled into the stacked frontal matrix, only those columns that correspond to columns of `F` are accessed.

8. THE SOLVE PHASE

An outline of the solve phase of HSL_MA77 is given in Figure 5. HSL_MA77 requires the right-hand side B to be held in full format. The simplifies the coding of the operations involving the right-hand side and avoids any actual input/output for it. To save memory, the user must supply B in an array X which is overwritten by the solution X.

MA77_solve performs forward substitution followed by back substitution unless only one of these is requested. The matrix factor must be accessed once for the forward substitution and once for the back substitution. If MA77_solve is called several times with the same factorization but different right-hand sides, HSL_OF01 will avoid actual input/output at the start of the forward substitution since the most recently data will still be in the buffer following the previous back substitution. In all cases, the amount of actual input/output is independent of the number of right-hand sides and so it is more efficient to solve for several right-hand sides at once rather than making repeated calls (this is illustrated in Table VII).

If the user calls MA77_factor_solve, the forward substitutions operations are performed as the factor entries are generated. Once the factorization is complete, the back substitutions are performed. This involves reading the factors only once from disk and so is faster than making separate calls to MA77_factor and MA77_solve (see Table VII).

MA77_solve includes options for performing partial solutions. The computed Cholesky factorization (2.1) may be expressed in the form

$$A = (PLP^T)(PL^T P^T). \tag{8.1}$$

MA77_solve may be used to solve one or more of the following systems:

$$AX = B, \quad (PLP^T)X = B, \quad (PL^T P^T)X = B. \tag{8.2}$$

Partial solutions are needed, for example, in optimization calculations, see Algorithm 7.3.1 of Conn et al. [2000].

A separate routine MA77_resid is available for computing the residuals $R = B - AX$. If out-of-core storage has been used, computing the residuals involves reading the matrix data from disk and so involves an input/output overhead. MA77_resid offers an option that, in the row-entry case, computes the infinity norm of A. In the element case, an upper bound on the infinity norm is computed (it is an upper bound because no account is taken of overlaps between elements).

9. NUMERICAL EXPERIMENTS

In this section, we illustrate the performance of HSL_MA77 on large positive-definite problems. Comparisons are made with the HSL sparse direct solver MA57. 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. It has Fujitsu MAT3147NP hard disks with sustained throughput of 132 MBs and average seek time of 4.5 ms for read and 5.0 ms for write. The

```
subroutine solve(tree, X)
    do for each root node i
        call forward(i)
        call back(i)
    end do
contains
subroutine forward(i)
do
    if (i has an unvisited non-leaf child node) then
        i = next_non-leaf_child(i) ! in the order determined by the analyse phase
    else
        read columns of L stored at node i
        copy components of X that are involved into a temporary full array
        perform partial forward substitution using HSL_MA54
        copy temporary full array back into appropriate components of X
        if (i == root) exit
        i = parent(i)
    end if
end do
end subroutine forward
subroutine back(i)
do
    read columns of L stored at node i
    copy components of X that are involved into a temporary full array
    perform partial back substitution using HSL_MA54
    copy temporary full array back into appropriate components of X
    if (i has an unvisited non-leaf child node) then
        i = next_non-leaf_child(i) ! in the reverse order to that determined
                                   ! by the analyse phase
    else
        if (i == root) exit
        i = parent(i)
    end if
end do
end subroutine back
end subroutine solve
```

Fig. 5. The solve phase of the multifrontal algorithm implemented within `MA77_solve`.

Nag f95 compiler with the -O3 option was used and we used ATLAS BLAS and LAPACK.[1]

The test problems used in our experiments are listed in Table I. Here $nz(A)$ denotes the millions of entries in the lower triangular part of the matrix (including the diagonal). An asterisk denotes that only the sparsity pattern is available. Most of the problems (including those from finite-element applications) are stored in assembled form; those held in element form are marked with a dagger and for these problems we use the element entry to `HSL_MA77`.

[1]math-atlas.sourceforge.net.

Table I. Positive Definite Test Matrices and Their Characteristics

Identifier	n	$nz(A)$	$nz(L)$	f_{max}	Application/description
1. thread	29.7	2.2	23.7	3.0	Threaded connector/contact
2. pkustk11*	87.8	2.7	28.5	2.0	Civil engineering. Cofferdam
3. pkustk13*	94.9	3.4	30.6	2.1	Machine element, 21 noded solid
4. crankseg_1	52.8	5.3	33.7	2.1	Linear static analysis
5. m_t1	97.6	4.9	34.6	1.9	Tubular joint
6. shipsec8	114.9	3.4	37.2	2.7	Ship section
7. gearbox*	153.7	4.6	39.3	2.2	Aircraft flap actuator
8. shipsec1	140.9	4.0	40.4	2.5	Ship section
9. nd6k	18.0	6.9	40.7	4.4	3D mesh problem
10. cfd2	123.4	1.6	40.9	2.5	CFD pressure matrix
11. crankseg_2	63.8	7.1	43.2	2.2	Linear static analysis
12. pwtk	217.9	5.9	50.4	1.1	Pressurized wind tunnel
13. shipsec5	179.9	5.1	55.0	3.2	Ship section
14. fcondp2*	201.8	5.7	55.2	3.3	Oil production platform
15. ship_003	121.7	4.1	62.2	3.3	Ship structure—production
16. thermal2	1228.0	4.9	63.0	1.4	Unstructured thermal FEM
17. troll*	213.5	6.1	63.7	2.6	Structural analysis
18. halfb*	224.6	6.3	66.2	3.3	Half-breadth barge
19. bmwcra_1	148.8	5.4	71.2	2.2	Automotive crankshaft model
20. fullb*	199.2	6.0	75.0	3.5	Full-breadth barge
21. af_shell3	504.9	17.6	97.7	2.2	Sheet metal forming matrix
22. pkustk14*	151.9	7.5	108.9	3.1	Civil engineering. Tall building
23. g3_circuit	1585.5	4.6	118.5	2.9	Circuit simulation
24. nd12k	36.0	14.2	118.5	7.7	3D mesh problem
25. ldoor	952.2	23.7	154.7	2.4	Large door
26. inline_1	503.7	18.7	179.3	3.3	Inline skater
27. bones10	914.9	28.2	287.6	4.7	Bone microfinite element model
28. nd24k	72.0	28.7	321.3	11.4	3D mesh problem
29. bone010	986.7	36.3	1089.1	10.7	Bone micro-finite element model
30. audikw_1	943.7	39.3	1264.9	11.2	Automotive crankshaft model

(n denotes the order of A in thousands; $nz(A)$ and $nz(L)$ are the number of entries in A and L, respectively, in millions; f_{max} is the maximum order of a frontal matrix in thousands. * indicates pattern only. † indicates stored in element form.)

Each test example arises from a practical application and are all available from the University of Florida Sparse Matrix Collection [Davis 2007]. We note that our test set comprises a subset of those used in the study of sparse direct solvers by Gould et al. [2007] together with a number of recent additions to the University of Florida Collection. As HSL_MA77 is specifically designed for solving large-scale problems, the subset was chosen by selecting only those problems that MA57 either failed to solve because of insufficient memory or took more than 20 seconds of CPU time to analyze, factorize and solve on our Dell computer.

For those matrices that are only available as a sparsity pattern, reproducible pseudorandom off-diagonal entries in the range $(0, 1)$ were generated using the HSL package FA14, while the ith diagonal entry, $1 \leq i \leq n$, was set to $\max(100, 10\rho_i)$, where ρ_i is the number of off-diagonal entries in row i of the matrix, thus ensuring that the generated matrix is positive definite. The right-hand side for each problem was generated so that the required solution is the vector of ones.

Table II. Comparison of the In-Core Factorization Phase Times Using HSL_MA54 with Different Blocksizes (nb) and DPOTRF

	HSL_MA54				
	nb =50	100	150	200	DPOTRF
1. thread	20.6	15.5	13.6	15.5	14.4
4. crankseg_1	22.0	17.2	16.1	17.7	17.2
12. pwtk	18.0	15.5	15.0	15.9	15.7
19. bmwcra_1	41.5	33.1	30.9	33.8	33.2
21. af_shell3	39.1	33.1	30.9	33.8	32.7

Unless stated otherwise, all control parameters were used with their default settings in our experiments. In particular, the size of each page (and file record) was 4096 scalars (reals in the real buffer or integers in the integer buffer) and the number of pages in the in-core buffers was 1600, which means that the real and integer buffers had sizes about 52 Mb and 26 Mb; the file size was 2^{21} scalars. These settings were chosen on the basis of the numerical experiments reported by Reid and Scott [2009a].

The analyze phase of the MA57 package was used to compute the pivot sequences for HSL_MA77. MA57 automatically chooses between an approximate minimum degree and a nested dissection ordering; in fact, for all our test problems, it selected a nested dissection ordering that is computed using METIS_NodeND. In Table I, we include the number of millions of entries in the matrix factor (denoted by $nz(L)$) and the maximum order of a frontal matrix (denoted by *front*) when this pivot order was used by HSL_MA77.

Throughout this section, the *complete solution time* for HSL_MA77 refers to the total time for inputting the matrix data, computing the pivot sequence, and calling the analyze, factorize and solve phases. The complete solution time for MA57 is the total time for calling the analyze, factorize and solve phases of MA57. Where appropriate, timings for HSL_MA77 include all input/output costs involved in holding the data in superfiles. With the exception of Table VII, all reported times are wall clock (elapsed) times in seconds.

9.1 Choice of Block Size and HSL_MA54 Versus LAPACK

Factorization timings for a subset of our test problems using a range of block sizes in the kernel code HSL_MA54 (Version 1.0.0) are presented in Table II. The results show that, on our test computer, nb =150 is a good choice; this is the default block size used within HSL_MA77. As noted in Section 5, we can simulate the effect of using LAPACK instead of HSL_MA54 by running HSL_MA77 with the block size for the blocked hybrid form set to the largest front size and using the LAPACK subroutine DPOTRF for the Cholesky factorization of the blocks on the diagonal. The main advantage of using HSL_MA54 in place of DPOTRF is the substantial storage savings; the results in Table II illustrate that, in our test environment, the speed improvements are modest.

9.2 The Effects of Node Amalgamation

Our strategy of amalgamating nodes of the tree was explained in Section 6.3. We amalgamate a child with its parent if both involve less than a given number,

Table III. 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										
nemin	16	1	4	8	16	32	1	4	8	16	32
12. pwtk	11.0	20.6	19.5	11.5	8.5	4.2	49	49	50	52	57
15. ship_003	6.2	11.8	11.8	6.8	4.8	2.5	61	61	62	64	70
19. bmwcra_1	6.4	16.7	10.0	6.7	3.7	2.4	69	70	71	74	77
24. nd12k	0.8	4.1	1.7	1.1	0.7	0.5	118	118	118	119	120

Table IV. Comparison of the In-Core Factorization and Solve Phase Times (Single Right-Hand Side) for Different Values of the Node Amalgamation Parameter nemin

	Factorization times					Solve times				
nemin	1	4	8	16	32	1	4	8	16	32
12. pktk	15.3	14.9	14.8	15.4	18.2	0.44	0.41	0.38	0.37	0.37
15. ship_003	39.0	38.6	38.3	39.8	40.5	0.44	0.44	0.41	0.41	0.41
19. bmwcra_1	33.2	30.9	31.1	35.2	33.4	0.80	0.56	0.48	0.48	0.47
24. nd12k	280	267	275	266	263	1.43	0.94	0.87	0.79	0.81

nemin, of eliminations. We show in Tables III and IV, a few of our results on the effect of varying nemin. We also show in Table III the number of nodes with eliminations that MA57 finds from the same pivot sequence with its default nemin value of 16. We note that it does far less amalgamations because of its restricted choice.

We have found that, provided nemin $>$ 1, the performance of our test problems is usually not very sensitive to the exact choice of nemin, probably because most of the work is performed in large frontal matrices. We have chosen 8 as the default value for nemin, which makes the number of nodes in the tree comparable with that of MA57. We considered 16, but this increases the storage (number of entries in L) and often increases the factorization time.

9.3 Assessing the Impact of the Guermouche-L'Excellent Algorithm in our Context

To assess the effectiveness of the algorithm of Guermouche and L'Excellent [2006] for ordering the processing of the children of a node and choosing the point at which assembly is commenced, we tried the following:

(1) Always set $p_i = 1$, as proposed in Liu [1986].
(2) Always set $p_i = 2$, since this is simple and never worse than setting $p_i = 1$ since the stack needs to hold the first generated element instead of the parent element while processing the second child.
(3) Always set $p_i = n_i$, which is the classical approach.
(4) Automatic choice of p_i using the algorithm of Guermouche and L'Excellent [2006], adapted to our storage; see Section 6.4.

We show in Table V how these four strategies performed for some of our problems. We show the maximum stack size and the number of Fortran records actually read and written, without taking into account whether the system buffers the operation.

Table V. Maximum Stack Size (Millions) and Number of File Records Read and Written for Different Child Ordering Strategies

Problem	Strategy	Maximum stack size	Thousands of records Read	Written	Total
3. pkustk13	$p_i = 1$	5.286	3.191	12.718	15.909
	$p_i = 2$	3.110	0.955	10.508	11.463
	$p_i = n_i$	2.787	0.949	11.087	12.036
	Automatic	2.787	0.949	11.087	12.036
6. shipsec8	$p_i = 1$	9.082	7.743	17.142	24.885
	$p_i = 2$	6.100	3.253	14.430	17.683
	$p_i = n_i$	5.083	1.773	13.784	15.557
	Automatic	5.083	1.773	13.784	15.557
21. af_shell3	$p_i = 1$	6.184	8.518	29.721	38.239
	$p_i = 2$	4.091	6.665	28.216	34.881
	$p_i = n_i$	3.067	5.523	27.712	33.235
	Automatic	3.003	5.792	27.913	33.705
23. g3_circuit	$p_i = 1$	8.944	11.402	37.249	48.651
	$p_i = 2$	5.856	6.980	34.132	41.112
	$p_i = n_i$	4.227	3.633	32.188	35.821
	Automatic	4.223	3.616	32.162	35.778
gupta2	$p_i = 1$	1.420	0.029	3.588	3.617
	$p_i = 2$	1.322	0.025	3.583	3.608
	$p_i = n_i$	77.375	38.162	31.391	69.553
	Automatic	1.303	0.026	3.584	3.610

Table VI. Characteristics of the Additional Test Matrix

Identifier	n	$nz(A)$	$nz(L)$	f_{max}	Application/description
gupta2	62.0	2.2	12.9	1.2	Linear programming

In all cases, as expected, the automatic method minimized the stack size. On our 30 test matrices, the actual I/O was usually equally good for the classical and automatic methods, but each was occasionally better than the other. In Table VI, we also show a case, which is not one of our 30 test problems, where the classical method is drastically worse. This seems a sufficient reason to reject the classical method. The choice $p = 1$ is clearly inferior, so we reject this too.

The choice $p = 2$ is obviously worse than the automatic method from the point of view of stack size and is usually worse from the I/O point of view. Our conclusion is to use the automatic method within HSL_MA77.

9.4 Times for Each Phase

In Section 3.4, we discussed the different phases of the HSL_MA77 package. In Table VII, we report the times for each phase for our eight largest test problems (note that the largest six problems cannot be solved in-core and so we only report times for running out-of-core). The input time is the time taken by the calls to MA77_input_vars and MA77_input_reals, and the ordering time is the time for MA57AD to compute the pivot sequence. MA77_factor(0) and MA77_factor(1) are the times for MA77_factor when called with no right-hand sides and with a single right-hand side, respectively. Similarly, MA77_solve(k) is the time for MA77_solve when called with k right-hand sides.

Table VII. Times for the Different Phases of HSL_MA77

Problem	23	24	25	26	27	28	29	30
Elapsed time								
Input	2.28	1.19	5.42	3.96	6.23	2.69	7.89	8.91
Ordering	26.8	7.45	8.70	14.4	23.5	17.1	37.2	43.7
MA77_analyse	12.3	9.44	9.23	4.06	11.7	24.9	27.5	26.7
MA77_factor(0)	52.6	263	62.9	84.2	155	1008	1440	1998
MA77_factor(1)	50.3	264	63.8	89.4	178	1064	1615	2390
MA77_solve(1)	6.02	3.55	7.57	5.70	31.9	34.3	313	303
MA77_solve(8)	14.8	6.41	12.4	11.2	43.6	25.3	321	305
MA77_solve(64)	98.8	32.9	70.1	62.5	152	92.2	510	505
CPU time								
MA77_factor(0)	51.0	261	59.0	81.9	146	1002	1227	1807
MA77_solve(1)	5.97	3.54	5.80	5.66	11.2	10.5	56.3	50.7

Table VIII. Real Records Read from and Written to Files (in Thousands) for the Factorization and Solve Phases of HSL_MA77

Problem		23	24	25	26	27	28	29	30
Phase									
MA77_factor(0)	Read	3.6	33.7	29.4	15.9	52.0	101	221	297
	Write	34.9	54.6	45.2	51.2	85.3	143	368	456
	Both	38.5	88.3	74.6	66.1	137	243	589	752
MA77_factor(1)	Read	32.5	62.6	67.2	59.7	122	179	486	606
	Write	35.3	54.6	45.4	51.2	85.3	143	368	456
	Both	67.8	117	112	111	207	322	854	1061
MA77_solve(1)	Read	56.2	57.9	74.0	85.9	140	157	532	618

As expected, because of the better use of high-level BLAS and because the amount of data to be read from 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.

Unfortunately, we found that the elapsed times can be very dependent on the other activity on our machine, as may be seen by the MA77_factor(0) time being greater than the MA77_factor(1) time for problem 23, and the solve time for a single right-hand side exceeding that for eight right-hand sides for problem 28. In Table VII, we also report CPU times for the factorize and solve phases. For problems 23 to 26, there is a relatively small difference between the CPU and elapsed times. But for the largest problems, the solve CPU time is much less than the elapsed time, providing an indication of the additional cost of reading the factors for these problems from disk. Another way to judge the performance is to look at the number of records actually read from or written to files using HSL_OF01; see Table VIII. By comparing the sum of the number of real records read and written for MA77_factor(0) and MA77_solve(1) with the number read and written for MA77_factor(1), we see that there are significant I/O savings if the solve is performed at the same time as the factorization.

We can also assess the overall performance using megaflop rates. On our test machine, a typical speed for DGEMM multiplying two square matrices of

Table IX. Mflop Rates for the Different Phases of HSL_MA77

Problem / Phase	23	24	25	26	27	28	29	30
MA77_factor(0)	1123	1992	1270	1769	1825	2048	2691	2635
MA77_factor(1)	1172	1984	1252	1666	1594	1941	2400	2458
MA77_solve(1)	99	166	102	157	45	47	17	17
MA77_solve(8)	319	736	500	639	263	507	136	136
MA77_solve(64)	382	1148	707	916	604	1114	683	629

Table X. Timings for Problem 30 Using the Default Buffer and a Large Buffer

Buffer	MA77_analyse	MA77_factor(0)	MA77_solve(1)	Complete time
Default	26.7	1998	303	2379
Large	21.5	1838	291	2200

Table XI. Times for the Different Phases of HSL_MA77 on a Machine with Less Memory

Problem / Phase	23	24	25	26	27	28	29	30
Input	39.3	36.8	150	114	193	83.5	218	241
MA77_analyse	28.1	16.4	44.2	26.6	49.1	47.7	95.2	113
MA77_factor(0)	81.4	396	276	231	417	1438	1997	2870
MA77_solve(1)	61.2	45.2	73.6	78.7	137	130	481	561
MA77_solve(8)	65.7	45.8	77.5	82.2	142	131	494	580

order 2000 is about 4250 Mflops. In Table IX, the megaflop rates corresponding to the results in Table VII are presented (note that these are computed using the wall clock times and so are affected by the level of activity on the machine). The low rates for the solve phase indicates that, for a small number of right-hand sides, the cost of reading in the factor data dominates the total cost and this is particularly true for the largest problems because they perform the most input/output (see Table VIII).

In Reid and Scott [2009a], we found that the timings for HSL_MA77 were not very sensitive to the buffer size. The experiments of Reid and Scott [2009a] did not include the largest problems in our test set so it is of interest here to note the effect of the buffer size on the performance of HSL_MA77 for the largest test problem. In Table X, timings are given for problem 30 for the default buffer (1600 pages each of length 2^{12} scalars) and for a large buffer (6400 pages each of length 2^{14} scalars). We see that it is advantageous to use a larger buffer but the saving is relatively modest (in this experiment, about 7% for the complete solution time).

We have also performed experiments on our large test problems on a machine with less memory, namely a Dell Dimension 4600 with a Pentium 4 2.80-GHz processor and 1 Gbyte of RAM; see Table XI. We used the g95 compiler with option -O and the GOTO BLAS.[2] We used the ordering found on the bigger machine and default settings for the buffer size.

[2]www.tacc.utexas.edu/resources/software.

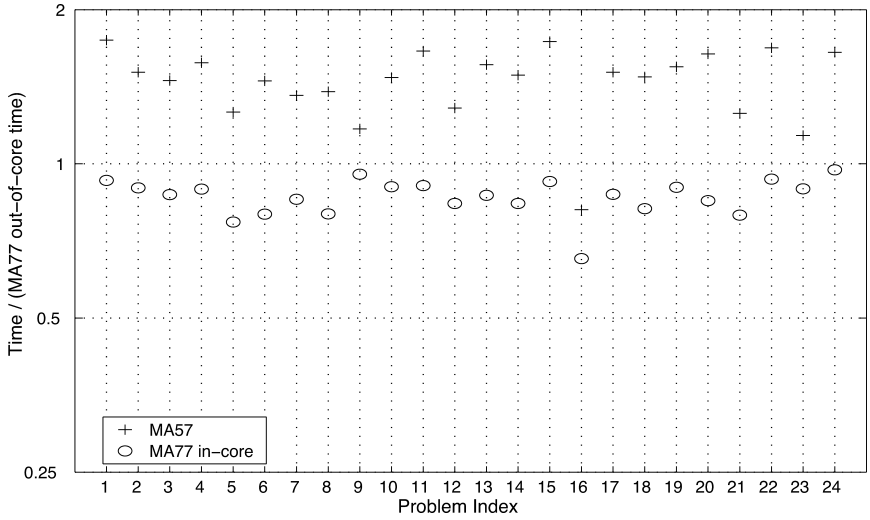

Fig. 6. The ratios of the MA57 and HSL_MA77 in-core factorize times to the HSL_MA77 out-of-core factorize times.

In these tests, the original matrix data is held in a file (for the larger problems, there is insufficent memory to hold the matrix in arrays) and the reported input time is the total time required to read each row of the matrix from this file and then call MA77_input_vars followed by MA77_input_reals. It is clear that reading the matrix from a file adds a significant overhead. Comparing the timings with those reported in Table VII, we see that for problems 23 to 28, on the machine with less memory the solve time accounts for a much greater proportion of the total solution time. Because the solve phase is dominated by input time, the extra cost for multiple right-hand sides is very small.

9.5 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. This is also a multifrontal code. Although primarily designed for indefinite problems, it can be used to solve either positive-definite or indefinite problems in assembled form. It does not offer out-of-core options. We have run Version 3.1.0 of MA57 on our test set. With the exception of the parameter that controls pivoting, the default settings were used for all control parameters. The pivoting control was set so that pivoting was switched off. For the test problems that are supplied in element form, we had to assemble the problem prior to running MA57; we have not included the time to do this within our reported timings.

In Figures 6 to 8, for those of our problems that MA57 can solve on our test computer (problems 1 to 24), we compare the factorize, solve and total solution times for MA57 and for HSL_MA77 in-core (using arrays in place of files) with those for HSL_MA77 out-of-core (using default settings). The figures show the ratios of the MA57 and HSL_MA77 in-core times to the HSL_MA77 out-of-core times.

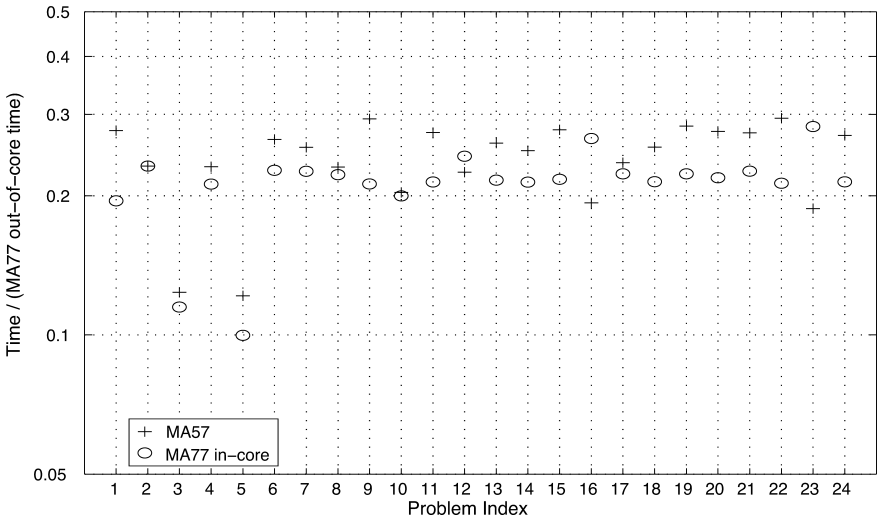

Fig. 7. The ratios of the MA57 and HSL_MA77 in-core solve times to the HSL_MA77 out-of-core solve times (single right-hand side).

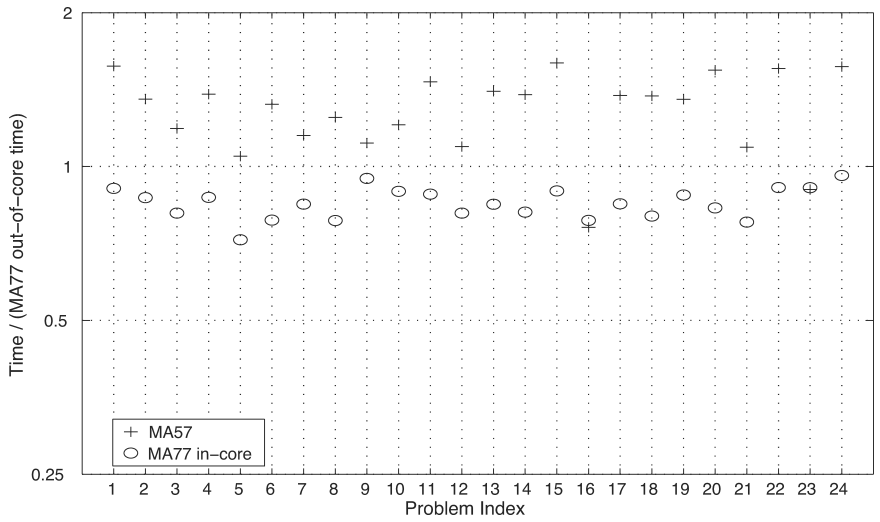

Fig. 8. The ratios of the MA57 and HSL_MA77 in-core complete solution times to the HSL_MA77 out-of-core complete solution times (single right-hand side).

From Figure 6, we see that for the HSL_MA77 factorization times, in-core working on our test machine usually increases the speed by 10 to 25%. For most of our test problems, MA57 factorization is slower than HSL_MA77 out-of-core factorization. If we compare MA57 with HSL_MA77 in-core, we see that the latter is almost always significantly faster. MA57 uses the same code for factorizing the frontal matrix in the positive-definite and indefinite cases, which means that before a column can be used as pivotal, it must be updated for the operations associated with the previous pivots of its block. We believe that this and its fewer node amalgamations are mainly responsible for the slower speed.

For the solution phase with a single right-hand side, the penalty for working out-of-core is much greater because the ratio of data movement to arithmetic operations is significantly higher than for the factorization. This is evident in Figure 7. We see that the HSL_MA77 in-core solve is usually faster than MA57, but for a small number of problems (notably problems 16 and 23), MA57 is the fastest.

The ratios of the total solution times are presented in Figure 8. Here we see that for most of the problems that can be solved without the use of any files, HSL_MA77 in-core is about 15% faster than HSL_MA77 out-of-core and is almost always faster than MA57. We note that MA57 does not offer the option of solving at the same time as the factorization. Our reported complete solution times for both packages are the total time for separate analyze, factorize and solve phases. If we used the option offered by HSL_MA77 for combining the factorize and solve phases, the difference between the out-of-core and in-core times would be slightly reduced and the improvement over MA57 slightly increased.

10. FUTURE DEVELOPMENTS AND CONCLUDING REMARKS

We have described a new out-of-core multifrontal Fortran code for sparse symmetric positive-definite systems of equations and demonstrated its performance using problems from actual applications. The code has a built-in option for in-core working and we have shown that, on our test machine, this usually performs better than MA57 [Duff 2004]. We have paid close attention to the reliable management of the input/output and have addressed the problem of having more data than a single file can hold. An attractive feature of the package is that, if the user requests in-core working but insufficient memory is available, the code automatically switches to out-of-core working, without the need to restart the computation.

The frontal matrix is held in packed storage and we have proposed a variant of the block hybrid format of Andersen et al. [2005] for applying floating-point operations. We have shown that this is economical of storage and efficient in execution.

We have considered the ordering of the children of each node of the assembly tree and the point at which the frontal matrix for the node is established and have found that exploiting the ideas of Guermouche and L'Excellent [2006] gives worthwhile gains.

We have described a new way to amalgamate nodes at which few eliminations are involved and shown that it performs more amalgamations than MA57.

Throughout careful attention has been paid to designing a robust and high-quality software package that is user friendly. We have employed a reverse communication interface, allowing input by rows or by elements. The package has a number of control parameters that the user can use to tune performance for his or her own machine and applications but, to assist less experienced users, default values are supplied that we have found generally result in good performance. Other important design facilities include routines to compute the residual, to extract the pivots, and to save the factors for later additional solves. Full details are given in the user documentation that accompanies the code.

The first release of the new solver HSL_MA77 was for positive definite matrices; it has recently been extended to the indefinite case. The need to handle interchanges makes the kernel for performing partial factorizations and solves of indefinite matrices significantly more complicated than in the positive-definite case. The indefinite code and, in particular, the incorporation of 1×1 and 2×2 pivots will be described elsewhere. We have also developed a version for unsymmetric matrices held as a sum of element matrices [Reid and Scott 2009b]; this version is called HSL_MA78.

HSL_MA77, together with the subsidiary packages HSL_MA54 and HSL_OF01, are included in the 2007 release of HSL. All use of HSL requires a license. Licenses are available without charge to individual academic users for their personal (noncommercial) research and for teaching; for other users, a fee is normally charged. Details of how to obtain a licence and further details of all HSL packages are available online.[3]

In the future, we plan to write a version of HSL_MA77 that will accept an assembled matrix as a whole, that is, without reverse communication. It will offer the option of computing a suitable pivot ordering. We expect to develop a version for complex arithmetic and possibly one that allows the frontal matrix to be held out of core to reduce main memory requirements further. We also plan to offer a number of options for inputting the right-hand sides B, including as a sum of element matrices (which may be a more convenient format for some finite-element users).

ACKNOWLEDGMENTS

We would like to express our appreciation of the help given by our colleagues Nick Gould and Iain Duff. Nick has constantly encouraged us and has made many suggestions for improving the functionality of the package. Iain reminded us about linking the elements and generated elements according to the supervariable that is earliest in the pivot sequence (see Section 6.2), which is a key strategy for the speed of MA77_analyse. He also made helpful comments on a draft of this article. We are also grateful to Jean-Yves L'Excellent of LIP-ENS Lyon and Abdou Guermouche of LaBRI, Bordeaux, for helpful discussions on their work on the efficient implementation of multifrontal algorithms. Finally, we would like to thank the three anonymous referees, each of whom made insightful and constructive criticisms that have led to improvements both in our codes and in this article.

REFERENCES

AMESTOY, P., DAVIS, T., AND DUFF, I. 1996. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl. 17*, 886–905.

AMESTOY, P., DAVIS, T., AND DUFF, I. 2004. Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw. 30*, 3, 381–388.

ANDERSEN, B., GUNNELS, J., GUSTAVSON, F., REID, J., AND WASNIEWSKI, J. 2005. A fully portable high performance minimal storage hybrid format cholesky algorithm. *ACM Trans. Math. Softw. 31*, 2, 201–227.

[3]www.cse.clrc.ac.uk/nag/hsl.

BCSLIB-EXT. 2003. BCSLIB-EXT—award winning sparse-matrix package. http://www.boeing.com/phantom/bcslib-ext/.

CONN, A. R., GOULD, N. I. M., AND TOINT, P. L. 2000. *Trust-Region Methods*. Society for Industrial and Applied Mathematics, Philadelphia, PA.

DAVIS, T. 2007. The University of Florida Sparse Matrix Collection. Tech. rep. University of Florida, Gainesville, FL. http://www.cise.ufl.edu/ davis/techreports/matrices.pdf.

DOBRIAN, F. AND POTHEN, A. 2003. A comparison between three external memory algorithms for factorising sparse matrices. In *Proceedings of the SIAM Conference on Applied Linear Algebra*.

DONGARRA, J., DU CROZ, J., DUFF, I. S., AND HAMMARLING, S. 1990. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw. 16*, 1, 1–17.

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

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

DUFF, I. AND REID, J. 1982. MA27—A set of Fortran subroutines for solving sparse symmetric sets of linear equations. Report AERE R10533. Her Majesty's Stationery Office, London, U.K.

DUFF, I. AND REID, J. 1983. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Trans. Math. Softw. 9*, 302–325.

DUFF, I. AND REID, J. 1996. Exploiting zeros on the diagonal in the direct solution of indefinite sparse symmetric linear systems. *ACM Trans. Math. Softw. 22*, 2, 227–257.

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

GOULD, N. AND SCOTT, J. 2004. A numerical evaluation of HSL packages for the direct solution of large sparse, symmetric linear systems of equations. *ACM Trans. Math. Softw. 30*, 3, 300–325.

GOULD, N., SCOTT, J., AND HU, Y. 2007. A numerical evaluation of sparse direct solvers for the solution of large, sparse, symmetric linear systems of equations. *ACM Trans. Math. Softw. 33*, 2, Article 10.

GUERMOUCHE, A. AND L'EXCELLENT, J.-Y. 2004. Optimal memory minimization algorithms for the multifrontal method. Tech. rep. RR5179. INRIA, Rocquencourt, France.

GUERMOUCHE, A. AND L'EXCELLENT, J.-Y. 2006. Constructing memory-minimizing schedules for multifrontal methods. *ACM Trans. Math. Softw. 32*, 1, 17–32.

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

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

ISO/IEC. 2001. TR 15581(E): Information technology—programming languages—Fortran—enhanced data type facilities (second edition), M. Cohen, Tech. rep., ISO/IEC, Geneva, Switzerland.

KARYPIS, G. AND KUMAR, V. 1998. METIS—family of multilevel partitioning algorithms. http://glaros.dtc.umn.edu/gkhome/views/metis.

KARYPIS, G. AND KUMAR, V. 1999. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput. 20*, 359–392.

LIU, J. 1985. Modification of the Minimum-Degree algorithm by multiple elimination. *ACM Trans. Math. Softw. 11*, 2, 141–153.

LIU, J. 1986. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Trans. Math. Softw. 12*, 3, 249–264.

MUMPS. 2007. MUMPS: A multifrontal massively parallel sparse direct solver. http://mumps.enseeiht.fr/.

REID, J. 1984. TREESOLV, a Fortran package for solving large sets of linear finite-element equations. Report CSS 155. AERE Harwell, Harwell, U.K.

REID, J. K. AND SCOTT, J. 2009a. Algorithm 891: A Fortran virtual memory system. *ACM Trans. Math. Softw. 36*, 1.

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

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

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

TINNEY, W. AND WALKER, J. 1967. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proc. IEEE 55*, 1801–1809.

Received October 2006; revised March 2007, December 2007, April 2008, September 2008; accepted September 2008