COMMUNICATIONS IN NUMERICAL METHODS IN ENGINEERING
Commun. Numer. Meth. Engng 2006; **22**:1015–1029
Published online 27 April 2006 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cnm.870

A frontal solver for the 21st century

Jennifer A. Scott*,†

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

SUMMARY

In recent years there have been a number of important developments in frontal algorithms for solving the large sparse linear systems of equations that arise from finite-element problems. We report on the design of a new fully portable and efficient frontal solver for large-scale real and complex unsymmetric linear systems from finite-element problems that incorporates these developments. The new package offers both a flexible reverse communication interface and a simple to use all-in-one interface, which is designed to make the package more accessible to new users. Other key features include automatic element ordering using a state-of-the-art hybrid multilevel spectral algorithm, minimal main memory requirements, the use of high-level BLAS, and facilities to allow the solver to be used as part of a parallel multiple front solver. The performance of the new solver, which is written in Fortran 95, is illustrated using a range of problems from practical applications. The solver is available as package `HSL_MA42_ELEMENT` within the HSL mathematical software library and, for element problems, supersedes the well-known `MA42` package. Copyright © 2006 John Wiley & Sons, Ltd.

KEY WORDS: large sparse linear systems; finite elements; frontal method; out-of-core; Fortran 95

1. INTRODUCTION

We are interested in the efficient solution of large sparse linear systems of equations

$$AX = B \tag{1}$$

where the system matrix A is of order $n \times n$, B is an $n \times nrhs$ ($nrhs \geqslant 1$) matrix of right-hand sides and X is the $n \times nrhs$ solution matrix. Such systems arise in many areas of computational science and engineering; our interest is in systems (1) that arise from finite-element

*Correspondence to: Jennifer A. Scott, Computational Science and Engineering Department, Atlas Centre, Rutherford Appleton Laboratory, Didcot, Oxon OX11 0QX, U.K.
†E-mail: j.a.scott@rl.ac.uk

Contract/grant sponsor: EPSRC; contract/grant number: GR/S42170

Received 19 October 2005
Revised 17 February 2006
Accepted 24 February 2006

applications. In this case, A is an elemental matrix, that is, A can be written as the sum

$$A = \sum_{k=1}^{\text{nelt}} A^{(k)} \tag{2}$$

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

One possible method for solving systems of this form is the frontal method. Frontal schemes have their origins in the early 1970s with the work of Irons [1]. At the time, there was a need to solve finite-element problems that were too large for the system matrix and the matrix factors to be held in main memory, so that existing direct methods could not be used. The frontal method was therefore designed to be a robust direct method that required only a small amount of main memory (that is, the main memory needed was small compared with the order n of the linear system). Today computers and their memories are much larger but so too are the problems that computational scientists and engineers wish to solve. Thus methods that require only limited main memory remain attractive.

The frontal method is a variant of Gaussian elimination and involves the matrix factorization

$$A = PLUQ \tag{3}$$

where P and Q are permutation matrices, and L and U are lower and upper triangular matrices, respectively. The solution process is completed by performing the forward elimination

$$PLY = B \tag{4}$$

followed by the back substitution

$$UQX = Y \tag{5}$$

The key feature of the method is that the contributions $A^{(k)}$ from the finite elements are assembled one at a time and the storage of the entire assembled coefficient matrix A is avoided by interleaving assembly and elimination operations. This allows the computation to be performed using a *frontal matrix* that at each stage may be expressed in the form

$$\begin{pmatrix} F_T & F_R \\ F_C & F_U \end{pmatrix} \tag{6}$$

where the rows and columns of the $r \times r$ matrix F_T are *fully summed*, that is, there are no other entries in these rows and columns in the overall matrix, while the rows and columns of the $s \times s$ matrix F_U are not yet fully summed. Assuming a suitable element assembly order is chosen, $r \ll s$ and the frontsize $r + s$ is much less than n, the order of A. Provided stable pivots can be chosen from F_T, the factorization $F_T = L_T U_T$ is computed. Then F_C and F_R are

updated as

$$F_C \leftarrow F_C U_T^{-1} \tag{7}$$

$$F_R \leftarrow L_T^{-1} F_R \tag{8}$$

and then Schur complement

$$F_U \leftarrow F_U - F_C F_R \tag{9}$$

is formed. At the next stage, another element matrix is assembled with this Schur complement to form another frontal matrix. As the rows and columns of the matrix factors are generated, they are written to buffers (work arrays), which are held in main memory. This frees up space in the frontal matrix, which can then be reused for further incoming elements. If a buffer becomes full, its contents are held out of core, that is, they are written to a direct access file. The data in these files is read back into main memory (one record at a time) during the forward elimination and back substitution phases.

Since the original work of Irons, the frontal method has been developed and generalized by a number of authors, including References [2–5]. The frontal solver `MA42` of Duff and Scott [6, 7] for real unsymmetric systems and its counterpart `ME42` for complex systems have been part of the mathematical software library HSL [8] for a decade. The codes have been widely used to solve problems from a variety of different application areas including fluid flow, structural analysis, and chemical process engineering. As well as being used by academics in the U.K. and elsewhere, they have been incorporated into a number of commercial products. They have also been used in the development of parallel frontal solvers for HSL. However, in recent years there has been further research into frontal algorithms that has led to important performance improvements in terms of both computational time and storage requirements (for example, References [9, 10]). Moreover, the older codes were written in Fortran 77, which does not offer many of the features of Fortran 95 that enable more user-friendly solvers to be developed. With the high-quality Fortran 95 compilers becoming more widely available (including the freely available gf95 compiler at `g95.sourceforge.net`), we have become interested in using Fortran 95 for the development of solvers for HSL. We therefore felt it was time to write a new frontal solver for the 21st century. The new Fortran 95 code, which is designed exclusively for the efficient solution of large-scale unsymmetric finite-element problems, is called `HSL_MA42_ELEMENT`. We end this section by listing the key features of `HSL_MA42_ELEMENT`, a number of which will be discussed further in later sections, as indicated.

- Two user interfaces are offered: a flexible reverse communication interface and a simpler all-in-one interface that is designed to appeal in particular to inexperienced users (see Section 2).
- Versions are included within the one package for factorizing and solving both real and complex systems. Once the matrix factors have been computed they can also be used to solve transpose systems $A^{\mathrm{T}}X = B$ (and, in the complex case, complex conjugate systems $A^{\mathrm{H}}X = B$).
- Through the use of control parameters and optional arguments, a wide range of options is available to the user. Key parameters and options are discussed in Sections 2.3 and 2.4 and full details are given in the user documentation.
- A number of state-of-the-art element ordering algorithms are incorporated within the package (see Section 5).

- The code automatically switches to holding the matrix factors out of core if there is insufficient main memory. During the factorization, data is put into explicitly held buffers, which are written to direct access files once they are full. The length of the records in each of the direct access files (which is equal to length of the associated buffer) is chosen either automatically by the code or by the user. Further details are given in Section 2.3.
- High level BLAS [11] are used for performing the dense linear algebra operations (8) and (9), and for performing the forward elimination and back substitution operations. Efficient use is made of level 3 BLAS by imposing a minimum pivot block size (see Section 3).
- To try and avoid unnecessary operations with zeros, the code follows the work of Scott [10] and exploits zeros in the frontal matrix (see Section 4).
- A number of features are offered that allow the code to be used within an implementation of the multiple front algorithm; this is discussed in Section 6.

Finally, we note that the naming convention adopted within HSL is for all Fortran 90 or 95 packages to have a name starting with `HSL_` (which distinguishes them from the Fortran 77 codes). For convenience, throughout the remainder of this paper, we abbreviate the full name `HSL_MA42_ELEMENT` of our new frontal solver to `MA42_ELEMENT`.

2. USER INTERFACE

In common with other sparse direct methods, the frontal method can be split into a number of distinct phases as follows:

1. An ordering phase that determines a suitable order for assembling the elements. For efficiency in terms of both storage and arithmetic operations, it is essential that the elements are assembled in an order that keeps the size of the frontal matrix small. That is, once a variable has entered the front, it needs to become fully summed as quickly as possible.
2. An analyse phase that takes the index lists for each of the elements in turn and determines a potential pivot sequence based on the sparsity pattern alone.
3. A factorization phase that uses the pivot sequence (modified if necessary to maintain numerical stability) to factorize the matrix.
4. A solve phase that performs forward elimination followed by back substitution using the stored factors.

`MA42_ELEMENT` offers the user two different interfaces: a reverse communication interface that requires separate calls to each of the different phases and a simpler (but slightly less flexible) all-in-one interface. Both make use of control parameters. These are parameters that, as their name implies, control the action within the package. They are given default values by a call to the initialization routine `MA42_ELEMENT_START`. The defaults have been chosen on the basis of our numerical experiments on a range of problems and computer platforms and are likely to be appropriate for most users. However, for maximum flexibility, these parameters may be reset by the user after the call to `MA42_ELEMENT_START`. The controls include parameters that determine the level of diagnostic printing, the choice of element ordering algorithm, the pivoting, and the action taken if A is found to be

singular (see Section 2.4). Full details of all the control parameters are provided in the user documentation.

In the remainder of this section, we describe the two interfaces and then look at the greater flexibility that is offered by the reverse communication interface.

2.1. Reverse communication

The key idea of the reverse communication interface is to keep main memory requirements for the matrix data to a minimum by requiring the user to supply the element matrices one at a time as they are needed. This gives the user maximum freedom as to how the element matrices are held; if convenient, the user may choose to generate the element data only as it is required. The ordering and analyse phases are optionally combined and, if the right-hand sides B are available in unassembled form, that is, $B = \sum_{k=1}^{\text{nelt}} B^{(k)}$, they may be passed with the element matrices $A^{(k)}$ to the factorization phase. In this case, forward elimination is performed at the same time as the matrix factorization and, once the factorization is complete, back substitution is performed to complete the solution.

There are three main routines that comprise the reverse communication interface:

`MA42_ELEMENT_ANALYSE`: must be called for each element in turn to specify which variables are associated with it. The calls may be made in any order. An element assembly order may be supplied by the user, otherwise an ordering is automatically generated. This is discussed further in Section 5. The output from the final call to `MA42_ELEMENT_ANALYSE` is the element assembly order for the factorization phase. In addition, the analyse phase determines when each variable becomes fully summed (that is, it determines a tentative pivot sequence) and computes estimates of the maximum frontsize and of the storage required for the matrix factors.

`MA42_ELEMENT_FACTORIZE`: must be called for each element to specify the entries of $A^{(k)}$ and, optionally, $B^{(k)}$. The calls *must* be made in the order determined by the analyse phase (no calls may be made until the analyse phase is complete). Data from `MA42_ELEMENT_ANALYSE` is used to factorize the matrix and, if $B^{(k)}$ are specified, the equations $AX = B$ with right-hand side(s) $B = \sum_{k=1}^{\text{nelt}} B^{(k)}$ are solved after the call for the last element. Note that more than one finite-element problem having elements with the same variable lists (but different numerical values) may be factorized and solved following a single set of calls to `MA42_ELEMENT_ANALYSE`. This is useful, for example, when solving the linear systems that result from using an iterative method (such as Newton's method) to solve a nonlinear problem; a sequence of problems with the matrix A having the same element structure and sparsity pattern but different numerical values must be solved.

`MA42_ELEMENT_SOLVE`: uses the computed factors to rapidly solve either further systems of the form $AX = B$ or systems of the form $A^{\mathrm{T}}X = B$ (or $A^{\mathrm{H}}X = B$, where A^{H} is the complex conjugate transpose of A), with the right-hand side vectors B input in assembled form. Any number of calls to `MA42_ELEMENT_SOLVE` may follow the final call to `MA42_ELEMENT_FACTORIZE`.

In addition to the above routines, `MA42_ELEMENT_RESIDUAL` may be called for each element after the final call to `MA42_ELEMENT_FACTORIZE`, or after a call to `MA42_ELEMENT_SOLVE`, to compute the residual matrix $RES = B - AX$ (or $RES = B - A^{\mathrm{T}}X$ or $RES = B - A^{\mathrm{H}}X$). This routine also optionally computes an upper bound on the infinity norm of the system

matrix (it is an upper bound because the absolute values of the element entries are taken before they are summed). Denoting this bound by $\|A\|_{b,\infty}$, the user can compute the scaled residuals

$$\frac{\|res_j\|_\infty}{\|A\|_{b,\infty}\|x_j\|_\infty + \|b_j\|_\infty} \tag{10}$$

where b_j is the jth right-hand side and x_j and res_j are the corresponding solution and residual vector, respectively. If the user decides that the computed residual is unacceptably large, iterative refinement can be performed by recalling `MA42_ELEMENT_SOLVE` with the right-hand side set to *RES*.

2.2. All-in-one interface

`MA42_ELEMENT` also offers a simpler all-in-one interface. This involves no reverse communication. Instead, the user makes a single call to routine `MA42_ELEMENT_AFS` to perform the analyse, factorize, and (optionally) solve phases. `MA42_ELEMENT_SOLVE` may be called to solve for further right-hand sides or to solve transpose (or complex conjugate transpose) systems.

When calling `MA42_ELEMENT_AFS`, the user supplies the lists of the variables for all the elements in a single integer array. The entries of the element matrices may be supplied using either an array or a direct access file (with a further array or file for element right-hand sides). Using files reduces the main memory requirements and will generally be needed for large problems (without this option, the important advantage that frontal solvers have of needing only a small amount of main memory is lost). Each record in the element data file must contain the entries for a single element matrix, with the elements held in the same order as in the array of element variable lists. Direct access (and not sequential) files are needed so that, during the factorization, the code can read the elements in the assembly order generated by the automatic element ordering algorithm.

2.3. Factorize options and flexibility

As well as offering the user greater flexibility in how to store or generate the element data, `MA42_ELEMENT_FACTORIZE` includes a number of options that are not available in the all-in-one interface. In particular, there are options to supply the lengths of the buffers (work arrays) used by the factorization and the maximum order of the frontal matrix. The latter must be at least as large as the estimate returned by the analyse phase. Because a dense matrix of order the maximum frontsize is needed, if memory restrictions are likely to be an issue, the user should not choose a maximum frontsize that is very large compared with the analyse estimate; a value larger than the analyse estimate is recommended to allow for possible increases to the frontsize because of delayed pivots. The analyse phase assumes that whenever there are at least `pivot_size` fully summed variables, they can be eliminated (`pivot_size` is a control parameter that is discussed further in Section 3). During the factorization, a potential pivot can only be used if it satisfies a numerical stability test. Specifically, a fully summed entry of the frontal matrix is only considered suitable for use as a pivot if it is of absolute value at least as large as `alpha` times the entry of largest absolute value in its column. The threshold parameter `alpha` ($0 <$ `alpha` < 1) is a control parameter with a default value of 0.01. If a potential pivot does not satisfy this condition, it is delayed and will be retested after further element assemblies. Values of `alpha` close to zero will generally result in a faster factorization

with fewer entries in the factors but values close to 1 are more likely to result in a stable factorization; the default of 0.01 is a compromise between stability and sparsity. If the user does not supply the maximum frontsize when calling MA42_ELEMENT_FACTORIZE, a maximum frontsize of 110% of the estimate from the analyse phase is used. Should a user have prior knowledge of his or her problem and know that a large number of pivots will be delayed, causing the maximum frontsize to increase substantially beyond that predicted by the analyse phase, the user can overwrite the automatically selected maximum frontsize with his or her own choice.

The earlier Fortran 77 code MA42 was unable to continue if the user-supplied frontsize was not large enough but using Fortran 95 allows MA42_ELEMENT_FACTORIZE to continue the computation by allocating a new larger frontal matrix. The action taken when the frontsize is too small depends upon the control parameter front_multiple. If this is less than or equal to 1, the computation terminates with an error message as soon as the frontsize is found to be too small. The user may then increase the maximum frontsize and restart the factorization (there is no need to repeat the analyse phase). If front_multiple is greater than 1 (the default is 1.1), the computation will attempt to continue. In this case, the contents of the internal arrays of size that depends on the maximum frontsize are written to scratch files, the arrays are deallocated and then reallocated with sizes sufficient to continue the computation. The data in the files is read back into these arrays, the scratch files are closed (and thus deleted), and the computation continues. Note, however, that the computation cannot continue if there is insufficient in-core memory to perform the reallocation. In this case, a suitable error flag is returned to the user. Note also that increasing the size of internal arrays will add to the factorization cost and may be done more than once during the factorization of a particular problem. At the end of the factorization details of the maximum frontsize used is returned to the user. If the user intends to factorize more than one matrix having the same sparsity pattern, for efficiency advantage should be taken of this maximum frontsize when calling MA42_ELEMENT_FACTORIZE for subsequent matrices.

Three direct-access files are optionally used by MA42_ELEMENT: one for the UQ factor (which is held with the corresponding right-hand sides), one for the PL factor, and one for the row and column indices of the variables in the factors. One of the complications of using MA42 was that the user had to provide data on both the sizes of the buffers to be used and the amount of storage required by the factors. Clearly, for a new problem or application area, this could be difficult. Estimates of the factor storage needed based on the assumption that no pivots are not delayed was returned by the analyse phase but, if numerical considerations caused a number of pivots to be delayed, the factorization could terminate before completion and would have to be completely restarted with increased parameter values. A key design aim of MA42_ELEMENT was to simplify the use of direct access files and minimize the input required from the user. For the all-in-one interface the user is not asked for any input relating to the buffers and files. In this case, buffer lengths of 2^{16} are used. For large problems, files will be required to hold the factors. The code chooses appropriate units on which to open files and these files are given the status SCRATCH. At the end of the computation, these files are closed (and hence lost). The only way the user will be aware that files have been used will be through the receipt of a warning flag.

However, this simple use of buffers and files may not be suitable for all users. In particular, the user may wish to use named direct access files that can be saved at the end of the computation for possible further solves in the future. The reverse communication interface

to MA42_ELEMENT offers a number of options aimed at more experienced users. The user can choose to supply the buffer lengths; if these are sufficiently large, using files may be avoided. The user's choice will normally depend upon the problem size and memory available. If the user does not supply buffer lengths then default values of 2^{16} are again used. The user may also optionally supply the names of the direct access files. If names are supplied, at the end of the computation the files are closed but not deleted.

We note that the L factor only needs to be stored during the factorization if the user wishes to call MA42_ELEMENT_SOLVE after the final call to MA42_ELEMENT_FACTORIZE (or after a call to MA42_ELEMENT_AFS). Not storing the L factor will clearly result in a substantial storage saving.

2.4. Coping with singular matrices

In some application areas, the system matrix A may be singular (or nearly singular). MA42_ELEMENT has two control parameters, lsingular and small, which are useful in this situation. lsingular is a logical scalar that controls the action is the matrix A is found to be singular. By default, lsingular = .true. and if A is found to be singular, a warning is issued and the computation continues. Components of the solution vector that correspond to zero pivots are set to zero. Otherwise, if lsingular = .false., the computation terminates with an error message immediately singularity is detected. small controls how small the pivots are allowed to become before they are treated as zero and A is declared singular. Specifically, if the entry of largest absolute value in any column of the reduced matrix at any stage of the factorization is found to less than or equal to small, A is singular. The default is small = 0 but by choosing a non-zero value, the user is able to control whether the computation terminates when A is found to be nearly singular.

3. MINIMUM PIVOT BLOCK

At each stage of the elimination process, once pivots have been chosen, it is essential to the overall efficiency of the frontal solver that the Schur complement (9) is formed as efficiently as possible. The frontal matrix is held as a dense matrix and so dense linear algebra kernels (in particular, the BLAS) may be used. If the frontal solver picks a single pivot at a time then it is only possible to use Level 2 BLAS but if $r > 1$ pivots are chosen, F_R may be updated using the Level 3 BLAS routine _TRSM and then the Schur complement (9) computed using the Level 3 BLAS routine _GEMM with interior dimension r. Our experience has been that, for some problems (notably those with only one variable per finite-element node), r can be small and there is then little advantage gained by using Level 3 BLAS. This prompted Cliffe *et al.* [9] to look at enhancing the use of the BLAS by delaying updating the frontal matrix until the number of pivot candidates is at least some prescribed minimum, say pivot_size. Suppose, at some stage, that the number of fully summed variables is k, then the maximum number of pivots which we can choose is k. If $k <$ pivot_size and not all the elements have been assembled, we do not look for pivots but assemble another element into the frontal matrix until the number of fully summed variables is at least pivot_size.

In MA42_ELEMENT, the minimum pivot block size pivot_size is a control parameter with default value 16. The best value to use is both problem and machine dependent. Increasing the minimum pivot block size in general increases the number of floating-point operations

and real storage requirements but reduces integer storage and, most importantly, can reduce the CPU time required by both the factorization and solve phases. Results illustrating this are given in Reference [9]. We note that `pivot_size` is used in both the analyse and the factorize phases. In particular, the estimates of the maximum frontsize and factor storage returned by `MA42_ELEMENT_ANALYSE` are dependent on `pivot_size`. Since the analyse phase is much less expensive than the factorization phase (especially as element reordering only needs to be performed on the first run of the analyse phase for a particular problem), the user can investigate the effect of varying `pivot_size` before factorizing the matrix.

4. ZEROS IN THE FRONT

During the factorization, the frontal matrix may contain some zero entries. Treating the frontal matrix as a dense matrix results in unnecessary operations being performed with these zeros and potentially a large number of explicit zeros being stored in the factors. Because Level 3 BLAS are used to perform the factorization operations, the cost of the operations with zeros may not be prohibitive but if the frontal matrix contains a significant number of zeros, Scott [10] found that it can be advantageous to exploit these zeros. To see how this can be done, suppose the frontal matrix has been permuted to the form (6) and that k is the number of fully summed variables. By performing further row and column permutations, the frontal matrix can be expressed in the form

$$F = \begin{pmatrix} F_T & F_{R_1} & 0_1 \\ F_{C_1} & F_{U_T} & F_{U_R} \\ 0_2 & F_{U_C} & F_{U_U} \end{pmatrix} \tag{11}$$

where 0_1 and 0_2 are zero matrices of order $k \times k_1$ and $k_2 \times k$, respectively. Assuming the current frontal matrix is of order $l \times l$, k_1 and k_2 satisfy $0 \leqslant k_1 \leqslant l - k$ and $0 \leqslant k_2 \leqslant l - k$.

In place of (7), (8) and (9), we now need only perform the updates

$$F_{C_1} \leftarrow F_{C_1} U_T^{-1} \tag{12}$$

$$F_{R_1} \leftarrow L_T^{-1} F_{R_1} \tag{13}$$

and

$$F_{U_T} \leftarrow F_{U_T} - F_{C_1} F_{R_1} \tag{14}$$

When writing to the buffers, F_{R_1} and F_{C_1}, rather than F_R and F_C, are stored, resulting in savings in both the real and integer factor storage.

If more than one pivot is chosen, the updated matrices F_{R_1} and F_{C_1} may still contain some zeros. However, experiments reported in Reference [10] indicated that, in general, the number of zeros remaining in the factors is small (typically less than 10% of the total number of entries in the factors). We do not, therefore, attempt to exploit zeros within F_{R_1} and F_{C_1}.

5. ELEMENT ORDERING

The efficiency of the frontal method, in terms of both storage and arithmetic operations, is dependent upon assembling the elements in an order that keeps the size of the frontal matrix, known as the *wavefront*, as small as possible. In other words, the elements need to be ordered so that partially summed variables become fully summed as soon as possible. `MA42_ELEMENT` offers a number of element ordering routines within the package, but also allows the user to supply his or her own ordering. Allowing the user to specify an ordering is particularly important if a number of matrices with the same (or similar) sparsity patterns are to be factorized. In this case, the reordering (which may add a significant CPU overhead to the analyse phase) need only be performed for the first matrix.

In a recent article, Scott [12] reported on the use of multilevel element ordering algorithms and compared their performance with a number of variants of Sloan's algorithm [13]. Scott considered both direct and indirect versions of the multilevel algorithm (an algorithm is referred to as a direct algorithm if it orders the elements directly and as an indirect algorithm if the variables are first resequenced and the new variable numbers then used to reorder the elements). Scott also used these variants in combination with spectral orderings to give the so-called hybrid orderings. Numerical experimentation on a range of large problems from practical applications showed that, in general, the best orderings are obtained using the indirect hybrid spectral-Sloan algorithm and so this has been chosen as the default element ordering algorithm within `MA42_ELEMENT`. The multilevel Fiedler code `HSL_MC73` of Hu and Scott [14] is called to compute the spectral ordering and then a modified version of `MC63` is used to obtain the hybrid ordering (`MC63` is an HSL package that offers an efficient implementation of a version of Sloan's algorithm; it is described in Reference [15]).

Because Sloan orderings can be computed cheaply using `MC63` and because they are generally of a similar quality to the hybrid spectral-Sloan orderings for relatively small problems, `MA42_ELEMENT` includes an option to reorder using `MC63`. An option is also available for computing both the direct and indirect orderings; the better ordering (in terms of the root mean squared wavefront) is then automatically selected and returned from the analyse phase. How much time the user wishes to spend on element ordering will generally depend either on whether memory restrictions make it important that the maximum frontsize is as small as possible or on the number of matrices with the same (or similar) sparsity patterns that are to be factorized. Clearly, if a large number of factorizations (or large number of solves following a factorization) are to be performed, it may well be worthwhile to experiment with a number of different ordering algorithms so that sparse factors are computed as rapidly as possible; `MA42_ELEMENT` has been designed to make this straightforward for the user.

6. FEATURES DESIGNED FOR A MULTIPLE FRONT ALGORITHM

One of the main deficiencies of the frontal solution scheme is that there is little scope for parallelism other than that which can be obtained within the high-level BLAS. One way of attempting to overcome this is to extend the basic frontal algorithm to use multiple fronts. While `MA42_ELEMENT` is not a multiple front code, it has been designed to include a number of options that will allow it to be used in a straightforward way within a multiple front code. These are discussed briefly in this section.

In a multiple front approach, the underlying finite-element domain Ω is first partitioned into non-overlapping subdomains Ω_i. This is equivalent to ordering the matrix A to doubly bordered block diagonal form

$$\begin{pmatrix} A_{11} & & & & C_1 \\ & A_{22} & & & C_2 \\ & & \ddots & & \vdots \\ & & & A_{NN} & C_N \\ \tilde{C}_1 & \tilde{C}_2 & \cdots & \tilde{C}_N & \sum_{i=1}^{N} E_i \end{pmatrix} \tag{15}$$

where the diagonal blocks A_{ii} are $n_i \times n_i$ and the border blocks C_i and $\tilde{C}_i$ are $n_i \times l$ and $l \times n_i$, respectively, with $l \ll n_i$. A partial frontal decomposition is performed on each of the matrices

$$\begin{pmatrix} A_{ii} & C_i \\ \tilde{C}_i & E_i \end{pmatrix} \tag{16}$$

This can be done in parallel. At the end of the assembly and elimination processes for each subdomain Ω_i, there will remain $1 \leqslant l_i \leqslant l$ interface variables. These variables cannot be eliminated since they are shared by more than one subdomain. Variables that have not been eliminated within the subdomain because of efficiency or stability considerations will also remain. These variables are added to the border and l is increased. If F_i holds the frontal matrix that remains when all possible eliminations on subdomain Ω_i have been performed, once each of the subdomains has been dealt with, formally, we have

$$A = P \begin{pmatrix} L_1 & & & & \\ & L_2 & & & \\ & & \ddots & & \vdots \\ & & & L_N & \\ \tilde{L}_1 & \tilde{L}_2 & \cdots & \tilde{L}_N & I \end{pmatrix} \begin{pmatrix} U_1 & & & & \tilde{U}_1 \\ & U_2 & & & \tilde{U}_2 \\ & & \ddots & & \vdots \\ & & & U_N & \tilde{U}_N \\ & & \cdots & & F \end{pmatrix} Q \tag{17}$$

where P and Q are permutation matrices and the $l \times l$ matrix F is a sum of the F_i's and is termed the *interface matrix*. It may also be factorized using the frontal method. Once the interface variables have been computed, the rest of the block back substitution can be performed in parallel.

When applying a frontal solver to a subdomain, elimination of the interface variables (which are not fully summed within the subdomain) must be prevented. A simple way of doing this is by introducing an extra element for each subdomain that contains only the interface variables for that subdomain. The extra element, which is called a *guard element* (see References [7, 16]), is passed as the last element to the analyse phase but is not passed to the factorize

phase. Since the factorize phase is not called for the guard element, variables in the guard element (that is, the interface variables) do not become fully summed but remain in the front after the assembly and elimination operations for the final element in Ω_i are complete. Thus if MA42_ELEMENT is to be used as part of a multiple front solver it needs to offer a means of extracting the remaining frontal matrix F_i from its internal structures. To do this, we have included within the package a separate subroutine MA42_ELEMENT_PARTIAL, which may be called by the user after one or more elements has been passed to the factorization phase to preserve the partial factorization. It writes the data remaining in the buffers to the direct access files and saves the data remaining in the frontal matrix and corresponding frontal right-hand side matrix in arrays that are held in main memory. In addition, we have included within routine MA42_ELEMENT_SOLVE options for performing the forward eliminations and back-substitutions on separate calls. This feature is needed by the solve phase of the multiple front algorithm.

7. NUMERICAL EXPERIMENTS

In this section, we report on using MA42_ELEMENT to solve a number of problems from practical applications. Comparisons are made with MA42. The test problems are listed in Table I. They range in size from fewer than 1000 elements to more than 70 000 elements with almost 225 000 degrees of freedom. If only the sparsity pattern is available, numerical values for the matrix entries are generated using the HSL pseudo-random number generator FA14. Our experiments are performed on a single Xeon 3.06 GHz processor of a Dell Precision Workstation 650

Table I. The test problems.

Identifier	n	nelt	Description/discipline
cham*	12 834	11 070	Part of an engine cylinder
crplat2*	18 010	3152	Corrugated plate field
fcondp2*	201 822	35 836	Oil production platform
fullb*	199 187	59 738	Full-breadth barge
halfb*	224 617	70 211	Half-breadth barge
inv-ext-2*	78 142	7193	Fluid flow
mt1*	97 578	5328	Tubular joint
opt1*	15 449	977	Part of condeep cylinder
ship_001	34 920	3431	Ship structure—predesign
ship_003	121 728	45 464	Ship structure—production
shipsec1*	140 874	41 037	Section of a ship
shipsec5	179 860	52 272	Section of a ship
shipsec8	114 919	32 580	Section of a ship
srb1*	54 924	9240	Space shuttle rocket booster
thread*	29 736	2176	Threaded connector
trdheim*	22 098	813	CFD simulation; mesh of Trondheim fjord
troll*	213 453	41 084	Structural analysis
tsyl201*	20 685	960	Part of condeep cylinder
tubu*	26 573	23 446	Engine cylinder model
x104	108 384	26 019	Beam joint

n and nelt denote the number of variables and elements, respectively.
*Only pattern available.

Table II. Timings (in seconds) for MA42 and MA42_ELEMENT.

	Analyse		Factorize		Solve	
Identifier	MA42	MA42_ ELEMENT	MA42	MA42_ ELEMENT	MA42	MA42_ ELEMENT
cham	0.09	0.20	8.9 (2.9)	2.8	0.38	0.23
crplat2	0.01	0.03	4.9 (3.9)	2.3	0.33	0.24
fcondp2	0.24	0.39	3299 (1695)	692	36.3	21.6
fullb	0.50	0.51	1806 (1065)	783	28.8	25.9
halfb	0.52	0.52	1345 (714)	507	26.0	19.5
inv-ext-2	0.25	0.55	1152 (1016)	448	15.9	9.8
mt1	0.10	0.22	243 (233)	172	6.4	6.1
opt1	0.02	0.07	7.6 (7.1)	7.8	0.40	0.41
ship_001	0.06	0.10	16 (17)	13	0.86	0.84
ship_003	0.37	0.35	1279 (1261)	399	13.0	9.2
shipsec1	0.34	0.34	1916 (1089)	342	23.9	11.1
shipsec5	0.44	0.43	3361 (4472)	1311	25.6	19.2
shipsec8	0.34	0.29	4684 (6522)	1680	21.8	15.1
srb1	0.05	0.08	14 (11)	11	0.99	0.97
thread	0.06	0.22	170 (147)	53	2.9	1.6
trdheim	0.02	0.03	1.0 (1.0)	0.9	0.23	0.12
troll	0.39	0.73	8060 (5257)	3408	62	47
tsyl201	0.01	0.03	9.2 (9.1)	13	0.54	0.58
tubu	0.27	0.42	29 (10)	10	0.90	0.61
x104	0.08	0.20	1453 (1651)	1031	9.1	7.1

with 4 GBytes of RAM under the Fedora Core 1 Linux operating system. The NAG Fortran 95 compiler is used with the compiler optimization flag -O. All reported timings are CPU times, measured using the Fortran 95 routine cpu_time and are given in seconds. In all our tests, the scaled residual (10) was computed; in each case, this was found to be less than 10^{-12}.

MA42 (Version 1.0.0) is run with all its control parameters set to their default values. Note that this means that the minimum pivot block size is 1 and zeros in the front are not exploited. The finite elements are preordered for MA42 using MC63 (both the indirect and direct algorithms are run and the best one selected). The time needed by MC63 to reorder the elements is added to the time for the analyse phase of MA42. The buffers sizes are chosen to be the same as those used by MA42_ELEMENT (that is, 2^{16}). MA42_ELEMENT is also run with its default control parameters. CPU timings for the analyse, factorize, and solve phases (for a single right-hand side) are reported in Table II. We see that the analyse phase of MA42_ELEMENT is more expensive than that of MA42. This is because it implements the hybrid spectral-Sloan algorithm, which is more expensive than the Sloan algorithms used by MC63 (see Reference [12]). However, for large problems, the analyse cost is clearly a very small proportion of the total cost. In almost every example, the factorization and solve times are significantly less for the new code. The reasons for this are the better ordering, the use of a minimum pivot block greater than 1, and the exploitation of zeros in the front (although closer examination reveals that, once we have a good ordering, the reductions achieved by exploiting zeros in the front are small compared with the total factorization time). We remark

Table III. Timings (in seconds), the number of entries in the factors ($\times 10^6$) and flop counts ($\times 10^9$) for MA42_ELEMENT for different pivot block sizes.

	AFS times			Entries in factors			Flops		
Identifier	1	16	32	1	16	32	1	16	32
cham	9.7	3.2	3.1	8.3	8.5	8.7	3	3	3
crplat2	2.8	2.5	2.8	8.5	8.7	9.0	2	2	2
fcondp2	1098	714	777	607	610	613	1147	1155	1165
fullb	1271	810	899	705	712	718	1313	1330	1347
halfb	906	527	594	586	592	599	813	826	840
inv-ext-2	720	459	450	306	306	307	756	759	762
mt1	221	179	189	216	216	318	267	268	272
opt1	11	8.3	8.0	15	15	15	9	9	10
ship_001	18	14	15	31	31	32	14	15	15
ship_003	684	408	382	307	319	327	438	452	465
shipsec1	612	354	404	374	378	381	561	568	577
shipsec5	1323	1331	1279	502	501	502	744	744	742
shipsec8	1754	1695	1753	423	423	422	853	844	844
srb1	14	12	14	34	35	36	11	12	12
thread	60	55	54	620	623	627	72	72	74
trdheim	0.9	1.0	1.1	4.4	4.4	4.7	0.6	0.6	0.7
troll	6352	3381	3242	1339	1342	1346	5500	5518	5540
tsyl201	10	10	10	21	21	21	11	11	11
tubu	37	10	10	22	22	23	10	11	11
x104	792	1039	984	239	253	248	368	430	408

that since its first inclusion within HSL, additional options have been made available in MA42. These include options for a pivot block of greater than 1 and for exploiting zeros in the front. Factorization times for MA42 with a pivot block of 16 and exploiting zeros in the front are given in parentheses in the MA42 factorize column of Table II. It is clear that, with these options, the performance of MA42 can be significantly enhanced (for example, fcondp2, fullb, and troll), but for some problems (including shipsec5 and shipsec8), the performance is worse and, in general, MA42_ELEMENT is still considerably faster.

In Table III, we compare running MA42_ELEMENT with a minimum pivot block pivot_size of 1 with using pivot_size = 16 and 32. The reported times are for analyse plus factorize plus solve for one right-hand side (AFS). The 'flop' counts are the number of floating-point operations in the inner-most loop of the factorization. For many problems, including halfb, shipsec1 and troll, there are substantial savings in time if pivot_size is greater than 1. The increases in the number of entries in the factors and the number of flops are generally small (typically less than 3%). On our test machine there is no consistent advantage is using pivot_size = 32 rather than 16; for some problems, 32 gives the faster time while for others the converse is true. Based on our findings and the previous results reported in Reference [9], 16 has been selected as the default within MA42_ELEMENT. We note that the only problem that is significantly slower using pivot_size > 1 is x104. For this problem, many pivots are delayed and, in this case, using a larger pivot block leads to a significant increase in the maximum frontsize, from 2417 for pivot_size = 1 to 2969 for pivot_size = 16, and this in turn leads to an increase of 17% in the number of flops.

8. CONCLUDING COMMENTS AND SOFTWARE AVAILABILITY

We have described the design and development of a new Fortran 95 frontal solver HSL_MA42_ELEMENT for finite-element applications. The code builds on the extensive experience we have of frontal software development and, for element problems, offers a replacement for our existing frontal codes MA42 and ME42. We have retained a reverse communication interface because of the flexibility it offers but we also now offer a simpler all-in-one interface that should appeal to inexperienced users. Incorporating element ordering within the package also simplifies its use. A number of options for holding the element matrices and/or the computed factors out-of-core are offered to allow very large problems to be solved on machines with limited main memory. The out-of-core facilities are simple to use; in fact, the user may only be aware that files have been used to hold the factors through a warning flag.

HSL_MA42_ELEMENT is included in the most recent release of the software library HSL (HSL 2004). Use of the package requires a licence; full details of how to obtain a licence may be found at www.cse.clrc.ac.uk/nag/hsl/hsl.shtml.

ACKNOWLEDGEMENTS

This work was supported by the EPSRC grant GR/S42170.

REFERENCES

1. Irons BM. A frontal solution program for finite-element analysis. *International Journal for Numerical Methods in Engineering* 1970; **2**:5–32.
2. Hood P. Frontal solution program for unsymmetric matrices. *International Journal for Numerical Methods in Engineering* 1976; **10**:379–400.
3. Duff IS. MA32—a package for solving sparse unsymmetric systems using the frontal method. *Report AERE R10079*, Her Majesty's Stationery Office, London, 1981.
4. Duff IS. Enhancements to the MA32 package for solving sparse unsymmetric equations. *Report AERE R11009*, Her Majesty's Stationery Office, London, 1983.
5. Duff IS. Design features of a frontal code for solving sparse unsymmetric linear systems out-of-core. *SIAM Journal on Scientific and Statistical Computing* 1984; **5**:270–280.
6. Duff IS. Scott JA. MA42—a new frontal code for solving sparse unsymmetric systems. *Technical Report RAL-93-064*, Rutherford Appleton Laboratory, 1993.
7. Duff IS, Scott JA. The design of a new frontal code for solving sparse unsymmetric systems. *ACM Transactions on Mathematical Software* 1996; **22**(1):30–45.
8. HSL. A collection of Fortran codes for large-scale scientific computation, 2004. See http://hsl.rl.ac.uk/
9. Cliffe KA, Duff IS, Scott JA. Performance issues for frontal schemes on a cache-based high performance computer. *International Journal for Numerical Methods in Engineering* 1998; **42**:127–143.
10. Scott JA. Exploiting zeros in frontal solvers. *Technical Report RAL-TR-98-041*, Rutherford Appleton Laboratory, 1997.
11. Dongarra JJ, DuCroz J, Duff IS, Hammarling S. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software* 1990; **16**(1):1–17.
12. Scott JA. Multilevel hybrid spectral element ordering algorithms. *Communications in Numerical Methods in Engineering* 2005; **21**:233–245.
13. Sloan SW. An algorithm for profile and wavefront reduction of sparse matrices. *International Journal for Numerical Methods in Engineering* 1986; **23**:1315–1324.
14. Hu YF, Scott JA. Ordering techniques for singly bordered block diagonal forms for unsymmetric parallel sparse direct solvers. *Technical Report RAL-TR-2003-020*, Rutherford Appleton Laboratory, 2003.
15. Scott JA. On ordering elements for a frontal solver. *Communications in Numerical Methods in Engineering* 1999; **15**:309–323.
16. Scott JA. A parallel solver for finite element applications. *International Journal for Numerical Methods in Engineering* 2001; **50**:1131–1141.

