RAL-TR-1999-075

The design of a parallel frontal solver

by

Jennifer A. Scott

Abstract

In a previous report (Rutherford Technical Report RAL-94-040), Duff and Scott looked at extending the frontal method for finite-element problems to the multiple front method. This generalization of the frontal method permits coarse-grained parallelism to be exploited in the solution process. In this report, we discuss the design and development of a general purpose software package HSL_MP42 that implements the multiple front algorithm. HSL_MP42 solves unsymmetric systems of finite-element equations on distributed or shared memory parallel computers. The code is written in Fortran 90 and uses MPI for message passing. A key design feature is a user interface that is straightforward to use while offering flexibility through a number of options. Numerical results for both a model problem and a practical problem arising from groundwater flow calculations illustrate the efficiency of the code and the advantages of the multiple front approach.

Keywords: finite-elements, unsymmetric linear systems, frontal method, parallel processing, Fortran 90, MPI.

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

December 21, 1999.

Contents

1 Introduction

Finite-element simulations involve the solution of large sparse linear systems of equations. Solving these systems is generally the most computationally expensive step in the simulation, often requiring in excess of 90 per cent of the total run time. As time-dependent three-dimensional simulations are now commonplace, there is a need to develop algorithms and software that can be used to efficiently solve such problems on parallel supercomputers.

The frontal solver MA42 of Duff and Scott (1993, 1996) (and its predecessor MA32 of Duff, 1981) was developed for solving unsymmetric linear systems. The code can be used to solve general sparse systems but was primarily designed for finite-element problems. The code is included in the Harwell Subroutine Library (1995) and has been used in recent years to solve problems from a range of application areas. A key feature of the frontal method is that, in the innermost loop of the computation, dense linear algebra kernels can be used. In particular, these are able to exploit high-level BLAS kernels (Dongarra, DuCroz, Duff and Hammarling, 1990). This makes the method efficient on a wide range of modern computer architectures, including RISC based processors and vector machines. Although MA42 uses level-3 BLAS, it does not exploit the multiprocessing architecture of parallel supercomputers. In this paper, we report on the design and development of a new general-purpose parallel frontal code for unsymmetric finite-element problems. The code, which may be run on distributed or shared memory parallel computers, exploits both multiprocessing and vector processing by using a multiple front approach (Duff and Scott, 1994*a*, 1994*b*).

This paper is organised as follows. In Section 2, we provide some background information on the multiple front method. The design of our parallel frontal solver is discussed in Section 3. The new code is called HSL_MP42. We describe the separate phases of the code and in Section 4 we look at how the code has been developed using our existing frontal solver MA42 as the basic computational tool together with MPI for message passing. In Section 5, the user interface is explained and the use of HSL_MP42 is illustrated by a simple example code. Numerical results are presented in Section 6. Finally, in Section 7, we make some concluding remarks.

2 Multiple front method

In this section, we describe the multiple front method. Since the method is based on partitioning the finite-element domain into subdomains and applying the frontal method to each subdomain, we first recall the key features of the frontal method.

2.1 The frontal method

Consider the linear system

$$AX = B \tag{2.1}$$

where the $n \times n$ matrix A is large and sparse. B is an $n \times nrhs$ $(nrhs \geq 1)$ matrix of right-hand sides and X is the $n \times nrhs$ solution matrix. In this paper, we are only interested in the case where the matrix A is an elemental matrix, that is, A is a sum of finite-element matrices

$$A = \sum_{l=1}^{m} A^{(l)}, \tag{2.2}$$

where each element matrix $A^{(l)}$ has nonzeros only in a few rows and columns and corresponds to the matrix from element l. In practice, each $A^{(l)}$ is held in packed form as a small dense matrix together with a list of the variables that are associated with element l, which identifies where the entries belong in A. Each $A^{(l)}$ 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. Frontal schemes have their origins in the work of Irons (1970) and the basis for our experience with them is discussed by Duff (1984). The method is a variant of Gaussian elimination and involves the matrix factorization

$$A = PLUQ, \tag{2.3}$$

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

$$PLY = B, \tag{2.4}$$

followed by the back-substitution

$$UQX = Y. \tag{2.5}$$

The main feature of the frontal method is that the contributions $A^{(l)}$ 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 relatively small *frontal matrix* that can be written as

$$\begin{pmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{pmatrix}, \tag{2.6}$$

where the rows and columns of F_{11} are *fully summed*, that is, there are no other entries in these rows and columns in the overall matrix. Provided stable pivots can be chosen from F_{11}, F_{11} is factorized, multipliers are stored over F_{12} and the Schur complement $F_{22} - F_{21}F_{11}^{-1}F_{12}$ is formed. At the next stage, another element matrix is assembled with this Schur complement to form another frontal matrix.

The most expensive part of the computation is the formation of the Schur complement . Since the frontal matrix is held as a dense matrix, dense linear algebra kernels can be used, and it is this that allows the frontal method to perform at high Megaflop rates (see, for example, Duff and Scott, 1994*a*).

By holding the matrix factors on disk (for example, in direct-access files), the frontal method may be implemented using only a small amount of main memory. The amount of memory required is dependent on the size of the largest frontal matrix. The number of floating-point operations and the storage requirements for the matrix factors are also dependent on the size of the frontal matrix at each stage of the computation. Since the size of the frontal matrix increases when a variable enters the frontal matrix for the first time and decreases whenever a variable is eliminated, the order in which the element matrices are assembled is crucial for efficiency. A number of element ordering algorithms have been proposed, many of which are similar to those for bandwidth reduction of assembled matrices (see, for example, Duff, Reid and Scott, 1989, and the references therein).

2.2 Multiple fronts

A major deficiency of the frontal solution scheme is the lack of scope for parallelism other than that which can be obtained within the high-level BLAS. To circumvent this shortcoming, Duff and Scott (1994*a*) proposed allowing a (small) number of independent fronts in a somewhat similar fashion to Benner, Montry and Weigand (1987) and Zang and Liu (1991) (see also Zone and Keunings, 1991 and, for non-element problems, Medeiros, Pimenta and Goldenberg, 1997).

In the 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

$$\begin{pmatrix} A_{11} & & & & C_1 \\ & A_{22} & & & C_2 \\ & & \cdots & & . \\ & & & A_{NN} & C_N \\ \tilde{C}_1 & \tilde{C}_2 & \cdots & \tilde{C}_N & \sum_{i=1}^{N} E_i \end{pmatrix}, \tag{2.7}$$

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

$$\begin{pmatrix} A_{ii} & C_i \\ \tilde{C}_i & E_i \end{pmatrix}. \tag{2.8}$$

This can be done in parallel. At the end of the assembly and elimination processes for each subdomain Ω_i, there will remain $1 \le k_i \le k$ interface variables. These variables cannot be eliminated since they are shared by more than one subdomain. In practice, there will also remain variables that were not eliminated within the subdomain because of efficiency or stability considerations. These variables are added to the border and k is increased. If F_i is the local Schur complement for subdomain Ω_i (that is, 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

$$A = P \begin{pmatrix} L_1 & & & & \\ & L_2 & & & \\ & & \dots & & . \\ & & & L_N & \\ \tilde{L}_1 & \tilde{L}_2 & \dots & \tilde{L}_N & I \end{pmatrix} \begin{pmatrix} U_1 & & & & \tilde{U}_1 \\ & U_2 & & & \tilde{U}_2 \\ & & \dots & & . \\ & & & U_N & \tilde{U}_N \\ & & \dots & & F \end{pmatrix} Q, \qquad (2.9)$$

where the $k \times k$ matrix

$$F = \sum_{i=1}^{N} F_i \qquad (2.10)$$

is termed the *interface matrix.* The interface matrix F 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.

In general, as the number of subdomains increases, so does the number of interface variables and, as a result, the cost of solving the interface problem rises and becomes a more significant part of the total computational cost. If the interface problem is solved using a single processor, this can constitute a serious computational bottleneck. Thus it is crucial to keep the size of the interface problem small to achieve good speedups for the overall solution process. It may be possible to alleviate the problem by nesting the multiple front approach, so that only small groups of subdomains are combined at each level. For example, if the domain is partitioned into 2^r subdomains, the subdomains may be combined 2 at a time. In this case, the multiple front algorithm would be nested to r levels. The success of the nested multiple front approach depends on being able to combine subdomains that have a large number of their interface variables in common. The logical conclusion of this approach is the so-called multifrontal algorithm (see, Duff and Reid, 1983), whereby a multitude of fronts is started simultaneously, each front eliminating its variables as soon as possible. Alternatively, depending on the size and sparsity of the interface matrix F, a solver other than a frontal solver might be more attractive for performing its factorization.

In this paper, we do not consider nesting the multiple front algorithm and currently we use the frontal method for the interface problem. However, our multiple front code is written using basic building blocks (see Section 4) and the intention is that this will allow us to investigate using other interface solvers at a later date.

3 Design of a parallel frontal solver

In this section, we consider the overall design of our multiple front solver `HSL_MP42`. The code is written in Fortran 90. `MPI` is used for message passing. `MPI` was chosen since the `MPI` Standard is internationally recognised and today `MPI` is widely available and accepted by users of parallel computers. The code requires a designated host process. The host performs the initial analysis of the data, distributes data to the remaining processes, collects the

Schur complements, solves the interface problem, and generally overseas the computation. With the other processes, the host also participates in subdomain calculations. For portability, it is **not** assumed that there is a single file system that can be accessed by all the process. This allows the code to be used on distributed memory parallel computers as well as on shared memory machines.

The module `HSL_MP42` is designed with five separate phases as follows:

1. **Initialize**
2. **Analyse**
3. **Factorize**
4. **Solve**
5. **Finalize**

The user must first initialize `MPI` by calling `MPI_INIT` on each process. The user must also define an `MPI` communicator for the package. The communicator defines the set of processes to be used by `HSL_MP42`. Each phase must then be called in order by each process (although the solve phase is optional). Before calling the next phase, the user must have completed all tasks with the defined communicator (and with any other communicator that overlaps the current `HSL_MP42` communicator). During the factorize phase, the matrix factors are generated. The user may optionally hold the matrix factors in direct-access files. This allows large problems to be solved in environments were each process has only a limited amount of main memory available. If element right-hand side matrices are supplied (that is, right-hand sides of the form $B = \sum_{l=1}^{m} B^{(l)}$), the solution X is returned to the user at the end of the factorize phase. If the right-hand sides are only available in assembled form, or if the user wishes to use the matrix factors generated by the factorize phase to solve for further right-hand sides, the solve phase should be called. The user may factorize more than one matrix at the same time by running more than one instance of the package; an instance of the package is terminated by calling the finalize phase. After the finalize phase and once the user has completed any other calls to `MPI` routines he or she wishes to make, the user should call `MPI_FINALIZE` to terminate the use of `MPI`. This is illustrated in the simple example code given in Figure 5.1 in Section 5.5. We now discuss each phase in further detail.

3.1 Initialize

This phase initializes an instance of the package. The code first calls `MPI_INITIALIZED` to check that `MPI` has been initialized by the user. `MPI_COMM_SIZE` is then called to determine the number of processes being used and `MPI_COMM_RANK` determines the rank of each process. The host is the process with rank zero. The control parameters are then initialized. These parameters control the action, including how the user wishes to supply the element data and whether or not direct-access files are to be used to minimise main memory requirements. If the user wishes to use values other than the defaults, the appropriate parameters should be reset on the host after the initialize phase and prior to calling the analyse phase. Full details of the control parameters are given in the user documentation for HSL_MP42 (see Appendix).

3.2 Analyse

The host first broadcasts the control parameters to the other processes. On the host, the user must supply, for each subdomain Ω_i, a list of its elements and, for each element, a list of its variables. The host checks this data and uses it to generate a list of the interface variables for each subdomain. If requested, the host also orders the elements within Ω_i. This is the order in which the elements will be assembled when the frontal solver is applied to Ω_i. An estimate of the floating-point operation (flop) count for each subdomain is made (assuming no stability restrictions on the pivot choice) and broadcast to the other processes. Unless the user wishes to choose which process factorizes which of the subdomain matrices, this estimate is used to assign the subdomains to processes. The subdomains are divided up between the processes so that, as far as possible, the loads (in terms of flops) are balanced.

3.3 Factorize

It is convenient to subdivide this phase.

- Distribution of data. The host uses the results of the analyse phase to send data to the other processes. For each process, the host only sends data for the subdomains that have been assigned to that process. The amount of data movement depends upon how the user has chosen to input the element data. By default, the element matrices $A^{(l)}$ (and the element right-hand side matrices $B^{(l)}$) required by a process are read by that process from a direct-access file one at a time as they are required. This minimises storage requirements and data movement. Alternatively, the user may supply the element data (and element right-hand sides) in unformatted sequential files so that the data required by a process is again be read by that process. If this option is used, the data for all the elements in a subdomain is read in at once, so more memory is required but, again, movement of data between processes is minimised. Options also exist for the host to read the element data for each of the subdomains from unformatted sequential files or, alternatively, the user may supply the element matrices as input arrays on the host. The later form of input is useful if the host has sufficient memory and the overhead for using direct-access or sequential files is high. If the data is input onto the host, there is an added overhead of sending the appropriate data from the host to the other processes. Since the host is also involved in the subdomain factorizations, this distribution of element data is done before the factorization commences.

- Subdomain factorization. The processes use MA42 to perform a frontal decomposition for each of the subdomains assigned to them. Threshold pivoting is used to maintain stability. The stability threshold is one of the control parameters (with default value 0.01). Once all possible eliminations have been performed, the integer data for the local Schur complement is sent to the host. Because, in general, MA42 uses off-diagonal pivoting, both the row and column indices of the local Schur complement need to be stored and sent to the host. We will refer to these indices as the *row and column Schur indices.* If the user wants to minimise main memory requirements, the processes write the reals for the local Schur complements (and corresponding right-hands sides) to sequential files.

- Interface factorization and solve. Treating the local Schur complements as element matrices, the host orders the subdomains and performs a frontal solution of the interface problem. When the local Schur complement for a subdomain is required, it is (optionally) read from its direct-access file by the process to which it was assigned and sent to the host. If element right-hand sides were not supplied, the factorize phase is complete. Otherwise, the element right-hand sides are also read and sent as required to the host. Once the host has completed the frontal elimination, the solution for the interface variables is broadcast to the processes. We observe that data must be read by the individual processes and sent to the host because we do not assume there is a single file system accessible by the host.

  When using MA42 to solve the interface problem, the column Schur indices are entered as the variable indices. Because off-diagonal pivoting is used by MA42 in the subdomains, row permutations are needed to restore the interface variables to the diagonal of the local Schur complement. These permutations (together with corresponding permutations to the right-hand side matrices) are performed by the host prior to calling the frontal factorization routine MA42B.

- Back-substitution. The processes perform back-substitution on their assigned subdomains. The solution is then assembled on the host. Information on the frontal eliminations (including the numbers of entries in the factors, the integer storage required by the factors, and the numbers of static condensations performed) is also sent to the host (see Section 5.3 for details).

3.4 Solve

The solve phase is optional and should be called if the right-hand sides were not available to the factorize phase in element form or if the user wants to use the factors to solve for further right-hand sides. The right-hand side vectors, which must be input to the host in assembled form, are broadcast from the host to the other processes. Forward elimination on the subdomains is performed by the processes. Once complete, the processes send their contributions to the Schur rows of the modified right-hand side matrices to the host. The host assembles these contributions and uses the factors for the interface problem to solve for the interface variables. The solution for the interface variables is broadcast to the processes, which then perform back-substitution on the subdomains. The final solution is assembled on the host. Note that any number of solves may follow the factorize phase but it is more efficient to solve for several right-hand sides at once because this allows better advantage to be taken of high-level BLAS.

3.5 Finalize

All arrays that have been allocated by the package for the current instance are deallocated and, optionally, all direct-access files used to hold the matrix factors for the current instance are deleted. This phase should be called after all other calls for the current instance of the package are complete.

4 Basic building blocks

Significant effort has been invested in making the frontal solver `MA42` both efficient and robust. We would like to use `MA42` as the frontal solver, both on the subdomains and for the interface problem. In our earlier report (Duff and Scott, 1994*b*), we described the design and development of a separate package `MA52` that can be used with `MA42` to implement the multiple front algorithm. Together with element ordering routines, `MA42` and `MA52` provide us with the basic building blocks needed to develop our parallel multiple front code. In the following sections, we briefly discuss these building blocks, highlighting the key features used by the multiple front code.

4.1 The frontal solver `MA42`

As with most sparse direct solvers, `MA42` has three distinct phases.

- A prepass phase (`MA42A`) that uses the integer data to determine at which assembly step each variable is fully summed. `MA42A` must be called once for each element; on each call, the integer data for the next element in the assembly order must be supplied.

- A factorization phase (`MA42B`) that factorizes the matrix into its upper and lower triangular factors using Gaussian elimination. Again, `MA42B` must be called once for each element, specifying the integer and element data for the next element in the assembly order. If right-hand side vectors B are included in the calls to `MA42B`, forward elimination on these right-hand sides is performed at the same time as the matrix factorization and, once the factorization is complete, `MA42B` calls a private subroutine (`MA42D`) to perform the back-substitution.

- A solution phase (`MA42C`) that solves for further right-hand sides. This phase is optional. `MA42C` calls a private subroutine `MA42E` to perform the forward elimination on the right-hand sides and then a private subroutine `MA42D` to perform the back-substitution and complete the solution.

A key feature of MA42 is that, by accepting the element one at a time, large problems can be solved using a relatively small amount of main memory. During the factorization, data is put into explicitly-held buffers and, whenever these buffers are full, they are optionally written to direct-access files. For further details of the MA42 package, including user documentation, the reader is referred to Duff and Scott (1993). The current Fortran 77 version of MA42 requires the user to choose stream numbers for the direct-access files but does not allow the user to name them (the Fortran 90 version allows named files). For a multiple front code, we decided the user needed the option of naming the files. This makes it straightforward for the user to retain the files for further computation and, for a distributed memory machine, enables the files to be written to a local disk. Our experience has been that this can be vital for efficiency (see Duff and Scott, 1994*a*). In MA42, the use of direct-access files is controlled by a separate user-called routine MA42P. In our multiple front code, this can be replaced by a similar subroutine that includes the direct-access file names in its argument list.

4.2 The MA52 package

The MA52 package comprises three simple routines that were primarily designed to allow MA42 to be run on subdomains.

4.2.1 MA52A

When applying a frontal solver to a subdomain Ω_i, the elimination of interface variables during the factorization phase has to be avoided. This can be achieved quite simply by making a final call to the prepass routine MA42A with an extra element containing all the interface variables. The extra element is termed the *guard element*; one guard element is associated with each subdomain. The guard elements are generated by MA52A.

4.2.2 MA52B and MA52F

When the last finite element is passed to the factorization routine MA42B, it is assembled into the frontal matrix and any fully summed variables that satisfy the stability criteria are eliminated. For a single domain problem, all the variables are fully summed at this stage and can be eliminated. If direct-access files are being used, MA42B writes the remaining contents of the buffers to the files being used to hold the matrix factors. If right-hand sides have been entered, forward elimination is performed at the same time as the factors are computed and only the private subroutine MA42D is needed for the back-substitution that completes the solution.

If the domain is partitioned into subdomains, after the assembly and eliminations for the final element in a subdomain, the interface variables remain in the front. In addition, there remain any variables not eliminated because of stability considerations plus any not chosen because the number of potential pivots was less than the minimum required by MA42 (see Cliffe, Duff and Scott, 1998 and Section 6.3). The frontal matrix and corresponding frontal right-hand side vectors are held within the work arrays used by MA42B. Subroutine MA52F is designed to preserve the partial factorization by extracting the frontal matrix and right-hand sides from these arrays and returning them in the form of an element matrix and element right-hand side, together with the row and column Schur indices. If direct-access files are being used by MA42, MA52F also writes any data left in the buffers to these files. If diagonal pivoting is used by MA42, the row and column Schur indices are identical. In this case, MA52B may be used to preserve the partial factorization. MA52B has the same interface as MA52F except that it returns only one set of indices.

4.2.3 MA52C

When designing the frontal MA42 code, we decided that the routines to perform forward elimination and back-substitution, namely MA42E and MA42D, should be private subroutines. This simplified the user interface because it reduced the number of subroutine calls the user was required to make. However, in the multiple front algorithm, the forward elimination and back-substitution need to be performed separately. Subroutine MA52C was written to provide an interface to MA42D and MA42E. The input parameter IND determines whether forward elimination or back-substitution is performed and, for back-substitution, whether the call follows a call to the factorization routine MA42B or to the solve routine MA42C for the interface problem (this distinction is needed because, during the factorization phase, the partial solution vectors are stored with the U factor). In the multiple front code, MA52C needs to be called for each subdomain but the calls with the same value of IND may be made in parallel.

4.3 Element ordering

As mentioned in Section 2.1, the efficiency of the frontal method is highly dependent on the order in which the elements are assembled into the frontal matrix. The ordering of elements within a subdomain was discussed by Scott (1999). This problem is more complicated than the problem of resequencing elements for a frontal solver on a single domain since it is necessary to distinguish between variables that are fully summed within the subdomain and the interface variables that are not. Once interface variables enter the front, they remain there, increasing both the amount of work required at each elimination and the storage for the frontal matrix and for the factors. The approach to element ordering in a subdomain adopted by Scott has its origins in the profile reduction algorithm of Sloan (1986). It uses the element connectivity graph representation of the subdomain. Using the graph and the guard element, an element lying as far from the interface boundary as possible is found. This element is used as the first element in the reordering and the guard element g is numbered last. The remaining elements are ordered using a priority function. The basic idea is to select the next element in the ordering by choosing, from a set of *eligible* elements, an element with maximum priority. The set of eligible elements comprises the neighbours of elements that have already been reordered and their neighbours. The priority P_i of element i given by

$$P_i = -W_1 * ngain(i) + W_2 * d(g,i) - W_3 * nadj(i), \tag{4.1}$$

where W_j, $j = 1, 2, 3$ are integer weights. The quantity $ngain(i)$ is the number of variables element i will introduce into the frontal matrix less the number of variables that may then be eliminated if it is assembled next. Since interface variables cannot be eliminated within a subdomain, $ngain$ will be large for elements containing a large number of new interface variables, giving such elements a low priority. $d(g,i)$ is the distance in the graph of element i from the guard element, and $nadj(i)$ is the number of elements adjacent to element i that have not yet been reordered. The aim of (4.1) is to give a high priority to elements that are a long way from the interface boundary, that keep the number of variables in the frontal matrix small, and keep the number of unordered adjacent elements small. A code MC53 implementing this approach is available in the Harwell Subroutine Library.

When using the frontal solver on the interface problem, it can be worthwhile to order the subdomains. In this case, an algorithm for element ordering for a single domain problem is needed. A number of algorithms for automatically ordering finite elements have been proposed in the literature (see Scott, 1999 for references). We have recently developed a new element ordering code MC63 for inclusion in the Harwell Subroutine Library (Scott, 1999). Numerical experiments reported by Scott have shown that this algorithm produces good orderings on a wide range of practical applications and as a result we will use the code MC63 for the interface problem in our parallel frontal solver.

5 User interface

A key design aim for our parallel frontal solver HSL_MP42 was to develop a user interface that is straightforward and, at the same time, offers flexibility through a variety of options. In the following, we discuss how the user interface has been written to allow the code to be used by those who have only a basic knowledge of MPI together with some experience of Fortran 90 programming. If a user was to write his or her own implementation of the multiple front algorithm using the basic building blocks discussed in the previous section, then he or she would need to possess quite an advanced knowledge of both Fortran and MPI.

An important early decision was not to include software to partition the domain Ω into subdomains within the package. Instead, the user must perform some preprocessing and must make lists of the elements belonging to each of the subdomains Ω_i available on the host (see below). This decision was made because the choice of a good partitioning is very problem dependent. Moreover, in many practical applications, a natural partitioning depending on the underlying geometry or physical properties of the problem is often available (for example, for a finite element model of an aircraft, it may be appropriate to consider the fuselage as one or more subdomains and the wings as two further subdomains). When no such partitioning is available, the user is advised to use a graph partitioning code, a number of which are now available in the public domain, including Chaco (Hendrickson and Leland, 1995) or METIS (http://winter.cs.umn.edu/~karypis/metis/).

HSL_MP42 has a single user-callable subroutine MP42A with a single parameter data of Fortran 90 derived datatype MP42_DATA, viz

```
TYPE (MP42_DATA) ::  data

CALL MP42A (data)
```

The derived datatype has many components, only some of which are of interest to the user. The other components are private. Full descriptions of the parameters of interest to the user are given in the HSL_MP42 user documentation (see Appendix). The user may factorize more than one matrix by running more than one instance of the package at once. In this case, a separate parameter of type MP42_DATA is needed for each instance. We remark that many components of the derived datatype are pointers but we use these pointers as if they were allocatable arrays. The current Fortran Standard does not permit components of derived datatypes to be allocatable arrays; when it does, we will replace all the pointers used by HSL_MP42 by allocatable arrays.

Workspace is allocated by the code as required. The amount of workspace needed is dependent upon how the element matrices are stored, on the length of the buffers used by MA42, and on whether the variables are numbered contiguously. More workspace is needed if variables are not numbered contiguously.

Prior to the first call to MP42A, the user must initialize MPI by calling MPI_INIT on each process. The user must also define an MPI communicator data%COMM for the package. data%COMM defines the set of processes to be used by HSL_MP42. Before each phase of the package is called, the user must have completed all tasks involving data%COMM (or involving any other communicator that overlaps data%COMM). Note that the code may be run using a single process; results illustrating this are given in Section 6.2.

The "job" parameter data%JOB determines which phase of the package is to be performed. It must be initialized to the **same** value on each process before each call to MP42A (a check for this is made within the code). The values 1 to 5 correspond to the five phases discussed in Section 3. A call with data%JOB = 3 must be preceded by a call with data%JOB = 2, which in turn must be preceded by a call with data%JOB = 1. Likewise, a call with data%JOB = 4 must be preceded by a call with data%JOB = 3, but several calls with data%JOB = 4 may follow a single call with data%JOB = 3. A call with data%JOB = 5 should be made after all other calls for the current instance are complete. The user may combine a call with data%JOB = 2 followed by a call with data%JOB = 3 by setting data%JOB = 23.

We now discuss the parameters that must be set by the user for each phase of the computation and the output returned to the user.

5.1 Initialize: data%JOB = 1

On a call to the initialize phase (data%JOB = 1), the user is not required to supply any components of data apart from the job parameter and the communicator. On exit, the arrays `data%ICNTL` and `data%CNTL` that control the action contain default values (see Appendix). If the user wishes to use values other than the defaults, the corresponding components of `data` should be reset before calling the analyse phase. In addition, on each process the following components of `data` are set by calls to the MPI routines `MPI_COMM_RANK` and `MPI_COMM_SIZE` during the initialize phase:

`data%RANK` holds the rank of the process in the communicator `data%COMM`. The host is defined to the process with rank 0.

`data%NPROC` holds the number of processes associated with the communicator `data%COMM`.

5.2 Analyse: `data%JOB = 2`

On a call to the analyse phase (`data%JOB = 2`), the user must specify the following integer data on the host process:

`data%NDOM`, the number of subdomains. This must be at least 2 (since if there is a single subdomain the user should use the frontal code `MA42`). In addition, the number of subdomains must be at least as large as `data%NPROC` (so that each process is responsible for at least one subdomain).

`data%NELT`, the total number of finite elements in the problem. Since we require that each subdomain has at least one element, `data%NELT` must be at least as large as `data%NDOM`.

`data%NELTSB` is a rank 1 pointer that must be allocated with size `data%NDOM`. On entry, `data%NELTSB(JDOM)` must be set by the user to hold the number of elements in subdomain $\Omega_{\mathtt{JDOM}}$ (`JDOM = 1, 2,..., data%NDOM`).

`data%ELTVAR` is a rank 1 pointer that must be allocated by the user and set to contain lists of the variable indices belonging to each of the finite elements. The lists of the variables in each of the elements belonging to Ω_1 must precede those for the elements belonging to Ω_2, and so on. Duplicate indices and variable indices less than `1` are not permitted. The code checks for such entries and, if found, terminates the computation with an error message. Note that, unless the user does not want the elements to be automatically ordered by `MC53`, the order of the elements within each subdomain is unimportant.

`data%ELTPTR` is a rank 1 pointer that must be allocated with size `data%NELT+1`. `data%ELTPTR(IELT)` must contain the position in `data%ELTVAR` of the first variable in the `IELT`-th element in the element variable lists (`IELT = 1, 2,..., data%NELT`), and `data%ELTPTR(data%NELT+1)` must be set to the position after the last variable in the last element.

Each of the above parameters must be unaltered by the user after the analyse phase. By default, the code performs a symbolic factorization to determine an estimate of the number of flops required by each subdomain and uses these to decide which subdomains should be assigned to which process. However, an option exists for the user to provide this information on the host in `data%INV_LIST`. In this case, `data%INV_LIST` is unchanged on exit. Otherwise, on exit from a call with `data%JOB = 2`, `data%INV_LIST(JDOM)` holds the rank of the process subdomain $\Omega_{\texttt{JDOM}}$ is assigned to (`JDOM = 1, 2,..., data%NDOM`). The remaining output from the analyse phase is the following information (on the host process only):

`data%LARGEST_INDEX` holds the largest integer used to index a variable. The size of arrays used by `MP42` is dependent upon `data%LARGEST_INDEX` so it is advisable (but not necessary) to number the variables consecutively so that the largest integer is equal to the order of the system.

`data%MAX_NDF` holds the maximum number of variables per element.

`data%NGUARD(JDOM)` holds the number of interface variables for subdomain `JDOM`.

`data%FLAG_52(JDOM)` holds the `MA52A` error flag for subdomain $\Omega_{\texttt{JDOM}}$.

`data%NDF(JDOM)` holds the number of variables in subdomain $\Omega_{\texttt{JDOM}}$.

`data%NFRONT(JDOM)` holds the maximum frontsize for the frontal elimination on subdomain $\Omega_{\texttt{JDOM}}$.

`data%RMS(JDOM)` holds the root mean squared frontsize for the frontal elimination on subdomain $\Omega_{\texttt{JDOM}}$.

`data%OPS(JDOM)` holds the number of floating-point operations in the innermost loops of the frontal elimination on subdomain `JDOM`.

`data%FLSIZE(J, JDOM)`, `J = 1, 2` hold, respectively, the real and integer storage needed for the matrix factors on subdomain $\Omega_{\texttt{JDOM}}$. Note that the storage for the U and L factors is equal but during the factorization, the modified right-hand sides are stored with the U factor and so the total storage for the U factor and the right-hand sides on subdomain $\Omega_{\texttt{JDOM}}$ is $\texttt{data\%FLSIZE(1,JDOM)}/2+\texttt{nrhs}*(\texttt{data\%NDF(JDOM)}-\texttt{data\%NGUARD(JDOM)})$, where `nrhs` is the number of right-hand sides.

`data%NORDER` holds the order in which the elements will be assembled during the factorization phase. Within subdomain $\Omega_{\texttt{JDOM}}$, the elements are locally labelled `1, 2,..., data%NELTSB(JDOM)`, according to the order in which the user inputs the variable lists in `data%ELTVAR`. On subdomain $\Omega_{\texttt{JDOM}}$, the elements will be assembled during the factorize phase in the order `data%NORDER(1, JDOM)`, `data%NORDER(2, JDOM)`,..., `data%NORDER(data%NELTSB(JDOM), JDOM)`.

Note that the statistics computed during the analyse phase are based solely on integer data. During the factorization, pivots may be delayed because of stability considerations and the actual frontsizes may, therefore, be larger than those computed with data%JOB = 2. This will also increase the flop count and factor storage. However, the analyse phase does not allow for the possible exploitation of zeros within the frontal matrix, which can reduce the flops and factor storage. Thus the analyse phase actually returns a lower bound on the frontsizes for the factorize phase and an estimate of the flop count and factor storage. These estimates enable the user to determine approximately the cost of the multiple front method. In addition, the user can assess how well balanced the computation is. Since the analyse phase is much cheaper to perform than the subsequent factorization phase, the user may decide to make modifications to the subdomains and rerun with the parameter data%JOB = 2 before proceeding to the factorization. For this reason, the analyse and factorize phases are separate (but may be combined in a single call by setting data%JOB = 23.

5.3 Factorize: data%JOB = 3

On a call to the factorize phase (data%JOB = 3), the user must specify on the host:

data%NRHS, the number of right-hand sides to be solved. If the user wishes to perform the factorization but not do any solves (for example, if the right-hand sides are only available in assembled form), data%NRHS should be set to zero.

The remaining components of the derived dataype data that must be set by the user for the factorization phase depend upon how the user wishes to input the element data. There are currently four options available. In each case, for each subdomain the element matrices $A^{(l)}$ (and, if data%NRHS $>$ 0, the element right-hand side matrices $B^{(l)}$) corresponding to the elements within that subdomain must be held as full matrices, ordered by columns. We now discuss how the element data may be input.

Option 1 (default)

The default is for the data required by a process to be read by that process from a direct-access file element-by-element as it is required by MA42. This minimises memory requirements and data movement between processes. For each subdomain assigned to a process, the user must set up an unformatted direct-access file, each record of which holds the values of the entries of an element matrix and, if data%NRHS $>$ 0, another file holding the values of the entries in the element right-hand side matrices. These files must be able to be read by the process responsible for the subdomain. The element matrices and element right-hand side matrices must be in the same order as in data%ELTVAR. The user must set the parameter data%MFRELT to be at least as large as data%MAX_NDF, the maximum number of variables per element, and the lengths of the records in the direct-access files holding the element matrices and element right-hand sides must be that required for real arrays of size data%MFRELT$**$2, and data%MFRELT $*$ data%NRHS, respectively. The names of the files must be supplied on the host in the arrays data%VALNAM and data%RHSNAM.

Option 2
The second option is for the matrix data for each subdomain to be in an unformatted sequential file that can be read by the process to which it is assigned. The file length will be less than for option 1, but additional memory is required because all the elements for a subdomain are read in at once. Data movement is again minimised. If data%NRHS $>$ 0, a corresponding unformatted sequential file holding the element right-hand side matrices must be set up.

Option 3

The third option is for the data for each subdomain to be read by the host process in turn and then sent to the other processes as required. In this case, for each subdomain, the user must set up on the host an unformatted sequential file holding the values of the entries in the element matrices and, if data%NRHS $>$ 0, another holding the element right-hand side matrices. This option is useful if the user finds it convenient to generate the element data on the host and the host has a sufficiently large file system to hold all the data.

Option 4
The last option requires the user to supply the element and element right-hand side matrices in memory on the host using pointers data%VALUES and data%RHSVAL. Data for each subdomain is sent to the other processes as required before the factorization commences. This option is suitable for smaller problems. It may also be useful for larger problems in a shared memory environment, particularly if the overhead of reading data from direct-access or sequential files is high.

In addition to choosing how to supply the element data, the user must decide whether or not to hold the matrix factors in direct-access files. The default is not to use direct-access files. In this case, each process must have sufficient memory to hold the factors for all the subdomains assigned to it. If files are used, the amount of memory each process needs in order to run MP42 successfully is essentially only that required to hold the frontal matrix on the subdomain (together with right-hand side vectors), allowing much larger problems to be solved. The penalty is the additional I/O cost for reading and writing data to the files.

When used, names for the direct-access files may be supplied on the host in data%FILES1. This is a rank-2 pointer of type default CHARACTER$*$128 that must be allocated by the user with size 3 by data%NDOM. On entry, for subdomain $\Omega_{\mathtt{JDOM}}$, data%FILES1(J, JDOM), J = 1, 2, 3, must hold the names (the full path name) of the direct-access files for the U factor, the L factor, and the integer data for the factors, respectively. Sequential files may be used to hold what remains in the front at the end of the subdomain factorization. If used, names may be supplied in data%FILES2. Otherwise, the code automatically names the files, which are written to the current directory. Note that to avoid name clashes, if the user wishes to run further instances of the package before the final call for the current instance (JOB = 5), for each instance files **must** be used and unique names **must** be provided by the user.

If data%NRHS$>$ 0, the main output from the factorize phase is data%X, a rank-2 pointer of size data%LARGEST_INDEX by data%NRHS. If I has been used to index a variable, on the host process data%X(I, J) holds the solution for variable I to the J-th system and is set to zero otherwise (J = 1, 2,..., data%NRHS). In addition, the following information is returned on the host:

data%STATIC holds the total number of static condensations performed.

data%SINGULAR is a logical variable that is set to .TRUE. if the matrix is found to be singular. In this case, the user has the option of continuing the computation.

data%FLOPS holds the total number of floating-point operations in the innermost loops of the factorization (this is the total for all the subdomains and the interface).

data%NZL holds the total number of entries in the L factors.

data%NZU holds the total number of entries in the U factors, plus the right-hand sides (if data%NRHS = 0, data%NZU = data%NZL).

data%STORINT holds the total storage for the row and column indices in integer words.

data%NLEFT(JDOM) holds the number of variables left in the front at the end of the elimination process for subdomain $\Omega_{\mathtt{JDOM}}$ (JDOM = 1, 2,..., data%NDOM). This is equal to the number of interface variables data%NGUARD(JDOM) plus the number of variables not eliminated either for stability reasons or because the number of fully summed variables was less than the minimum pivot block size (given by the control parameter data%ICNTL(16)).

data%INFO_42(:, JDOM) and data%RINFO_42(:, JDOM) hold the MA42B integer and real information arrays for subdomain $\Omega_{\mathtt{JDOM}}$ (JDOM = 1, 2,...,data%NDOM) and, for JDOM = data%NDOM+1, they hold the corresponding information for the interface problem.

data%INFO_63 and data%RINFO_63 hold the MC63 integer and real information arrays for the interface problem.

5.4 Solve: data%JOB = 4

For the solve phase, on the host the user must supply the number of right-hand sides data%NRHS and the rank-2 pointer data%B of size data%LARGEST_INDEX by data%NRHS must be allocated and set so that if I is used to index a variable, data%B(I, J) is the corresponding component of the right-hand side for the J-th system (J = 1, 2,..., data%NRHS). On exit, the solution is returned on the host in data%X.

5.5 Example of the use of MP42

In Figure 5.1, an example of the use of HSL_MP42 is given. Access to the package requires a USE statement and the MPI file mpif.h must be included. The user must perform the initialization and termination of MPI via calls on each process to MPI_INIT and MPI_FINALIZE.

All the calls to MP42A are made on each process. The package is initialized by calling MP42A with data%JOB = 1, the element data is read in by the host, and the solution is computed by a call with data%JOB = 23 (analyse and factorize phases are performed in turn). A call with data%JOB = 4 is made to solve for further right-hand sides. Finally, a call with data%JOB = 5 deallocates the data structures used by the instance of the package. Default settings are used for all control parameters,

6 Numerical results

In this section, we illustrate the performance of MP42, first on a model problem and then on three problems arising from groundwater flow calculations. The experiments in Sections 6.1 and 6.2 were performed on the SGI Origin 2000 and Cray T3E at Manchester, and those in Section 6.3 were carried out on the SGI Origin 2000 located at Parallab (Bergen).

6.1 Model problem

We first present results for a model problem designed to simulate those actually occurring in some CFD calculations. The elements are 9-noded rectangular elements with nodes at the corners, mid-points of the sides, and centre of the element. A parameter to the element generation routine determines the number of variables per node. This parameter has been set to 5 for the numerical experiments reported in this section. In Tables 6.1 to 6.3, we present results for a square domain subdivided into 4 and 8 subdomains. MP42 is run using 1, 2, 4 and, in the case of the 8 subdomain problem, 8 processes. For the 4 subdomain problem, the subdomains are all square and of equal size. For the 8 subdomain problem, we use a grid of 4 × 2 subdomains and, in this case, to achieve good load balancing, we use subdomains of unequal size (see Duff and Scott, 1994*b*). The "corner" subdomains with 2 interface boundaries are of size 15 × 24 elements and 30 × 48 elements for the problems of order 48 × 48 and 96 × 96, respectively, and the remaining subdomains, which have 3 interface boundaries, are of size 9 × 24 and 18 × 48 elements, respectively. In each test, we solve for 2 right-hand sides. The element data is held in memory on the host (Option 4 in Section 5.3). On the Origin (a shared memory machine), the default values are used for all other control parameters (so that files are not used for the matrix factors); on the T3E, the matrix factors are

```
      PROGRAMM MP42_TEST
      USE HSL_MP42_DOUBLE
      INCLUDE 'mpif.h'
      TYPE (MP42_DATA) data
      INTEGER ERCODE
! Start MPI
      CALL MPI_INIT(ERCODE)
! Define a communicator for the package
      data%COMM = MPI_COMM_WORLD
! Initialize instance (default settings used)
      data%JOB = 1
      CALL MP42A (data)
! Read data on host
      IF (data%RANK .EQ. 0) THEN
         READ (5,*) data%NDOM, data%NRHS, data%NELT
! Read number of elements in each subdomain
         ALLOCATE ( data%NELTSB(1:data%NDOM) )
         READ (5,*) data%NELTSB(1:data%NDOM)
! Read element variable lists
         ALLOCATE ( data%ELTPTR(1:data%NELT+1) )
         READ (5,FMT=*) data%ELTPTR(1:data%NELT+1)
         NE = data%ELTPTR(data%NELT+1) - 1
         ALLOCATE ( data%ELTVAR(1:NE) )
         READ (5,FMT=*) data%ELTVAR(1:NE)
! Read in names of files holding real element data
         ALLOCATE ( data%VALNAM(1:data%NDOM) )
         ALLOCATE ( data%RHSNAM(1:data%NDOM) )
         READ (5,FMT=*) data%VALNAM(1:data%NDOM)
         READ (5,FMT=*) data%RHSNAM(1:data%NDOM)
      END IF

! Call analyse phase followed by factorize phase
      data%JOB = 23
      CALL MP42A (data)
      IF (data%RANK.EQ.0) WRITE (*,*)
     &  ' The solution is:', data%X(1:data%LARGEST_INDEX,1:data%NRHS)
! Solve for further right-hand sides.
      IF (data%RANK.EQ.0) THEN
         READ (5,*) data%NRHS
         ALLOCATE ( data%B(1:data%LARGEST_INDEX,1:data%NRHS) )
         READ (5,*) data%B(1:data%LARGEST_INDEX,1:data%NRHS)
      END IF
      data%JOB = 4
      CALL MP42A (data)
      IF (data%RANK.EQ.0) WRITE (*,*)
     &  ' The solution is:', data%X(1:data%LARGEST_INDEX,1:data%NRHS)
! Finalize
      data%JOB = 5
      CALL MP42A (data)
      CALL MPI_FINALIZE(ERCODE)
      STOP
      END PROGRAMM MP42_TEST
```

Figure 5.1: Example program using HSL_MP42.

Table 6.1: Wall clock timings (in seconds) for MP42 on the Origin 2000 for the model problem with 4 subdomains. The numbers in parentheses are the times taken to factor and solve the interface problem.

Dimension	No. of variables	No. of processes	Factor+ Solve		Speedup	Solve	Speedup
32 × 32	21,125	1	10.5	(1.0)	-	0.50	-
		2	6.2		1.7	0.35	1.4
		4	4.0		2.6	0.25	2.0
48 × 48	47,045	1	41	(3.2)	-	1.5	-
		2	23		1.8	1.0	1.5
		4	14		2.9	0.7	2.1
64 × 64	83,205	1	118	(7.2)	-	4.0	-
		2	67		1.8	2.8	1.4
		4	41		2.9	1.7	2.3
96 × 96	186,245	1	546	(27)	-	17.7	-
		2	304		1.8	10.2	1.7
		4	168		3.2	5.8	3.0

Table 6.2: Wall clock timings (in seconds) for MP42 on the T3E for the model problem with 4 subdomains. The numbers in parentheses are the times taken to factor and solve the interface problem.

Dimension	No. of variables	No. of processes	Factor+ Solve		Speedup	Solve	Speedup
32 × 32	21,125	1	32.8	(2.3)	-	10.8	-
		2	18.0		1.8	5.7	1.9
		4	10.5		3.1	3.3	3.3
48 × 48	47,045	1	116	(6)	-	34	-
		2	59		2.0	18	1.9
		4	33		3.5	9	3.8
64 × 64	83,205	1	269	(12)	-	77	-
		2	144		1.9	40	1.9
		4	78		3.4	20	3.8
80 × 80	129,605	1	531	(22)	-	152	-
		2	275		1.9	76	2.0
		4	150		3.5	39	3.9

written to direct-access files. We see that for sufficiently large 4 subdomain problems, for "Factor+Solve" we achieve speedups of around 1.8 and 3 using 2 and 4 processors of the Origin. Slightly better speedups are achieved on the T3E. The time taken for the interface factor and solve is independent of the number of processes. Observe that, as the problem size increases, the percentage of time required to solve the interface problem decreases. This emphasizes the suitability of our parallel code for solving very large problems. For the 8 subdomain problem, the speedups when using 8 processes in place of 4 processes are modest. This is because the interface problem, which is solved on a single processor, is becoming a more significant part of the computation. For the 96 × 96 problem, for 8 subdomains the interface problem involves 3920 variables and requires $10 * 10^9$ flops out of a total of $126 * 10^9$ flops, while for 4 subdomains the corresponding statistics are 1925 variables and $3 * 10^9$ out of a total of $143 * 10^9$ flops.

Table 6.3: Wall clock timings (in seconds) for MP42 on the Origin 2000 for the model problem with 8 subdomains. The numbers in parentheses are the times taken to factor and solve the interface problem.

Dimension	No. of variables	No. of processes	Factor+ Solve	Speedup	Solve	Speedup
48 × 48	47,045	1	46 (9)	-	2.5	-
		2	28	1.6	1.3	1.9
		4	20	2.3	1.0	2.5
		8	14	3.3	0.8	3.1
96 × 96	186,245	1	539 (77)	-	19.5	-
		2	316	1.7	10.5	1.9
		4	218	2.5	6.8	2.9
		8	164	3.4	9	3.3

6.2 MP42 versus MA42

We now compare the performance of the frontal code MA42 with that of MP42 on a single process. In Table 6.4, we present factor storage requirements and flop counts for the two codes for the model problem. MA42 treats the problem as a single domain while for MP42 the domain is divided into 4 equal subdomains and the flop count is the total for the 4 subdomains plus the interface problem.

We see that the amount of work and storage can be significantly reduced by partitioning the domain and using a multiple front approach. The savings in the storage and flop counts increase with the problem size and, for large problems, the flop count is reduced by a factor of more than 2. In Table 6.5, we compare CPU timings (in seconds) for MA42 and MP42 run on a single process of the Origin 2000 and Cray T3E. We observe that the savings in flops translate to significant savings in the CPU times for the factorize phase and the reduction in the number of entries in the factors leads to savings in the solve phase.

Table 6.4: A comparison of the factor storage requirements and flop counts for MA42 and MP42 on model problem.

Dimension	Code	Factor Storage (Kwords)		Flops ($*10^8$)
		Real	Integer	
32×32	MA42	14233	995	36
	MP42	11065	752	23
48×48	MA42	46390	3259	179
	MP42	34725	2350	102
64×64	MA42	111686	7844	596
	MP42	78625	5306	301
80×80	MA42	223070	15709	1547
	MP42	149654	10077	707

Table 6.5: A comparison of the CPU times (in seconds) for MA42 and MP42 on model problem (single process).

Dimension	Code	Origin 2000		Cray T3E	
		Factor+ Solve	Solve	Factor+ Solve	Solve
32×32	MA42	15.2	0.75	18.6	1.8
	MP42	9.9	0.53	13.3	1.2
48×48	MA42	66	2.0	73	5.6
	MP42	39	1.5	49	3.8
64×64	MA42	215	6.7	206	13.2
	MP42	114	3.8	133	8.5
80×80	MA42	630	17.8	502	25.9
	MP42	269	12.3	262	16.3

6.3 Groundwater flow computations

We now report results for a set of three test problems supplied by Steve Joyce of AEA Technology. These problems are from the finite-element modelling of groundwater flow through a porous medium. The problems are all defined on regular grids and were subdivided into 4 equal subdomains that have a small interface by Steve Joyce. The first problem is a 2 dimensional problem with 40000 square elements; problems 2 and 3 are 3 dimensional with 27000 and 125000 8-noded cubic elements, respectively. There is a single variable at each node, representing pressure. The number of variables and interface variables are included in Table 6.7. In Table 6.6, we present results for the groundwater flow problems run on 1, 2, and 4 processes. Again, we achieve good speedups, although because the number of interface variables is proportionally higher for the 3 dimensional problems, the speedups for the factor times for these problems is not quite as good as for the 2 dimensional problem.

Table 6.6: Wall clock timings (in seconds) for MP42 on 1, 2, and 4 processors of the Origin 2000 for the groundwater flow problems (4 subdomains).

Problem	No. of processes	Factor+ Solve	Speedup	Solve	Speedup
1	1	138	-	11.3	-
	2	77	1.8	6.2	1.8
	4	42	3.3	3.6	3.1
2	1	204	-	3.2	-
	2	125	1.6	2.0	1.6
	4	85	2.4	1.2	2.6
4	1	5823	-	48	-
	2	3560	1.6	28	1.7
	4	2050	2.8	15	3.2

6.3.1 The effect of the minimum pivot block size

In a recent paper, Cliffe et al. (1998) performed experiments using the frontal solver MA42 and found that it can be advantageous to delay pivoting until a minimum number of pivots are available. The advantage comes from using the level-3 BLAS routine GEMM with a larger internal dimension than would occur if elimination operations are performed whenever possible after an assembly step. In Tables 6.7 and 6.8, we present results for different pivot block sizes for the groundwater flow test examples. These results were obtained on a single process of the Origin 2000 at Parallab. Problem 3 required too much CPU time for us to test each of the block sizes but it is clear from our results that using a block size greater than 1 can lead to significant savings in both time and storage. Using a minimum pivot block size greater than 1 is particularly important when there is a single variable at each node because, in this case, the number of pivots that become fully summed following an element assembly is often 1 and as a result, for each real stored in the L and U factors, one

Table 6.7: The effect of varying the minimum pivot block size on the wall clock time (in seconds) for the factorization (single process of Origin 2000).

Problem	Number of variables	Interface variables	Number of elements	Minimum pivot block size 1	16	32	64
1	159999	859	40000	165	142	138	157
2	29785	1978	27000	559	234	204	210
3	132651	5159	125000	25328	-	5823	-

Table 6.8: The effect of varying the minimum pivot block size on the real and integer factor storage (in Kwords) (NT indicates not tested).

Problem	Minimum pivot block size							
	1		16		32		64	
	Real	Integer	Real	Integer	Real	Integer	Real	Integer
1	86692	43791	89083	26373	91508	25325	96676	25658
2	40164	35951	40649	2384	41184	1231	42252	650
3	474732	443181	NT	NT	479002	14410	NT	NT

integer is stored. We can see this by comparing the real and integer storage for a minimum pivot block size of 1. For the groundwater flow problems, the integer storage is approximately equal to half the real storage, which is equal to the storage for the L factor plus the storage for the U factor (which are both the same). Increasing the minimum pivot block size does not add greatly to the real storage but leads to substantial savings in the integer storage. Based on our results and those of Cliffe et al. (1998), we have chosen the default minimum pivot block size in MP42 to be 32, but this is a control parameter that the user may choose to reset. Note that the experiments using MP42 and MA42 reported on in the previous sections all used the default pivot block size.

7 Conclusions and future work

We have designed and developed a multiple front code for solving systems of unsymmetric unassembled finite-element equations in parallel. The code is written in Fortran 90 with MPI for message passing. An important design feature is the straightforward but flexible user interface, which offers the user a number of important options, including how the element data is to be input and allowing the possibility of holding the matrix factors in direct-access files. This enables the code to be used to solve large problems efficiently on a range of modern parallel machines. We have tested the code on a model problem and on a practical application and, in both cases, we have achieved good speedups using a small number of processes. Numerical results have also shown that the new code can perform significantly better than our established frontal solver MA42 on a single process. The results are particularly encouraging for 2 dimensional problems.

A limitation of our new code is that the interface problem is currently solved by a frontal scheme on a single process. In the future, we plan to look at solving the interface problem using other sparse direct solvers (such as the multifrontal code MA41 of Amestoy and Duff, 1989 from the Harwell Subroutine Library). An alternative approach is to assemble the local Schur complements and treat the resulting system as a dense system that can be solved using (for example) ScaLAPACK routines (see http://www.netlib.org/scalapack/). The design of HSL_MP42 using library subroutines as building blocks should allow us to try different solvers for the interface problem within the existing code.

A version of the code for symmetric positive-definite systems is currently being developed.

The code MP42 is available for use under licence and will be included in the next release of the Harwell Subroutine Library. Anyone interested in using the code should contact the author for details (or see http://www.cse.clrc.ac.uk/Activity/HSL).

8 Acknowledgements

I would like to thank Jacko Koster who, while holding a postdoctoral position at the Rutherford Appleton Laboratory, helped with the testing of HSL_MP42 and performed the experiments reported in Section 6.3. I also had many invaluable discussions with Jacko while I was learning MPI. I am also grateful to Andrew Cliffe and Steve Joyce of AEA Technology for the groundwater flow test data, and to my colleagues Iain Duff and John Reid at the Rutherford Appleton Laboratory for reading and providing many useful comments on a draft of this report.

References

P.R. Amestoy and I.S. Duff. Vectorization of a multiprocessor multifrontal code. *Inter. Journal of Supercomputer Applics*, **3**, 41–59, 1989.

R.E. Benner, G.R. Montry, and G.G. Weigand. Concurrent multifrontal methods: shared memory, cache, and frontwidth issues. *Inter. Journal of Supercomputer Applics*, **1**, 26–44, 1987.

K.A. Cliffe, I.S. Duff, and J.A. Scott. Performance issues for frontal schemes on a cache-based high performance computer. *Inter. Journal on Numerical Methods in Engineering*, **42**, 127–143, 1998.

J.J. Dongarra, J. DuCroz, I.S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Mathematical Software*, **16**(1), 1–17, 1990.

I.S. Duff. MA32 - a package for solving sparse unsymmetric systems using the frontal method. Report AERE R10079, Her Majesty's Stationery Office, London, 1981.

I.S. Duff. Design features of a frontal code for solving sparse unsymmetric linear systems out-of-core. *SIAM J. Scientific and Statistical Computing*, **5**, 270–280, 1984.

I.S. Duff and J.K. Reid. The multifrontal solution of unsymmetric sets of linear equations. *ACM Transactions on Mathematical Software*, **9**, 302–325, 1983.

I.S. Duff and J.A. Scott. MA42 – a new frontal code for solving sparse unsymmetric systems. Technical Report RAL-93-064, Rutherford Appleton Laboratory, 1993.

I.S. Duff and J.A. Scott. The use of multiple fronts in Gaussian elimination. Technical Report RAL-94-040, Rutherford Appleton Laboratory, 1994*a*.

I.S. Duff and J.A. Scott. The use of multiple fronts in Gaussian elimination. *in* J. Lewis, ed., 'Proceedings of the Fifth SIAM Conference Applied Linear Algebra', pp. 567–571. SIAM, 1994*b*.

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

I.S. Duff, J.K. Reid, and J.A. Scott. The use of profile reduction algorithms with a frontal code. *Inter. Journal on Numerical Methods in Engineering*, **28**, 2555–2568, 1989.

Harwell Subroutine Library. *A Catalogue of Subroutines (Release 12)*. Advanced Computing Department, AEA Technology, Harwell Laboratory, Oxfordshire, England, 1995.

B. Hendrickson and R. Leland. The Chaco user's guide: Version 2.0. Technical Report SAND94-2692, Sandia National Laboratories, Albuquerque, NM, 1995.

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

S.R.P. Medeiros, P.M. Pimenta, and P. Goldenberg. A parallel block frontal solver for large scale process simulation: reordering effects. *Computers in Chemical Engineering*, **21**, 439–444, 1997.

J.A. Scott. On ordering elements for a frontal solver. *Communications in Numerical Methods in Engineering*, **15**, 309–323, 1999.

S.W. Sloan. An algorithm for profile and wavefront reduction of sparse matrices. *Inter. Journal on Numerical Methods in Engineering*, **23**, 1315–1324, 1986.

W.P. Zang and E.M. Liu. A parallel frontal solver on the Alliant. *Computers and Structures*, **38**, 202–215, 1991.

O. Zone and R. Keunings. Direct solution of two-dimensional finite element equations on distributed memory parallel computers. *in* M. Durand and F. E. Dabaghi, eds, 'High Performance Computing'. Elsevier Science Publications, 1991.