INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN ENGINEERING
Int. J. Numer. Meth. Engng 2001; **50**:1131–1144

A parallel frontal solver for finite element applications

Jennifer A. Scott*,†

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

SUMMARY

In finite element simulations, the overall computing time is dominated by the time needed to solve large sparse linear systems of equations. We report on the design and development of a parallel frontal code that can significantly reduce the wallclock time needed for the solution of these systems. The algorithm used is based on dividing the finite element domain into subdomains and applying the frontal method to each subdomain in parallel. The so-called multiple front approach is shown to reduce the amount of work and memory required compared with the frontal method and, when run on a small number of processes, achieves good speedups. The code, `HSL_MP42`, has been developed for the Harwell Subroutine Library (http://www.numerical.rl.ac.uk/hsl). It is written in Fortran 90 and, by using `MPI` for message passing, achieves portability across a wide range of modern computer architectures. Copyright © 2001 John Wiley & Sons, Ltd.

KEY WORDS: finite-elements; unsymmetric linear systems; frontal method; parallel processing; Fortran 90; `MPI`

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 [1, 2] (and its predecessor `MA32` of Duff [3]) 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 [4] 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 [5]. 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

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

Received 19 November 1999
Revised 10 April 2000

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 large-scale 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 [6, 7].

This paper is organized as follows. In Section 2, we outline the multiple front method. The design and development of our parallel frontal solver `HSL_MP42` is then discussed in Section 3. Numerical results for a model problem and a practical application are presented in Section 4. The performance of `HSL_MP42` is also compared with that of the frontal code `MA42`. Finally, in Section 5, 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

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

where the $n \times n$ matrix A is large and sparse. B is an $n \times nrhs$ ($nrhs \geqslant 1$) 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

$$A = \sum_{l=1}^{m} A^{(l)} \tag{2}$$

where each element matrix $A^{(l)}$ has non-zeros 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 [8] and the basis for our experience with them is discussed by Duff [9]. The 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 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

$$\begin{pmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{pmatrix} \tag{6}$$

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 [6]).

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 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 *et al.* [10], 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 [6] proposed allowing a (small) number of independent fronts in a somewhat similar fashion to Benner *et al.* [11] and Zang and Liu [12] (see also Zone and Keunings [13] and, for non-element problems, Mallya *et al.* [14]).

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

$$\begin{pmatrix} A_{11} & & & & C_1 \\ & A_{22} & & & C_2 \\ & & \cdots & & \cdot \\ & & & A_{NN} & C_N \\ \tilde{C}_1 & \tilde{C}_2 & \cdots & \tilde{C}_N & \sum_{i=1}^{N} E_i \end{pmatrix} \tag{7}$$

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

$$\begin{pmatrix} A_{ii} & C_i \\ \tilde{C}_i & E_i \end{pmatrix} \tag{8}$$

This can be done in parallel. At the end of the assembly and elimination processes for each subdomain Ω_i, there will remain $1 \leqslant k_i \leqslant k$ 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

$$A = P \begin{pmatrix} L_1 & & & & \\ & L_2 & & & \\ & & \cdots & & \cdot \\ & & & L_N & \\ \tilde{L}_1 & \tilde{L}_2 & \cdots & \tilde{L}_N & I \end{pmatrix} \begin{pmatrix} U_1 & & & & \tilde{U}_1 \\ & U_2 & & & \tilde{U}_2 \\ & & \cdots & & \cdot \\ & & & U_N & \tilde{U}_N \\ & & \cdots & & F \end{pmatrix} Q \tag{9}$$

where the $k \times k$ matrix

$$F = \sum_{i=1}^{N} F_i \tag{10}$$

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.

3. DESIGN OF A PARALLEL FRONTAL SOLVER

In this section, we consider the design of the new multiple front solver `HSL_MP42`. A full description of how to use the code is given in the report of Scott [15]. The code is written in Fortran 90 and `MPI` is used for message passing. `MPI` was chosen since the `MPI` Standard is internationally recognized and today `MPI` is widely available and accepted by users of parallel computers. The code requires a designated host process (see below). 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 processes. This allows the code to be used on distributed memory parallel computers as well as on shared memory machines.

When writing a program to run `HSL_MP42`, the user must include the file `mpif.h` at the top of his or her program. This is done by placing the line

```
INCLUDE 'mpif.h'
```

after the program statement at the start of the program. The file `mpif.h` contains a number of predefined constants and data types used by `MPI` functions. In addition, the user must initialize `MPI` by calling `MPI_INIT` on each process and must define a communicator for the package. A communicator is essentially a collection of processes that can communicate with each other. The most basic communicator is `MPI_COMM_WORLD`, which is predefined by `MPI` and consists of all the processes running when the program execution begins. In many applications, this communicator is adequate.

The module HSL_MP42 has five separate phases:

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

Each phase must be called in order by each process (although the solve phase is optional). 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 where 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. We now discuss each phase in further detail.

3.1. *Initialize*

This phase initializes an instance of the package. The code first checks that MPI has been initialized by the user, determines the number of processes being used, and the rank of each process. The processes involved in the execution of an MPI program are identified by a sequence of non-negative contiguous integers. If there are p processes executing a program, they will have rank $0, 1, \ldots, p-1$. The *host* is defined to be 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 minimize 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 Scott [15]).

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. The ordering algorithm used is that of Scott [16]. The order generated for Ω_i 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. This count is made by assuming that as soon as a variable becomes fully summed it is available for elimination. It is only an estimate because, during the factorization, stability considerations may mean delaying pivots, causing a growth in the front size and in the actual flop count. Moreover, the factorization attempts to take advantage of blocks of zeros within the front to avoid unnecessary operations being performed on zeros (see Scott [17]). This can reduce the number of flops needed. The estimated flop count is 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 shared 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 minimizes 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 are again be read by that process. If this option is used, the data for all the elements in a subdomain are read in at once, so more memory is required but, again, movement of data between processes is minimized. 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 are 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 minimize 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 uses `MA42` to perform 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 and the integer storage required by the factors) is also sent to the host.

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.

3.6. Partitioning the domain

An important early decision when designing HSL_MP42 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. Our decision was made partly because the choice of a good partitioning is very problem dependent and also this is still a very active research area and no single approach has yet emerged as being ideally suited for our application. Moreover, in many practical problems, 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 [18] or METIS (http://winter.cs.umn.edu/~karypis/metis/).

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. Because our parallel frontal code solves the interface problem using a single processor, to achieve good speedups for the overall solution process it is crucial that the user partitions the domain into subdomains in such a way as to keep the size of the interface problem small. Our experience has also been that to achieve good load balancing in terms of the amount of work and memory required by each processes, the subdomains should have a similar

number of interface variables. This can mean that the subdomains are not balanced in terms of the number of elements they contain (see Duff and Scott [7]).

4. NUMERICAL RESULTS

In this section, we illustrate the performance of HSL_MP42, first on a model problem and then on three problems arising from groundwater flow calculations. The experiments in Sections 4.1 and 4.2 were performed on the SGI Origin 2000 and Cray T3E at Manchester, and those in Section 4.3 were carried out on the SGI Origin 2000 located at Parallab (Bergen).

4.1. 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 I–III, we present results for a square domain subdivided into 4 and 8 subdomains. HSL_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 [7]). 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 are held in memory on the host. On the Origin (a non-uniform access shared memory machine), files are not used for the matrix factors; on the T3E, the matrix factors are 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.

4.2. HSL_MP42 versus MA42

We now compare the performance of the frontal code MA42 with that of HSL_MP42 on a single process. In Table IV, 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 HSL_MP42 the domain is divided into 4 equal subdomains and the flop count is the total for the 4 subdomains plus the interface problem.

Table I. Wall clock timings (in s) for HSL_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 II. Wall clock timings (in s) for HSL_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

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 V, we compare CPU timings (in s) for MA42 and HSL_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. We conclude that, although the main motivation for our work is the development of a parallel code, the code may also be of advantage on a uniprocessor.

Table III. Wall clock timings (in s) for HSL_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	5.9	3.3

Table IV. A comparison of the factor storage requirements and flop counts for MA42 and HSL_MP42 on model problem.

Dimension	Code	Factor storage (Kwords) Real	Integer	Flops ($*10^8$)
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 V. A comparison of the CPU times (in s) for MA42 and HSL_MP42 on model problem (single process).

Dimension	Code	Origin 2000 Factor + Solve	Solve	Cray T3E 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

4.3. 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 two-dimensional problem with 40000 square elements; problems 2 and 3 are three-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 VII. In Table VI, 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 three-dimensional problems, the speedups for the factor times for these problems is not quite as good as for the two-dimensional problem.

4.3.1. The effect of the minimum pivot block size. In a recent paper, Cliffe *et al*. [19] 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 VII and VIII, 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 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

Table VI. Wall clock timings (in s) for `HSL_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

Table VII. The effect of varying the minimum pivot block size on the wall clock time (in s) 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 VIII. The effect of varying the minimum pivot block size on the real and integer factor storage (in Kwords) (NT indicates not tested).

	Minimum pivot block size							
	1		16		32		64	
Problem	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

leads to substantial savings in the integer storage. Based on our results and those of Cliffe *et al.* [19] we have chosen the default minimum pivot block size in `HSL_MP42` to be 32, but this is a control parameter that the user may choose to reset. Note that the experiments using `HSL_MP42` and `MA42` reported on in the previous sections all used the default pivot block size.

5. CONCLUSIONS AND FUTURE WORK

We have designed and developed a multiple front code for solving large systems of unsymmetric unassembled finite-element equations in parallel. The code `HSL_MP42`, which is in Fortran 90 with `MPI` for message passing, has been written using our extensive knowledge and experience of frontal methods and, in particular, uses the established frontal solver `MA42` combined with the subdomain element ordering algorithm of Scott [16]. Experiments have been run using a model problem and 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 `MA42` on a single process. The results are particularly encouraging for two-dimensional problems. We remark that, for non-element problems, Mallya *et al.* [20] reported similar findings.

In this paper, we have presented `HSL_MP42` timings for runs performed on an Origin 2000 and on a Cray T3E. The code can, however, be run on any system with `MPI` available. In particular, a cluster of workstations that can communicate using `MPI` could be used. Results reported by Duff and Scott [7] illustrate that this kind of approach can be very effective. When working in a network-based environment, it is important to consider how the element data are input to the code and where the matrix factors are stored. For efficiency, the amount of data movement between processes needs to be minimized. Because of this `HSL_MA42` was designed with a number of different input data options (see Section 3.3). On a cluster of workstations, the default option

is recommended whereby all the element data required by a process are read by that process. In addition, if direct-access files are needed to hold the matrix factors, the user should ensure that the files are held locally. This can be achieved by appropriately setting the parameters for the direct-access file names. Full details are given in the User Documentation.

A limitation of the new code is that the interface problem is currently solved by a frontal scheme on a single process, making it vital for good performance that the number of interface variables is kept small. 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 [21] 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 finite-element problems has been developed. This version is called HSL_MP62.

Both HSL_MP42 and HSL_MP62 are available for use under licence and will be included in the next release of the Harwell Subroutine Library [4]. Anyone interested in using the codes may contact the author for details (or see http://www.numerical.rl.ac.uk/hsl).

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 4.3. 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 paper.

REFERENCES

1. Duff IS, Scott JA. MA42—a new frontal code for solving sparse unsymmetric systems. *Technical Report RAL-93-064*, Rutherford Appleton Laboratory, 1993.
2. Duff IS, Scott JA. The design of a new frontal code for solving sparse unsymmetric systems. *ACM Transactions of the Mathematical Software* 1996; **22**(1):30–45.
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. HSL. A Collection of Fortran Codes for Large Scale Scientific Computation, 2000. Full details from http://www.numerical.rl.ac.uk/hsl.
5. Dongarra JJ, DuCroz J, Duff IS, Hammarling S. A set of level 3 basic linear algebra subprograms. *ACM Transactions of the Mathematical Software* 1990; **16**(1):1–17.
6. Duff IS, Scott JA. The use of multiple fronts in Gaussian elimination. *Technical Report RAL-94-040*, Rutherford Appleton Laboratory, 1994.
7. Duff IS, Scott JA. The use of multiple fronts in Gaussian elimination. In: *Proceedings of the 5th SIAM Conference Applied Linear Algebra*, Lewis J (ed). SIAM, Philadelphia, PA, 1994; 567–571.
8. Irons BM. A frontal solution program for finite-element analysis. *International Journal for Numerical Methods in Engineering* 1970; **2**:5–32.
9. Duff IS. Design features of a frontal code for solving sparse unsymmetric linear systems out-of-core. *SIAM Journal of Scientific and Statistical Computing* 1984; **5**:270–280.
10. Duff IS, Reid JK, Scott JA. The use of profile reduction algorithms with a frontal code. *International Journal for Numerical Methods in Engineering* 1989; **28**:2555–2568.
11. Benner RE, Montry GR, Weigand GG. Concurrent multifrontal methods: shared memory, cache, and frontwidth issues. *International Journal of Supercomputer Applications* 1987; **1**:26–44.

12. Zang WP, Liu EM. A parallel frontal solver on the Alliant. *Computers and Structures* 1991; **38**:202–215.
13. Zone O, Keunings R. Direct solution of two-dimensional finite element equations on distributed memory parallel computers. In: *High Performance Computing*, Durand M, Dabaghi FE. (eds). Elsevier: Amsterdam, 1991.
14. Mallya JU, Zitney SE, Choudhary S, Stadtherr MA. A parallel block frontal solver for large scale process simulation: reordering effects. *Computers in Chemical Engineering* 1997; **21**:S439–S444.
15. Scott JA. The design of a parallel frontal solver. *Technical Report RAL-TR-99-075*, Rutherford Appleton Laboratory, 1999.
16. Scott JA. On ordering elements for a frontal solver. *Communications in Numerical Methods in Engineering* 1999; **15**:309–323.
17. Scott JA. Exploiting zeros in frontal solvers. *Technical Report RAL-TR-98-041*, Rutherford Appleton Laboratory, 1997.
18. Hendrickson B, Leland R. The Chaco user's guide: Version 2.0. *Technical Report SAND94-2692*, Sandia National Laboratories, Albuquerque, NM, 1995.
19. 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.
20. Mallya JU, Zitney SE, Stadtherr MA. Parallel frontal solver for large scale process simulation and optimization. *A.I.Ch.E. Journal* 1997; **43**:1032–1040.
21. Amestoy PR, Duff IS. Vectorization of a multiprocessor multifrontal code. *International Journal of Supercomputer Applications* 1989; **3**:41–59.

