RAL-TR-2002-012

Parallel frontal solvers for large sparse linear systems

by

Jennifer A. Scott*

Abstract

Many applications in science and engineering give rise to large sparse linear systems of equations that need to be solved as efficiently as possible. As the size of the problems of interest increases, it can become necessary to consider exploiting multiprocessors to solve these systems. We report on the design and development of parallel frontal solvers for the numerical solution of large sparse linear systems. Three codes have been developed for the mathematical software library HSL (www.cse.clrc.ac.uk/Activity/HSL). The first is for unsymmetric finite-element problems; the second is for symmetric positive definite finite-element problems; and the third is for highly unsymmetric linear systems such as those that arise in chemical process engineering. In each case, the problem is subdivided into a small number of loosely connected subproblems and a frontal method is then applied to each of the subproblems in parallel. We discuss how our software is designed to achieve the goals of portability, ease of use, efficiency, and flexibility, and illustrate the performance on an SGI Origin 2000 using problems arising from real applications.

Keywords: large sparse linear systems, finite-elements, frontal method, multiple front method, parallel processing, Fortran 90, `MPI`.

---

*j.a.scott@rl.ac.uk
Computational Science and Engineering Department, Atlas Centre, Rutherford Appleton Laboratory, Oxon OX11 0QX, England.
This work was funded by the EPSRC Grant GR/R46441.

April 8, 2002.

1 Introduction

Large-scale simulations in many areas of science and engineering involve the repeated solution of large sparse linear systems of equations of the form

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

Solving these systems is generally the most computationally expensive step in the simulation. 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 method is a variant of Gaussian elimination that offers one approach to solving linear systems in either element or non-element form. It was first developed in the 1970s by Irons (1970) (see also Hood, 1976) for finite element problems at a time when there was a need to solve problems for which the matrix A and its LU factors were too large to be stored in the main memory of the available computers. For finite element problems, A is a sum of elemental matrices

$$A = \sum_{k=1}^{m} A^{(k)}, \tag{1.2}$$

where each $A^{(k)}$ has nonzeros only in a few rows and columns and corresponds to the matrix from element k. The principal idea is to avoid assembling all the elements into one large sparse system matrix A but to assemble the $A^{(k)}$ one at a time into a small dense *frontal* matrix. As soon as a variable becomes *fully summed* (that is, it is not involved in any of the elemental matrices still to be assembled), it becomes a candidate for elimination. In this way, it is possible to interleave the assembly and elimination operations. Provided the elemental matrices are assembled in a suitable order, the size of the frontal matrix can be kept to a fraction of the total number of variables and, by writing the rows and columns of the matrix factors to secondary storage (such as direct access files) as soon as they are generated, the amount of main memory required is small. This allows large problems to be solved. Furthermore, since the frontal matrix is held as a full matrix, dense linear algebra kernels can be used is the innermost loop of the computation. In particular, high-level BLAS kernels (Dongarra, DuCroz, Duff and Hammarling, 1990) can be exploited, making the method efficient on a wide range of computer architectures.

The frontal method is not limited to finite-element applications: the method was extended to non-element problems by Duff (1984). In this case, the rows of the matrix are assembled one at a time into the frontal matrix which, in this case, is rectangular. A variable can be eliminated as soon as the last row in which it has a nonzero entry has been assembled. The method can also be generalised to incorporate pivoting for numerical stability. An advantage of the frontal method over many other direct sparse solvers for non-element problems is that it does not require the system matrix to have any special structural or numerical properties such as symmetry, positive definiteness, diagonal dominance, or bandedness. As chemical process simulation matrices

possess none of these desirable properties, the choice of suitable solvers is restricted and the frontal method is an attractive option.

Over the last thirty years, the frontal method has been successfully used to efficiently solve a wide range of problems from a variety of application areas. It also lies at the heart of many commercial finite-element packages. However, a major deficiency of the frontal solution scheme for modern computers is the lack of scope for parallelism other than that which can be obtained within the high-level BLAS. In an attempt to circumvent this shortcoming, Duff and Scott (1994) proposed subdividing the problem into a (small) number of loosely connected subproblems and allowing an independent front on each subproblem 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, Mallya, Zitney, Choudhary and Stadtherr, 1997). It is this so-called *multiple front* approach that is implemented in the software reported on in this paper.

The aim of this paper is to report on the design and development of three software packages that respectively implement the multiple front method for unsymmetric finite element problems, for symmetric positive definite finite element problems, and for highly unsymmetric sparse non-element problems. These codes have been developed for the software library HSL 2002. The paper is organised as follows. In Section 2, the multiple front method is reviewed and its implementation is discussed in Section 3. In Section 4, we look at software considerations in the design of our parallel codes and then, in Section 5, we present numerical results for each of the solvers. Finally, in Section 6, concluding remarks are made.

We end this section by noting that, unless stated otherwise, the numerical results presented in this paper were computed on a 12 processor SGI Origin2000 using the "cpuset" facility to give exclusive access to processors and their local memory. The Fortran 90 compiler was used in 64 bit mode with optimization flags `-O3 -OPT:Olimite=0`. All presented timings are wallclock times given in seconds.

2 Multiple fronts

2.1 Element problems

We first introduce the multiple front approach for finite-element problems. We start by partitioning the underlying finite-element domain Ω into a chosen number N of non-overlapping subdomains Ω_l. This is equivalent to preordering the original matrix A to doubly-bordered block diagonal form

$$\begin{pmatrix} A_{11} & & & & C_1 \\ & A_{22} & & & C_2 \\ & & \ldots & & . \\ & & & A_{NN} & C_N \\ \tilde{C}_1 & \tilde{C}_2 & \ldots & \tilde{C}_N & \sum_{l=1}^{N} E_l \end{pmatrix}, \tag{2.1}$$

where the diagonal blocks A_{ll} are $n_l \times n_l$ and the border blocks C_l and $\tilde{C}_l$ are $n_l \times k$ and $k \times n_l$, respectively (with $\tilde{C}_l = C_l^T$ in the symmetric case). If $k_l \leq k$ is the number of columns of C_l with at least one nonzero entry, the division into subdomains should be chosen so that $k_l \ll n_l$. A partial frontal decomposition is performed on each of the matrices

$$\begin{pmatrix} A_{ll} & C_l \\ \tilde{C}_l & E_l \end{pmatrix}, \tag{2.2}$$

(with any zero columns removed). These decompositions may be performed in parallel. At the end of the assembly and elimination processes for each subdomain Ω_l, there will remain k_l *interface variables*. These variables are shared by more than one subdomain and so are not fully summed and cannot be eliminated within the subdomain. In practice, there may also remain variables that are not eliminated within the subdomain because of efficiency or stability considerations (see Section 4.2). These variables are added to the border and k is increased. If F_l is the local Schur complement for subdomain Ω_l (that is, F_l holds the frontal matrix that remains when all possible eliminations on subdomain Ω_l have been performed), once each of the subdomains has been partially factorized, we have

$$A = P \begin{pmatrix} L_1 & & & & \\ & L_2 & & & \\ & & \ldots & & . \\ & & & L_N & \\ \tilde{L}_1 & \tilde{L}_2 & \ldots & \tilde{L}_N & I \end{pmatrix} \begin{pmatrix} U_1 & & & & \tilde{U}_1 \\ & U_2 & & & \tilde{U}_2 \\ & & \ldots & & . \\ & & & U_N & \tilde{U}_N \\ & & \ldots & & F \end{pmatrix} Q, \tag{2.3}$$

where P and Q are permutation matrices and the $k \times k$ matrix F is the sum of the F_l matrices and is termed the *interface matrix*. By treating each of the subdomain frontal matrices F_l as an elemental matrix, the interface matrix F may also be factorized using the frontal method. Forward eliminations and back-substitutions can be then be performed (in parallel) on the subdomains to complete the solution.

2.2 Non-element problems

To extend the multiple front approach to general sparse systems with an unsymmetric sparsity pattern, we preorder the matrix to singly bordered block diagonal form

$$\begin{pmatrix} A_{11} & & & C_1 \\ & A_{22} & & C_2 \\ & & \ldots & . \\ & & A_{NN} & C_N \end{pmatrix}, \tag{2.4}$$

where the diagonal blocks A_{ll} are now rectangular $m_l \times n_l$ matrices with $m_l \geq n_l$, and the border blocks C_l are $m_l \times k$ with $k \ll n_l$. In this case, the frontal method is used to perform a partial LU decomposition on each of the matrices

$$\begin{pmatrix} A_{ll} & C_l \end{pmatrix} \tag{2.5}$$

(again, any zero columns are assumed to have been removed). Assuming A is nonsingular, as the rows of (2.5) are assembled, n_l variables become fully summed and may be eliminated. These variables correspond to the columns of A_{ll}; the k_l nonzero columns of C_l do not become fully summed because they have entries in at least one other border block C_j $(j \neq l)$. At the end of the assembly and elimination operations, for each block there will remain a frontal matrix F_l of order $(m_l - n_l) \times k_l$ (if A is singular, fewer than n_l eliminations may be possible so that the size of F_l will increase). The sum F of these remaining frontal matrices may again be factorized using the frontal method, followed by block forward eliminations and back-substitutions.

In the remainder of this paper, to unify the discussion of element and non-element problems, we will denote by $\{A_{ll}, C_l\}$ the subproblem which, for element problems, corresponds to (2.2) and, for non-element problems, to (2.5).

2.3 Finding a good preordering

As already stated, the multiple front method requires the matrix A to be first preordered to bordered block diagonal form. Each subproblem $\{A_{ll}, C_l\}$ is than partially factorized. As this may be done in parallel, the number N of blocks should be at least as large as the number p of processes used. For each subproblem there will remain interface variables that are not fully summed within the subproblem (plus any variables that cannot be eliminated for stability reasons). In the software reported on in this paper, the interface matrix F corresponding to these variables is factorized by a single process. Thus to avoid the solution of the interface problem becoming a serious bottleneck, the size of the interface matrix needs to be as small as possible. In other words, the preordering to block diagonal form should aim to minimise the interconnection between the subproblems. As the number of interface variables increases with the number of subproblems, if good overall speedup is to be achieved, the multiple front approach is most suited for use with a relatively small number of processors (typically, 8 or 16). Furthermore, if the subproblems are of a similar size, with a similar number of interface variables, good load balance can often be achieved by choosing the number p of processors equal to a multiple of the number of subproblems N.

An important early decision when designing our software was not to include within it (at least initially) software for preordering the problem to bordered block diagonal form. Instead, the user must perform some preprocessing and make available a list of the elements or rows belonging to each of the subproblems. Our decision was made partly because the choice of a good preordering is very problem dependent and also because this is still a very active research area and no single approach has yet emerged as being ideally suited for the full range of applications of interest to us. 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 subproblems and the wings as two further subproblems. Again, in chemical process engineering applications, the matrix can naturally occur

in the required form (see Mallya et al., 1997) and this may render reordering unnecessary.

When no such ordering is available, for finite-element problems 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 (winter.cs.umn.edu/~karypis/metis/). For the highly unsymmetric problems that arise in chemical process engineering, the MONET algorithm of Hu, Maguire and Blake (2000) has been shown to perform extremely well, producing very small interfaces for modest-sized N. An implementation of this algorithm is available within the mathematical software library HSL as routine `HSL_MC66`. Alternatively, if the matrix can be permuted to banded form, the user could consider partitioning the rows into N blocks each with an equal (or nearly equal) number of rows. For example, if $N = 2$, the banded linear system may be represented as

$$\begin{pmatrix} A_{11} & C_1 & 0 \\ 0 & C_2 & A_{22} \end{pmatrix}, \tag{2.6}$$

and this can be permuted to the bordered form

$$\begin{pmatrix} A_{11} & & C_1 \\ & A_{22} & C_2 \end{pmatrix}. \tag{2.7}$$

For the interface problem to be small, the bandwidth must be narrow, ideally with sparsity within the band. This approach to subdividing the matrix is used in a recent paper by Golub, Sameh and Sarin (2001). Preordering a matrix with a symmetric sparsity pattern to banded form may be performed using the HSL code `MC60`, which includes an efficient implementation of the Reverse Cuthill McKee algorithm (Cuthill and McKee, 1969).

We remark that for the very sparse matrices arising from circuit simulation problems, a possible approach to ordering the matrix to doubly bordered block diagonal form is suggested by Bomhof and van der Vorst (2000).

Note that in many applications, the user is interested in factorizing a number of matrices with the same sparsity pattern. For example, in solving a nonlinear system using Newton's method, a single Newton step will correspond to solving (1.1). In such cases, the cost of the preordering the sparsity pattern of A to bordered form may be amortized over a number of Newton steps.

3 Implementation of the multiple front algorithm

3.1 The frontal method and `MA42`

Before discussing the design and development of our software for the multiple front algorithm, we briefly consider how the frontal algorithm for solving (1.1) is implemented within the HSL frontal solver `MA42`. `MA42` is quite a complicated code and represents considerable programming effort. It has also proved to be very robust and reliable. We were therefore keen to exploit `MA42` within our multiple front software, with a minimum number of modifications.

MA42 was developed in the early 1990s by Duff and Scott (1993, 1996), superseding the earlier well known code MA32 (Duff, 1981). Although primarily designed for finite-element problems, by offering the user the option of entering the matrix data by rows, MA42 can also be used to solve general sparse linear systems. A later code MA62 (Duff and Scott, 1998), which adopted similar design principles to those used by MA42, is for symmetric positive definite finite-element systems.

In common with many sparse direct solvers, the algorithm used by MA42 (and MA62) may be split into three phases as follows:

MA42 Frontal Algorithm:

Symbolic analyse.
A symbolic analysis using the sparsity pattern of A is performed. This determines when each column of A is fully summed and, optionally, computes estimates of memory requirements (maximum frontsizes and factor storage) for the frontal elimination.

Factorize.
The frontal elimination on A, if necessary incorporating pivoting for numerical stability, is carried out. The columns of L and rows of U are optionally written to direct access files as they are generated. High level BLAS routines are used in the innermost loop of the factorization.

Solve.
Forward elimination is followed by back-substitution.

The above algorithm is modified if right-hand side vectors b (in elemental form) are available at the time of the frontal factorization. In this case, forward elimination operations are performed as the L and U factors are generated. In the solve phase, back-substitution is performed to give the final solution. If the user does not want to solve for further right-hand sides, storage way be reduced by not retaining the L factor.

More than one right-hand side may be solved for at once. In this case, MA42 is able to take advantage of Level 3 BLAS in the forward elimination and back-substitution stages.

A key design feature of MA42 is its use of a reverse communication interface for the analyse and factorize phases. To avoid the need for the user to have all the elements or rows of A available at once, control is returned to the user each time an element (or row) is needed. Each element (or row) has to be passed to the symbolic analyse and factorize phases just once (integer data only for the symbolic phase). This use of reverse communication minimises main memory requirements and allows the user to generate the matrix entries (or read them from secondary storage) as they are required.

The efficiency of the frontal method depends upon keeping the maximum frontsize small. MA42 does not preorder the elements for the user: this may optionally be done using another HSL code, MC63. MC63 (Scott, 1999*b*) offers

an efficient implementation of a modified version of Sloan's algorithm (Sloan, 1989). For non-element problems with an unsymmetric structure, `MC62`, which implements the MSRO algorithm of Scott (1999*a*), may be used. If the problem has a symmetric (or almost symmetric) sparsity pattern, the HSL code `MC60` (Reid and Scott, 1999) for profile reduction is appropriate.

3.2 The multiple front algorithm

We now outline how the multiple front algorithm is implemented within our software. A key idea is the use of a *guard* element or row. If we pass the subproblem $\{A_{ll}, C_l\}$ to a frontal solver, a complete LU factorization of the subproblem will be performed. However, we do not want this since we cannot eliminate the interface variables until we have assembled all the contributions from the other subproblems. To use our existing frontal software, we have to prevent the interface variables from being eliminated; this is done by flagging them as not being fully summed within the subproblem. Consider the non-element case and the submatrix (2.5). Before the start of the symbolic phase, we add an extra row d_l to the submatrix. This row (which we term the *guard row*) has a nonzero entry in position j if and only if column j of C_l has at least one nonzero entry. We thus have the $(m_l + 1) \times (n_l + k_l)$ matrix $\hat{A}_l$ given by

$$\hat{A}_l = \begin{pmatrix} A_{ll} & C_l \\ 0 & d_l^T \end{pmatrix}. \tag{3.1}$$

The matrix $\hat{A}_l$ is passed to the symbolic phase of the frontal code. This flags all the columns of C_l with a nonzero entry as becoming fully summed after the assembly of row $m_l + 1$, that is, after d_l has been assembled. The original submatrix $(A_{ll}\ C_l)$ is then passed to the factorize phase. Because the symbolic analysis was performed on the matrix $\hat{A}_l$, which has an additional row, the columns of C_l do not become fully summed, ensuring the eliminations are restricted to the columns of A_{ll}.

The element case is handled in a similar way. For each subproblem, an additional element (the *guard element*), which comprises a list of the interface variables in subdomain l, is passed to the symbolic phase. This element is entered last and, by not passing it to the factorize phase, the interface variables are prevented from becoming fully summed and remain in the front.

The flexibility of the `MA42` reverse communication interface makes it straightforward for us to pass the extra guard element (or row) to the symbolic phase and thus, to use `MA42` within our multiple front software, it was only necessary to write an additional routine that could extract from the `MA42` data structures the data remaining in the front after an incomplete factorization.

We emphasise that our multiple front software does **not** have a reverse communication interface : it is the "user" of `MA42` and handles the `MA42` reverse communication in a way that is shielded from the user of the multiple front software.

The main steps in the multiple front algorithm are as follows:

Multiple Front Algorithm:

Initialize: performed by a single process.

- Assign an equal (or nearly equal) number of subproblems to each of the p processes.
- Generate the guard element (or row) for each subproblem $\{A_{ll}, C_l\}$.

Reorder (optional) (parallel)

- The elements (or rows) within each subproblem $\{A_{ll}, C_l\}$ are reordered and estimated flop counts for the frontal solver applied to the subproblem based on this local ordering are computed. We denote by P_l the permutation matrix that corresponds to the local ordering for subproblem l.
- The flop counts are used to distribute the subproblems between the processes in preparation for the factorization. The aim is to achieve a good load balance in terms of flops.

Frontal factorization (parallel)
For each of its assigned subproblems, process p_j performs the following steps:

- A symbolic analysis on the sparsity pattern of the permuted subproblem $\{P_l A_{ll}, P_l C_l\}$ with the addition of the guard element (or row).
- The frontal elimination on $\{P_l A_{ll}, P_l C_l\}$, incorporating pivoting as necessary for numerical stability.
- Storage of the computed rows of U and, optionally, the columns of L. The rows and columns remaining in the frontal matrix are passed to the host for assembly.

Interface problem: (single process).
The frontal method is used on a single process to factorize the interface matrix. Note that other sparse direct solvers could be used but, by using the frontal method, explicit assembly of the interface matrix is avoided.

Solve (parallel and single process).
Forward elimination in parallel on the submatrices is followed, on a single process, by forward elimination and back-substitution for the interface problem. Back-substitution in parallel on the submatrices completes the computation of the solution.

4 Software considerations

Having outlined the multiple front algorithm, we now discuss some of the details of our software. The codes for general and symmetric positive definite

finite-element problems are HSL_MP42 and HSL_MP62, respectively; the code for unsymmetric assembled systems is HSL_MP43. The prefix HSL is used within the HSL Library to flag codes that are written in Fortran 90 but, for clarity of notation, throughout the rest of the paper we will omit this prefix. Fortran 90 was chosen not only for its efficiency for scientific computation but also because it offers many more features than Fortran 77. In particular, our software makes extensive use of dynamic memory allocation and this allows a much cleaner user interface. MPI is used for message passing. This choice was made because the MPI Standard (MPI, 1994) is internationally recognised and today it is widely available and accepted by users of parallel computers. Our software does **not** assume that there is a single file system that can be accessed by all the processes. This enables the code to be used on distributed memory parallel computers as well as on shared memory machines. The use of standard Fortran 90 and MPI, together with BLAS kernels for the innermost loop of the computation, allows us to achieve our goal of producing portable software.

Each of our three codes has a similar interface and follows the same design principles. This has the advantage of making moving from using one code to another relatively straightforward and at the same time simplifies our software maintenance. We decided against incorporating all the codes within a single package since, in our experience of developing HSL codes, offering too many options within one package adds to the complexity and length of the user documentation and makes the software both harder to use and to maintain.

Each package requires that one process is designated as the *host*. The host performs the initial checking of the user's data, distributes data to the remaining processes, collects computed data from the processes, solves the interface problem, and generally overseas the computation. With the other processes, the host may also participate in local ordering and in generating the partial LU decompositions.

4.1 Local ordering

The efficiency of the multiple front method is very dependent on the ordering of the elements (or rows) within each subproblem (see, for example, Scott, 2001*c*). Obtaining a good ordering is of particular importance if a number of matrices having the same sparsity pattern are to be factorized or if the factors generated are to be used repeatedly for solving for different right-hand sides b. In such instances, the effort spent on generating a good local ordering pays dividends in the resulting reductions in the overall computational times and the factor storage requirements.

Existing reordering algorithms for frontal solvers were unsuitable because of their assumption that once a variable has appeared for the last time it can be eliminated. It was therefore necessary to develop new ordering algorithms for use with the multiple front method.

For element problems, the local ordering algorithm used is that described by Scott (1996) and implemented within the HSL code MC53. The basic idea is that, for each subdomain Ω_l, the algorithm looks for elements which lie as far away as possible from the interface between Ω_l and its adjacent subdomains

(subdomains are said to be adjacent if they have one or more variables in common). Then, starting with such an element, the reordering moves towards the interface using a generalisation of Sloan's algorithm. The aim is to balance keeping the frontsize small and delaying bringing interface variables into the front for as long as possible since, having entered the front, these variables cannot be eliminated.

In the last few years, a number of algorithms for automatically ordering the rows of a matrix with an unsymmetric sparsity pattern for use with frontal solvers have been proposed (see Scott, 2000, for an overview). The most successful currently available are the MSRO methods of Scott (1999*a*), which again have their origins in Sloan's profile reduction algorithm. For the multiple front method, Scott (2001*c*) proposed modifying the MSRO algorithm to take into account the columns that are not fully summed within the submatrix. Again, the idea is to delay assembling rows containing interface variables while keeping local frontsizes small. A Fortran code `MC62` implementing the modified MSRO algorithm has been developed for HSL and is available within our multiple front software. `MC62` may also be optionally used to order the rows of the interface problem.

Our software allows the user to decide whether or not reordering of the subproblems is to be performed. Thus, if a matrix with the same (or similar) sparsity pattern has already been factorized, the user may choose to reuse the previously computed ordering. Numerical experimentation has shown that

Table 4.1: Timings for the factorize phase of `MP43` with and without row ordering.

Identifier	Factorize no ordering	Factorize with ordering	Row ordering	Ordering plus factorize
`4cols`	0.67	0.12	0.07	0.19
`10cols`	3.85	0.31	0.18	0.49
`bayer01`	1.33	0.93	0.49	1.42
`bayer04`	0.63	0.39	0.37	0.76
`icomp`	1.32	0.65	0.54	1.19
`lhr14c`	0.69	0.56	0.79	1.35
`lhr34c`	2.30	2.30	1.92	4.22
`lhr71c`	4.92	4.61	3.92	8.53
`graham1`	9.61	9.79	1.08	10.9
`pesa`	12.4	0.34	0.10	0.44
`poli_large`	0.70	0.31	0.06	0.37
`Zhao2`	330.1	46.8	0.26	47.1

reusing an ordering can be particularly advantageous for `MP43`. In Table 4.1, we compare the time for the factorize phase of `MP43` with and without local row ordering; we also include the time taken for the row ordering. The entries in column 5 are the sum of the corresponding entries in columns 3 and 4 (that is, the time for row ordering plus factorizing the matrix). Details of the test problems are given in Table 5.1 in Section 5. The number of blocks in the singly bordered block diagonal form is 4 and 4 processors are used. We see

that, with the exception of the `lhrc` problems and `graham1`, the reductions in the factorize times achieved by row ordering are substantial. However, we also observe that for a number of our test cases (including the `bayer` problems), the savings in the factor time achieved by reordering are not sufficient to offset the reordering cost: in such cases more than one factorization is needed to justify this overhead.

4.2 Numerical pivoting

For assembled systems, we observe that preordering the matrix to the singly bordered block diagonal form (2.4) allows the multiple front method to incorporate partial pivoting to ensure numerical stability. For each submatrix

$$\begin{pmatrix} A_{ll} & C_l \end{pmatrix}, \tag{4.1}$$

when a column of A_{ll} becomes fully summed, the largest entry in the column is selected as the pivot. `MP43` does not perform an elimination immediately a column becomes fully summed but, while unassembled rows remain, it allows their assembly into the front to continue until the number of fully summed columns is at least as large as the minimum pivot blocksize. The minimum pivot blocksize is a parameter under the user's control and in `MP43` it has the default value of 8. This choice was made on the basis of numerical experimentation (see Scott, 2001*a*). Using a minimum pivot blocksize greater than one enhances the proportion of the computation performed by Level 3 BLAS at the cost of (possibly) increasing the frontal matrix size and hence the number of flops and the number of entries in the factors. There is thus a trade-off between the use of Level 3 BLAS and the computational cost, and the user may wish to experiment with different blocksizes to obtain optimal performance on his or her computer and problems.

For unsymmetric element problems, `MP42` employs threshold pivoting (see, for example, Duff, Erisman and Reid, 1986, section 5.4). Consider the submatrix

$$\begin{pmatrix} A_{ll} & C_l \\ \tilde{C}_l & E_l \end{pmatrix}. \tag{4.2}$$

Once an entry of A_{ll} becomes fully summed it is considered suitable for use as a pivot if it is of absolute value at least as large as the threshold parameter times the entry of largest absolute value in its column (the threshold parameter may be chosen by the user and has default value 0.01). If large entries in $\tilde{C}_l$ prevent a pivot from being chosen, pivoting is delayed, which may lead to an increase in the size of the interface problem (although, in our experience, this increase is generally small when compared to the size of the interface problem). The value of the threshold parameter is under the control of the user; in `MP42` it has a default value of 0.01. `MP42` also uses a minimum pivot blocksize; the default value is 32 (see Scott, 2001*b*).

The code `MP62` is designed for symmetric definite problems and so does not incorporate pivoting. If a zero pivot is encountered (or a pivot with absolute value less than a user-defined quantity), the computation terminates with an error flag. `MP62` again has a default minimum pivot blocksize of 32.

4.3 Use of files for factors and matrix data

The user may choose whether to hold the partial LU factors in main memory or in direct-access files. Sequential files may also be used to store the data that remains in the local frontal matrices after the partial LU decompositions. `MP42` and `MP43` use separate files for the L and U factors plus another for the integer factor data. `MP62` exploits symmetry and so only requires two files, one for the reals and the other for the integer data. Moreover, since the unsymmetric codes use off-diagonal pivoting to maintain stability, they must hold both row and column indices of the variables in the frontal matrix. `MP62` requires only one set of indices and so uses substantially less real and integer storage than the general code `MP42`.

The use of files for the factors reduces main memory requirements and allows larger problems than could otherwise be handled to be solved. It also enables the user to retain the factors for later use. However, the extra I/O involved can increase the overall computational time and so we advise holding the factors and local frontal matrices in main memory unless the factors are to be kept or the problem is too large to be accommodated. In a distributed memory environment, for efficiency it is important that the files are held locally.

If the user only wishes to solve for right-hand sides at the same time as the factorization is performed (that is, the user does not wish to call the solve phase for additional right-hand sides), `MP42` and `MP43` save storage by not retaining the L factor.

The amount of main memory required is also influenced by how the user chooses to supply the matrix data. A number of options are provided. By default, at the start of the computation, the real data for each subproblem $\{A_{ll}, C_l\}$ is held in a direct-access file and the software requires that the data needed by a particular process must be readable by that process. For each subproblem, the data is read element-by-element (or row-by-row) as required by the process to which it is assigned. This minimises main memory requirements and data movement between processes. Alternatively, the user may hold the subproblem data in unformatted sequential files so that each process again reads the data it requires but, in this case, the data for all the elements (or rows) in a subproblem is read in at once. This clearly demands more memory but, again, movement of data between processes is minimised.

Options also exist for the host to read the data for each of the subproblems from sequential files or, alternatively, the user may supply the matrix data using input arrays on the host. The latter form is useful if the host has sufficient memory and the cost of using direct-access or sequential files is high. However, if the data is input or read onto the host, there is an added overhead of sending the appropriate data from the host to the other processes and efficiency will depend on the speed of this communication. Since by default the host is also involved in the block factorizations, this distribution of subproblem data is carried out before the factorization commences.

Finally, to minimise data movement while avoiding the cost of reading data from files, on each process, the user may use input arrays to supply the data for the whole problem or just for the submatrices assigned to it. The former

is suitable for shared memory machines while Stadtherr and Yudong (private communication, 2001) have reported finding the latter useful when using a network of Sun workstations.

Table 4.2: Timings for `MP43` using different input options on a network of Sun workstations.

Identifier	Files on processes	Arrays on host	Arrays on processes
`4cols`	0.36	0.33	0.30
`10cols`	1.32	1.35	1.18
`bayer01`	1.76	1.53	1.36
`bayer04`	1.53	1.57	1.33
`lhr14c`	1.59	1.59	0.79
`lhr71c`	8.99	9.80	7.10

In Table 4.2, we present `MP43` timings using different input options for a subset of our chemical process engineering test problems (see Table 5.1). These results are for analyse, factorize and solve and were obtained by Stadtherr and Lin of the University of Notre Dame using a cluster of Sun Ultra 2/2400 nodes connected using 100 Mbps switched Ethernet. In each case, the singly bordered block diagonal form has 8 blocks and 8 processes are used. In column 2, timings are for the default option of holding the submatrix data in files on the individual processes; the column 3 timings are for the matrix data held in arrays on the host; the final results are for holding the submatrix data in arrays on the individual processes. By comparing columns 2 and 4 we can see that, in this environment, the cost of I/O can represent a significant overhead, while a comparison of columns 3 and 4 illustrates the overhead resulting from sending data from the host to the processes.

5 Numerical results

Throughout this section, N denotes the number of subproblems and p the number of processes.

5.1 `MP43`

We illustrate the performance of `MP43` using the problems listed in Table 5.1. Those in the top half of the table (`4cols` to `lhr71c`) are from chemical process engineering. The remaining problems are included in the sparse matrix collection of Tim Davis (see www.cise.ufl.edu/ davis/sparse). Problem `graham1` is a Jacobian from a Galerkin finite element discretization of the Navier-Stokes equations applied to a two-phase fluid flow problem; `Zhao2` is an electromagnetic matrix; `pesa` is from the subcollection Gaertner and `poli_large` is from Grund. The application areas for `pesa` and `poli_large` are unknown. The *symmetry index* $s(A)$ of a matrix A is the number of matched nonzero off-diagonal entries (that is, the number of nonzero entries a_{ij}, $i \neq j$, for which a_{ji} is also nonzero)

Table 5.1: The test problems for MP43. The problems in the top half of the table are from chemical process engineering.

Identifier	Order	Number of entries	Symmetry index	Interface variables
4cols	11770	43668	0.0159	287
10cols	29496	109588	0.0167	313
bayer01	57735	277774	0.0002	299
bayer04	20545	159082	0.0016	490
icomp	75724	338711	0.0010	314
lhr14c	14270	307858	0.0066	716
lhr34c	35152	764014	0.0015	870
lhr71c	70304	1528092	0.0016	1213
graham1	9035	335504	0.7182	2253
pesa	11738	79566	1.0000	423
poli_large	15575	33074	0.0035	628
Zhao2	33861	166453	0.9225	2784

divided by the total number of off-diagonal nonzero entries. Small values of $s(A)$ indicate a matrix is far from symmetric while values close to 1 indicate an almost symmetric sparsity pattern. We see that the chemical process engineering test problems are all highly unsymmetric and very sparse, while problem pesa has a symmetric symmetric structure. For each problem, we have also included in Table 5.1 the number of interface variables when the problem is partitioned into 8 subdomains using the MONET algorithm of Hu et al. (2000). MONET applied to the chemical process engineering problems produces partitionings with a small interface variables. The interface is also small (less than 10 per cent of the total number of variables) for each of the other test problems except graham1. We anticipate that the relatively large interface for this problem will lead to the solution of the interface problem causing a bottleneck within MP43.

The performance of MP43 is compared with that of the serial frontal code MA42 together with the well-known HSL general sparse direct solver MA48 (Duff and Reid, 1996). We remark that MA48 (and its predecessor MA28) is a benchmark standard in the solution of sparse unsymmetric systems and is frequently used for chemical process engineering problems because it is ideally suited to solving problems that are highly unsymmetric and very sparse. Default values are used for all MA48 control parameters, with diagonal pivoting for problems graham1, pesa, and Zhao2. MA42 and MA48 do not use the bordered block diagonal form (2.4). For MA42, with the exception of problems pesa and Zhao2, the rows of A are preordered using MC62. Since these two problems have a (nearly) symmetric structure, MC60 applied to $A+A^T$ is used. For each solver, wallclock timings (in seconds) are presented for three execution paths, namely:

1. Analyse + Factorize + Solve (AFS) : This is the time required to perform the analyse phase, to determine a pivot sequence, to compute the L and U factors of A, and to perform the forward elimination and back-substitution

operations to solve $Ax = b$ for a single right-hand side b.

2. Fast Factorize (FF) : This is the time taken to factorize a matrix having the same sparsity pattern as one that has already been factorized.

3. Solve (S): This is the time to solve $Ax = b$ by performing forward elimination and back-substitution operations using previously computed L and U factors of A.

In our tests, the factors are held in main memory. A version of `MA42` that does not use BLAS during the solve phase is employed because this has been found to be more efficient for our non-element problems when solving for a single right-hand side.

We first present timings for `MP43` run on $p = 1$, 2, 4 and 8 processors, and compare it with `MA42` and `MA48` run on a single processor. For `MP43`, the number of blocks in the singly bordered block diagonal form is $N = 8$. The times required for AFS are given in Table 5.2. The time taken to preorder the rows of A for `MA42` using `MC62` is included in the analyse time. In Tables 5.3 and 5.4, timings are presented for the fast factorize FF and the solve S, respectively.

Table 5.2: Timings for Analyse + Factorize + Solve (AFS). Numbers in parentheses are times for factorizing the interface problem. The numbers in italics are the speedups for `MP43` compared with using a single process.

Identifier	`MA42`	`MA48`	`MP43` ($N = 8$)					
			$p = 1$	2		4		8	
`4cols`	1.26	2.34	0.77 (0.02)	0.42	*1.83*	0.23	*3.35*	0.18	*4.28*
`10cols`	3.64	16.54	1.92 (0.03)	1.05	*1.83*	0.62	*3.10*	0.43	*4.47*
`bayer01`	8.70	6.52	4.78 (0.03)	2.67	*1.79*	1.74	*2.75*	1.05	*4.42*
`bayer04`	3.52	2.96	2.77 (0.09)	1.61	*1.72*	0.97	*2.86*	0.61	*4.54*
`icomp`	7.61	0.88	4.51 (0.02)	2.45	*1.84*	1.66	*2.72*	0.96	*4.70*
`lhr14c`	4.90	7.38	5.29 (0.14)	2.91	*1.82*	1.71	*3.09*	1.18	*4.48*
`lhr34c`	13.56	24.20	15.82 (0.54)	8.52	*1.86*	5.00	*3.16*	3.69	*4.29*
`lhr71c`	32.86	51.26	35.01 (1.10)	18.92	*1.85*	10.26	*3.41*	7.10	*4.93*
`graham1`	18.10	184.0	17.29 (9.02)	13.54	*1.28*	11.43	*1.51*	10.51	*1.65*
`pesa`	2.47	1.90	1.22 (0.05)	0.69	*1.77*	0.44	*2.77*	0.32	*3.81*
`poli_large`	1.87	0.06	0.90 (0.04)	0.53	*1.70*	0.36	*2.50*	0.21	*4.28*
`Zhao2`	67.54	656.5	45.35 (11.2)	28.89	*1.57*	20.56	*2.21*	18.12	*2.50*

In Table 5.2, the numbers in parentheses in column 4 are the times taken to solve the interface problem and the numbers in italics are the speedups for `MP43` on 2, 4, 8 processes compared with the time on a single process. As noted earlier, the interface problem is solved by a single process. The chemical process engineering problems and problems `pesa` and `poli_large` have very small interfaces (see Table 5.1) and thus solving the interface problem represents a tiny fraction of the overall factorization time and does not cause a bottleneck. However, for problem `graham1`, which has a much larger interface, the time

taken taken for the interface problem dominates the total time and limits the speed up obtained on increasing the number of processes.

Table 5.3: Timings for Fast Factorize (FF).

Identifier	MA42	MA48	MP43 ($N = 8$)			
			$p = 1$	2	4	8
4cols	0.66	0.32	0.45	0.26	0.19	0.12
10cols	1.94	2.08	1.13	0.64	0.38	0.28
bayer01	5.25	0.65	2.69	1.54	1.05	0.63
bayer04	1.60	0.31	1.17	0.71	0.45	0.36
icomp	3.53	0.13	2.04	1.16	0.92	0.51
lhr14c	1.58	1.10	1.80	1.08	0.75	0.63
lhr34c	4.99	3.66	6.58	3.82	2.45	2.19
lhr71c	14.45	7.61	14.82	8.77	5.28	4.39
graham1	13.12	35.06	13.57	11.58	10.40	9.84
pesa	1.97	0.31	0.81	0.47	0.32	0.26
poli_large	1.33	0.01	0.64	0.39	0.27	0.16
Zhao2	63.83	171.7	43.67	28.01	19.94	17.72

We see that, on a single processor the AFS time for MP43 is competitive with that for MA42: for mnay problems MP43 is faster but for the lhr problems the converse is true. MP43 is, of course, designed to be run on more than one processor and, for each problem, using only two processors, MP43 significantly outperforms MA42. For the problems with a small interface, the performance of MP43 improves as the number of processors increases to 4 and to 8, although for the smallest problems the speedups achieved are less than for the larger problems.

Table 5.4: Timings for Solve (S).

Identifier	MA42	MA48	MP43 ($N = 8$)			
			$p = 1$	2	4	8
4cols	0.043	0.017	0.039	0.024	0.019	0.016
10cols	0.134	0.082	0.103	0.072	0.050	0.038
bayer01	0.284	0.111	0.245	0.171	0.126	0.108
bayer04	0.092	0.038	0.092	0.064	0.045	0.036
icomp	0.182	0.065	0.198	0.144	0.121	0.100
lhr14c	0.082	0.070	0.105	0.069	0.048	0.042
lhr34c	0.246	0.195	0.307	0.203	0.139	0.130
lhr71c	0.651	0.404	0.654	0.436	0.267	0.260
graham1	0.333	0.294	0.228	0.186	0.142	0.155
pesa	0.100	0.034	0.061	0.046	0.025	0.020
poli_large	0.039	0.007	0.045	0.034	0.026	0.021
Zhao2	1.317	0.950	0.919	0.606	0.384	0.351

The analyse phase of MA48 can be relatively slow so that, for most of our test problems, the AFS time for MP43 on a single processor is faster than for MA48. We remark that, although on a single processor the FF time for MA48 is

generally faster than for MP43, the MA48 fast factorization is more restrictive. This is because MP43 only reuses the row assembly order and employs partial pivoting for stability both for the initial and for all subsequent factorizations. By contrast, the FF phase of MA48 uses exactly the same pivot sequence as was computed on the initial factorization. It should, therefore, only be used if the user is confident that the changes to the numerical values of the matrix have not made this sequence unsuitable. To check the computed solution, the user may decide to use the iterative refinement option offered by the solve phase of MA48. This can add a significant overhead to the execution time, but is not included in our results (see Duff and Reid, 1996). We remark that MA48 does offer an additional factorization option for the case where some columns are unchanged since a previous factorization but we do not present results for this "semi-fast" option.

In each test case, the solve time for MA48 is less than for MA42 and, on one or two processes, is usually less than for MP43. By increasing the number of processes to 4 or 8, the MP43 solve is generally able to outperform MP48.

5.2 MP42

We now illustrate the performance of MP42 using the finite element problems listed in Table 5.5. These problems were supplied by Christian Damhaug of Det Norske Veritas, Norway. In each case, the problem is divided into 4 subproblems using Chaco (Hendrickson and Leland, 1995). We list the number of elements and interface variables for each subproblem, together with the total number of interface variables. We observe that, in general, Chaco is able to produce

Table 5.5: Test problems for MP42 ($N = 4$).

Identifier	Order	Elements	Total Interface	Subdomain	Elements/ Interface
opt1	15449	977	1338	1	294 / 584
				2	252 / 588
				3	190 / 838
				4	241 / 678
trdheim	22098	813	348	1	206 / 150
				2	212 / 120
				3	196 / 240
				4	199 / 108
thread	29736	2176	3809	1	586 / 1266
				2	597 / 1275
				3	497 / 2532
				4	496 / 2541
mt1	97578	5328	2748	1	586 / 1266
				2	597 / 1275
				3	497 / 2532
				4	496 / 2541

a partitioning with a (nearly) equal number of subdomains but some of the subdomains have a significantly larger number of interface variables.

In Table 5.6 wallclock timings (in seconds) are given for analyzing, factorizing and solving (AFS) for a single right-hand side using `MP42` run on 1 to 4 processes; for comparison, results are included for `MA42`. For each problem except `trdheim`, `MP43` outperforms `MA42` on a single process. This is because, for these examples, the flop count is significantly reduced by partitioning the domain and using the multiple front algorithm. For example, for `opt1` the flop counts for `MA42` and `MP42` are, respectively, $1.15 * 10^{10}$ and $6.57 * 10^{9}$. Similar findings were reported by Duff and Scott (1994).

Table 5.6: AFS timings for `MA42` and `MP42` ($N = 4$). Numbers in parentheses are times for factorizing the interface problem. The numbers in italics are the speedups for `MP42` compared with using a single process.

Identifier	MA42	MP42							
		$p = 1$		2		3		4	
opt1	60.6	37.7	(10.4)	25.8	*1.46*	22.0	*1.71*	20.5	*1.84*
trdheim	5.8	7.2	(0.18)	4.6	*1.56*	2.9	*2.48*	2.9	*2.48*
thread	1621	846	(241)	558	*1.52*	452	*1.86*	450	*1.86*
mt1	2444	1130	(97)	663	*1.70*	618	*1.83*	447	*2.53*

Good speedups for `MP42` are achieved by using 2 processes but the gains are smaller when the number of processes is increased further. In particular, we observe that, for the first three test cases, there is essentially no speedup when 4 processes are used rather than 3. We recall that we are using 4 subdomains and so, if `MP42` is run on 3 processes, one process must factorize two subdomains and we might anticipate this producing a load imbalance. However, if we look for example in greater detail at problem `thread`, we see from Table 5.6 that, although subdomains 3 and 4 have fewer elements than subdomains 1 and 2, subdomains 3 and 4 each have approximately twice the number of interface variables. As already noted, once an interface variable enters the front, it remains there, increasing the flop count and storage requirements. This is what happens in the case of `thread`: the `MP42` flop counts for subdomains 3 and 4 are almost double those for subdomains 1 and 2. Thus good load balance is achieved by using 3 processes and factorizing both subdomains 1 and 2 on a single process.

For our finite-element test examples, the solution of the interface problem is generally a much more significant proportion of the total factorization time than we found for most of the problems reported on in the previous section. For problem `opt1`, on a single process, the interface factorization represents approximately 27 per cent of the total factorization time; as the number of processes increases, this proportion increases to 50 per cent on 4 processes and represents a bottleneck in the computation. If the number of blocks is increased, so will the number of interface variables and solving the interface problem will quickly come to dominate the overall computational cost. This demonstrates the importance of obtaining a good initial partitioning. For some problems on irregular domains, it is not clear how this can best be achieved.

5.3 MP62

Finally, we present some results to demonstrate the efficiency of the symmetric positive definite finite element parallel solver MP62. These were provided by Dr Milan Mihajlovic of The University of Manchester. In these tests, MP62 is used

Table 5.7: Factorization times for MP62. N denotes the number of subdomains and n the number of variables.

Subdomains	$n =$	20449	36481	82369	146689	330625
$N = 2 \times 2$	$p = 1$	9.9	29.2	141	447	2219
	2	5.0	14.8	71.3	222	1114
	4	2.6	7.5	36.1	112	560
$N = 4 \times 4$	$p = 1$	3.5	10.3	46.6	137.9	670
	2	2.0	5.4	24.1	70.8	342
	4	1.1	3.0	12.9	37.3	177
	8	0.7	1.8	7.2	20.5	95
$N = 8 \times 8$	$p = 1$	1.8	4.3	17.1	47.3	190
	2	1.1	2.6	9.9	26.4	101
	4	0.8	1.7	6.3	15.8	54
	8	0.6	1.3	4.5	10.6	33

to factorize a discrete Dirichlet Laplacian on a unit square domain Ω using a uniform mesh and piecewise linear elements. Ω is split into $N = 2 \times 2$, 4×4 and 8×8 equal subdomains. In Tables 5.7 and 5.8 factorization and solve times are presented for a number of different mesh sizes, using up to $p = 8$ processes (n is the total number of variables). For this two dimensional model problem with

Table 5.8: Solve times for MP62. N denotes the number of subdomains and n the number of variables.

Subdomains	$n =$	20449	36481	82369	146689	330625
$N = 2 \times 2$	$p = 1$	0.17	0.39	1.25	2.88	12.68
	2	0.11	0.25	0.78	1.78	7.30
	4	0.06	0.14	0.43	0.98	3.97
$N = 4 \times 4$	$p = 1$	0.16	1.07	2.40	9.40	9.40
	2	0.11	0.68	1.51	5.80	5.80
	4	0.06	0.41	0.89	3.35	3.35
	8	0.05	0.27	0.58	1.78	1.78
$N = 8 \times 8$	$p = 1$	0.20	0.40	1.11	2.12	7.11
	2	0.13	0.26	0.68	1.36	4.49
	4	0.08	0.16	0.42	0.83	2.54
	8	0.06	0.12	0.30	0.58	1.49

equal subdomains, good speedups are obtained for the factorize and solve phases as the number of processes increases. In particular, for the factorization phase for the largest problems, speedups close to 2 are achieved using 2 processes and exceed 3.5 using 4 processes. We deduce that load balancing is good and that

the solution of the interface problem is not significantly effecting the overall performance.

6 Concluding remarks

We have designed and developed general-purpose multiple front codes for solving large sparse systems of linear equations in parallel. The performance of the codes has been illustrated on an Origin2000 using a range of practical problems. The software is fully portable and may be used on any computer on which a Fortran 90 compiler and MPI are available. Results on a Cray T3E and a 2-processor Compaq DS20 have been presented in Scott (2001*b*, 2001*a*); Stadtherr and Lin (private communication, 2001) have also reported using `MP43` successsfully on a network of Sun workstations.

A potential limitation of the current software is the requirement for the user to carry out the partitioning into subproblems before the start of the computation. Furthermore, the sequential solution of the interface problem can cause a bottleneck as the number of processes increases. This effectively restricts the use of the multiple front approach to a relatively small number of subproblems and processes. Nevertheless, our results demonstrate that significant improvements over serial direct solvers can be achieved.

The parallel frontal solvers `HSL_MP42`, `HSL_MP43`, and `HSL_MP62`, together with the implementation `HSL_MC66` of the MONET algorithm used in this paper, and the ordering routines `MC60`, `MC62`, and `MC63`, are all available for use under licence through HSL. Anyone interested in using these codes (or the serial sparse direct solvers `MA42` and `MA48`) should refer to the website www.cse.clrc.ac.uk/Activity/HSL for further details.

7 Acknowledgements

I am very grateful to Milan Mihajlovic for providing me with the results presented in Section 5.3 and also for helpful feedback concerning his experiences using the frontal solvers. I am indebted to the Department of Computer Science at The University of Manchester and, in particular, to Michael Bane, for providing me with access to their Origin2000 and use of the cpuset facility. I would also like to thank Mark Stadtherr and Youdong Lin of The University of Notre Dame for the results given in Table 4.2, and my colleague Iain Duff for comments on a draft of this paper.

References

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.

C.W. Bomhof and H.A. van der Vorst. A parallel linear system solver for circuit simulation problems. *Numerical Linear Algebra with Applications*, **7**, 649–665, 2000.

E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. *in* 'Proceedings of the 24th National Conference of the ACM'. Brandon Systems Press, 1969.

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 design of MA48, a code for the direct solution of sparse unsymmetric linear systems of equations. *ACM Trans. Mathematical Software*, **22**, 187–226, 1996.

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.

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 and J.A. Scott. MA62 – a frontal code for sparse positive-definite symmetric systems from finite-element applications. *in* M. Papadrakakis and B. Topping', eds, 'Innovative Computational Methods for Structural Mechanics', pp. 1–25, Edinburgh, 1998. Saxe-Coburg Publications.

I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, England, 1986.

G.H. Golub, A.H. Sameh, and V. Sarin. A parallel balanced scheme for banded linear systems. *Numerical Linear Algebra with Applications*, **8**, 297–316, 2001.

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

P. Hood. Frontal solution program for unsymmetric matrices. *Inter. Journal on Numerical Methods in Engineering*, **10**, 379–400, 1976.

HSL. A collection of Fortran codes for large scale scientific computation, 2002. Full details from www.cse.clrc.ac.uk/Activity/HSL.

Y.F. Hu, K.C.F. Maguire, and R.J. Blake. A multilevel unsymmetric matrix ordering for parallel process simulation. *Computers in Chemical Engineering*, **23**, 1631–1647, 2000.

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

J.U. Mallya, S.E. Zitney, S. Choudhary, and M.A. Stadtherr. A parallel block frontal solver for large scale process simulation: reordering effects. *Computers in Chemical Engineering*, **21**, S439–S444, 1997.

MPI. A message-passing interface standard. *Inter. Journal of Supercomputer Applics*, **8**, 1994. Special edition on MPI.

J.K. Reid and J.A. Scott. Ordering symmetric sparse matrices for small profile and wavefront. *Inter. Journal on Numerical Methods in Engineering*, **45**, 1737–1755, 1999.

J.A. Scott. Element resequencing for use with a multiple front algorithm. *Inter. Journal on Numerical Methods in Engineering*, **39**, 3999–4020, 1996.

J.A. Scott. A new row ordering strategy for frontal solvers. *Numerical Linear Algebra with Applications*, **6**, 1–23, 1999*a*.

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

J.A. Scott. Row ordering for frontal solvers in chemical process engineering. *Computers in Chemical Engineering*, **24**, 1865–1880, 2000.

J.A. Scott. The design of a portable parallel frontal solver for chemical process engineering problems. *Computers in Chemical Engineering*, **25**, 1699–1709, 2001*a*.

J.A. Scott. A parallel solver for finite element applications. *Inter. Journal on Numerical Methods in Engineering*, **50**, 1131–1141, 2001*b*.

J.A. Scott. Two-stage ordering for unsymmetric parallel row-by-row frontal solvers. *Computers in Chemical Engineering*, **25**, 323–332, 2001*c*.

S.W. Sloan. A FORTRAN program for profile and wavefront reduction. *Inter. Journal on Numerical Methods in Engineering*, **28**, 2651–2679, 1989.

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.