Parallel Frontal Solvers for Large Sparse Linear Systems

JENNIFER A. SCOTT
Rutherford Appleton Laboratory

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 using problems arising from real applications.

Categories and Subject Descriptors: G.1.0 [**Numerical Analysis**]: General—*Numerical algorithms*; G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*Sparse, structured, and very large systems*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Sparse matrices, linear systems, finite-elements, frontal method, parallel processing

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

$$Ax = b. \tag{1}$$

Solving these systems is generally the most computationally expensive step in the simulation. As time-dependent three-dimensional simulations are now

This work was funded by the EPSRC grant GR/R46441.
Author's address: Computational Science and Engineering Department, Atlas Centre, Rutherford Appleton Laboratory, Oxon OX11 0QX, England; email: J.A.Scott@rl.ac.uk.

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

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

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 in the innermost loop of the computation. In particular, high-level BLAS kernels [Dongarra et al. 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 et al. [1987] and Zang and Liu [1991] (see also Zone and Keunings [1991] and, for non-element problems, Mallya et al. [1997]). It is this so-called *multiple front* approach that is implemented in the software reported on in this paper.

The aim of this article 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 article is organised as follows. In Section 2, the multiple front method is reviewed; we then look at how the method can be implemented and, in particular, how the frontal method can be used to perform partial LU factorizations of the subproblems. In Section 3, we discuss software considerations in the design of our parallel codes and then, in Section 4, we present some numerical results to illustrate the performance of each of the parallel frontal solvers. Finally, concluding remarks are made in Section 5.

2. THE MULTIPLE FRONT ALGORITHM

2.1 Element Problems

We start by recalling the multiple front approach for finite-element problems. We first partition 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

$$\begin{pmatrix} A_{11} & & & & C_1 \\ & A_{22} & & & C_2 \\ & & \dots & & . \\ & & & A_{NN} & C_N \\ \tilde{C}_1 & \tilde{C}_2 & \dots & \tilde{C}_N & \sum_{l=1}^{N} E_l \end{pmatrix}, \tag{3}$$

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 $k \times n_l$ matrices, respectively (with $\tilde{C}_l = C_l^T$ in the symmetric case). If $k_l \le k$ 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

$$\begin{pmatrix} A_{ll} & C_l \\ \tilde{C}_l & E_l \end{pmatrix}, \tag{4}$$

(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 3.2). These variables are added to the border and k is increased. If F_l is the local Schur complement matrix 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

$$A = P \begin{pmatrix} L_1 & & & & \\ & L_2 & & & \\ & & \dots & & . \\ & & & L_N & \\ \tilde{L}_1 & \tilde{L}_2 & \dots & \tilde{L}_N & I \end{pmatrix} \begin{pmatrix} U_1 & & & & \tilde{U}_1 \\ & U_2 & & & \tilde{U}_2 \\ & & \dots & & . \\ & & & U_N & \tilde{U}_N \\ & & \dots & & F \end{pmatrix} Q, \tag{5}$$

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 Schur complement matrices F_l as an elemental matrix, the interface matrix F may also be factorized using the frontal method. Forward eliminations and back-substitutions complete the solution.

2.2 Non-Element Problems

The multiple front approach can be extended to general sparse systems with an unsymmetric sparsity pattern by preordering the matrix to singly bordered block diagonal form

$$\begin{pmatrix} A_{11} & & & & C_1 \\ & A_{22} & & & C_2 \\ & & \dots & & . \\ & & & A_{NN} & C_N \end{pmatrix}, \tag{6}$$

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

$$\begin{pmatrix} A_{ll} & C_l \end{pmatrix} \tag{7}$$

(again, any zero columns are assumed to have been removed). Assuming A is nonsingular, as the rows of (7) 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 Schur complement 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 matrices may again be factorized using the frontal method, followed by block forward eliminations and back-substitutions.

In the remainder of this article, 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 (4) and, for non-element problems, to (7).

2.3 Implementing the Multiple Front Algorithm Using a Frontal Solver

We now outline how the multiple front algorithm is implemented within our software. Our frontal code MA42 [Duff and Scott 1996] is quite complicated and represents considerable programming effort; furthermore, experience has shown it to be both reliable and efficient. Rather than writing a multiple front code completely from scratch, we felt a more efficient approach would be to try and exploit MA42 within our multiple front software, with a minimum number of modifications. In this section, we look at how this is done.

We first need to understand the structure of the frontal solver MA42. The code has three distinct phases:

—*Symbolic analyze:* A symbolic analysis of the sparsity pattern of the matrix determines when each column of A is fully summed.

—*Factorize:* Using the information from the symbolic analyze, the frontal factorization is carried out (if necessary incorporating pivoting for numerical stability).

—*Solve:* The computed L and U factors are used to perform forward elimination followed by back-substitution.

An important design feature of MA42 is its use of a reverse communication interface for the analyze and factorize phases. To avoid the need for the user to have all the elements or rows of the matrix 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 analyze and factorize phases just once (integer data only for the analyze phase). This minimizes main memory requirements and allows the user to generate the matrix entries (or read them from secondary storage) as they are required. It also enables us to use MA42 for our multiple front software, as we now discuss.

A key idea is the introduction of a *guard* element or row for each subproblem. If we simply pass the subproblem $\{A_{ll}, C_l\}$ element-by-element (or row-by-row) to the analyze and factorize phases of MA42, a complete LU factorization of the subproblem will be performed. However, the interface variables are not fully summed within the subproblem, so we must prevent them from being eliminated until all the contributions from the other subproblems have been assembled. We do this by flagging them as not being fully summed within the subproblem. Consider the non-element case and the submatrix (7). We add an extra row d_l to the end of this submatrix. The added 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 pass to the analyze phase of MA42 the $(m_l + 1) \times (n_l + k_l)$ matrix $\hat{A}_l$ given by

$$\hat{A}_l = \begin{pmatrix} A_{ll} & C_l \\ 0 & d_l^T \end{pmatrix}. \tag{8}$$

On exit from the final call, all the columns of C_l with a nonzero entry are flagged 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 of MA42. 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} and only a partial factorization is performed.

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 analyze 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 introduction of a guard element (or row) allows us to use the analyze and factorize phases of MA42 unchanged. All that is needed is an additional subroutine to save, in a convenient form, the Schur complement matrices from the partial factorizations for assembly for the interface problem. In the multiple front approach it is necessary to perform the forward elimination and back substitution on the subproblems quite separately. The solve routine in MA42 combines these two steps but as it has one internal subroutine for the forward elimination and another for the back substitution, it is straightforward to call these separately.

3. SOFTWARE CONSIDERATIONS

Following the discussion in the previous section, we can now summarize the main steps in the multiple front algorithm as follows:

Multiple Front Algorithm (p processors):

Initialize (single processor):
Assign an equal (or nearly equal) number of subproblems to each of the p processors, and for each subproblem generate the guard element (or row).

Reorder (optional) (parallel):

—The elements (or rows) within each subproblem $\{A_{ll}, C_l\}$ are reordered to minimize the frontsize 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 processors 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, processor p 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 L and U. The remaining Schur complement matrix is sent to the processor responsible for interface problem.

Interface problem (single processor):
Factorization of the interface matrix.

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

In the remainder of this section, we discuss some of the details of our software that implements this algorithm. The multiple front 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. HSL_MP62 is written using the symmetric positive definite finite-element frontal solver MA62 [Duff and Scott 1998]; the other two codes use MA42. We remark that our software adopts the naming convention of the HSL Library. The prefix HSL is used within the 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. The "MP" denotes the codes that belong to the chapter of MPI dependent packages. 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 was chosen for message passing 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 processors. 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 other processes, factorizes and solves the interface problem, and generally overseas the computation. The host also participates by default with the other processes in local ordering and in generating the partial LU decompositions.

3.1 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 [2001c]). 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.

Standard 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 that 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 generalization of Sloan's profile reduction algorithm [Sloan 1989]. 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 [1999a], which also have their origins in Sloan's profile reduction algorithm. For the row-by-row multiple front method, Scott [2001c] 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 used by default within our multiple front code MP43. MC62 is also 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 reusing an ordering can be particularly advantageous for MP43. In Table I, we compare the time on an SGI Origin 2000 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

Table I. Timing in Seconds on an Origin 2000 for the Factorize Phase of MP43 With and Without Row Ordering

Identifier	Factorize no ordering	Factorize with ordering	Row ordering	Ordering plus factorize
4cols	0.36	0.06	0.04	0.10
10cols	1.59	0.16	0.09	0.25
bayer01	0.69	0.44	0.25	0.67
bayer04	0.44	0.24	0.14	0.38
icomp	0.65	0.30	0.28	0.58
lhr14c	0.35	0.32	0.31	0.63
lhr34c	1.10	1.00	0.87	1.87
lhr71c	2.51	2.20	1.81	4.01
graham1	4.94	5.57	0.41	5.98
pesa	3.56	0.17	0.04	0.21
poli_large	0.44	0.24	0.03	0.27
Zhao2	172.30	7.73	0.11	7.84

and 4 (that is, the time for row ordering plus factorizing the matrix). Details of the test problems are given in Table III in Section 4. 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 problem graham1, row ordering leads to a reduction in the factorize time and for many problems the reduction is substantial. However, for a number of our test cases (including the lhr 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.

3.2 Numerical Pivoting

For non-element problems, we observe that preordering the matrix to the singly bordered block diagonal form (6) allows the multiple front method to incorporate partial pivoting to ensure numerical stability. For each submatrix

$$(A_{ll} \quad C_l), \tag{9}$$

when a column of A_{ll} becomes fully summed, the largest entry in that column is selected as the pivot. MP43 does not perform an elimination as soon as a column becomes fully summed but continues to assemble rows into the front until either the number of fully summed columns is at least as large as the minimum pivot blocksize or no further rows remain. 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 [Scott 2001a]. 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 finite element problems, MP42 employs threshold pivoting (see, for example, Duff et al. [1986], Section 5.4). Consider the submatrix

$$\begin{pmatrix} A_{ll} & C_l \\ \tilde{C}_l & E_l \end{pmatrix}. \tag{10}$$

Once an entry of A_{ll} becomes fully summed it is tested for use as a pivot. Suppose the pivot candidate lies in column k of A_{ll}. The whole of column k in submatrix (10) (including the border $\tilde{C}_l$) is searched for the entry of largest absolute value. Since column k has no entries in any other submatrix, this entry is the largest in its column within the whole matrix (3). The pivot candidate is accepted if it is of absolute value at least as large as the threshold parameter times the largest entry. Testing against **all** the entries in its column ensures threshold pivoting is implemented correctly but can result in large entries in $\tilde{C}_l$ preventing a pivot from being chosen. If it is not possible to stably choose a pivot from within the submatrix, pivoting is delayed and the frontal matrix that remains after all possible stable eliminations have been performed within the submatrix will be larger than was predicted during the analyze phase (that is, the frontal matrix will contain rows and columns that are additonal to those corresponding to the k_l interface variables). As a result, the size of the interface problem will also be larger than predicted (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 [Scott 2001b].

The code MP62 is designed only for symmetric definite problems and so does not incorporate pivoting. Clearly, if used for an indefinite problem, the factorization may not be stable but the computation will only terminate if a zero pivot is encountered (or a pivot with absolute value less than a user-defined quantity). MP62 again has a default minimum pivot blocksize of 32.

3.3 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

Table II. Timing in Seconds for MP43 Using Different Input Options on a Network of 8 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

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 processor must be readable by the processor executing the 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 minimizes main memory requirements and data movement between processors. Alternatively, the user may hold the subproblem data in unformatted sequential files so that each processor 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 processors is minimized.

Options also exist for the user to supply all the subproblem data on the host, either in sequential files or in input arrays. This may be most convenient for the user but involves the added overhead of sending the appropriate subproblem data from the host to the remaining processors. Since by default the host is also involved in the subproblem factorizations, this distribution of data is carried out before the factorizations commence, thus each processor must have sufficient memory to store the data for all the subproblems assigned to it.

Finally, if the user wishes to avoid the overhead of reading from files, he or she may instead use input arrays on each processor to supply the data for the subproblems assigned to it. Stadtherr and Lin (private communication, 2001) have reported finding this option useful when using a network of Sun workstations. In Table II, we present MP43 timing using different input options for a subset of our chemical process engineering test problems (see Table III). These results are for analyze, 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 processors are used. In column 2, the timings are for the default option of holding the submatrix data in files on the individual processors; in column 3 the 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 processors. 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 other processors.

3.4 Finding a Good Preordering

As already stated, the multiple front method requires the matrix A to be preordered to bordered block diagonal form. Once the partial factorization of the subproblems is complete, there remains an interface problem. In the software reported on in this paper, the interface problem is solved using a single processor. To prevent this step from 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 minimize 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 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 [Mallya et al. 1997] and this may render reordering unnecessary.

When no suitable 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 (www–users.cs.umn.edu/∼karypis/metis/). For the highly unsymmetric problems that arise in chemical process engineering, the MONET algorithm of Hu et al. [2000] has been shown to perform extremely well, producing very small interfaces for modest-sized N with good row balance (blocks with similar number of rows). An implementation of this algorithm is available within the mathematical software library HSL as routine HSL_MC66 and is reported on by Duff and Scott [2002]. Alternatively, if the matrix can be permuted to banded

Table III. 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
4cols	11770	43668	0.0159
10cols	29496	109588	0.0167
bayer01	57735	277774	0.0002
bayer04	20545	159082	0.0016
icomp	75724	338711	0.0010
lhr14c	14270	307858	0.0066
lhr34c	35152	764014	0.0015
lhr71c	70304	1528092	0.0016
graham1	9035	335504	0.7182
pesa	11738	79566	1.0000
poli_large	15575	33074	0.0035
Zhao2	33861	166453	0.9225

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

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

and this can be permuted to the bordered form

$$\begin{pmatrix} A_{11} & & C_1 \\ & A_{22} & C_2 \end{pmatrix}. \tag{12}$$

For the interface problem to be small, the bandwidth needs to be narrow, ideally with sparsity within the band. This approach to subdividing the matrix is used in a recent paper by Golub et al. [2001].

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].

4. NUMERICAL RESULTS

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

4.1 MP43

We first present results for the row-by-row multiple front solver MP43 computed on an SGI Origin 2000 at the University of Manchester with 128 R12000 processors 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.

We illustrate the performance of MP43 using the problems listed in Table III. Those in the top half of the table (4cols to lhr71c) are from chemical process

Table IV. The Number of Interface Variables and the Timing (in seconds) for Preordering the Test Problems and Factorizing Using MP43 on an Origin 2000 ($p = N = 8$)

	MONET			MC62		
Identifier	Interface variables	Preorder time	Factorize time	Interface variables	Preorder time	Factorize time
4cols	212	0.90	0.05	440	0.15	0.08
10cols	312	1.95	0.10	633	0.37	0.15
bayer01	375	4.45	0.22	904	0.92	0.53
bayer04	627	2.74	0.15	954	0.65	0.19
icomp	474	6.22	0.21	941	1.12	0.25
lhr14c	664	3.67	0.22	1061	1.33	0.27
lhr34c	909	9.86	0.60	1509	3.54	0.81
lhr71c	863	18.69	1.63	1845	7.80	2.72
graham1	2207	4.69	4.72	4202	1.88	9.55
pesa	442	1.01	0.11	1163	0.19	0.43
poli_large	709	0.94	0.07	1524	0.10	0.16
Zhao2	2998	3.27	10.19	5734	0.57	41.88

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) 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 structure.

4.1.1 *Preordering.* In Table IV, we list the number of interface variables for the test problems after preordering using the HSL_MC66 implementation of the MONET algorithm ($N = 8$). The times for preordering to singly bordered form and the subsequent factorization using MP43 with $p = 8$ are also given. Additionally, we present the corresponding statistics for preordering to singly bordered block form by ordering the rows using the row ordering code MC62 [Scott 1999a] and then assigning the first n/N rows to block 1, followed rows $n/N + 1$ to $2 * n/N$ to block 2, and so on. We see that the MONET algorithm applied to the chemical process engineering problems produces partitionings with a small number of interface variables. The interface is also small (less than 5 per cent of the total number of variables) for problems pesa and poli_large.

The simpler approach of employing MC62 produces significantly larger interfaces (for some problems, the interface size is more than twice that for HSL_MC66 ordering). Furthermore, although the MC62 ordering produces blocks with an equal (or almost equal) number of rows, closer examination shows that the number of interface variables in the blocks can be very different. For example, for problem bayer01 the number of columns lying in the border ranges from 62 to 370. This leads to a significant imbalance between the number of numerical operations and the time required to factor each of the blocks (the times vary from 0.12 to 0.28 seconds). For the MONET ordering applied to the same problem, the number of border columns ranges from 49 to 159 (and the factor times vary from 0.12 to 0.18 seconds). The greater imbalance between the blocks for the MC62 orderings together with the larger interfaces results in slower total factorization times. However, the MC62 orderings are significantly cheaper to compute. Whether the extra cost of using HSL_MC66 is justified may depend on the number of times the user wishes to factorize and solve a matrix having the same (or a similar) sparsity pattern. In many applications, the user will need to do this. For example, in solving a nonlinear system using Newton's method, a single Newton step will correspond to solving (1) and repeated factorizations of matrices with the same pattern as A will be needed. 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.

The MP43 results presented in the remainder of this section use the HSL_MC66 ordering (with $N = 8$).

4.1.2 MP43 *Timing.* We now compare the performance of MP43 with that of the serial frontal code MA42 and the well-known HSL general sparse direct solver MA48 [Duff and Reid 1996]. MA48 implements a sparse variant of Gaussian elimination, using conventional sparse data structures and incorporating threshold pivoting for numerical stability. To maintain sparsity, the pivot ordering is based on a modification of the Markowitz criterion. We remark that MA48 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 the repeated solution of problems that are highly unsymmetric and very sparse. Default values are used for all MA42 and MA48 control parameters, with diagonal pivoting in MA48 for the (almost) symmetrically structured problems graham1, pesa, and Zhao2. MA42 and MA48 do not use the bordered block diagonal form (6) but for MA42 it is necessary to obtain a good assembly order for the rows. The problems with a (nearly) symmetric structure are preordered using the profile reduction code MC60 [Reid and Scott 1999] applied to the sparsity pattern of $A + A^T$; the remaining problems are preordered using the MSRO algorithm implemented within MC62 [Scott 1999a]. We include the row ordering time within the MA42 analyze time.

For each solver, timing is presented for three execution paths, namely:

(1) Analyze + Factorize + Solve (AFS) : This is the time required to perform the analyze 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) Factorize (F2) : This is the time taken to factorize a matrix having the same sparsity pattern as one that has already been factorized.

Table V. Timing in Seconds for Analyze + 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 Processor

Identifier	MA42	MA48	MP43 ($N = 8$)						
			$p = 1$	2		4		8	
4cols	0.62	1.07	0.37 (0.01)	0.20	*1.83*	0.12	*3.14*	0.08	*4.42*
10cols	1.69	2.93	0.98 (0.02)	0.55	*1.82*	0.32	*3.16*	0.20	*4.89*
bayer01	4.17	2.75	2.42 (0.02)	1.34	*1.80*	0.77	*3.13*	0.48	*4.99*
bayer04	1.64	1.15	1.28 (0.06)	0.72	*1.77*	0.50	*2.58*	0.30	*4.29*
icomp	3.70	0.42	2.35 (0.01)	1.44	*1.64*	0.89	*2.64*	0.54	*4.33*
lhr14c	2.27	3.08	2.14 (0.08)	1.19	*1.79*	0.69	*3.11*	0.47	*4.55*
lhr34c	6.36	9.90	5.95 (0.19)	3.27	*1.82*	2.03	*2.93*	1.20	*4.94*
lhr71c	15.72	20.47	13.23 (0.19)	7.29	*1.85*	4.36	*3.03*	2.91	*4.54*
graham1	8.06	62.86	8.07 (4.12)	6.38	*1.26*	5.35	*1.51*	5.03	*1.61*
pesa	1.16	0.84	0.59 (0.03)	0.33	*1.79*	0.23	*2.54*	0.15	*3.83*
poli_large	0.92	0.03	0.45 (0.02)	0.24	*1.84*	0.15	*3.00*	0.10	*4.47*
Zhao2	29.08	359.20	22.68 (6.52)	15.22	*1.49*	11.16	*2.03*	10.36	*2.19*

(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. The times required for AFS are given in Table V. In Tables VI and VII, the timings are presented for factorize F2 and solve S, respectively. The F2 time for MP43 is less than for the first factorization of a matrix of a given sparsity pattern because the first factorization includes performing the analysis phase for the interface problem; on a subsequent factorization, the row assembly order for the interface problem is reused. However, for problems with a small interface, the difference between the cost of a first and a subsequent factorization is small, typically less than 20%. We remark that MA48 has a fast factorize option for factorizing a matrix having the same sparsity pattern as one that has already been factorized, using the **same** pivot sequence as the earlier factorization. Results for this option are included in Table VI since it can be substantially faster than a standard factorization but it may be numerically unstable if the matrix entries are markedly different from the earlier factorization.

In Table V, the numbers in parentheses in column 4 are the times taken to solve the interface problem and the numbers in italics in columns 5 to 8 are the speedups for MP43 on 2, 4, and 8 processors compared with the time on a single processor. As noted earlier, the interface problem is solved by a single processor. The chemical process engineering problems and problems pesa and poli_large have very small interfaces (see Table IV), thus solving the interface problem represents a tiny fraction of the overall factorization time and does not cause a bottleneck. However, graham1 and Zhao2 have much larger interfaces and, for

Table VI. Timing in Seconds for Factorize (F2). Numbers in Parentheses are MA48 Fast Factorize Times

Identifier	MA42	MA48	MP43 ($N = 8$)			
			$p = 1$	2	4	8
4cols	0.66	0.23 (0.16)	0.18	0.09	0.05	0.03
10cols	1.94	0.59 (0.27)	0.52	0.28	0.14	0.09
bayer01	5.25	0.58 (0.31)	1.22	0.65	0.36	0.20
bayer04	1.60	0.23 (0.14)	0.57	0.34	0.26	0.14
icomp	3.53	0.09 (0.06)	1.01	0.62	0.38	0.20
lhr14c	1.58	0.73 (0.55)	0.82	0.46	0.28	0.22
lhr34c	4.99	2.25 (1.78)	2.51	1.43	0.99	0.56
lhr71c	14.45	5.05 (3.82)	5.88	3.25	1.99	1.52
graham1	13.12	11.01 (10.4)	6.27	5.31	4.74	4.58
pesa	1.97	0.22 (0.15)	0.37	0.20	0.15	0.10
poli_large	1.33	0.01 (0.01)	0.29	0.15	0.09	0.06
Zhao2	63.83	99.34 (96.4)	21.63	14.42	10.74	10.06

Table VII. Timing in Seconds for Solve (S)

Identifier	MA42	MA48	MP43 ($N = 8$)			
			$p = 1$	2	4	8
4cols	0.043	0.009	0.018	0.012	0.011	0.009
10cols	0.134	0.037	0.066	0.043	0.027	0.023
bayer01	0.300	0.071	0.161	0.124	0.071	0.054
bayer04	0.097	0.016	0.055	0.039	0.025	0.019
icomp	0.291	0.038	0.130	0.116	0.077	0.058
lhr14c	0.065	0.041	0.074	0.047	0.028	0.021
lhr34c	0.202	0.126	0.199	0.144	0.100	0.051
lhr71c	0.545	0.263	0.458	0.334	0.202	0.145
graham1	0.240	0.172	0.168	0.144	0.109	0.090
pesa	0.085	0.010	0.035	0.020	0.014	0.011
poli_large	0.042	0.003	0.022	0.014	0.012	0.011
Zhao2	1.005	0.619	0.670	0.501	0.351	0.256

these problems, the interface time is a significant proportion of the total time on a single processor and limits the speedup obtained by increasing the number of processors.

Looking at the AFS times, we see that MP43 outperforms MA42 on a single processor. It is also faster than MA48 for a significant number of our test problems. However, if the cost of preordering to bordered block form is included (see Table IV), MP43 becomes more expensive than the serial solvers. Since HSL_MC66 is a serial code, these findings suggest that, if only a single AFS is wanted, it may be important to develop a parallel version of the preordering routine. It also suggests that the most appropriate use for the parallel codes is likely to be for solving problems that are too large for the traditional serial solvers. MA48 has no out-of-core storage facilities and this limits the size (the order and density) of problem that it can successfully solve.

From Table VI, MA48 factorize is generally faster than MP43 factorize on a single processor but, provided the interface is small, MP43 achieves good speedups on increasing the number of processors and is generally able to outperform both the standard and fast MA48 factorizations using a modest number of processors.

Table VIII. 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

As already noted, the solve phase of MA48 is very efficient and, from Table VII, we see that for each test case, the MA48 solve time is less than for MA42. On one or two processors it is also generally less than for MP43 but, by increasing the number of processors to 4 or 8, the MP43 is faster than MP48 on many of our problems.

4.2 MP42

We now illustrate the performance of MP42 using the finite element problems listed in Table VIII. These problems were supplied by Christian Damhaug of Det Norske Veritas, Norway. In each case, the problem is ordered to bordered block diagonal form with $N = 4$ 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 a partitioning with a (nearly) equal number of subdomains but some of the subdomains have a significantly larger number of interface variables.

The timings presented in this section and in Section 4.3 are obtained on a 12 processor SGI Origin 2000 at the University of Manchester, again using the cpuset facility and optimization flags `-O3 -OPT:Olimite=0`. In Table IX timings are given for the analyze plus factorize plus solve phases (AFS) for a single right-hand side using MP42 run on 1 to 4 processors. Note that, by default, MP42 distributes the subdomains between the processors according to the predicted flop count (the number of floating-point operations) computed by the analyze phase. The aim is to achieve good load balance by assigning the subdomains to the processors so that the total predicted flop count for the partial LU factorizations to be performed by each processor is as equal as possible. Flop estimates are computed using two assumptions: first, that a variable may be eliminated as soon as it is fully summed (that is, numerical considerations do not delay elimination) and second, that the frontal matrix is treated as dense (in practice,

Table IX. AFS Timings in Seconds 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 Processor

		MP42							
Identifier	MA42	$p = 1$		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*

the code is able to take some advantage of zeros in the front during the factorization and this can result in the predicted flop count being greater than the actual number of flops needed). The subdomains are ordered in descending order of their predicted flop counts. Each of the first p subdomains in the ordered list is assigned to a different processor. If unassigned subdomains remain, the next unassigned subdomain in the list is assigned to the processor p_j with the smallest predicted flop count, the predicted flop count for p_j is then updated and the process continued until there are no unassigned subdomains.

For comparison, results are included in Table IX for the serial frontal solver MA42. For MA42, the element assembly order is generated using the HSL routine MC63 [Scott 1999b]. We see that, for each problem except trdheim, MP43 outperforms MA42 on a single processor. This is because, for these examples, the flop count is significantly reduced by partitioning the domain and using the multiple front approach. 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].

Good speedups for MP42 are achieved on 2 processors but the gains are smaller when the number of processors is increased further. In particular, we observe that, for the first three test cases, there is essentially no speedup when 4 processors are used rather than 3. We recall that we are using 4 subdomains so, if MP42 is run on 3 processors, two subdomains are assigned to one of the processors, and one to each of the other two; we might anticipate this would produce a load imbalance. However, if we look, for example, in greater detail at problem thread, we see from Table IX that, although subdomains 3 and 4 have fewer elements than subdomains 1 and 2, they 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 processors and factorizing both subdomains 1 and 2 using a single processor.

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 opt1, on a single processor, the interface factorization accounts for approximately 27 per cent of the total factorization time; this increases to 50 per cent on 4 processors and thus represents a significant bottleneck in the computation. If

Table X. Factorization Times in Seconds 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

Table XI. 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

the number of subdomains is increased, so will the order of the interface problem so that solving it 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. It also suggests that in the future we may want to look at solving the interface problem using more than one processor.

4.3 MP62

Finally, we present some results to illustrate 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 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 X and XI factorization and solve times are presented for a number of different mesh sizes, using up to $p = 8$ processors (n is the total number of variables). For this two dimensional model problem with equal subdomains, good speedups are obtained for factorize and solve as the number of processes increases. In particular, for the factorization phase for the largest problems, speedups close to 2 are achieved using 2 processors and exceed 3.5 using 4 processors. We deduce that load balancing is good and that the solution of the interface problem is not significantly affecting the overall performance.

5. 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 Origin 2000 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. Further results on a Cray T3E and a 2-processor Compaq DS20 have been presented in Scott [2001b, 2001a]; Stadtherr and Lin (private communication, 2001) have also reported using MP43 successfully 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. For unsymmetric problems we used the HSL implementation HSL_MC66 of the MONET algorithm to effect an *a priori* permutation to singly bordered block diagonal form. The ordering is so effective for highly unsymmetric problems such as those that arise in chemical process engineering that a single processor factorization that exploits this form can be faster and less memory demanding than using a Markowitz threshold algorithm on the whole matrix. However, the partitioning cost using the present serial code can be more expensive than the subsequent factorization and hence represents a significant overhead. Thus, if we are only going to do a single analyze/factorize/solve, it is important to develop parallel preordering routines.

Our numerical results demonstrate that the performance of our parallel codes is very sensitive to the size of the interface problem. The sequential solution of the interface problem can cause a bottleneck as the number of processors increases. This effectively restricts the use of the multiple front approach to a relatively small number of subproblems and processors. Nevertheless, our results demonstrate that significant improvements over serial direct solvers can be achieved. Furthermore, by distributing the matrix data and the factors across the processors and optionally using files for the data, the parallel approach has the potential to solve much larger problems than the serial solvers.

The parallel frontal solvers HSL_MP42, HSL_MP43, and HSL_MP62, together with the implementation HSL_MC66 of the MONET algorithm used in this article, and the sparse matrix 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) may find further details on the website www.cse.clrc.ac.uk/Activity/HSL.

ACKNOWLEDGMENTS

I am very grateful to Milan Mihajlovic for providing me with the results presented in Section 4.3 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 SGI Origin 2000 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 II, my colleague Iain Duff for comments on a draft of this article, and the anonymous referees for their careful reading and constructive criticism of the article.

REFERENCES

BENNER, R., MONTRY, G., AND WEIGAND, G. 1987. Concurrent multifrontal methods: shared memory, cache, and frontwidth issues. *Int. J. Supercomput. Applics. 1*, 26–44.

BOMHOF, C. AND VAN DER VORST, H. 2000. A parallel linear system solver for circuit simulation problems. *Numerical Linear Algebra with Applications 7*, 649–665.

DONGARRA, J., DUCROZ, J., DUFF, I., AND HAMMARLING, S. 1990. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft. 16*, 1, 1–17.

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

DUFF, I., ERISMAN, A., AND REID, J. 1986. *Direct Methods for Sparse Matrices*. Oxford University Press, England.

DUFF, I. AND REID, J. 1996. The design of MA48, a code for the direct solution of sparse unsymmetric linear systems of equations. *ACM Trans. Math. Soft. 22*, 187–226.

DUFF, I. AND SCOTT, J. 1994. The use of multiple fronts in Gaussian elimination. Tech. Rep. RAL-94-040, Rutherford Appleton Laboratory.

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

DUFF, I. AND SCOTT, J. 1998. MA62—a frontal code for sparse positive-definite symmetric systems from finite-element applications. In *Innovative Computational Methods for Structural Mechanics*, M. Papadrakakis and B. Topping, Eds. Saxe-Coburg Publications, Edinburgh, 1–25.

DUFF, I. AND SCOTT, J. 2002. A parallel direct solver for large sparse highly unsymmetric linear systems. Tech. Rep. RAL-TR-2002-033, Rutherford Appleton Laboratory, Didcot, Oxfordshire.

GOLUB, G., SAMEH, A., AND SARIN, V. 2001. A parallel balanced scheme for banded linear systems. *Numerical Linear Algebra with Applications 8*, 297–316.

HENDRICKSON, B. AND LELAND, R. 1995. The Chaco user's guide: Version 2.0. Tech. Rep. SAND94-2692, Sandia National Laboratories, Albuquerque, NM.

HOOD, P. 1976. Frontal solution program for unsymmetric matrices. *Int. J. Num. Meth. Eng. 10*, 379–400.

HU, Y., MAGUIRE, K., AND BLAKE, R. 2000. A multilevel unsymmetric matrix ordering for parallel process simulation. *Computers in Chemical Engineering 23*, 1631–1647.

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

MALLYA, J., ZITNEY, S., CHOUDHARY, S., AND STADTHERR, M. 1997. A parallel block frontal solver for large scale process simulation: reordering effects. *Computers in Chemical Engineering 21*, S439–S444.

MPI. 1994. A message-passing interface standard. *Int. J. Supercomp. Applic. 8*. Special edition on MPI.

REID, J. AND SCOTT, J. 1999. Ordering symmetric sparse matrices for small profile and wavefront. *Int. J. Num. Meth. Eng 45*, 1737–1755.

SCOTT, J. 1996. Element resequencing for use with a multiple front algorithm. *Int. J. Num. Meth. Eng. 39*, 3999–4020.

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

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

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

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

SCOTT, J. 2001b. A parallel solver for finite element applications. *Int. J. Num. Meth. Eng. 50*, 1131–1141.

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

SLOAN, S. 1989. A FORTRAN program for profile and wavefront reduction. *Int. J. Num. Meth. Eng. 28*, 2651–2679.

ZANG, W. AND LIU, E. 1991. A parallel frontal solver on the Alliant. *Computers and Structures 38*, 202–215.

ZONE, O. AND KEUNINGS, R. 1991. Direct solution of two-dimensional finite element equations on distributed memory parallel computers. In *High Performance Computing*, M. Durand and F. E. Dabaghi, Eds. Elsevier Science Publications.

Received April 2002; revised January 2003 and July 2003; accepted July 2003