

A parallel direct solver for large sparse highly unsymmetric linear systems

I.S. Duff and J.A. Scott

Abstract

The need to solve large sparse linear systems of equations efficiently lies at the heart of many applications in computational science and engineering. For very large systems when using direct factorization methods of solution, it can be beneficial and sometimes necessary to use multiple processors, because of increased memory availability as well as reduced factorization time. We report on the development of a new parallel code that is designed to solve linear systems with a highly unsymmetric sparsity structure using a modest number of processors (typically up to about 16). The problem is first subdivided into a number of loosely connected subproblems and a variant of sparse Gaussian elimination is then applied to each of the subproblems in parallel. An interface problem in the variables on the boundaries of the subproblems must also be factorized. We discuss how our software is designed to achieve the goals of portability, ease of use, efficiency, and flexibility, and illustrate its performance on an SGI Origin 2000, a Cray T3E, and a 2-processor Compaq DS20, using problems arising from real applications.

Keywords: large sparse linear systems, highly unsymmetric matrices, Gaussian elimination, parallel processing, Fortran 90, MPI.

This report is available by anonymous ftp to `ftp.numerical.rl.ac.uk` in directory `pub/reports`. This report is in the file `duscRAL2002033.ps.gz` or `duscRAL2002033.pdf`. The report is also available through the URL `www.numerical.rl.ac.uk/reports/reports.html`. This work was funded by the EPSRC Grant GR/R46441.

Computational Science and Engineering Department
Atlas Centre
Rutherford Appleton Laboratory
Oxon OX11 0QX

December 20, 2002.

Contents

1	Introduction	1
2	Test problems and computing environment	2
3	Parallel approach	3
3.1	Preordering to bordered form	4
4	MA48 algorithm, code, and modifications	6
4.1	MA48 and MA50	6
4.2	Modifications to MA50	7
5	Software design	8
5.1	Portability	8
5.2	User interface	8
5.3	Flexibility	10
5.3.1	Input of matrix data	10
5.3.2	The use of files	11
5.3.3	Load balancing	11
5.3.4	MA48 options	12
6	Numerical results	12
6.1	Timings on other platforms	16
7	Concluding comments	19
8	Availability	20

1 Introduction

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

$$Ax = b.$$

Solving these systems usually dominates the computational cost of the simulation. As time-dependent three-dimensional simulations are now commonplace and workstations and computers with several CPUs are now more widely available, there is increasing interest in developing algorithms and software that can be used to solve such problems efficiently on parallel computers. This paper reports on the design and development of a new parallel unsymmetric solver, **HSL_MP48**, that has at its heart the well known software package **MA48** from the mathematical software library **HSL** (www.cse.clrc.ac.uk/nag/hsl).

MA48 is a serial code that was developed in the early 1990s by Duff and Reid (1993). It 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. During the last decade **MA48** and, before that, its predecessor **MA28** (Duff, 1977), has been very widely used; it has been incorporated into a number of commercial packages and has become a benchmark against which other sparse direct solvers are frequently compared. **MA48** has proved particularly successful when used for solving problems where the system matrix A is very sparse and highly unsymmetric. One application area in which such systems arise is chemical process engineering. It has also been found to be extremely efficient when there is a need to solve repeatedly for different right-hand sides.

MA48 stores both the matrix A and its factors in main memory. The size of problem that can be solved using **MA48** is therefore limited by the amount of computer memory available. To solve larger problems, as well as to solve problems more quickly, our aim is to develop a parallel version of **MA48**. Because **MA48** is a well-established code that represents substantial programming effort and expertise, we were anxious to exploit the existing code as far as possible in developing a parallel version. However, there is limited scope for parallelism within **MA48**, beyond that which is available through block triangularization and through its use of high-level BLAS kernels (Dongarra, DuCroz, Duff and Hammarling, 1990). The idea behind our parallel approach is to partition the matrix into a (small) number of loosely connected submatrices and to apply a modified version of **MA48** to each of the submatrices in parallel. This approach was disussed by Duff and Scott (1994) and was used successfully by Scott (2001*b*, 2002) to develop parallel frontal solvers. The results reported by Scott demonstrate that, for a range of practical problems, the parallel frontal approach is significantly faster than using the **HSL** serial frontal code **MA42** (Duff and Scott, 1993). In particular, for problems arising from chemical process engineering, speedups in the range of 3 to 8 over the serial code **MA42** are typically achieved on 8 processors of an Origin 2000 (Scott, 2001*a*).

This paper is organised as follows. We describe, in Section 2, the test problems and computing environment that we use for our numerical experiments. In Section 3, we discuss our parallel approach. In Section 4, we briefly review the algorithm implemented by **MA48** and outline the modifications needed for our parallel implementation. Software

considerations in the design of our new code are discussed in Section 5. Numerical results for our test problems are presented in Section 6 and finally, in Section 7, some concluding remarks are made. The new code is called HSL_MP48 and is available for use under licence from HSL (see Section 8 for details).

2 Test problems and computing environment

In this section, we introduce the test problems that we will use throughout this paper to illustrate the performance of our parallel approach. The test matrices are all highly unsymmetric and are listed in order of increasing size in Table 2.1. We have selected problems that are of order at least 10,000 and are highly unsymmetric. A † indicates that the problem is included in the University of Florida Sparse Matrix Collection (Davis, 1997). The remaining problems were supplied by Mark Stadtherr of the University of Notre Dame and Tony Garrett of AspenTech, UK. The *symmetry index* $s(A)$ of a matrix A is defined to be 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.

Identifier	n	nz	$s(A)$ index
ethylene-2	10353	78004	0.3020
ethylene-1	10673	80904	0.2973
Matrix10876	10876	77494	0.0010
4cols	11770	43668	0.0159
1hr14c†	14270	307858	0.0066
bayer04†	20545	159082	0.0016
10cols	29496	109588	0.0167
Matrix32406	32406	1035989	0.0014
1hr34c†	35152	764014	0.0015
Matrix35640	35640	146880	0.0001
bayer01†	57735	277774	0.0002
1hr71c†	70304	1528092	0.0016
icomp	75724	338711	0.0010

Table 2.1: Test problems. n , nz denote the order of the system and the number of entries, respectively. $s(A)$ denotes the symmetry index. Problems marked † are available from the University of Florida Sparse Matrix Collection.

Unless stated otherwise, the numerical results presented in this paper were computed on a 12 processor SGI Origin 2000 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 timings are for the elapsed times in seconds, measured using `MPI_WTIME` on the host process.

3 Parallel approach

Our parallel approach is based upon partitioning the matrix A into a singly bordered block diagonal (SBBD) form

$$\begin{pmatrix} A_{11} & & & C_1 \\ & A_{22} & & C_2 \\ & & \dots & \vdots \\ & & & A_{NN} & C_N \end{pmatrix}, \quad (3.1)$$

where the rectangular blocks on the diagonal A_{ll} are $m_l \times n_l$ matrices with $m_l \geq n_l$, and the border blocks C_l are $m_l \times k$ with $k \ll n_l$. We note that in general many columns of the C_l are zero so that the matrix can be stored as an $m_l \times k_l$ matrix with $k_l \leq k$. We discuss further how to obtain this partition in Section 3.1. The principal way that we exploit parallelism is to perform a partial LU decomposition of the matrices (A_{ll}, C_l) simultaneously. We note that since the matrix A is square, we have $\sum_{k=1}^N (m_l - n_l) = k$, and if the border has only a few columns, the submatrices A_{ll} will be nearly square. The overall solution scheme is given by the following steps:

1. Factorization
 - (a) partial LU decomposition of each of the submatrices (A_{ll}, C_l)
 - (b) formation of an *interface problem* by summing the Schur complement matrices that remain after all possible eliminations have been performed on the submatrices
 - (c) factorization of the interface matrix
2. Forward elimination
 - (a) forward elimination on the submatrices
 - (b) sending of data to the processor responsible for the interface matrix
 - (c) forward elimination on the interface matrix
3. Back substitution
 - (a) back substitution on the interface matrix
 - (b) sending of the interface solution to the other processors
 - (c) back substitution on the submatrices.

Step 2(c) and step 3(a) can be combined as the solve step on the interface problem. The operations on submatrices in steps 1(a), 2(a), and 3(c) can be performed in parallel.

Using this approach has significant advantages over attempting to design a general parallel sparse direct solver from scratch. Firstly, each processor can be preassigned all the matrix data required for the computations that it will perform before the factorization starts. Secondly, the factorizations on the submatrices can be performed in parallel. Communications in the costly “Factorization” step (step 1) are required only to send the Schur complement matrices to the processor responsible for the interface problem.

Interprocessor communication is thus both limited and structured. Finally, the interface matrix, which is much smaller than the original matrix, may be factorized using any existing sparse direct solver.

The partial decomposition of the submatrices at step 1(a) cannot be performed using a standard direct solver without some modifications. This is because a standard solver is designed to factorize the whole of the matrix A . When factorizing a submatrix (A_{ll}, C_l) , the variables corresponding to the columns of A_{ll} are candidates for elimination, but those corresponding to the k_l nonzero columns of C_l may not be eliminated since these columns have entries in at least one other border block C_j ($j \neq l$). Modifications are thus needed to enable a distinction to be made between the columns of A_{ll} that may be eliminated and those of C_l that must be passed to the interface problem. We use a modified version of **MA48** (or, more precisely, a modified version of the code **MA50** that lies at the heart of **MA48**) to perform the partial decompositions of the submatrices. The modifications we have made are discussed in Section 4.2. We use the unmodified **MA48** code to solve the interface problem.

This approach to developing a parallel version of **MA48** is suitable for a relatively small number, N , of submatrices and for use with no more than N processors. In our experiments, we have used $N \leq 16$. Restricting N is necessary because the size of the interface problem generally grows rapidly as the number of submatrices increases (see Table 3.1 in Section 3.1). The cost of solving the interface problem thus increases and becomes a more significant part of the total computational cost, potentially causing a serious bottleneck.

3.1 Preordering to bordered form

Our parallel approach currently solves the interface problem using a single processor. Thus to achieve good speedups for the overall solution time, it is crucial that the SBBB form (3.1) has a narrow border. The parallel code **HSL_MP48** requires the user to perform the preordering before the computation starts (or at least requires the user to specify to which submatrix each row of A belongs). The design decision not to incorporate software for preordering A within the **HSL_MP48** package was made early in the code's development. We made this choice firstly because, in some applications, the matrix naturally occurs in the required form and, secondly, the best approach to obtaining a good ordering is problem dependent and the development of effective and efficient algorithms for partitioning is still a subject of active research.

For ordering chemical process engineering problems, Hu, Maguire and Blake (2000) have developed the **MONET** algorithm. **MONET** implements a multilevel recursive bisection algorithm, with the aim of minimizing the border size of the final SBBB form for a user-chosen number of submatrices, while maintaining good *row balance* (each submatrix A_{ii} , $i = 1, N$ has a similar number of rows). Starting with the original matrix A , the multilevel approach generates a sequence of matrices of smaller and smaller size. The matrix on the coarsest level (which is the one of smallest order) is bisected into an SBBB form with two submatrices using several runs of the Kernighan-Lin algorithm (Kernighan and Lin, 1970), starting with a random initial ordering. The best bisection obtained from this is prolonged to the larger matrix in the sequence at the next level and the bisection

and associated ordering is further refined using the Kernighan-Lin algorithm. This is repeated on larger matrices from successive levels in the sequence until a bisection of the original matrix is obtained. This whole multilevel process is then recursively applied on the partitions of the original matrix until the desired number of submatrices, N , is achieved.

Identifier	n	$N = 2$	$N = 4$	$N = 8$	$N = 16$
ethylene-2	10353	21 (0.20%)	73 (0.71%)	147 (1.42%)	271 (6.79%)
ethylene-1	10673	42 (0.39%)	118 (1.11%)	181 (1.70%)	276 (6.29%)
Matrix10876	10876	128 (1.18%)	644 (5.92%)	1123 (10.3%)	2088 (19.2%)
4cols	11770	30 (0.25%)	60 (0.51%)	184 (1.56%)	433 (3.68%)
lhr14c	14270	68 (0.68%)	266 (1.58%)	635 (4.45%)	1262 (8.84%)
bayer04	20545	185 (0.90%)	453 (2.29%)	599 (2.92%)	886 (7.71%)
10cols	29496	30 (0.10%)	137 (0.46%)	288 (0.98%)	600 (2.03%)
Matrix32406	32406	574 (1.77%)	1021 (3.15%)	1387 (4.28%)	2289 (7.06%)
lhr34c	35152	94 (0.27%)	384 (1.09%)	885 (2.52%)	1766 (5.02%)
Matrix35640	35640	312 (0.88%)	624 (1.75%)	1464 (4.11%)	2520 (7.07%)
bayer01	57735	71 (0.12%)	233 (0.46%)	354 (0.61%)	613 (1.06%)
lhr71c	70304	64 (0.09%)	252 (0.36%)	832 (1.18%)	1834 (2.61%)
icomp	75724	186 (0.25%)	250 (0.33%)	449 (0.59%)	719 (0.95%)

Table 3.1: Number of interface variables in the SBBB form generated by HSL_MC66. n is the order of the problem and N is the number of submatrices. The figures in parentheses are the percentage of the columns in the border.

The performance of MONET as implemented in the HSL code HSL_MC66 is illustrated in Table 3.1. In this table, we show the number of interface variables for 2, 4, 8, and 16 submatrices. We see that MONET is very successful at producing an SBBB form for highly unsymmetric matrices that has a narrow border (for all matrices except `Matrix10876`, the border represents less than 5% of the total number of columns for up to 8 submatrices).

We remark that, in all our tests, a good row balance between the submatrices was achieved, although the imbalance generally increases with N . We define the row imbalance to be the difference between the maximum submatrix row dimension and the average submatrix row dimension, divided by the average submatrix row dimension, expressed as a percentage. That is, if m_i is the row dimension of submatrix A_{ii} , then

$$\text{row imbalance} = \frac{\max\{m_i\} - n/N}{n/N} \times 100.$$

For our test examples, with $N = 4$ the row imbalance ranges from less than 2% to 9% while for $N = 8$ it ranges from 2.3% to 10.5%. Note that there is a trade off between the row imbalance and the border size of the final ordering; allowing a larger row imbalance tends to give a smaller border. In HSL_MC66, the desired row imbalance at each stage of the multilevel algorithm is a parameter under user control.

4 MA48 algorithm, code, and modifications

4.1 MA48 and MA50

The HSL code **MA48** is a driver for the HSL code **MA50**. Although it is **MA50** that we will modify, we first discuss **MA48** since it is used unchanged for the factorization and solution of the interface problem. In common with many sparse direct solvers, **MA48** divides the computation into a number of distinct phases. The phases of **MA48** are:

- **Analyse** permutes A to block upper triangular form and chooses pivots using a criterion that combines good sparsity preservation with a numerical threshold test.
- **Factorize** factorizes a matrix with the same sparsity pattern (but possibly different numerical values) using a given column sequence and with row interchanges guided by a recommended row sequence. Threshold pivoting ensures stability.
- **Fast factorize** factorizes another matrix with exactly the same sparsity pattern using exactly the same pivot sequence.
- **Solve** uses the LU factors generated by factorize to perform forward elimination followed by back substitution to solve the linear system.

The **MA50** package that lies at the heart of **MA48** factorizes a single block on the diagonal of the block upper triangular form. It will also factorize rectangular systems. It requires the entries to have been sorted by columns, with no duplicates. Provided a user accepts these restrictions, **MA50** may be called directly. In our case, **HSL_MP48** will be the “user” of **MA50**, and a modified version of **MA50** will be used in the factorization and in the forward elimination and back substitution for the submatrices (A_{II}, C_I) of (3.1).

The **MA50** package has four user-callable subroutines:

- **Initialize** provides default values for the parameters that control the execution of the package. The user may reset one or more of these before passing the parameters to the other subroutines.
- **Analyse** uses a sparse variant of Gaussian elimination to compute a pivot ordering for the decomposition of A into its LU factors. When the fill-in to the reduced matrix reaches a user-defined density, the elimination operations are terminated and an arbitrary permutation of the remaining matrix is used to complete the pivot ordering, putting the dense block at the end of the ordering. The analyse subroutine thus does not complete the factorization, nor does it keep the factors, which makes it less likely to fail through lack of storage.
- **Factorize** accepts a matrix A together with recommended row and column permutations and the size of the final dense block. It performs the factorization $PAQ = LU$, using the Gilbert and Peierls algorithm (Gilbert and Peierls 1988) with threshold pivoting on the sparse part and slightly modified LAPACK routines on the dense part. An option exists for the subsequent ‘fast’ factorization of matrices with the same sparsity pattern under the assumption that exactly the same permutations are suitable.

- **Solve** performs simple forward-elimination and back-substitution operations using the factors from the factorize phase. For the full matrix processing, options are included for the use of Level 1 or Level 2 BLAS.

When MA50 is used to factorize a singular or a rectangular matrix, the components of the solution vector corresponding to a zero pivot are set to zero.

A key feature of MA50 that enables both it and MA48 to achieve high performance compared with the earlier MA28 code is the switch from sparse to full-matrix processing once the reduced matrix is sufficiently dense. This allows Level 3 BLAS (Dongarra et al., 1990) to be used. The point at which the move to full-matrix processing is made is controlled by the user; by default the switch is made when the ratio of the number of entries in the reduced matrix to the number it would have as a full matrix is greater than 0.5.

For stability, each pivot, a_{kj} , is required to satisfy the column threshold test

$$|a_{kj}| \geq u \max_i |a_{ij}|$$

within the reduced matrix, where u is the threshold parameter that may be set by the user (in MA48 and MA50 it has default value 0.1). Within the reduced matrix at each stage a pivot is chosen from entries that satisfy the stability test with least Markowitz cost (Markowitz, 1957) (product of the number of other entries in the row of the reduced matrix and the number of other entries in the column). The strategy of Zlatev (1980) for only searching a small number of columns of the reduced matrix (those with least number of entries) for a pivot is offered as an option (the default is 3).

For a full discussion of the design of both MA48 and MA50, the reader is referred to Duff and Reid (1993, 1996).

4.2 Modifications to MA50

In this section, we briefly describe some of the changes that we had to make to MA50 in order to use it to factorize the submatrices from the SBBB form. Although the changes were non-trivial, the majority of the code was not altered allowing us to benefit from our earlier investment in this software. The modified routines are presently internal to the HSL_MP48 package but may later form the basis for a user-callable package.

The task for our modified factorization routine is to perform an LU factorization on the rectangular matrix $(A_{ll} \ C_l)$ and then to use these factors in forward elimination and back substitution. This corresponds to steps 1(a), 2(a), and 3(c) of our algorithm outlined in Section 3.

Although MA50 will factorize rectangular matrices, we have the added complication that only n_l pivots can be chosen from within the A_{ll} block. The original code did allow for a set of columns to be kept to the end of the pivoting sequence and, although we use some of this mechanism, modifications were needed to terminate the factorization after the selection of n_l pivots and to provide the Schur complement matrix from this partial factorization for assembly for the interface problem. An added problem is that the rectangular matrix can be singular even when the whole matrix is not so that, although we

flag the singular columns, we must allow them to be pivoted on later after the assemblies for the interface problem have been performed.

In our new code, it is necessary to perform the forward elimination and the back substitution (steps 2(a) and 3(c), respectively) on the submatrices quite separately. The solve routines in the MA50 code combined these two steps and so we had to disentangle these in the new code.

5 Software design

Our aim was to design and develop a general parallel sparse direct solver that is portable and straightforward to use, while being both flexible and efficient. In this section, we address how each of these goals was achieved. Key features include: a number of different options for supplying the matrix data to take advantage of the computer architecture being used, optionally holding the LU factors from the partial decomposition in sequential files to enable larger problems to be solved, and a fast factorization for factorizing a matrix where only the numerical values have changed.

5.1 Portability

To ensure the code can be used on a wide range of modern computers, HSL_MP48 is written in standard Fortran 90 and uses MPI for message passing. 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 and more readable code, which in turn assists with code maintenance. The choice of using MPI for message passing was made because the MPI Standard (MPI, 1994) is internationally recognised and today is widely available and accepted by users of parallel computers. It is also available on most shared memory multiprocessors. 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 the use BLAS kernels within the modified MA50, allows us to achieve our goal of producing portable software.

One of the options offered by HSL_MP48 is to write the LU factors from the partial decomposition to sequential files at the end of the factorization of a submatrix. If the number N of submatrices exceeds the number p of processors, this can allow larger problems to be solved than would otherwise be possible. It also further increases the portability of the code by enabling it to be run on machines where each processor has only a limited amount of main memory.

5.2 User interface

A key design aim for HSL_MP48 was a user interface that is straightforward and, at the same time, offers flexibility through a variety of options. Our intention is that it should be possible for the code to be used by those who have only a basic knowledge of MPI together with limited experience of Fortran 90 programming.

Access to the module `HSL_MP48` requires a `USE` statement and an `INCLUDE` statement is needed for MPI. In addition, the user must declare a structure `data` of Fortran 90 derived datatype `MP48_DATA` defined by the module. `HSL_MP48` has a single user-callable subroutine `MP48A` (`MP48AD` in the double precision version) with `data` of type `MP48_DATA` as the only parameter. This derived datatype has many components, only some of which are of interest to the user and must be set by him or her. These include the components that define the sparsity pattern of the matrix and the SBBB form. Other components are used by the package to provide the user with information on the computation (including flop counts and the storage used or required). Full details of the derived datatype are provided in the user documentation.

The `HSL_MP48` code is divided into six separate phases (all operations on the submatrices are performed in parallel):

- **Initialize** provides default values for the components of `data` that control the execution of the package. The user may reset one or more of these before passing the parameters to the subsequent phases.
- **Preliminary analyse** checks the user-supplied data, computes lists of border columns for each submatrix and optionally assigns each submatrix to a processor.
- **Analyse** uses the modified `MA50` analyse to compute, for each submatrix (A_{ii} , C_i), permutations P_i and Q_i suitable for the partial LU factorization.
- **Factorize** uses the modified `MA50` factorize to generate the factors for the submatrices using the information from the analyse phase, altering the matrices P_i if necessary for numerical stability. This allows the stable factorization of submatrices with the same sparsity pattern but different numerical values without recalling the analyse phase. The interface matrix is then assembled and the analyse followed by the factorize phase of `MA48` compute the LU factors for the interface matrix. An option exists for subsequent calls for matrices with the same sparsity pattern to be made faster on the assumption that the same pivot sequence is still suitable.
- **Solve** uses the factors to solve a system of the form $Ax = b$. Repeated calls may be made to solve for more than one right-hand side.
- **Finalize** terminates the computation by deallocating all arrays allocated by `HSL_MP48` and, optionally, deletes the files holding the matrix factors (see Section 5.3.2).

Prior to the call to the initialize phase, the user must choose the number of processes, p , that are to be used and must initialize MPI by calling `MPI_INIT` on each process. The user must also define an MPI communicator for the package. Among other things, the communicator defines the set of processes to be used by `HSL_MP48`. Each such process is labelled by a process *rank*, viz. $0, 1, 2, \dots, p-1$. The *host* is the process with rank zero. The host performs the initial checking of the data, distributes data to the remaining processes, collects computed data from the processes, factorizes and solves the interface problem, and generally oversees the computation. The host also participates by default with the other processes in generating the partial LU decompositions of the submatrices.

Having defined the communicator, each phase must be called in order by each processor in the communicator; a job parameter (`data%JOB`) determines which phase of the package is to be performed. The user may either do this by calling each phase from his or her calling program or, for convenience, having called the initialize phase there exists an option of making a single call that initiates calls to each remaining phase in turn. Before calling each phase, the user must have completed all other tasks that he or she was performing with the defined communicator (and with any other communicator that overlaps the `HSL_MP48` communicator). This can be ensured using an MPI barrier. After the finalize phase and once the user has completed any other calls to MPI routines he or she wishes to make, the user should call `MPI_FINALIZE` to terminate the use of MPI.

The following pseudocode illustrates how `HSL_MP48` can be used.

```

USE HSL_MP48_DOUBLE
INCLUDE 'mpif.h'
...
INTEGER ERCODE
TYPE (MP48_DATA) :: data
...
CALL MPI_INIT(ERCODE)
...
! Set components of data, including data%JOB
...
CALL MP48AD (data)
...
CALL MPI_FINALIZE(ERCODE)

```

5.3 Flexibility

The call to the initialize phase of `HSL_MP48` assigns default parameters to the control parameters. These parameters control the action and offer the user a number of options. It is these options that make the package flexible. We now discuss some of the main options; full details of all the control parameters and options are given in the user documentation for `HSL_MP48`.

5.3.1 Input of matrix data

The amount of main memory required by `HSL_MP48` is influenced by how the user chooses to supply the matrix data. A number of options are provided. By default, the submatrices (A_{ll}, C_l) are held in unformatted sequential files and the data required by a particular processor must be readable by the processor executing the process. This has the advantage that each processor only needs to have sufficient memory to read and generate the LU factors of one submatrix at a time; it also minimises the movement of data between processors. If the user wishes to avoid the overhead of reading data from files, he or she may instead use input arrays on each processor to supply the data for the submatrices assigned to it. Options also exist for the user to supply all the submatrix 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 submatrix data from the host to the other processors. Since the host is also involved in the submatrix factorizations, this distribution of data is carried out by HSL_MP48 before the factorization commences, and thus each processor must have sufficient memory to store the data for all the submatrices assigned to it.

5.3.2 The use of files

As well as using files to supply the submatrix data, the user may choose to hold the LU factors for the partial decompositions in sequential files. Using files reduces storage requirements when one processor factorizes more than one submatrix and thus allows the solution of larger problems than could otherwise be handled. However, depending upon the computing environment, the extra I/O involved may increase the overall execution time. Thus we suggest that files are only used if either the problem is too large to be accommodated or if the user wishes to retain the factors to solve later for further right-hand sides.

5.3.3 Load balancing

By default, the submatrices are assigned to processors by the code but an option is available for the user to decide which processor is to factorize which submatrix. For load balancing, we rely on the SBBB form providing a partitioning so that the factorization of each submatrix involves a similar amount of work. If this is so and we are working in a homogeneous computing environment, we would normally advise that N be a multiple of p . Unfortunately, there is no obvious *a priori* way to determine from the order, density, and number of interface variables the time to compute the factorization of an unsymmetric matrix. Thus the current version of the code simply assigns the submatrices to the processors in an arbitrary order so that each processor has either N/p or $N/p + 1$ submatrices to factorize. If the user wants to factorize a matrix with the same sparsity pattern as an earlier matrix, he or she should check that the assignment of the submatrices for the earlier matrix gave good load balance. If not, the submatrix flop counts (or timings or storage) from the earlier factorization may be used to more appropriately assign the submatrices to processors for the subsequent factorization. The user may also wish to assign submatrices to processors if it is known that one or more of the processors is faster or has more memory. Moreover, if a sequence of problems is to be solved, information from a first assignment might be used to better assign the submatrices to processors for further problems in the sequence.

We have already seen that, with N small, the MONET algorithm is successful in generating an SBBB form with a good row balance. In Table 5.1, timings are given for the analyse and factorize phases of HSL_MP48 for each submatrix of the SBBB form obtained using MONET ($N = 4$). The load balancing is good if, for a given problem, T_{diff} is small, where T_{diff} is defined to be the difference between the slowest and fastest submatrix times divided by the slowest submatrix time. We see that, for a number of our test problems, we have not achieved a good load balance; this will limit the speedup that we obtain when running on more than one processor. Since the interface problem is

Identifier	Submatrix				
	1	2	3	4	T_{diff}
ethylene-2	0.30/0.08	0.15/0.04	0.21/0.06	0.30/0.07	0.50/0.50
ethylene-1	0.39/0.08	0.28/0.08	0.21/0.05	0.21/0.05	0.46/0.37
Matrix10876	0.60/0.13	0.46/0.13	0.32/0.09	0.45/0.12	0.47/0.30
4cols	0.11/0.08	0.12/0.06	0.12/0.09	0.12/0.09	0.09/0.33
lhr14c	2.03/0.53	1.63/0.47	1.66/0.53	1.51/0.41	0.26/0.23
bayer04	0.42/0.16	0.40/0.14	0.34/0.13	0.28/0.11	0.33/0.31
10cols	0.68/0.20	0.72/0.21	0.76/0.22	0.43/0.14	0.46/0.36
Matrix32406	9.82/2.78	10.1/2.78	3.43/1.39	1.17/0.56	0.88/0.80
lhr34c	6.33/1.67	6.74/1.98	6.15/1.66	10.7/2.62	0.42/0.36
Matrix35640	25.1/8.60	19.1/8.03	22.5/6.97	26.2/11.6	0.27/0.40
bayer01	0.45/0.19	0.43/0.18	0.86/0.32	0.70/0.26	0.50/0.44
lhr71c	17.4/4.31	15.4/4.06	17.6/4.94	21.0/5.78	0.22/0.30
icomp	0.19/0.13	0.20/0.12	0.18/0.13	0.20/0.13	0.10/0.08

Table 5.1: Analyse/Factorize times for each submatrix ($N = 4$) in seconds on an Origin 2000.

always solved on the host and can have a significant memory requirement, the user can choose to avoid assigning any of the submatrix calculations to the host.

5.3.4 MA48 options

The interface and initialization phase of HSL_MP48 also allow the user to exploit some of the options available in the MA48 package. In particular, the switch to full-matrix processing is a parameter under the user's control, as is the stability threshold and the strategy for maintaining sparsity in the factors (see Section 4). The default value for the threshold parameter used by HSL_MP48 is 0.01. The user can also use the dropping strategy within MA48 and the modified MA50 which will allow a sparser inaccurate factorization that may, for example, be useful as a preconditioner for iterative methods.

6 Numerical results

We now present numerical results for our test problems and compare the performance of our new code HSL_MP48 with the serial code MA48. Unless stated otherwise, we do not include the time taken to preorder to the SBBB form (3.1) used by HSL_MP48, although we give times separately for this preordering in Table 6.7. For both solvers, the elapsed time (in seconds) is presented for four 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. Factorize (F2). This is the time taken to factorize a matrix having the same sparsity pattern as one that has already been factorized but different numerical values, incorporating numerical pivoting for stability.

3. Fast Factorize (FF). As F2, but no numerical pivoting is performed.
4. 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 all our tests, the threshold parameter was set to 0.1; otherwise default settings were used for both codes. We note that, for HSL_MP48 the first factorization of a matrix with a given sparsity pattern involves calling both the analyse and factorize phases of MA48 for the interface problem. For subsequent factorizations of matrices with the same sparsity pattern, the analyse phase of MA48 does not have to be repeated. Thus the F2 and FF times do not include the interface problem analyse time. While F2 incorporates numerical pivoting for stability, FF uses the pivot sequence from the first factorization. Thus FF is faster than F2 but may be numerically unstable if the matrix entries are markedly different from the earlier factorization.

We first present timings on the Origin 2000 for HSL_MP48 run on 1, 2, 4 and 8 processors, and compare it with MA48 run on a single processor. For HSL_MP48, the number of blocks in the singly bordered block diagonal form is 8. The times for AFS are given in Table 6.1. In Tables 6.2, Tables 6.3, and 6.4, timings are presented for the factorize (F2), fast factorize (FF), and solve (S) phases, respectively. Flop counts (the number of floating-point operations) are given in Table 6.5. We did not solve problem Matrix35640 using MA48 since the analyse phase alone took in excess of 2100 seconds.

Identifier	MA48	HSL_MP48 ($N = 8$)			
		No. processors			
		1	2	4	8
ethylene-2	0.77	1.15 (0.01)	0.61 <i>1.88</i>	0.43 <i>2.67</i>	0.31 <i>3.70</i>
ethylene-1	0.76	1.14 (0.01)	0.71 <i>1.60</i>	0.40 <i>2.85</i>	0.28 <i>4.07</i>
Matrix10876	3.48	5.39 (3.38)	4.66 <i>1.15</i>	4.28 <i>1.25</i>	4.11 <i>1.31</i>
4cols	2.47	0.67 (0.04)	0.41 <i>1.63</i>	0.29 <i>2.31</i>	0.23 <i>2.91</i>
1hr14c	7.23	8.87 (0.21)	4.88 <i>1.82</i>	2.87 <i>3.09</i>	1.74 <i>5.09</i>
bayer04	2.91	2.37 (0.39)	1.54 <i>1.53</i>	1.06 <i>2.23</i>	0.81 <i>2.96</i>
10cols	16.4	2.75 (0.15)	1.60 <i>1.72</i>	0.93 <i>2.96</i>	0.65 <i>4.23</i>
Matrix32406	40.8	28.8 (3.66)	20.2 <i>1.42</i>	12.6 <i>2.28</i>	10.3 <i>2.80</i>
1hr34c	24.2	30.1 (0.70)	16.2 <i>1.85</i>	9.80 <i>3.07</i>	5.89 <i>5.11</i>
Matrix35640	NS	50.3 (12.7)	34.8 <i>1.44</i>	25.6 <i>1.96</i>	20.7 <i>2.43</i>
bayer01	6.37	4.23 (0.06)	2.39 <i>1.77</i>	1.48 <i>2.86</i>	0.97 <i>4.36</i>
1hr71c	50.6	71.2 (0.68)	39.8 <i>1.79</i>	22.3 <i>3.19</i>	12.4 <i>5.74</i>
icomp	0.84	1.59 (0.01)	0.97 <i>1.64</i>	0.65 <i>2.45</i>	0.50 <i>3.18</i>

Table 6.1: Timings for Analyse+Factorize+Solve in seconds on an Origin 2000. NS denotes not solved. Numbers in parentheses are times for analysing and factorizing the interface problem. The numbers in italics are the speedups for HSL_MP48 compared with using a single processor.

We first consider the AFS timings presented in Table 6.1. For the majority of our test cases, using only 2 processors, HSL_MP48 outperforms MA48. As the number of processors is increased, HSL_MP48 achieves good speedups for most of our problems, with speedups on 8 processors in excess of 5 being achieved by the 1hr problems.

In column 3 of Table 6.1, we include in parentheses the time required for analysing and factorizing the interface problem. Similarly, in Table 6.5 we include the flop count for the interface problem. We observe that, in most cases, the interface time and flop count are small compared with the total AFS time and flop count. However, for a few problems, in particular **Matrix10876**, the interface represents a bottleneck and limits the speedup we can obtain when increasing the number of processors. From Table 3.1, we see that for **Matrix10876** with 8 submatrices, more than 10 per cent of the columns are in the border. Although the interface problem is smaller than the original problem, it is denser so that factorizing it using **MA48** requires more flops than **MA48** applied to the original matrix (see Table 6.5). We conclude that as a rule of thumb for **HSL_MP48** to work well, it is essential that the border is as narrow as possible; our results suggest that less than about 5 per cent of the columns should be in the border to avoid the solution of the interface problem dominating the total cost. We also note that this by itself does not guarantee good performance as we see in the case of **bayer04** where the interface problem is relatively expensive although the border is small. We have performed some additional experiments on problem **Matrix10876** using the **SBB** form with 4 submatrices. In this case, the AFS times for **HSL_MP48** on 2 and 4 processors are reduced to 2.43 and 1.98 seconds, respectively (the interface time is reduced to 1.15 seconds).

Identifier	MA48	HSL_MP48 ($N = 8$)							
		No. processors							
		1	2	4	8	1	2	4	8
ethylene-2	0.17	0.25 (0.00)	0.13	<i>1.92</i>	0.09	<i>2.78</i>	0.06	<i>4.17</i>	
ethylene-1	0.17	0.25 (0.00)	0.15	<i>1.67</i>	0.09	<i>2.78</i>	0.06	<i>4.17</i>	
Matrix10876	0.64	1.70 (1.29)	1.55	<i>1.10</i>	1.46	<i>1.16</i>	1.40	<i>1.21</i>	
4cols	0.56	0.20 (0.01)	0.11	<i>1.82</i>	0.08	<i>2.50</i>	0.06	<i>3.33</i>	
lhr14c	1.50	2.07 (0.11)	1.11	<i>1.86</i>	0.72	<i>2.87</i>	0.43	<i>4.81</i>	
bayer04	0.51	0.67 (0.17)	0.45	<i>1.49</i>	0.32	<i>2.09</i>	0.26	<i>2.58</i>	
10cols	2.65	0.64 (0.04)	0.36	<i>1.78</i>	0.20	<i>3.20</i>	0.13	<i>4.92</i>	
Matrix32406	7.58	8.10 (1.36)	5.52	<i>1.47</i>	3.71	<i>2.18</i>	3.04	<i>2.66</i>	
lhr34c	4.88	6.69 (0.37)	3.62	<i>1.84</i>	2.29	<i>2.92</i>	1.46	<i>4.58</i>	
Matrix35640	NS	17.2 (4.34)	11.2	<i>1.54</i>	7.82	<i>2.20</i>	6.34	<i>2.71</i>	
bayer01	1.22	1.05 (0.02)	0.59	<i>1.78</i>	0.36	<i>2.92</i>	0.22	<i>4.77</i>	
lhr71c	9.99	15.1 (0.41)	8.74	<i>1.72</i>	4.82	<i>3.13</i>	2.84	<i>5.32</i>	
icom	0.18	0.53 (0.01)	0.30	<i>1.77</i>	0.18	<i>2.94</i>	0.12	<i>4.45</i>	

Table 6.2: Timings for Factorize in seconds on an Origin 2000. NS denotes not solved. Numbers in parentheses are times for factorizing the interface problem. The numbers in italics are the speedups for **HSL_MP48** compared with using a single processor.

The factorize F2, fast factorize FF, and solve S times are reported in Tables 6.2 to 6.4. Good speedups are again achieved for the factorizations, particularly for the F2 case, but those for the solve phase are less encouraging. In particular, for many of the problems, very little time is saved by increasing the number of processors from 4 to 8 for the solve phase. Nevertheless, with only 2 processors, **HSL_MP48** generally outperforms **MA48** on both the factorize and solve phases.

Identifier	MA48	HSL_MP48 ($N = 8$)							
		No. processors							
		1	2	4	8				
ethylene-2	0.09	0.14 (0.00)	0.07	<i>2.00</i>	0.05	<i>2.80</i>	0.04	<i>3.50</i>	
ethylene-1	0.08	0.14 (0.00)	0.09	<i>1.56</i>	0.06	<i>2.33</i>	0.04	<i>3.50</i>	
Matrix10876	0.46	1.49 (1.20)	1.40	<i>1.06</i>	1.34	<i>1.11</i>	1.30	<i>1.15</i>	
4cols	0.32	0.14 (0.01)	0.07	<i>2.00</i>	0.05	<i>2.80</i>	0.04	<i>3.50</i>	
lhr14c	1.10	1.56 (0.10)	0.86	<i>1.81</i>	0.58	<i>2.69</i>	0.35	<i>4.46</i>	
bayer04	0.31	0.44 (0.16)	0.32	<i>1.37</i>	0.25	<i>1.76</i>	0.21	<i>2.09</i>	
10cols	2.07	0.39 (0.03)	0.22	<i>1.78</i>	0.13	<i>3.20</i>	0.10	<i>3.90</i>	
Matrix32406	6.52	6.89 (1.27)	4.79	<i>1.44</i>	3.29	<i>2.09</i>	2.75	<i>2.50</i>	
lhr34c	3.66	5.17 (0.35)	2.84	<i>1.82</i>	1.88	<i>2.75</i>	1.22	<i>4.24</i>	
Matrix35640	NS	16.0 (4.15)	10.6	<i>1.50</i>	7.35	<i>2.18</i>	6.04	<i>2.65</i>	
bayer01	0.65	0.61 (0.11)	0.36	<i>1.69</i>	0.23	<i>2.65</i>	0.15	<i>4.07</i>	
lhr71c	7.56	11.7 (0.39)	7.08	<i>1.65</i>	3.90	<i>3.00</i>	2.37	<i>4.97</i>	
icomp	0.13	0.30 (0.01)	0.18	<i>1.67</i>	0.11	<i>2.72</i>	0.08	<i>3.75</i>	

Table 6.3: Timings for Fast Factorize in seconds on an Origin 2000. NS denotes not solved. Numbers in parentheses are times for factorizing the interface problem. The numbers in italics are the speedups for HSL_MP48 compared with using a single processor.

Identifier	MA48	HSL_MP48 ($N = 8$)							
		No. processors							
		1	2	4	8				
ethylene-2	0.013	0.019	0.013	<i>1.46</i>	0.010	<i>1.90</i>	0.009	<i>2.11</i>	
ethylene-1	0.013	0.019	0.011	<i>1.72</i>	0.010	<i>1.90</i>	0.010	<i>1.90</i>	
Matrix10876	0.024	0.045	0.037	<i>1.22</i>	0.032	<i>1.41</i>	0.028	<i>1.61</i>	
4cols	0.017	0.017	0.012	<i>1.42</i>	0.010	<i>1.70</i>	0.010	<i>1.70</i>	
lhr14c	0.068	0.091	0.060	<i>1.52</i>	0.035	<i>2.60</i>	0.022	<i>2.72</i>	
bayer04	0.035	0.045	0.031	<i>1.45</i>	0.020	<i>2.25</i>	0.019	<i>2.37</i>	
10cols	0.074	0.055	0.034	<i>1.62</i>	0.027	<i>2.03</i>	0.023	<i>2.39</i>	
Matrix32406	0.177	0.220	0.154	<i>1.43</i>	0.105	<i>2.09</i>	0.086	<i>2.56</i>	
lhr34c	0.195	0.243	0.167	<i>1.45</i>	0.101	<i>2.41</i>	0.071	<i>3.42</i>	
Matrix35640	NS	0.221	0.165	<i>1.34</i>	0.120	<i>1.84</i>	0.090	<i>2.45</i>	
bayer01	0.105	0.116	0.082	<i>1.41</i>	0.050	<i>2.32</i>	0.047	<i>2.47</i>	
lhr71c	0.393	0.514	0.324	<i>1.59</i>	0.199	<i>2.58</i>	0.126	<i>4.08</i>	
icomp	0.065	0.085	0.063	<i>1.35</i>	0.052	<i>1.63</i>	0.052	<i>1.63</i>	

Table 6.4: Timings for Solve in seconds on an Origin 2000. NS denotes not solved. The numbers in italics are the speedups for HSL_MP48 compared with using a single processor.

Identifier	MA48	HSL_MP48 ($N = 8$)
ethylene-2	47	58 (0.9)
ethylene-1	52	55 (0.8)
Matrix10876	555	1944 (1234)
4cols	276	95 (6)
lhr14c	937	1069 (90)
bayer04	161	347 (138)
10cols	1611	183 (19)
Matrix32406	4986	10585 (1178)
lhr34c	3053	3618 (399)
Matrix35640	543554	15794 (3897)
bayer01	251	268 (5)
lhr71c	6291	7531 (608)
icomp	2	189 (0.4)

Table 6.5: Flops ($\times 10^5$) required by MA48 and HSL_MP48. Numbers in parentheses are the flops for factorizing the interface problem.

6.1 Timings on other platforms

Experiments with HSL_MP48 have also been performed on other platforms. In Table 6.6 we present timings for HSL_MP48 on a Compaq DS20, which has 2 processors. Desktop computers with a small number of processors are increasingly common as they become more affordable and our results illustrate the very worthwhile speedups that can be achieved using just 2 processors.

Identifier	AFS			F2			FF			S		
	1	2	<i>1.81</i>	1	2	<i>1.69</i>	1	2	<i>2.07</i>	1	2	<i>1.40</i>
ethylene-2	0.42	0.23	<i>1.81</i>	0.10	0.06	<i>1.69</i>	0.06	0.03	<i>2.07</i>	0.007	0.005	<i>1.40</i>
ethylene-1	0.42	0.26	<i>1.60</i>	0.09	0.05	<i>1.84</i>	0.05	0.03	<i>1.59</i>	0.008	0.005	<i>1.60</i>
Matrix10876	1.91	1.65	<i>1.16</i>	0.59	0.54	<i>1.08</i>	0.50	0.47	<i>1.07</i>	0.019	0.015	<i>1.27</i>
4cols	0.24	0.14	<i>1.64</i>	0.07	0.04	<i>1.73</i>	0.05	0.03	<i>1.82</i>	0.006	0.004	<i>1.50</i>
lhr14c	3.19	1.77	<i>1.80</i>	0.73	0.40	<i>1.80</i>	0.56	0.31	<i>1.80</i>	0.034	0.023	<i>1.48</i>
bayer04	0.86	0.60	<i>1.43</i>	0.24	0.16	<i>1.55</i>	0.17	0.11	<i>1.50</i>	0.019	0.012	<i>1.58</i>
10cols	1.06	1.00	<i>1.06</i>	0.27	0.14	<i>1.85</i>	0.16	0.09	<i>1.85</i>	0.019	0.013	<i>1.46</i>
Matrix32406	10.42	7.80	<i>1.34</i>	2.81	1.95	<i>1.44</i>	1.94	1.10	<i>1.48</i>	0.077	0.060	<i>1.28</i>
lhr34c	11.22	6.21	<i>1.81</i>	2.48	1.39	<i>1.78</i>	0.27	0.15	<i>1.77</i>	0.090	0.065	<i>1.38</i>
Matrix35640	18.64	13.01	<i>1.43</i>	6.01	3.98	<i>1.51</i>	4.76	2.85	<i>1.48</i>	0.078	0.063	<i>1.24</i>
bayer01	1.62	0.90	<i>1.80</i>	0.41	0.22	<i>1.90</i>	0.14	0.08	<i>1.88</i>	0.044	0.030	<i>1.47</i>
lhr71c	27.26	15.39	<i>1.77</i>	6.08	3.44	<i>1.77</i>	5.49	3.71	<i>1.67</i>	0.182	0.131	<i>1.39</i>
icomp	0.60	0.39	<i>1.53</i>	0.20	0.11	<i>1.74</i>	2.45	1.65	<i>1.68</i>	0.033	0.025	<i>1.32</i>

Table 6.6: Timings in seconds and speedups for HSL_MP48 run on 1 and 2 processors of a Compaq DS20. Speedups are given in italics.

In Table 6.7, we present timings for the analyse phase of MA48 and HSL_MP48 on the Compaq DS20. In addition, we include timings for HSL_MC66, the HSL implementation of the MONET algorithm used to preorder the matrices to SBBD form before running HSL_MP48. We see that for some problems the HSL_MC66 time can be greater than the

time for the analyse phase of HSL_MP48, resulting in the total analyse cost (that is, the preordering time plus the time for the analyse phase of HSL_MP48) being greater than the MA48 analyse cost. 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.

Identifier	MA48	HSL_MC66	HSL_MP48 ($N = 8$)	
			No. processors	
			1	2
ethylene-2	0.24	1.73	0.31	0.17
ethylene-1	0.23	1.60	0.31	0.19
Matrix10876	1.08	1.67	0.44	0.25
4cols	0.81	0.38	0.15	0.09
lhr14c	2.09	2.22	2.35	1.29
bayer04	0.87	1.49	0.52	0.33
10cols	2.23	0.94	0.74	0.42
Matrix32406	12.18	14.68	6.45	4.65
lhr34c	6.72	5.93	8.41	4.56
Matrix35640	NS	1.80	9.62	5.90
bayer01	1.97	2.18	1.15	0.64
lhr71c	13.84	11.41	20.84	11.55
icom	0.23	2.78	0.36	0.24

Table 6.7: Timings for Analyse in seconds on a Compaq DS20. NS denotes not solved.

Identifier	AFS				F2			
	1	2	4	8	1	2	4	8
ethylene-2	1.35	0.74	0.53	0.39	0.33	0.17	0.11	0.08
ethylene-1	1.31	0.79	0.44	0.31	0.33	0.19	0.10	0.07
Matrix10876	5.97	5.06	4.55	4.31	1.71	1.39	1.25	1.21
4cols	0.74	0.44	0.28	0.20	0.22	0.12	0.07	0.05
lhr14c	10.93	5.85	3.56	2.37	3.14	1.55	0.98	0.54
bayer04	2.89	1.69	1.13	0.86	0.82	0.45	0.32	0.22
10cols	3.86	2.14	1.19	0.76	1.12	0.65	0.33	0.21
lhr34c	39.63	20.00	11.77	7.05	11.73	5.66	3.14	1.80
bayer01	5.32	2.81	1.70	1.10	1.64	0.85	0.50	0.30
Matrix35640	NS	38.32	27.86	22.41	NS	15.06	7.47	5.91
lhr71c	NS	49.16	27.21	14.56	NS	13.10	7.23	3.93
icom	2.07	1.27	0.78	0.47	1.07	0.48	0.27	0.16
Matrix35640*	71.47	47.98	34.33	33.93	21.63	15.03	11.22	9.31
lhr71c*	107.61	59.30	33.78	18.42	30.59	20.72	9.78	5.65

Table 6.8: HSL_MP48 timings in seconds for AFS and F2 run on 1, 2, 4, and 8 processors of a Cray T3E. NS denotes not solved. * denotes factors stored in files

In Tables 6.8 and 6.9 timings are given for HSL_MP48 when run on 1, 2, 4 and 8 processors of the Cray T3E-1200E at Manchester, UK. As in the other experiments reported on in this paper, the factors were held in main memory and no use was made of sequential files. Working in main memory, it was not possible to solve test problems

Identifier	FF				S			
	1	2	4	8	1	2	4	8
ethylene-2	0.23	0.12	0.08	0.05	0.027	0.015	0.010	0.007
ethylene-1	0.23	0.13	0.07	0.05	0.028	0.017	0.010	0.007
Matrix10876	1.32	1.16	1.04	1.02	0.041	0.030	0.024	0.022
4cols	0.14	0.08	0.05	0.03	0.021	0.013	0.009	0.007
lhr14c	2.46	1.28	0.79	0.46	0.087	0.070	0.026	0.018
bayer04	0.61	0.36	0.25	0.18	0.048	0.028	0.019	0.015
10cols	0.74	0.40	0.22	0.14	0.060	0.032	0.021	0.015
lhr34c	8.30	4.33	2.58	1.48	0.205	0.110	0.065	0.044
bayer01	1.20	0.62	0.37	0.22	0.121	0.069	0.042	0.030
Matrix35640	NS	9.17	6.64	5.34	NS	0.118	0.080	0.064
lhr71c	NS	10.76	6.01	3.22	NS	0.223	0.128	0.079
icomp	0.72	0.40	0.23	0.14	0.085	0.055	0.036	0.028
Matrix35640*	21.54	15.78	10.75	9.35	6.256	5.179	3.054	2.989
lhr71c*	30.45	16.88	9.40	5.49	12.929	6.775	4.191	2.033

Table 6.9: HSL_MP48 timings in seconds for FF and S run on 1, 2, 4, and 8 processors of a Cray T3E. NS denotes not solved. * denotes factors stored in files

Identifier	AFS					F2				
	1	2	4	8	16	1	2	4	8	16
ethylene-2	1.03	0.55	0.34	0.23	0.18	0.23	0.13	0.07	0.05	0.03
ethylene-1	1.26	0.57	0.36	0.21	0.16	0.25	0.14	0.09	0.05	0.03
Matrix10876	14.80	14.03	13.70	13.57	13.52	3.59	3.40	3.32	3.29	3.28
4cols	0.59	0.39	0.30	0.29	0.23	0.15	0.10	0.07	0.06	0.05
lhr14c	6.90	4.17	2.71	2.00	1.60	1.81	1.11	0.72	0.54	0.43
bayer04	2.55	1.78	1.41	1.24	1.16	0.69	0.46	0.34	0.30	0.27
10cols	2.45	1.47	0.95	0.70	0.57	0.66	0.38	0.23	0.19	0.13
lhr34c	32.94	18.05	10.67	6.41	4.29	10.51	4.90	2.93	1.73	1.15
bayer01	4.70	2.62	1.47	0.96	0.65	1.40	0.76	0.40	0.24	0.16
Matrix35640 [†]	64.20	45.12	49.04	46.30	44.12	17.37	14.32	12.82	11.95	15.92
lhr71c	NS	NS	24.50	15.76	9.93	NS	NS	6.75	4.08	2.50
icomp	2.19	1.21	0.71	0.49	0.36	0.84	0.46	0.24	0.15	0.09
lhr71c*	96.40	55.84	31.61	27.73	16.15	29.90	16.64	9.97	7.06	5.43

Table 6.10: HSL_MP48 timings in seconds for AFS and F2 run on 1, 2, 4, 8, and 16 processors of a Cray T3E. NS denotes not solved. * denotes that factors are stored in files. [†] denotes that the host process is run on a separate (extra) processor.

Identifier	FF					S				
	1	2	4	8	16	1	2	4	8	16
ethylene-2	0.16	0.09	0.05	0.03	0.02	0.027	0.016	0.010	0.008	0.006
ethylene-1	0.21	0.10	0.06	0.03	0.03	0.034	0.016	0.011	0.008	0.006
Matrix10876	3.12	2.98	2.91	2.90	2.88	0.057	0.048	0.043	0.041	0.041
4cols	0.10	0.07	0.07	0.04	0.04	0.021	0.014	0.010	0.008	0.007
1hr14c	1.52	0.92	0.61	0.46	0.37	0.074	0.044	0.030	0.023	0.019
bayer04	0.50	0.34	0.26	0.23	0.21	0.047	0.029	0.020	0.017	0.015
10cols	0.43	0.16	0.25	0.11	0.09	0.053	0.031	0.021	0.017	0.014
1hr34c	7.36	4.06	2.42	1.42	0.94	0.209	0.114	0.069	0.048	0.036
bayer01	1.00	0.55	0.29	0.18	0.12	0.114	0.065	0.040	0.030	0.025
Matrix35640 [†]	15.41	12.94	11.67	10.93	10.53	0.201	0.146	0.116	0.103	0.097
1hr71c	NS	NS	5.61	3.33	2.09	NS	NS	0.141	0.088	0.064
icomp	0.73	0.37	0.19	0.12	0.08	0.099	0.056	0.036	0.029	0.025
1hr71c*	34.80	18.00	10.25	7.30	5.54	10.911	7.347	4.090	3.702	3.270

Table 6.11: HSL_MP48 timings in seconds for FF and S run on 1, 2, 4, 8, and 16 processors of a Cray T3E. NS denotes not solved. * denotes that factors are stored in files. [†] denotes that the host process is run on a separate (extra) processor.

Matrix35640 and 1hr71c using a single processor (denoted by NS). For these problems, we also experimented with holding the factors in files and timings for these runs are included at the end of the tables. We see that using files can add a significant overhead, particularly to the solve phase, and hence our recommendation is that this option is used only when the user wishes to retain the factors for future use or the memory requirements are too large. In order to demonstrate the scalability of our code on our test problems on up to 16 processors, we show in Tables 6.10 and 6.11 the results of running on 1, 2, 4, 8, and 16 processors of the Cray T3E with a partitioning of the matrices into an SBBD form with 16 blocks on the diagonal. Because of the size of the interface problem for Matrix35640, we were unable to run subsequent factorizations of this problem as an allocation error was returned by the host. However, when we used the host process to solve only the interface and did not use this for any of the subproblems, we obtained the results shown in Tables 6.10 and 6.11. We see that, in most cases, we still have good scalability although in nearly half the cases the larger interface system causes the times (on the same number of processors) to be worse than the partitioning with 8 blocks. We thus conclude that, for our set of test problems, we can sometimes benefit from using 16 processors.

7 Concluding comments

We have designed and developed an algorithm and code for the solution of large sparse unsymmetric systems on parallel computers. We have seen that our new code HSL_MP48 performs well in all phases of the solution and has good speedup on our realistic test problems for up to 8 and sometimes 16 processors.

It is crucial to first order the matrix to Singly Bordered Block Diagonal form (3.1), and we have used the HSL implementation HSL_MC66 of the MONET algorithm to effect an *a priori* permutation to this form. Contrary to earlier experiments of Arioli and Duff (1990)

on ordering to bordered form, we find that, for most of our problems, HSL_MC66 is able to keep the border small while balancing the size of the subproblems. This is in part due to the fact that the partitioning algorithm leaves rectangular blocks on the diagonal. These in turn are a powerful tool for obtaining a dynamic and stable decomposition to doubly bordered form of a similar kind to that obtained in domain decomposition partitioning. Indeed, the ordering is so effective 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 cost of ordering a matrix to SBBD form using the present serial code can be more expensive than the subsequent analyse phase and thus represents a significant overhead. Thus, if we are only going to do a single analyse/factorize/solve, it is important to develop a parallel version of the partitioning routine.

The performance of HSL_MP48 is very sensitive to the amount of work in the interface problem. Although, this cannot in general be predicted in advance, we note that the proportion of columns in the border generally gives a good indication of performance. As a rule of thumb, our numerical experiments suggest that the algorithm works well if less than 5% of the columns lie in the border. This is the principal reason that our approach would not normally extend to more than 16 processors.

8 Availability

HSL_MP48 is written in standard Fortran 90 and calls the MPI package for message passing. HSL_MC66 is written in standard Fortran 95. Both codes are distributed through HSL. Further information on HSL can be obtained from the Web page www.cse.clrc.ac.uk/nag/hsl.

References

- M. Arioli and I. S. Duff. Experiments in tearing large sparse systems. *in* M. G. Cox and S. Hammarling, eds, 'Reliable Numerical Computation', pp. 207–226, Oxford, 1990. Oxford University Press.
- T. Davis. University of Florida Sparse Matrix Collection. *NA Digest*, **97**(23), 1997. Full details from www.cise.ufl.edu/~davis/sparse/.
- 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. MA28 - a set of Fortran subroutines for sparse unsymmetric systems. Report AERE R8730, Her Majesty's Stationery Office, London, 1977.
- I.S. Duff and J.K. Reid. MA48, a Fortran code for direct solution of sparse unsymmetric linear systems of equations. Report RAL-93-072, Rutherford Appleton Laboratory, 1993.

- 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. in J. Lewis, ed., ‘Proceedings of the Fifth SIAM Conference Applied Linear Algebra’, pp. 567–571. SIAM, 1994.
- J. R. Gilbert and T. Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Scientific and Statistical Computing*, **9**, 862–874, 1988.
- 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.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Systems Tech. J.*, **49**, 291–308, 1970.
- H.M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Science*, **3**, 255–269, 1957.
- MPI. A message-passing interface standard. *Inter. Journal of Supercomputer Applics*, **8**, 1994. Special edition on MPI.
- J.A. Scott. The design of a portable parallel frontal solver for chemical process engineering problems. *Computers in Chemical Engineering*, **25**, 1699–1709, 2001a.
- J.A. Scott. A parallel solver for finite element applications. *Inter. Journal on Numerical Methods in Engineering*, **50**, 1131–1141, 2001b.
- J.A. Scott. Parallel frontal solvers for large sparse linear systems. Technical Report RAL-TR-2002-012, Rutherford Appleton Laboratory, 2002.
- Z. Zlatev. On some pivotal strategies in Gaussian elimination by sparse technique. *SIAM J. Numerical Analysis*, **17**, 18–30, 1980.