RAL-TR-2000-030

Two-stage ordering for unsymmetric parallel row-by-row frontal solvers.

by

Jennifer A. Scott

Abstract

The row-by-row frontal method may be used to solve general large sparse linear systems of equations. By partitioning the matrix into (nearly) independent blocks and applying the frontal method to each block, a coarse-grained parallel frontal algorithm is obtained. The success of this approach depends on preordering the matrix. This can be done in two stages: (1) order the matrix to bordered block diagonal form (2) order the rows within each block to minimise the size of the frontal matrix. A number of recent papers have considered stage (1). In this paper, an algorithm is proposed for stage (2). For a range of practical examples from chemical process engineering it is shown that the proposed algorithm substantially reduces the block frontal matrix size and, for sufficiently large problems, this can lead to significant reductions in the factorization times when the row-by-row frontal method is implemented in parallel.

Keywords: Row ordering, parallel frontal method, unsymmetric sparse matrices.

Computational Science and Engineering Department,
Atlas Centre, Rutherford Appleton Laboratory,
Oxon OX11 0QX, England.
j.scott@rl.ac.uk

July 12, 2000.

Contents

1 Introduction

The frontal method is often used for solving the large sparse systems of linear equations that arise in large-scale chemical process simulation and optimization problems. One reason for this is that the frontal method can be used to solve any general sparse linear system, because 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 process simulation matrices possess none of these desirable properties, the choice of suitable solvers is restricted.

The frontal method is able to achieve good performance on a wide range of modern computer architectures (including RISC based processors and shared memory parallel processors) through the exploitation of dense linear algebra kernels in the innermost loop of the factorization. However, a major deficiency of the method is the lack of scope for parallelism beyond that which can be obtained within the dense kernels. One way of attempting to overcome this shortcoming is by generalising the method to the multiple front method (Duff and Scott, 1994*a*, 1994*b*).

The multiple front method uses a problem decomposition corresponding to a bordered block diagonal matrix and factorizes each of the diagonal blocks using the frontal method. This can be done in parallel. The solvers `PFAMP` and `MP42` follow this approach. `PFAMP` was developed by Cray Research and is specifically designed for process engineering problems. The code and the algorithm it implements is described in Mallya, Zitney, Choudhary and Stadtherr (1997*b*, 1997*a*). The package `MP42` of Scott (1999*a*) is a general multiple front code for finite element problems and was developed using the established frontal solver `MA42` of Duff and Scott (1996). Both `PFAMP` and `MP42` require the user to have preordered the matrix to bordered block diagonal form. For good load balancing, it is desirable that the diagonal blocks are of nearly equal size. In addition, the blocks need to be as independent as possible (that is, the interaction between the blocks through the number of columns that have entries in more than one block should be kept as small as possible). In the simulation and optimization of large-scale chemical processes, the natural unit-stream structure may provide an ordering with little overlap between the blocks. However, the blocks are likely to vary significantly in size. Moreover, commercial simulators do not always generate a matrix with a suitable block diagonal form. A reordering algorithm is thus essential. Recently, a number of such algorithms have been proposed for unsymmetric matrices. These include the GPA-SUM algorithm of Camarda and Stadtherr (1998) and the MONET algorithm of Hu, Maguire and Blake (2000).

Having performed an appropriate reordering to bordered block diagonal form, the efficiency of the multiple front method depends on the assembly order used by the frontal method within each block. This requires a second stage of ordering and it is this row ordering within the diagonal blocks that is of interest to us. When investigating the potential of the multiple front method for process simulation problems, Mallya et al. (1997*a*) realised the need to order within the blocks and conjectured that the performance of their parallel multiple front

solver PFAMP "may depend strongly on the ordering of the rows within each block". However, Mallya et al. did not provide any results to support this and in their work they made no attempt to reorder the rows within the blocks. The purpose of this paper is to illustrate the importance of having a good row ordering within each block and to propose an algorithm for achieving such an ordering.

This paper is organised as follows. In Section 2, we recall key features of the row-by-row frontal method and its generalisation to the multiple front method. In Section 3, we describe a row ordering algorithm for use with the frontal method and look at how we can extend the method so that it only orders a subset of the rows of the matrix; this extension allows the method to be used for ordering the rows within the matrix blocks of the multiple front method. Numerical results for a range of test examples taken from practical chemical process engineering problems are presented in Section 4. Finally, in Section 5, some concluding comments are made.

We remark that for problems arising from finite-element applications, the diagonal blocks have a symmetric structure and an appropriate ordering algorithm for the elements within each block has been developed by Scott (1996).

2 Background

In this section, we briefly recall the row-by-row frontal method and its generalisation to the multiple front method.

2.1 Frontal method

Consider the linear system of equations

$$Ax = b, \tag{2.1}$$

where the $n \times n$ matrix A is large and sparse, and the right-hand side vector b and solution vector x are of length n. The frontal method is a variant of Gaussian elimination that was originally developed in the 1970s for the solution of finite-element problems in which A is a sum of elemental matrices (see Irons, 1970, Hood, 1976). The original motiviation was the need to solve problems from finite-element applications that were large by the standard of the day using only the limited amount of high-speed memory then available. This was achieved by limiting the computational work to a relatively small matrix, termed the *frontal matrix*. The method was subsequently extended to the solution of general sparse linear systems by Duff (1981, 1984). When the method is used for a general system, we refer to it as the *row-by-row frontal method* (and, to distinguish it from the multiple front approach, we also refer to it as the unifrontal method). Today the method is widely used on vector supercomputers because, by treating the frontal matrix as a full matrix, most of the arithmetic operations can be performed using highly efficient vectorized dense kernels.

At each stage, the row-by-row frontal method comprises the following steps:

- Assemble a row of A into the frontal matrix.
- Determine if any columns are fully summed. Column l is defined as being *fully summed* once the last row with an entry in column l has been assembled.
- If there are any fully summed columns, perform partial pivoting in those columns, eliminating the pivot rows and columns and performing an outer-product update on the remaining frontal matrix.
- Optionally write the rows and columns of the matrix factors generated by the eliminations to auxiliary storage (for example, direct-access files).

In this way, the method proceeds by interleaving assembly and elimination operations until, once all the rows have been assembled and the final eliminations performed, a decomposition of a permutation of A is computed, that is,

$$PAQ = LU,$$

where L is unit lower triangular and U is upper triangular. The system (2.1) can then be solved by a simple forward substitution

$$Ly = Pb,$$

followed by a backsubstitution

$$Uz = y.$$

The required solution

$$x = Qz$$

follows.

Since a variable can only be eliminated after its column is fully summed, the order in which the rows are assembled will determine both how long each variable remains in the front and the order in which the variables are eliminated. For the row-by-row frontal method to be efficient, both in terms of storage and arithmetic operations, the rows need to be assembled in an order that keeps both the row and column frontsizes as small as possible. In recent years, a number of algorithms for automatically ordering the rows of A have been proposed. These methods are reviewed by Scott (1999*c*). The most successful methods currently available are the MSRO methods of Scott (1999*b*), which we discuss in Section 3.1.

2.2 Multiple front method

The multiple front method is a coarse-grained parallel approach in which the frontal method is applied simultaneously to multiple independent or loosely connected blocks. For the unsymmetric case, the matrix must first be ordered to singly bordered block diagonal form

$$\begin{pmatrix} A_{11} & & & C_1 \\ & A_{22} & & C_2 \\ & & \ldots & . \\ & & A_{NN} & C_N \end{pmatrix}, \tag{2.2}$$

where the rectangular diagonal blocks A_{ll} are $m_l \times n_l$ matrices with $m_l \geq n_l$, and the border blocks C_l are $m_l \times k_l$. This ordering needs to be chosen so that $k_l \ll n_l$. A partial LU decomposition is performed on each of the matrices

$$\begin{pmatrix} A_{ll} & C_l \end{pmatrix} \tag{2.3}$$

using the frontal method. This can be done in parallel. As the rows of (2.3) are assembled, n_l variables become fully summed. These variables correspond to the columns of A_{ll}; the columns of C_l do not become fully summed because they have entries in at least one other border block C_j $(j \neq l)$. The fully summed variables may be eliminated, using partial pivoting to ensure numerical stability. Because the A_{ll} are, in general, rectangular, at the end of the assembly and elimination processes, for each block there will remain a frontal matrix F_l of order $(m_l - n_l) \times k_l$. The variables that remain in the front are termed the *interface variables* and

$$F = \sum_{l=1}^{N} F_l \tag{2.4}$$

is the *interface matrix*. The interface matrix F may also be factorized using the frontal method. Once the interface variables have been computed, the rest of the block back-substitution can be performed (in parallel) to complete the solution.

Provided the rows of A have been ordered so that there is limited interaction between the blocks, the size of the interface problem will be very small compared to the overall problem size. In this case, the solution time will be dominated by the most expensive block LU decomposition. The problem of obtaining a bordered block form with (approximately) equal diagonal blocks and a small interface has been addressed in a number of recent papers (see, for example, Camarda and Stadtherr, 1998 and Hu et al., 2000). In this study, we use the MONET code of Hu et al. (2000) to generate the block form (2.2).

3 Row ordering within blocks

In this section, we recall the MSRO algorithm for reordering **all** the rows of a sparse unsymmetric matrix for use with a row-by-row frontal solver and look at how it may be extended to obtain orderings for the multiple front algorithm.

3.1 The MSRO algorithm

The MSRO method of Scott (1999*b*) is an algorithm for ordering the rows of an unsymmetric matrix A. The algorithm has its origins in the work on profile reduction of Sloan (1986). It involves selecting a global row ordering and then refining this global row ordering to obtain a row reordering for A. The algorithm uses the row graph of A. Row graphs were originally introduced by Mayoh (1965). The *row graph* G_R of A is defined to be the undirected graph of the symmetric matrix $B = A * A^T$, where $*$ denotes matrix multiplication without taking cancellations into account (so that, if an entry in B is zero as

a result of numerical cancellation, it is considered as a nonzero entry and the corresponding edge is included in the row graph). The nodes of G_R correspond to the rows of A and two rows i and j $(i \neq j)$ are adjacent if and only if there is at least one column k of A for which a_{ik} and a_{jk} are both nonzero. Row permutations of A correspond to relabelling the nodes of the row graph.

The chosen global ordering is used by the MSRO algorithm to define the global priority of each row. The row with the highest global priority is chosen as the *start* row (that is, the row that is first in the global ordering is ordered first in the new ordering). In the second phase of the MSRO algorithm, the global ordering is used to guide the reordering, with rows having a low global priority being chosen towards the end of the ordering. Examples of global orderings used by Scott (1999*b*) are the pseudodiameter of G_R (also used by Sloan, 1986) and the spectral ordering for G_R (see Kumfert and Pothen, 1997).

A row is defined to be *active* if it has not yet been reordered but is adjacent in the row graph to a row that has already been reordered. The MSRO algorithm aims to reduce the row and column frontsizes within the frontal method by reducing the number of rows that are active at each stage of the method. This is achieved by a local reordering (or refinement) of the global ordering. For each row i, the MSRO algorithm computes the priority function

$$P_i = -W_1 \; rcgain_i + -W_2 \; g_i. \tag{3.1}$$

Here W_1 and W_2 are positive weights, g_i is the (positive) global priority for row i, and $rcgain_i$ is the sum of the increases to the row and column frontsizes resulting from assembling (ordering) row i next. A first pass through the rows counts how many times each column index appears. Assembling a row into the frontal matrix causes the row frontsize to either increase by one, to remain the same, or to decrease. The row frontsize increases by one if each column index appears in at least one of the rows that has not yet been assembled, it remains the same if a single column index appears for the last time, and it decreases if more than one column appears for the last time. The increase in the column frontsize is the difference between the number of column indices that appear in the front for the first time and the number that appear for the last time. If this difference is negative, the column frontsize decreases. Hence, if s_i is the number of column indices that appear for the last time when row i is assembled, and $newc_i$ is the number of column indices that enter the front for the first time, then

$$rcgain_i = 1 + newc_i - 2s_i. \tag{3.2}$$

This is small when assembling row i brings a small number of new columns into the front but results in a large number of columns appearing for the final time.

The start row is ordered first then, at each stage, the next row in the ordering is chosen from a list of eligible rows to maximise P_i. The *eligible rows* are the active rows plus their neighbours. A list of eligible rows is maintained using the connectivity lists for the row graph. Thus, the MSRO algorithm attempts to keep a balance between having only a small number of rows and columns in the front and including rows that have a high global priority. The

balance is determined by the choice of weights. On the basis of extensive numerical experimentation, Scott (1999*b*) recommends using two sets of weights and choosing the better order. Scott uses the pairs (2,1) and (32,1) if the global priority is a pseudodiameter of the row graph and (1,2) and (32,1) if the spectral ordering is used. Throughout the remainder of this paper, we use a pseudodiameter of the row graph to define g_i and, unless stated otherwise, the weights used are the default pairs (2,1) and (32,1).

Reid and Scott (1999) have shown that the maximum and mean column frontsizes are invariant if a given row order is reversed. However, the maximum and mean row frontsizes and the mean frontal matrix size are, in general, different if the given order is reversed. Numerical experimentation has shown that, for some examples, the mean frontal matrix size can be significantly reduced by reversing a given row order while for other examples, the converse is true. Thus, the MSRO algorithm computes an ordering as described above and calculates the mean frontal matrix sizes for this ordering and for the reverse ordering; the ordering for which the mean frontal matrix size is the smallest is then selected as the MSRO ordering.

We remark that although the MSRO algorithm was designed for ordering the rows of a square matrix, because the algorithm orders the nodes of the row graph and the row graph is defined for any $m \times n$ matrix, it may also be used to order the rows of a rectangular matrix. In the next section, we discuss how we can generalise the method to order the rows in the rectangular blocks of the block diagonal matrix (2.2).

3.2 A two-stage approach

A two-stage approach to row ordering for a parallel frontal solver comprises the stages:

1. preorder the matrix to block diagonal form (2.2)
2. reorder the rows within each block matrix $(A_{ll} \;\; C_l)$.

As already mentioned, Stage (1) can be performed using, for example, the MONET algorithm of Hu et al. (2000). Our interest lies in Stage (2). The simplest approach (which we will refer to as the MSRO(1) method) is to apply the MSRO algorithm directly to each block matrix $(A_{ll} \;\; C_l)$. However, the MSRO algorithm is designed to reordered **all** the rows of a matrix. Thus, when choosing which row to order next, it is assumed that when any column index appears for the last time, the column is fully summed and so can be eliminated. If we apply the MSRO algorithm to $(A_{ll} \;\; C_l)$ this assumption will not be valid for columns belonging to the border C_l. Consequently, during the factorization of the block, the elimination order predicted by the MSRO ordering must be modified and, as interface variables cannot be eliminated, the frontsize will increase beyond that predicted. If the interface variables are brought into the front early in the ordering because their rows have a low priority P_i, this will lead to a large number of additional operations being performed as well as a need for more memory.

An alternative approach is to apply the MSRO algorithm only to the matrix A_{ll}. If we do this, then when the frontal method is applied to (A_{ll} C_l), the row frontsize at each stage will be as predicted by the MSRO algorithm but the column frontsize will increase by up to k_l, where k_l is the number of columns of C_l with at least one entry. This suggests that this approach will work well provided the number of interface variables is small compared with the number of non-interface variables. We will denote this method by MSRO(2).

A third approach is to modify the second step of the MSRO algorithm so that, when applied to (A_{ll} C_l), the columns within the border C_l are recognised as not being fully summed and so are not removed from the frontal matrix. When implementing the MSRO algorithm, the initial priority of the ith row in (A_{ll} C_l) is given by

$$P_i = -W_1\ (1 + len_i) - W_2\ g_i, \tag{3.3}$$

where len_i is the number of entries in row i. As each row i is assembled, the priority of each of its neighbours j is updated in two phases as follows.

1. For each column index k that appears for the first time in row i with a_{ik} and a_{jk} nonzero,

$$P_j \leftarrow P_j + W_1. \tag{3.4}$$

2. For each column index k that appears for the last time in row i with a_{ik} and a_{jk} nonzero,

$$P_j \leftarrow P_j + 2\ W_1. \tag{3.5}$$

If A_{ll} has m_l rows and we flag each column with a nonzero entry in C_l as appearing for the final time in row $m_l + 1$, no updates of the form (3.5) will be made for these columns. Thus each row j that has nonzero entries in C_l will have a lower priority value than it would otherwise have had and hence the selection of such rows will be delayed. We will denote this method by MSRO(3).

A weakness of approach MSRO(3) is that, when selecting the next row to assemble, it is does not distinguish between the interface and non-interface variables within the rows. We can modify the method further to allow for this. We do this by replacing the priority function (3.1) for row i with the following priority function

$$P_i = -W_1\ rcgain_i - W_2\ g_i + W_3\ nold_i, \tag{3.6}$$

where W_3 is another (positive) weight and $nold_i$ is the number of non-interface variables in row i that have already been introduced into the front. Initially, $nold_i = 0$. As rows are assembled, $nold_i$ increases, so that rows with a large number of non-interface variables already lying in the front are given preference. This term acts as a tie-breaker for, if two rows result in the same increase to the frontal matrix and have the same global priority, then the one that has the most non-interface variables already in the front is selected. The aim here is to try and ensure non-interface variables become fully summed as soon as possible after they have entered the front. We will denote this method by MSRO(3,W_3) (with MSRO(3,0) = MSRO(3)).

4 Numerical results

The test problems used in our numerical experiments are listed in Table 4.1. Each problem comes from chemical process engineering. Problems marked with a † were taken from the University of Florida Sparse Matrix Collection (Davis, 1997). The remaining problems were supplied by Mark Stadtherr of the University of Notre Dame; further details of these problems may be found in Mallya et al. (1997*b*).

Table 4.1: The test problems. † indicates problem taken from University of Florida Sparse Matrix Collection.

Identifier	Order	Number of entries
`4cols`	11770	43668
`10cols`	29496	109588
`bayer01`†	57735	277774
`bayer03`†	6747	56196
`bayer04`†	20545	159082
`bayer09`†	3083	21216
`ethylene-1`	10673	80904
`ethylene-2`	10353	78004
`icomp`	75724	338711
`lhr07c`†	7337	156508
`lhr14c`†	14270	307858
`lhr34c`†	35152	764014
`lhr71c`†	70304	1528092

The reported numerical results were all obtained on the SGI Origin 2000 at Manchester, UK. The MONET code was used to order the matrix to singly bordered block diagonal form (2.2). In our experiments, we have chosen to set the number of blocks to $N = 4$ but MONET may be used with any value of N.

In Table 4.2, we compare the performance of the different variants of the MSRO algorithm introduced in the previous section. The comparison is based on the mean frontal matrix size f_{avg} within each block, which is defined to be

$$f_{avg} = \frac{1}{n_l} \sum_{1}^{n_l} (frow_i * fcol_i), \tag{4.1}$$

where $frow_i$ and $fcol_i$ denote the row and column frontsizes before the ith elimination in the matrix block (n_l is the number of columns in the block). Note that $f_{avg} * n_l$ provides a prediction of the number of floating-point operations that must be performed by the frontal solver to (partially) factorize the matrix block (assuming zeros within the frontal matrix are not exploited).

We see that, in general, applying the MSRO algorithm directly to the matrix block $\begin{pmatrix} A_{ll} & C_l \end{pmatrix}$ (approach MSRO(1)) substantially reduces the initial mean frontal matrix size (that is, the frontsize of the block resulting from the use of

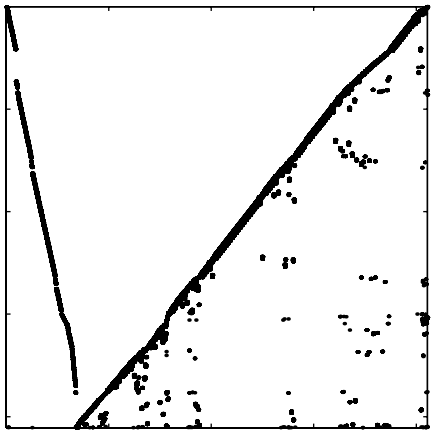
Table 4.2: The mean frontal matrix size (f_{avg}) for different variants of the MSRO algorithm. The smallest values are highlighted. $*$ indicates f_{avg} not improved by reordering.

Identifier	Block	Interface variables	Initial	MSRO(1)	MSRO(2)	MSRO(3)	MSRO(3,W_3)	MSRO(3,W_3) (default W_3)
4cols	1	78	68345	5498	3396	3104	**2840**	2876
	2	78	83183	3960	2464	2238	**2154**	**2194**
	3	30	50435	2238	1210	1206	**1064**	1203
	4	30	145145	955	970	970	**909**	919
10cols	1	77	232908	5874	4493	3993	**3802**	3836
	2	76	450245	4227	3608	**3441**	**3441**	3447
	3	61	100312	3372	**2455**	2825	2762	2781
	4	62	258601	1794	1792	1802	**1796**	**1796**
bayer01	1	120	14572	9342	4412	3971	**3362**	3713
	2	65	42068	9005	6834	6884	**6545**	7135
	3	69	29154	6034	**3663**	4443	4121	4453
	4	125	34226	7100	**4133**	4909	4755	4755
bayer03	1	65	4462	3313	3420	**3352**	**3352**	3474
	2	35	8542	4119	3286	3703	**2269**	2357
	3	32	6827	3241	1123	1670	**957**	1046
	4	65	10059	3462	3317	3431	**2978**	**2978**
bayer04	1	81	49818	8853	10660	8291	**6885**	7590
	2	192	65261	19260	14024	13193	**13036**	**13036**
	3	144	22573	13810	7356	9982	**6174**	7278
	4	199	33127	16930	15106	7404	**6872**	**6872**
bayer09	1	63	9167	3179	2195	1858	**1731**	**1731**
	2	64	4156	2925	2855	2821	**2813**	**2813**
	3	65	7136	3099	2740	2697	**1904**	2506
	4	81	5194	$*$	**4710**	5900	5235	5480
ethylene-1	1	50	134203	48928	14363	48818	**4648**	6961
	2	60	4183	1683	**873**	1190	1183	1183
	3	56	14489	5733	5072	5723	**4207**	8117
	4	48	2724	$*$	$*$	$*$	**2147**	5768
ethylene-2	1	30	3843	$*$	**421**	$*$	752	$*$
	2	83	21480	9566	**2719**	12999	5220	7170
	3	52	85351	48593	$*$	48450	**7281**	15823
	4	45	18727	$*$	$*$	$*$	**6000**	14665
icomp	1	98	9772	3265	3321	3258	**3116**	3141
	2	86	22383	5006	5040	**5006**	**5006**	5119
	3	65	47070	**1623**	1709	1626	1626	1698
	4	51	17673	3178	2090	2066	**1850**	**1850**
lhr07c	1	126	34289	22963	12160	10585	**10047**	**10047**
	2	218	31283	26692	21055	19336	**18530**	18730
	3	156	35419	30464	19492	19978	**18786**	19600
	4	94	17598	17578	7204	5051	**4541**	4989
lhr14c	1	164	59792	40184	18385	35239	**7745**	29576
	2	212	79449	20453	21118	12666	**12658**	**12658**
	3	142	32068	27085	14224	16672	**13736**	15868
	4	92	48389	22145	14349	13933	**11641**	11728
lhr34c	1	221	69322	29286	23579	28954	**21442**	27854
	2	161	67257	54472	30665	29794	**22694**	23192
	3	253	128511	59575	46357	55640	**44682**	45992
	4	194	52182	44410	28689	16810	**15289**	**15289**
lhr71c	1	292	97312	63359	**47150**	56007	50442	53285
	2	188	172738	57443	59848	40513	**35665**	38176
	3	384	104890	$*$	77739	85713	**58202**	66548
	4	286	152902	118920	51980	42125	**40481**	44441

the MONET algorithm). However, in the majority of cases, it is better to apply the MSRO algorithm only to A_{ll} (approach MSRO(2)). This is because, for our test examples, the MONET code is successful in producing partitionings of the original problem that have only a small number of interface variables compared to the order of the problem. If we modify the MSRO algorithm to recognise interface variables (MSRO(3)) then in about half the cases, we are able to reduce f_{avg} further. By introducing a third term into the priority function (MSRO(3,W_3)) we obtain the smallest mean frontal matrix sizes for most of the test examples.

We have performed experiments using a range of values for W_3. With $(W_1, W_2) = (2, 1)$ and $(32, 1)$, we set W_3 in turn to $0.0, 0.167, 0.2, 0.5, 1.0$, and 1.5. The results given in column 8 of Table 4.2 are the best results obtained using this range of values for W_3. Our experience is that some problems are sensitive to the choice of W_3 while for others, the precise choice for W_3 is less important. Based on our experiments, we have chosen the default value for W_3 to be 0.2; this choice was used to give the results in column 9. Although this value often gives mean frontal matrix sizes that are close to the best we have computed, we see that for some problems, including `ethylene-2`, it may be worthwhile to try different choices for W_3 before selecting the final ordering to present to the frontal solver.

Figure 4.1: Problem `bayer04` with original ordering



In Figures 4.1 to 4.3 we show the sparsity pattern for problem `bayer04` using the initial supplied ordering, after it has been reordered to bordered block diagonal form, and after the MSRO(3,W_3) algorithm has been used to reorder the rows within the blocks. This clearly illustrates the effectiveness of MSRO(3,W_3) in reducing the frontsize within the diagonal blocks.

In Table 4.3 we present timings for factorizing the matrix A using a new parallel row-by-row frontal solver that is currently under development. In

Figure 4.2: Problem `bayer04` in bordered block diagonal form

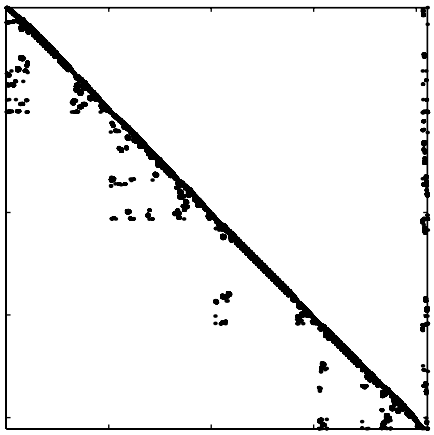
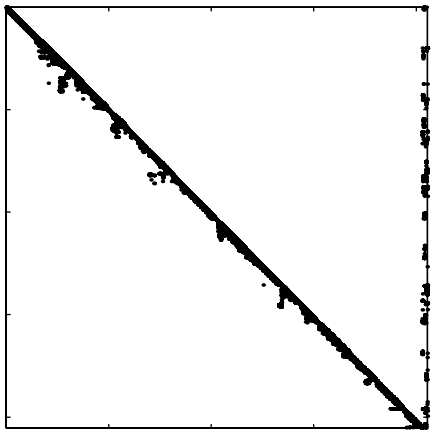


Figure 4.3: Problem `bayer04` after reordering with MSRO(3,W_3)



each case, the MONET code was again used to partition the matrix into 4 blocks and runs were performed using 2 and 4 processors. Results are given both for no ordering of the rows within the blocks and for reordering using the MSRO(3,W_3) approach. Wallclock times are given in seconds and are the minimum times over 10 runs. We see that, for the large test problems of order greater than 10,000, reordering the rows can lead to substantial savings in the time required for the matrix factorization (notably, problems `4cols`, `10cols`, and `icomp`). However, the savings are not always as large as the reductions

in the frontal matrix size might lead us to expect. This is because the frontal solver is able to take advantage of some zeros in the frontal matrix and does not, in fact, treat the frontal matrix as completely dense. This can lessen the effect of a poor ordering. For the smaller problems, although the MSRO(3,W_3) algorithm generally reduces the frontal matrix size considerably (Table 4.2), these reductions do not lead to large savings in the factorization times. There are a number of reasons, we believe, for this. Again, there is the exploitation by the frontal solver of zeros in the front. Secondly, for our test problems, the number of interface variables resulting from the MONET ordering appears to be largely independent of the problem size and so, for the smaller problems, the solution of the interface problem (which is independent of whether or not the rows within the blocks are reordered) accounts for a larger proportion of the total solution time. Furthermore, for the small problems, the amount of data movement between processors is high compared with the computation performed by each processor. If the rows are poorly ordered, more operations may be performed on each subdomain than if the rows are well ordered but, in each case, the data movement between processors is the same.

Table 4.3: Factorization timings for a parallel frontal solver with and without ordering of the rows within the blocks.

Identifier	2 processors		4 processors	
	No ordering	MSRO(3,W_3) ordering	No ordering	MSRO(3,W_3) ordering
`4cols`	0.73	0.19	0.48	0.11
`10cols`	3.93	0.47	2.91	0.27
`bayer01`	1.80	1.31	1.08	0.77
`bayer03`	0.17	0.12	0.09	0.08
`bayer04`	0.88	0.58	0.52	0.33
`bayer09`	0.08	0.08	0.05	0.05
`ethylene-1`	0.39	0.27	0.36	0.19
`ethylene-2`	0.36	0.23	0.33	0.16
`icomp`	2.05	1.36	1.16	0.66
`lhr07c`	0.48	0.45	0.33	0.33
`lhr14c`	1.00	1.00	0.58	0.50
`lhr34c`	2.86	2.65	1.69	1.88
`lhr71c`	6.55	6.46	3.88	3.84

When the number of processors is equal to the number of matrix blocks, the factorization time is determined by the slowest block matrix factorization time. It is therefore important that the subdomains are well balanced. The MONET code produces matrix blocks with an (almost) equal number of rows but, as shown in Table 4.2, the average frontal matrix size on each of the blocks can vary substantially. Reordering the rows can help reduce this imbalance. For example, for problem `10cols`, before reordering, f_{avg} for block 2 is more than 4 times that for block 3 while the time for factorizing block 2 is more than 3 times that for block 3. However, after reordering, these 2 blocks have similar average frontal matrix sizes and comparable factorization times.

4.1 A note on ordering for the unifrontal code

In Section 3.2, we introduced a third term into the priority function to give (3.6). This third term can also be included if we are ordering the rows of the whole matrix A for use with a frontal solver (that is, for a unifrontal code). In this case, **all** the variables are non-interface variables and so $nold_i$ reduces to the number variables in row i that are already in the front. In Table 4.4 we present the mean frontal matrix size for the original matrix ordering and for the MSRO algorithm with and without this third term (again, we use $W_3 = 0.2$). We see that, in many of our test cases, we are able to reduce the mean frontal

Table 4.4: The mean frontal matrix size ($f_{avg} * 10^2$) for original ordering and for the MSRO reordering algorithm with and without the third term.

Identifier	Original	$W_3 = 0$	$W_3 = 0.2$
`4cols`	2218	30	21
`10cols`	7091	39	28
`bayer01`	1183	180	194
`bayer03`	200	26	24
`bayer04`	1911	342	161
`bayer09`	249	17	16
`ethylene-1`	1452	3910	566
`ethylene-2`	451	2818	494
`icomp`	1217	73	81
`lhr07c`	521	62	49
`lhr14c`	1076	153	166
`lhr34c`	1499	283	232
`lhr71c`	1548	835	449

matrix size through the inclusion of the additional term. In a number of cases, the improvements are significant, notably for problems `bayer04`, `ethylene-1`, and `lhr71c`.

5 Concluding remarks

In this paper, we have taken an unsymmetric matrix that has been ordered to border block diagonal form and extended the MSRO row ordering algorithm to order the rows within each of the blocks. The resulting orderings have been used with a parallel frontal solver. We have shown that it is possible to substantially reduce the size of the frontal matrix on the blocks, and this can in turn result in significant speed-ups in the frontal solver. The best orderings were obtained by explicitly taking into account the columns that are not fully summed within the block. A Fortran code implementing the MSRO(3,W_3) algorithm has been developed and will be included in the next release of the Harwell Subroutine Library (HSL, 2000); for further details, please contact the author. We are currently developing our parallel row-by-row frontal solver into a general-purpose sparse code for the Harwell Subroutine Library. This code is being written in Fortran 90 and, for portability, employs MPI for message passing. We are investigating the performance of this new code and comparing

it with other sparse solvers that are also designed for unsymmetric systems.

An additional outcome of this study is that, through the use of a third term in the priority function, we have improved the MSRO algorithm for ordering all the rows of a matrix.

6 Acknowledgements

I would like to thank my colleague Yifan Hu for supplying a copy of his MONET code, which was used to order the matrices in singly bordered block diagonal form. I am also grateful to Mark Stadtherr of the University of Notre Dame for providing me with some of the test problems. My thanks to Iain Duff for reading and commenting on a draft of this paper.

References

K.V. Camarda and M.A. Stadtherr. Frontal solvers for process engineering: local row ordering strategies. *Computers in Chemical Engineering*, **22**, 333–341, 1998.

T. Davis. University of Florida Sparse Matrix Collection. *NA Digest*, **97**(23), 1997.

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

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

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

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

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

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

HSL. A collection of Fortran codes for large scale scientific computation, 2000. Full details from http://www.numerical.rl.ac.uk/hsl.

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

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

G. Kumfert and A. Pothen. Two improved algorithms for envelope and wavefront reduction. *BIT*, **18**, 559–590, 1997.

J.U. Mallya, S.E. Zitney, and M.A. Stadtherr. Parallel frontal solver for large scale process simulation and optimization. *AIChE J.*, **43**, 1032–1040, 1997*a*.

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

B.H. Mayoh. A graph technique for inverting certain matrices. *Mathematics of Computation*, **19**, 644–646, 1965.

J.K. Reid and J.A. Scott. Reversing the row order for the row-by-row frontal method. Technical Report RAL-TR-1999-037, Rutherford Appleton Laboratory, 1999. To appear in *Numerical Linear Algebra with Applications*.

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

J.A. Scott. The design of a parallel frontal solver. Technical Report RAL-TR-99-075, Rutherford Appleton Laboratory, 1999*a*.

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

J.A. Scott. Row ordering for frontal solvers in chemical process engineering. Technical Report RAL-TR-99-035, Rutherford Appleton Laboratory, 1999*c*. To appear in *Computers in Chemical Engineering*.

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