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

Jennifer A. Scott *

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

Received 27 July 2000; received in revised form 14 November 2000; accepted 14 November 2000

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 minimize 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. © 2001 Elsevier Science Ltd. All rights reserved.

Keywords: Row ordering; Parallel frontal method; Unsymmetric sparse matrices

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 generalizing the method to the multiple front method (Duff & Scott, 1994a,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 and Stadtherr (1997a), Mallya, Zitney, Choudhary and Stadtherr (1997b). The package MP42 of Scott (2001a) 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. (1997a) realized 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 organized as follows. In Section 2, we recall key features of the row-by-row frontal method and its generalization 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 generalization to the multiple front method.

2.1. Frontal method

Consider the linear system of equations

$$Ax = b,$$

(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 motivation 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 Eq. (2.1) can then be solved by a simple forward substitution,

$$Ly = Pb,$$

followed by a backsubstitution

$$Uz = y.$$
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 front sizes 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 (2000). The most successful methods currently available are the MSRO methods of Scott (1999b), 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

\[
\begin{bmatrix}
A_{11} & C_1 \\
A_{22} & C_2 \\
\vdots & \vdots \\
A_{NN} & C_N
\end{bmatrix},
\]

(2.2)

where the rectangular diagonal blocks \(A_{ii}\) are \(m_i \times n_i\) matrices with \(m_i \geq n_i\), and the border blocks \(C_i\) are \(m_i \times k_i\). This ordering needs to be chosen so that \(k_i \ll n_i\). A partial LU decomposition is performed on each of the matrices

\[
(A_{ii} \ C_i)
\]

(2.3)

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

\[
F = \sum_{i=1}^{N} F_i
\]

(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 with 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 & Stadtherr, 1998; Hu et al., 2000). In this study, we use the MONET code of Hu et al. (2000) to generate the block form Eq. (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 (1999b) 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 \(\ast\) 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 relabeling 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 (1999b) are the pseudodiameter of $G_R$ (also used by Sloan, 1986) and the spectral ordering for $G_R$ (see Kumfert & 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 front sizes 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

$$P_i = -W_1 \text{regain}_i - W_2 g_i.$$  \hfill (3.1)

Here $W_1$ and $W_2$ are positive weights; $g_i$ is the (positive) global priority for row $i$; and $\text{regain}_i$ is the sum of the increases to the row and column front sizes 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 front size to either increase by one, to remain the same, or to decrease. The row front size 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 front size 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 front size decreases. Hence, if $s_i$ is the number of column indices that appear for the first time when row $i$ is assembled, and $\text{newc}_i$ is the number of column indices that enter the front for the first time, then

$$\text{regain}_i = 1 + \text{newc}_i - 2s_i.$$  \hfill (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 maximize $P_r$. The eligible rows are the active rows plus their neighbors. 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 (1999b) 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 (2001) have shown that the maximum and mean column front sizes are invariant if a given row order is reversed. However, the maximum and mean row front sizes 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 generalize the method to order the rows in the rectangular blocks of the block diagonal matrix given by Eq. (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 (Eq. (2.2)).
2. Reorder the rows within each block matrix $(A_{ii} \ C_i)$.

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 $(A_{ii} \ C_i)$. However, the MSRO algorithm is designed to reorder 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 $(A_{ii} \ C_i)$ this assumption will not be valid for columns belonging to the border $C_i$. 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 front size will increase beyond that predicted. If the interface variables are brought into the front early in the ordering because their rows have a high priority $P_r$, 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 $A_H$. If we do this, then when the frontal method is applied to $(A_H C_I)$, 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 $k_I$ is the number of columns of $C_I$ 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 $(A_H C_I)$, the columns within the border $C_I$ 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 $i$th row in $(A_H C_I)$ is given by

$$P_i = -W_1(1 + l_{ni}) - W_2g_i,$$

(3.3)

where $l_{ni}$ is the number of entries in row $i$. As each row $i$ is assembled, the priority of each of its neighbors $j$ 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,
   $$P_j \leftarrow P_j + W_1.$$

(3.4)

2. For each column index $k$ that appears for the last time in row $i$ with $a_{ik}$ and $a_{jk}$ nonzero,
   $$P_j \leftarrow P_j + 2W_1.$$

(3.5)

If $A_H$ has $m_I$ rows and we flag each column with a second nonzero entry in $C_I$ as appearing for the final time in row $m_I + 1$, no updates of the form (Eq. (3.5)) will be made for these columns. Thus each row $j$ that has nonzero entries in $C_I$ 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 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 (Eq. (3.1)) for row $i$ with the following priority function

$$P_i = -W_1r_{regain_i} - W_2g_i + W_3n_{old_i},$$

(3.6)

where $W_1$ is another (positive) weight and $n_{old_i}$ is the number of non-interface variables in row $i$ that have already been introduced into the front. Initially, $n_{old_i} = 0$. As rows are assembled, $n_{old_i}$ 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, $W_3$) $= MSRO(3)$).

4. Numerical results

The test problems used in our numerical experiments are listed in Table 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. The bayer problems are from the German chemical industry. The ethylene problems arise from the

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Order</th>
<th>Number of entries</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4cols</td>
<td>11770</td>
<td>43 668</td>
<td>9-Component plant with 4 interlinked distillation columns</td>
</tr>
<tr>
<td>10cols</td>
<td>29496</td>
<td>109 588</td>
<td>9-Component plant with 10 interlinked distillation columns</td>
</tr>
<tr>
<td>bayer01†</td>
<td>57735</td>
<td>277 774</td>
<td>German chemical industry process simulation</td>
</tr>
<tr>
<td>bayer03†</td>
<td>6747</td>
<td>56 196</td>
<td>German chemical industry process simulation</td>
</tr>
<tr>
<td>bayer04†</td>
<td>20545</td>
<td>159 082</td>
<td>German chemical industry process simulation</td>
</tr>
<tr>
<td>bayer09†</td>
<td>3083</td>
<td>21 216</td>
<td>German chemical industry process simulation</td>
</tr>
<tr>
<td>ethylene-1</td>
<td>10673</td>
<td>80 904</td>
<td>Optimization of an ethylene plant</td>
</tr>
<tr>
<td>ethylene-2</td>
<td>10353</td>
<td>78 004</td>
<td>Optimization of an ethylene plant</td>
</tr>
<tr>
<td>icomp</td>
<td>75724</td>
<td>338 711</td>
<td>Dynamic plant simulation problem</td>
</tr>
<tr>
<td>1hr07c†</td>
<td>7337</td>
<td>156 508</td>
<td>Light hydrocarbon recovery process</td>
</tr>
<tr>
<td>1hr14c†</td>
<td>14270</td>
<td>307 858</td>
<td>Light hydrocarbon recovery process</td>
</tr>
<tr>
<td>1hr34c†</td>
<td>35152</td>
<td>764 014</td>
<td>Light hydrocarbon recovery process</td>
</tr>
<tr>
<td>1hr71c†</td>
<td>70304</td>
<td>1528 092</td>
<td>Light hydrocarbon recovery process</td>
</tr>
</tbody>
</table>

* † Indicates problem taken from University of Florida Sparse Matrix Collection.
optimization of an ethylene plant. Each problem involves a flowsheet of 43 units, including five distillation columns. The problems differ in the number of stages in the distillation columns. The 1hr problems are derived using a chemical process simulator and are based on a light hydrocarbon recovery process. The larger problems involve more chemical species and/or more chemical processing units.

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 (Eq. (2.2)).

In Table 2, we compare the performance of the different variants of the MSRO algorithm introduced in the previous section. In each case, we have chosen the number of blocks to \( N = 4 \). The comparison is based on the mean frontal matrix size \( f_{\text{avg}} \) within each block, which is defined to be

\[
f_{\text{avg}} = \frac{1}{n_i} \sum_{i=1}^{n_i} (f_{\text{row},i} \cdot f_{\text{col},i})
\]

(4.1)

where \( f_{\text{row},i} \) and \( f_{\text{col},i} \) denote the row and column front sizes before the \( i \)th elimination in the matrix block (\( n_i \) is the number of columns in the block). Note that \( f_{\text{avg}} n_i \) 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 \( (A_{ii}, C_i) \) (approach MSRO(1)) substantially reduces the initial mean frontal matrix size (that is, the frontsize of the block resulting from the use of the MONET algorithm). However, in the majority of cases, it is better to apply the MSRO algorithm only to \( A_{ii} \) (approach MSRO(2)). This is because, for our test examples from chemical process engineering, the MONET code is successful in producing partitionings of the original problem that have only a small number of interface variables compared with the order of the problem. If we modify the MSRO algorithm to recognize interface variables (MSRO(3)) then in about half the cases, we are able to reduce \( f_{\text{avg}} \) 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 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.

In Figs. 1–3 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 3 we present timings for factorizing the matrix \( A \) using a new parallel row-by-row frontal solver MP43; see Scott (2001b) for details. In each case,
Table 3
Factorization timings for a parallel frontal solver with and without ordering of the rows within the blocks

<table>
<thead>
<tr>
<th>Identifier</th>
<th>2 processors</th>
<th>4 processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No ordering</td>
<td>MSRO(3, W₃) ordering</td>
</tr>
<tr>
<td>4cols</td>
<td>0.73</td>
<td>0.19</td>
</tr>
<tr>
<td>10cols</td>
<td>3.93</td>
<td>0.47</td>
</tr>
<tr>
<td>bayer01</td>
<td>1.80</td>
<td>1.31</td>
</tr>
<tr>
<td>bayer03</td>
<td>0.17</td>
<td>0.12</td>
</tr>
<tr>
<td>bayer04</td>
<td>0.88</td>
<td>0.58</td>
</tr>
<tr>
<td>bayer09</td>
<td>0.08</td>
<td>0.08</td>
</tr>
<tr>
<td>ethylene-1</td>
<td>0.39</td>
<td>0.27</td>
</tr>
<tr>
<td>ethylene-2</td>
<td>0.36</td>
<td>0.23</td>
</tr>
<tr>
<td>icomp</td>
<td>2.05</td>
<td>1.36</td>
</tr>
<tr>
<td>1hr07c</td>
<td>0.48</td>
<td>0.45</td>
</tr>
<tr>
<td>1hr14c</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>1hr34c</td>
<td>2.86</td>
<td>2.65</td>
</tr>
<tr>
<td>1hr71c</td>
<td>6.55</td>
<td>6.46</td>
</tr>
</tbody>
</table>

In each case, the MONET code was again used to partition the matrix into four blocks and runs were performed using two and four processors. Results are given both for no ordering of the rows within the blocks and for reordering using the MSRO(3, W₃) approach. Wallclock times are given in seconds and are the minimum times over ten 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, W₃) algorithm generally reduces the frontal matrix size considerably (Table 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.

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 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 four times that for block 3 while the time for factorizing block 2 is more than three times that for block 3. However, after reordering, these two blocks have similar average frontal matrix sizes and comparable factorization times.

Table 4
The mean frontal matrix size (\( f_{avg} \times 10^2 \)) for original ordering and for the MSRO reordering algorithm with and without the third term

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Original</th>
<th>( W₃ = 0 )</th>
<th>( W₃ = 0.2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>4cols</td>
<td>2218</td>
<td>30</td>
<td>21</td>
</tr>
<tr>
<td>10cols</td>
<td>7091</td>
<td>39</td>
<td>28</td>
</tr>
<tr>
<td>bayer01</td>
<td>1183</td>
<td>180</td>
<td>194</td>
</tr>
<tr>
<td>bayer03</td>
<td>200</td>
<td>26</td>
<td>24</td>
</tr>
<tr>
<td>bayer04</td>
<td>1911</td>
<td>342</td>
<td>161</td>
</tr>
<tr>
<td>bayer09</td>
<td>249</td>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>ethylene-1</td>
<td>1452</td>
<td>3910</td>
<td>566</td>
</tr>
<tr>
<td>ethylene-2</td>
<td>451</td>
<td>2818</td>
<td>494</td>
</tr>
<tr>
<td>icomp</td>
<td>1217</td>
<td>73</td>
<td>81</td>
</tr>
<tr>
<td>1hr07c</td>
<td>521</td>
<td>62</td>
<td>49</td>
</tr>
<tr>
<td>1hr14c</td>
<td>1076</td>
<td>153</td>
<td>166</td>
</tr>
<tr>
<td>1hr34c</td>
<td>1499</td>
<td>283</td>
<td>232</td>
</tr>
<tr>
<td>1hr71c</td>
<td>1548</td>
<td>835</td>
<td>449</td>
</tr>
</tbody>
</table>
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 Eq. (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 $n_{old_i}$ reduces to the number variables in row $i$ that are already in the front. In Table 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 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 1hr7ic.

4.2. A comparison of the unifrontal and multiple front codes

In Table 5, we compare the factorization times for the unifrontal code MA42 of Duff & Scott (1996) run on a single processor with those for the multiple front code run on two, four, and eight processors. In each case, the number of blocks $N$ was chosen to be equal to the number of processors. For both the unifrontal and multiple front code, the rows of the matrices were preordered using the MSRO algorithm with $W_3 = 0.2$. We see that it is clearly advantageous to use the multiple front approach with at least two processors. For the smaller problems (fewer than about 10 000 variables), we do not get an improvement in performance by using more than two processors. This is because, as the number of blocks increases, there is less work to be done on each block and the factorization time becomes dominated by the time required to solve the interface problem. However, for the largest problems, notably bayer01 and icomp we get speedups of about 2 as we double the number processors to four and to eight (see Fig. 4). We tried running these problems with 16 blocks on 16 processors but the times increased. We conclude that the multiple front approach is suitable for use on a small number of processors and that more processors should be used for larger problems.

5. Concluding remarks

In this paper, we have taken an unsymmetric matrix that has been ordered to bordered 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′) algorithm has been developed and is available as part of the latest release of the HSL subroutine library (HSL, 2000); for further details, please contact the author. Our unifrontal and parallel row-by-row frontal solvers are also included in HSL 2000 as routines MA42 and MP43, respectively. MP43 is written in Fortran 90 and, for portability, employs MPI for message passing. We are currently 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.

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


