RAL-TR-97-046

A combined unifrontal/multifrontal method for unsymmetric sparse matrices [1]

Timothy A. Davis[2] and Iain S. Duff[3]

ABSTRACT

We discuss the organization of frontal matrices in multifrontal methods for the solution of large sparse sets of unsymmetric linear equations. In the multifrontal method, work on a frontal matrix can be suspended, the frontal matrix can be stored for later reuse, and a new frontal matrix can be generated. There are thus several frontal matrices stored during the factorization and one or more or these are assembled (summed) when creating a new frontal matrix. Although this means that arbitrary sparsity patterns can be handled efficiently, extra work is required to sum the frontal matrices together and can be costly because indirect addressing is required. The (uni-)frontal method avoids this extra work by factorizing the matrix with a single frontal matrix. Rows and columns are added to the frontal matrix, and pivot rows and columns are removed. Data movement is simpler, but higher fill-in can result if the matrix cannot be permuted into a variable-band form with small profile. We consider a combined unifrontal/multifrontal algorithm to enable general fill-in reduction orderings to be applied without the data movement of previous multifrontal approaches. We discuss this technique in the context of a code designed for the solution of sparse systems with unsymmetric pattern.

Categories and Subject Descriptors:
G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra - *linear systems (direct methods), sparse and very large systems;*
G.4 [**Mathematics of Computing**]: Mathematical Software - *algorithm analysis, efficiency.*
General Terms: Algorithms, Experimentation, Performance.
Additional Key Words and Phrases: sparse unsymmetric matrices, linear equations, multifrontal methods, frontal methods.

[1] Current reports available by anonymous ftp from matisa.cc.rl.ac.uk (internet 130.246.8.22) in the directory "pub/reports". This report is in file daduRAL97046.ps.gz. Also appeared as Technical Report TR-97-016, Computer and Information Science and Engineering Department, University of Florida.
[2] Computer and Information Science and Engineering Department, University of Florida, Gainesville, Florida, USA. (352) 392-1481, email: davis@cise.ufl.edu. The work of this author was supported by National Science Foundation grant DMS-9504974.
[3] isd@rl.ac.uk.

Department for Computation and Information
Atlas Centre
Rutherford Appleton Laboratory
Oxon OX11 0QX

September 20, 1997.

Contents

1 Introduction

We consider the direct solution of sets of linear equations $\mathbf{Ax} = \mathbf{b}$, where the coefficient matrix $\mathbf{A}$ is sparse, unsymmetric, and has a general nonzero pattern. A permutation of the matrix $\mathbf{A}$ is factorized into its LU factors, $\mathbf{PAQ} = \mathbf{LU}$, where $\mathbf{P}$ and $\mathbf{Q}$ are permutation matrices chosen to preserve sparsity and maintain numerical accuracy. Many recent algorithms and software for the direct solution of sparse systems are based on a multifrontal approach (Amestoy and Duff 1989, Davis and Duff 1997, Duff and Reid 1983, Liu 1992). In this paper, we will examine a new frontal matrix strategy to be used within a multifrontal approach. We use the term "unifrontal" for what is usually called the "frontal" method so that the term "frontal" can be used generically for both unifrontal and multifrontal methods.

Unifrontal and multifrontal methods compute the LU factors of $\mathbf{A}$ by using data structures that permit regular access of memory and the use of dense matrix kernels (such as the BLAS) in their innermost loops. On supercomputers and high-performance workstations, this can lead to a significant increase in performance over methods that have irregular memory access and that do not use dense matrix kernels.

We discuss unifrontal methods in Section 2. We summarize the multifrontal method in Section 3, and in particular our earlier work on an unsymmetric-pattern multifrontal method. We refer to this prior method as `UMFPACK V1.1` (Davis 1995, Davis and Duff 1991, Davis and Duff 1997). The combination of unifrontal and multifrontal methods is discussed in Section 4. The combined algorithm is based on `UMFPACK V1.1` and the new frontal matrix strategy discussed here. This combined algorithm is available in Release 12 of the Harwell Subroutine Library (HSL 1996) as the package `MA38`. In the remainder of this paper, we refer to the combined unifrontal/multifrontal algorithm as `MA38`. We describe our sparse matrix test set, and how we selected it, in Section 5. In Section 6, we consider the influence of a key parameter that is present in both the `UMFPACK V1.1` and the `MA38` versions of our unsymmetric-pattern multifrontal method. The performance of `MA38` is discussed in Section 7, before a few concluding remarks and information on the availability of our codes are given in Section 8.

2 Unifrontal methods

In a unifrontal scheme (Duff 1984*a*, Irons 1970, Zitney, Mallya, Davis and Stadtherr 1996, Zitney and Stadtherr 1993), the factorization proceeds as a sequence of partial factorizations and eliminations on dense submatrices, called frontal matrices. Although unifrontal methods were originally designed for the solution of finite-element problems (Irons 1970), they can be used on assembled systems (Duff 1984*a*) and it is this version that we study in this paper. For assembled systems, the frontal matrices can be written as

$$\begin{pmatrix} \mathbf{F}_{11} & \mathbf{F}_{12} \\ \mathbf{F}_{21} & \mathbf{F}_{22} \end{pmatrix}, \tag{2.1}$$

where all rows are fully summed (that is, there are no further contributions to come to the rows in (2.1)) and the first block column is fully summed. This means that pivots can be chosen from anywhere in the first block column and, within these columns, numerical pivoting with arbitrary row interchanges can be accommodated since all rows in the frontal matrix are fully summed.

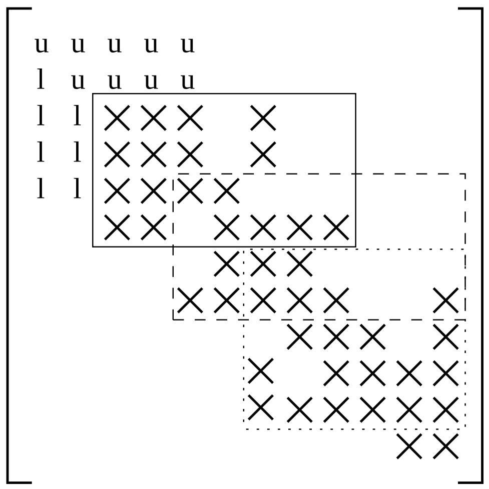


Figure 2.1: Frontal method example

We assume, without loss of generality, that the pivots that have been chosen are in the square matrix $\mathbf{F}_{11}$ of (2.1). $\mathbf{F}_{11}$ is factorized, the Gaussian elimination multipliers overwrite $\mathbf{F}_{21}$ and the Schur complement

$$\mathbf{F}_{22} - \mathbf{F}_{21}\mathbf{F}_{11}^{-1}\mathbf{F}_{12}, \tag{2.2}$$

is formed using dense matrix kernels. The submatrix consisting of the rows and columns of the frontal matrix from which pivots have not yet been selected is called the contribution block. In the case above, this is the same as the Schur complement matrix (2.2).

At the next stage, further rows from the original matrix are assembled with the Schur complement to form another frontal matrix. The frontal matrix is extended in size, if necessary, to accommodate the incoming rows. The overhead is low (compared to a multifrontal method) since each row is assembled only once and there is never any assembly of two (or more) frontal matrices. The entire sequence of frontal matrices is held in the same working array. Data movement is limited to assembling rows of the original matrix into the frontal matrix, and storing rows and columns as they become pivotal. There is never any need to move or assemble the Schur complement into another working array. One important advantage of the method is that only this single working array need reside in memory. Rows of $\mathbf{A}$ can be read sequentially from disk into the frontal matrix. Entries in $\mathbf{L}$ and $\mathbf{U}$ can be written sequentially to disk in the order they are computed. A detailed description of frontal methods for assembled problems is given by Zitney (1992).

An example is shown in Figure 2.1, where two pivot steps have already been performed on a 5-by-7 frontal matrix (computing the first two rows of $\mathbf{U}$ and columns of $\mathbf{L}$, respectively), the columns are in pivotal order. Entries in $\mathbf{L}$ and $\mathbf{U}$ are shown in lower case. Row 6 has just been assembled into the current 4-by-7 frontal matrix (shown as a solid box). Columns 3 and 4 are now fully summed and can be eliminated. After this step, rows 7 and 8 must both be assembled before columns 5 and 6 can be eliminated (the dashed box, a 4-by-6 frontal matrix containing rows 5 through 8 and columns 5, 6, 7, 8, 9, and 12). The frontal matrix is, of course, stored without the zero columns, columns 6 and 7 in the dashed box. The dotted box shows the state of the frontal matrix when the next four pivots can be eliminated. To factorize the 12-by-12 sparse matrix in Figure 2.1, a (dense) working array of size 5-by-7 is sufficient to hold all frontal matrices.

The unifrontal method works well for matrices with small profile, where the profile of a matrix is a measure of how close the nonzero entries are to the diagonal and is given by the expression:

$$\sum_{i=1}^{n} \{ \max_{a_{ij} \neq 0} (i-j) + \max_{a_{ji} \neq 0} (i-j) \},$$

where it is assumed the diagonal is nonzero so all terms in the summation are non-negative. For matrices that are symmetric or nearly so, the unifrontal method is typically preceded by an ordering method to reduce the profile such as reverse Cuthill-McKee (RCM) (Chan and George 1980, Cuthill and McKee 1969, Liu and Sherman 1976). This is typically faster than the sparsity-preserving orderings commonly used by a multifrontal method (such as nested dissection (George and Liu 1981) and minimum degree (Amestoy, Davis and Duff 1996, George and Liu 1989)). However, for matrices with large profile, the frontal matrix can be large, and an unacceptable amount of fill-in can occur. In particular, we lack effective profile reduction strategies for matrices whose pattern is very unsymmetric.

The unifrontal scheme can easily accommodate numerical pivoting. Because all rows in (2.1) are fully summed and regular partial pivoting can be performed, it is always possible to choose pivots from all the fully summed columns (unless the matrix is structurally singular).

3 Multifrontal methods

In a multifrontal method (Amestoy and Duff 1989, Duff and Reid 1983, Duff and Reid 1984, Liu 1992) for a matrix with a symmetric sparsity pattern, it is common to use an ordering such as minimum degree to reduce the fill-in. An example of a code that is primarily designed for matrices with a symmetric pattern is the Harwell Subroutine Library (HSL) code `MA41` (Amestoy and Duff 1989) that will also solve general unsymmetric systems by holding explicit zeros so that the unsymmetric matrix is embedded in one of symmetric structure. Orderings like minimum degree tend to reduce fill-in much more than profile reduction orderings. The ordering is combined with a symbolic analysis to generate an assembly tree, where each node represents the elimination operations on a frontal matrix and each edge represents an assembly operation. When using the tree to drive the numerical factorization, the only requirement is that eliminations at any node cannot complete until those at the child nodes have been completed, giving added flexibility for issues such as exploitation of parallelism. As in the unifrontal scheme, the complete frontal matrix (2.1) cannot normally be factorized but only a few steps of Gaussian elimination are possible, after which the Schur complement $\mathbf{F}_{22}-\mathbf{F}_{21}\mathbf{F}_{11}^{-1}\mathbf{F}_{12}$ (contribution block) needs to be summed (assembled) with other data at the parent node.

In the unsymmetric-pattern multifrontal method (Davis 1995, Davis and Duff 1991, Davis and Duff 1997), the ordering, symbolic analysis, and numerical factorization are performed at the same time. The tree is replaced by a directed acyclic graph (dag). A contribution block may be assembled into more than one subsequent frontal matrix. For example, consider the LU

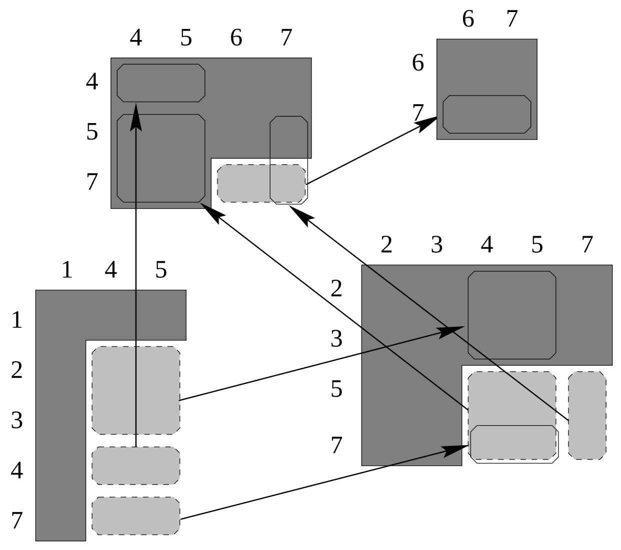


Figure 3.2: Assembly dag for the unsymmetric-pattern multifrontal method

factors of a matrix from Davis and Duff (1997),

$$\mathbf{L}+\mathbf{U}-\mathbf{I} = \begin{bmatrix} u_{11} & 0 & 0 & u_{14} & u_{15} & 0 & 0 \\ l_{21} & u_{22} & u_{23} & u_{24} & u_{25} & 0 & u_{27} \\ l_{31} & l_{32} & u_{33} & u_{34} & u_{35} & 0 & u_{37} \\ l_{41} & 0 & 0 & u_{44} & u_{45} & u_{46} & u_{47} \\ 0 & l_{52} & l_{53} & l_{54} & u_{55} & u_{56} & u_{57} \\ 0 & 0 & 0 & 0 & 0 & u_{66} & u_{67} \\ l_{71} & l_{72} & l_{73} & l_{74} & l_{75} & l_{76} & u_{77} \end{bmatrix}. \qquad (3.3)$$

The unsymmetric-pattern multifrontal factorization of this matrix is depicted in Figure 3.2. The heavily shaded regions are the rows and columns of the factors, the lightly shaded regions are the contribution blocks. The arrows represent the assembly operations from the contribution blocks into solid-lined regions of the same shape in the frontal matrices of the parents. Note that the contribution that the frontal matrix in the lower left of the figure (with the u_{11} pivot) makes to rows 2 and 3 must be assembled into the frontal matrix at the lower right of Figure 3.3, whereas its contribution to row 4 cannot be assembled into the same frontal matrix.

The first pivot within a frontal matrix (called the "seed" pivot by Davis and Duff (1997)) defines its size. This new frontal matrix is held in a larger working array, to allow room for the assembly of subsequent pivot rows and columns. The next pivots can reside either in the fully summed part, or the non-fully summed part. If a potential pivot lies in the non-fully summed part of the frontal matrix then it is necessary to sum its row and column before it can be used as a pivot. This is possible as long as its fully summed row and column can be accommodated within the larger working array, along with the contribution block and all previous pivot rows and columns of the frontal matrix. This is similar to the node amalgamation (Duff and Reid 1983) used in the symmetric-pattern multifrontal method, except that here we determine the amalgamation during numerical factorization, rather than during the symbolic analysis.

We use the term *normal* multifrontal method to denote a multifrontal method where each

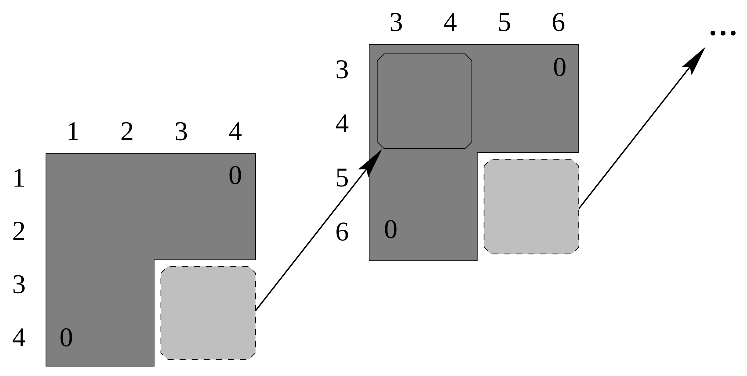


Figure 4.3: First two frontal matrices for a pentadiagonal matrix

frontal matrix is first assembled, then all eliminations are performed, and the contribution block is held (stacked) for assembly at the parent node. Data movement is required both for stacking the contribution block and later assembling it into the parent frontal matrix. Both `MA41` and `UMFPACK V1.1` are examples of a normal multifrontal method. This terminology is used primarily to distinguish this earlier approach from the new multifrontal approach introduced in this paper.

4 Combining the two methods

Let us now consider an approach that combines some of the best features of the two methods: namely, the lower data movement of unifrontal methods, and the lower fill-in of normal multifrontal methods. We give an outline of the approach, followed by some implementation details.

4.1 Outline

Assume we have chosen a pivot and determined a frontal matrix as in a normal multifrontal method. At this stage, a normal multifrontal method will select as many pivots as it can from the fully summed part of the frontal matrix, perform the eliminations corresponding to these pivots, store the pivotal rows and columns, and store the contribution block for later assembly at the parent node of the assembly tree.

In `UMFPACK V1.1`, a potential pivot can be selected from the non-fully summed part, its row and column can be assembled, and be moved to the fully-summed part. Suppose that the rows and columns of such a potential pivot can fit in the working array, but not at the same time as the previous pivotal rows and columns of this frontal matrix. `UMFPACK V1.1` would stop the factorization at this point, store the contribution block for later assembly, and continue with a new frontal matrix. In our combined strategy (`MA38`), we instead perform any arithmetic operations corresponding to the earlier pivots, store their rows and columns (removing them from the frontal matrix), and then assemble the new pivot row and column into the current working array. The contribution block thus need not be stored for later assembly. In this way, we avoid some of the data movement and assemblies of the multifrontal method.

In Figure 4.3, we show the first two frontal matrices for an n-by-n pentadiagonal matrix. Suppose `UMFPACK V1.1` uses a 4-by-4 working array to hold each frontal matrix, and that the pivots are on the diagonal in order. The first two pivot rows and columns fit in the first frontal matrix; this amalgamation causes a_{14} and a_{41} to be treated as "nonzero" entries in the matrix (shown as a zero in the heavily shaded regions). The third pivot row and column do not fit. Once the factorization operations have been completed for the first frontal matrix (using a rank-2 update to the contribution block, a Level 3 BLAS operation) its 2-by-2 contribution block must then be added into the second frontal matrix, in a different working array. A total of $2n$ floating-point values are copied between working arrays, for the $n/2$ contribution blocks.

In MA38, the rank-2 update for the first two pivots is applied, and their pivot rows and columns are stored and are removed from the frontal matrix (just as in the unifrontal method). The 2-by-2 contribution block remains in place, and the third and fourth pivot rows and columns can then be assembled into the working array. The working array now holds what UMFPACK V1.1 would have for its second frontal matrix, but no data has been moved for the 2-by-2 contribution block from the first frontal matrix. As a result, the entire matrix is factorized in a single 4-by-4 working array and the contribution blocks are held within the working array and are never stacked, in contrast with the normal multifrontal method discussed in the previous paragraph.

The combined strategy (MA38) allows the use of a general fill-reducing ordering (as does the normal multifrontal method), rather than a profile-reducing ordering. When it encounters submatrices with good profile (as in the pentadiagonal case just considered), it takes advantage of them with a unifrontal strategy, and thus has less data movement than the normal multifrontal method.

Although the motivation is different, the idea of continuing with a frontal matrix for some steps before moving to another frontal matrix is similar to recent work in implementing frontal schemes within a domain decomposition environment, for example Duff and Scott (1994), where several fronts are used within a unifrontal context. However, in the case of Duff and Scott (1994), the ordering is done a priori and no attempt is made to use a minimum degree ordering.

4.2 Implementation

We now describe how this new frontal matrix strategy is implemented in MA38, which uses a modified version of the minimum degree algorithm, called the approximate minimum degree algorithm (Amestoy et al. 1996, Davis and Duff 1997) that uses an upper bound on the degree counts.

MA38 consists of several major steps, each of which comprises several pivot selection and elimination operations. To start a major step, MA38 selects a few (by default 4) columns from those of minimum upper bound degree (Amestoy et al. 1996, Davis and Duff 1997) and computes their patterns, true degrees, and numerical values. A pivot row is selected on the basis of the upper bound on the row degree from those rows with nonzero entries in the selected columns. The pivot must also satisfy a numerical threshold test (it must be at least as large as u times the largest entry in the pivot column, where u is 0.1 by default). Suppose the pivot row and column degrees are r and c, respectively. A k-by-l working array is allocated (k and l are gc and gr, respectively, where by default $g = 2$). The pivot row and column are fully assembled into the working array and define the active frontal matrix. This active frontal matrix is c-by-r but is stored in the k-by-l working array. The approximate degree update and assembly phase computes the bounds on the degrees of all the rows and columns in this active frontal matrix and assembles previous contribution blocks into the active frontal matrix. A row i in a previous contribution block is assembled into the active frontal matrix if

1. the row index i is in the nonzero pattern of the current pivot column, and

2. the column indices of the remaining entries in the row are all present in the nonzero pattern of the current pivot row.

Columns of previous contribution blocks are assembled in an analogous manner.

The major step then continues with a sequence of minor steps at each of which another pivot is sought from within the current frontal matrix. These minor steps are repeated until the factorization can no longer continue within the current working array, at which point a new major step is started. When a pivot is chosen in a minor step, its rows and columns are fully assembled into the working array and redefine the active frontal matrix. If there are new rows or columns in the frontal matrix, then the approximate degree update and assembly phase is repeated, as described above. Otherwise, the update and assembly phase can be skipped (analogous to mass elimination in a minimum degree ordering algorithm (George and Liu 1989)).

After a pivot is selected (at the start of a major step, or in a minor step) the corresponding row and column of $\mathbf{U}$ and $\mathbf{L}$ are computed, but the updates from this pivot are not necessarily performed immediately. For efficient use of the Level 3 BLAS, it is better to accumulate a few updates (typically up to 16, if possible) and perform them at the same time.

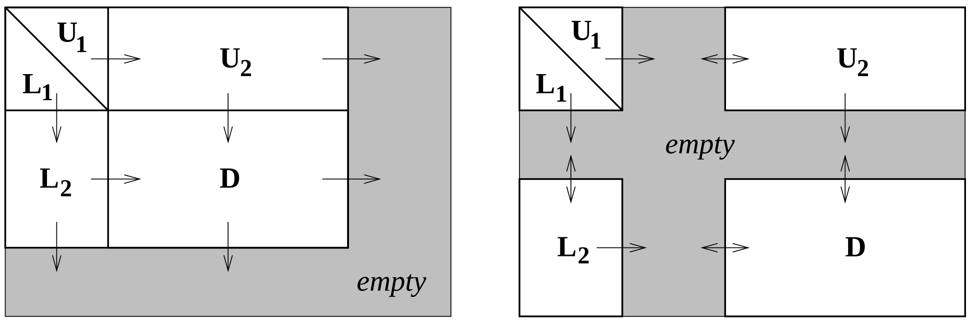
`MA38` and `UMFPACK V1.1` differ in how the minor step is performed, although both use the same amalgamation parameter, g. To find a pivot in this minor step, a single candidate column from the non-fully summed block is first selected, choosing one with least value for the upper bound of the column degree, and any pending updates are applied to this column. The column is assembled into a separate work vector, and a pivot row is selected on the basis of the upper bound on the row degrees and a numerical threshold test. Suppose the candidate pivot row and column have degrees r' and c', respectively. Three conditions apply (where p is the number of pivots currently stored in the active frontal matrix):

1. If $r' > l$ or $c' > k$, then factorization can no longer continue within the active frontal matrix. Any pending updates are applied. The LU factors are stored. The active contribution block is saved for later assembly into a subsequent frontal matrix. The major step is now complete.

2. If $r' \leq l - p$ and $c' \leq k - p$, then the candidate pivot can fit into the active frontal matrix without removing the p pivots already stored there. Set $p \leftarrow p+1$. Factorization continues within the active frontal matrix by starting another minor step.

3. Otherwise, if $l - p < r' \leq l$ or $k - p < c' \leq k$, then the candidate pivot can fit, but only if some of the previous p pivots are shifted out of the current frontal matrix. Any pending updates are applied. The LU factors corresponding to the pivot rows and columns are removed from the front and stored. The active contribution block is left in place. Set $p \leftarrow 1$. Factorization continues within the active frontal matrix by commencing another minor step.

`UMFPACK V1.1` uses the same strategy as `MA38`, described above, except for Case 3. In Case 3, `UMFPACK V1.1` saves the contribution block for later assembly, and terminates the major step. We know of no other multifrontal method that allows the factorization to proceed within the



(a) UMFPACK V1.1 working array *(b) MA38 working array*

Figure 4.4: Data structures for the active frontal matrix

current frontal matrix, as in Case 3. Case 1 does not occur in unifrontal methods, which are given a working array large enough to hold the largest frontal matrix. Cases 2 and 3 do occur in unifrontal methods. Taking simultaneous advantage of all three cases can significantly reduce the memory requirements, data movement, and assembly operations, while still allowing the use of orderings that reduce fill-in.

Figure 4.4 illustrates how the working array can be organized. The matrices $\mathbf{L}_1$, $\mathbf{L}_2$, $\mathbf{U}_1$, and $\mathbf{U}_2$ in the figure are the columns and rows of the LU factors corresponding to the pivots eliminated within this frontal matrix. The matrix $\mathbf{D}$ is the contribution block. The arrows denote how these matrices grow as new pivots are added. When pivots are removed from the working array in Figure 4.4(b), for Case 3 above, the contribution block does not need to be moved, thus accommodating the more dynamic situation without further data movement. For coding reasons, the working array in `MA38` is held in reverse order but the data movement is similar to that shown in the figure[1].

5 Test matrices and computing platform

In the next two sections, we discuss some experiments on the selection of the amalgamation parameter for g for `MA38` and `UMFPACK V1.1` and compare the performance of `MA38` with other sparse matrix codes. In both sections, we use the same set of sparse test matrices.

The sparse matrix collection at the University of Florida (Davis 1997) contains 264 unsymmetric sparse matrices in assembled equation form (as opposed to unassembled finite-element form). This set includes the Harwell-Boeing Sparse Matrix Collection (Duff, Grimes and Lewis 1989), Saad's collection (Saad 1994), Bai's collection (Bai, Day, Demmel and Dongarra 1996), and matrices from other sources (Feldmann, Melville and Long 1996, Vavasis 1993, Zitney 1992, Zitney et al. 1996).

The ten matrices that we will use in the next two sections are listed in Table 5.1. The table lists the name, order, number of entries, structural symmetry, the discipline from which the matrix comes, and additional comments. The structural symmetry is the ratio of the number of *matched* off-diagonal entries to the total number of off-diagonal entries. An entry, a_{ij} ($j \neq i$), is

[1]A similar data organization is employed by the unifrontal HSL code `MA42` (Duff and Scott 1993, Duff and Scott 1996*b*).

Table 5.1: Test matrices. Sources are: [1] Vavasis (1993), [2] Feldman et al. (1996), [3] Duff et al. (1989), [4] Zitney et al. (1996), [5] Bai et al. (1996), and [6] Zitney (1992).

name	n	no. entries	sym.	discipline	comments, and source
AV41092	41,092	1,683,902	0.001	partial diff. eqn.	2D, wild coefficients [1]
TWOTONE	120,750	1,224,224	0.245	circuit simul.	harmonic balance method [2]
PSMIGR_1	3,140	543,162	0.479	demography	US county-to-county migration [3]
LHR71C	70,304	1,528,092	0.002	chemical eng.	light hydrocarbon recovery (corrected) [4]
ONETONE1	36,057	341,088	0.074	circuit simul.	harmonic balance method [2]
ONETONE2	36,057	227,628	0.113	circuit simul.	harmonic balance method [2]
LHR14C	14,270	307,858	0.007	chemical eng.	light hydrocarbon recovery (corrected) [4]
RW5151	5,151	20,199	0.490	probability	random walk Markov chain [5]
ORANI678	2,529	90,158	0.071	economics	Australia [3]
RDIST1	4,134	94,408	0.059	chemical eng.	reactive distillation [6]

matched if a_{ji} is also an entry. This means that a symmetric matrix has a structural symmetry value of 1.0. This measure is one minus the index of asymmetry, originally defined by Duff (1984*b*).

We chose the test matrices using the following strategy. Since we are considering a method for unsymmetric-patterned matrices, we selected all nonsingular matrices from our collection with structural symmetry less than or equal to 0.5 (89 matrices). We ran all six codes discussed in Section 7 for each of the 89 matrices, and selected a matrix for our comparisons in this paper if the fastest factorization time of these six codes was greater than 0.5 seconds. This leaves 29 matrices from which we discarded one matrix too large for most methods on the workstation used for these experiments (from the same application as TWOTONE) and eighteen matrices from sets already represented (two similar to PSMIGR_1, and all but two of the LHR series[2]).

All experiments reported in this paper are on a SUN UltraSparc Model 170, with 256 Mbytes of main memory and a single 167 Mhz processor. Version 4.0 of the SUN Fortran and C compilers were used, with identical optimization parameters (`SuperLU` is written in C, the other codes in Fortran 77). The BLAS that we use are from Daydé and Duff (1996). The double precision matrix-matrix multiply routine, `DGEMM`, which most of the methods use, runs at about 80 Mflops on this workstation. (Note that the single precision version, `SGEMM`, runs at about 145 Mflops.) The theoretical peak performance is 333 Mflops for both double and single precision.

6 Amalgamation parameter

We ran `MA38` and `UMFPACK V1.1` on all matrices in Table 5.1, with g ranging from 1.0 to 4.0 in increments of 0.1. For each matrix, we found the median, with respect to g, of the `MA38` run times. The `MA38` and `UMFPACK V1.1` run times for this matrix were divided by this median

[2]Due to a modeling error, some of the original LHR matrices (Zitney et al. 1996) were extremely ill-conditioned. We used the following procedure to correct the matrices. If $\mathbf{A}$ is the original matrix, the corrected matrix is $\mathbf{A} + \mathbf{P}^T\mathbf{F}\mathbf{Q}^T$, where $\mathbf{P}$ and $\mathbf{Q}$ are permutation matrices such that $\mathbf{PAQ}$ is in block upper triangular form. The matrix $\mathbf{F}$ is diagonal, with $\mathbf{F}_{ii} = 0.001$ if $[\mathbf{PAQ}]_{ii} \geq 0$ and -0.001 otherwise. The corrected matrix has the same nonzero pattern as the original matrix. We selected the value 0.001 by trial and error. We used the smallest value we found for which Matlab could compute at least 2 digits of accuracy in the solution.

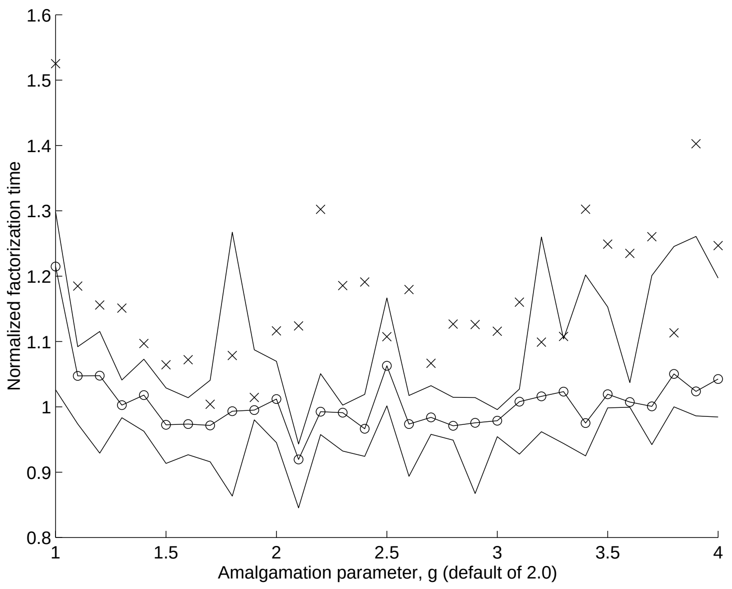


Figure 6.5: Normalized run times for ten matrices, as a function of the amalgamation parameter, g. The median and quartiles for `MA38` are given by the solid lines and the median for `UMFPACK V1.1` by the crosses.

to obtain normalized `MA38` and `UMFPACK V1.1` run times. Then, for each value of g, we found the median (and quartiles) of the normalized `MA38` and `UMFPACK V1.1` run times for all ten matrices. Figure 6.5 shows the median and quartiles for `MA38` and the median for `UMFPACK V1.1` of the normalized times for the ten matrices, plotted as a function of g. The `MA38` normalized factorization times are plotted as solid lines, with the median time data points circled. The mean normalized factorization time for `UMFPACK V1.1` is given by x's. A similar comparison of the normalized memory requirements (excluding fragmentation of the work arrays) of both methods is shown in Figure 6.6.

Although the fill-in and operation count (not shown) are typically lowest when the minimum amount of memory is allocated for each frontal matrix ($g = 1$), the factorization time is often high because of the additional data movement required to assemble the contribution blocks and the fact that the dense matrix kernels are more efficient for the larger frontal matrices that are produced when $g > 1$.

From the results in Figures 6.5 and 6.6, we examine the effect and sensitivity of execution time and memory requirements to changes in the value of g. There are some rapid fluctuations in the graphs, particularly for `UMFPACK V1.1`. This is caused by changes in pivot order with different values of g. A small change in the working array size can affect whether or not a candidate pivot is selected in a minor step. If it is not selected, we start a new frontal matrix, and a new pivot candidate is found in a major step which considers all columns. This is a very different pivot selection strategy from that used in the minor steps. This high sensitivity is not so noticeable with `MA38` because it can continue for longer with the minor step strategy after removing pivot rows and columns from the active frontal matrix. Although the frontal matrices become larger with larger values of g, there are fewer of them, and thus the median

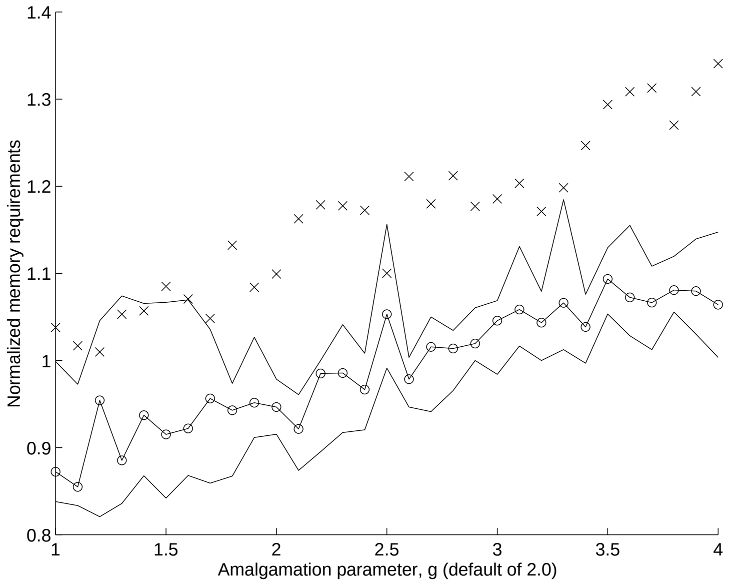


Figure 6.6: Normalized memory requirements for ten matrices, as a function of the amalgamation parameter, g. The median and quartiles for `MA38` are given by the solid lines and the median for `UMFPACK V1.1` by the crosses.

memory requirements increase only slightly as g increases. For these ten matrices, the median factorization time is lowest when g is 2.1. From experiments with a wider range of matrices, we have found that a default value of $g = 2$ is a reasonable tradeoff between memory usage and factorization time.

We also notice, from the results in this section, that exploiting a unifrontal strategy within a multifrontal code (`MA38`) can reduce both factorization time and memory requirements. Comparing `MA38` with its predecessor, `UMFPACK V1.1`, Figures 6.5 and 6.6 show a median improvement of 18% in the run time and 14% in the memory requirements for these ten matrices. Although not shown in the plots, `MA38` is almost four times faster than `UMFPACK V1.1` for one matrix (LHR71C).

7 Performance

In this section, we compare the performance of the combined unsymmetric-pattern unifrontal/multifrontal code `MA38`, with the unsymmetric-pattern multifrontal code `UMFPACK V1.1` (Davis 1995, Davis and Duff 1991, Davis and Duff 1997), the symmetric-pattern multifrontal code `MA41` (Amestoy, Daydé and Duff 1989), the unifrontal code `MA42`, (Duff 1984*a*, Duff and Scott 1993, Duff and Scott 1996*b*), and two sparse matrix factorization codes based on partial pivoting `MA48`, (Duff and Reid 1993, Duff and Reid 1996), and `SuperLU Version 1.0` (Demmel, Eisenstat, Gilbert, Li and Liu 1995, Demmel, Gilbert and Li 1997).

Each code can factorize general unsymmetric matrices and all use dense matrix kernels (Dongarra, Du Croz, Duff and Hammarling 1990) to some extent. Each code was given 1.8

Gbytes of (virtual) memory to factorize the matrices listed in Table 5.1. Each code has a set of input parameters that control its behavior. We used the recommended defaults for most of these, with a few exceptions that we now indicate.

The default numerical threshold u varies with each code. For MA41, it is 0.01, and the test is by rows, not columns. SuperLU and MA42 select the largest entry in each column (effectively, $u = 1.0$). The rest use $u = 0.1$. We used the default value in each case.

Although MA41 has an option for exploiting shared-memory parallelism, we did not use this option in the runs for this paper. Using a non-default option, MA41 can preorder a matrix, using a maximum transversal algorithm, to ensure the diagonal of the permuted matrix is zero-free. We selected this option because it is recommended for matrices with highly unsymmetric nonzero patterns. This is followed by a default approximate minimum degree ordering (using the HSL code MC47 (Amestoy et al. 1996)) on the nonzero pattern of $\mathbf{A} + \mathbf{A}^T$. The supernodal partial pivoting code SuperLU preorders the columns via its default method, a multiple minimum degree ordering (MMD (George and Liu 1989)) on the nonzero pattern of $\mathbf{A}^T\mathbf{A}$.

We used a revised version of the unifrontal code MA42 that takes advantage of explicit zeros in the frontal matrix. MA42 is able to operate both in-core and out-of-core, using direct access files. It was primarily designed for a finite-element entry, but we used a simplified driver (MA43), that does not use out-of-core storage and assumes the matrix is held in equation form (although one matrix, AV41092, is obtained from finite-element calculations). In columns "mem. used" and "min. mem.", we show the storage required for the factorization if the factors are held in-core and an estimate of the memory required for the out-of-core factorization, respectively. In one case (TWOTONE), there was insufficient storage to hold the factors but we compute an estimate that we display.

The results, shown in Tables 7.2 and 7.3, include the following statistics for each code (all times are CPU times):

1. Factorization time, which includes preordering and symbolic factorization, if any.
2. Refactorization time, which is the numerical factorization of a matrix whose pivot ordering and symbolic factors are known. It excludes preordering and symbolic factorization.
3. Solve time, excluding iterative refinement.
4. Solve time, including iterative refinement. MA41, MA38, MA48, and SuperLU all use the same method (Arioli, Demmel and Duff 1989). MA42 and UMFPACK V1.1 do not provide this option.
5. Entries in the LU factors, in millions.
6. Memory used to obtain the timings listed in the tables, in millions of 8-byte double precision words. For MA38, UMFPACK V1.1, and MA48 this includes fragmentation within the work arrays[3]. Recall that the workstation used for the experiments has 33.6 million words of main memory (256 Mbytes), and about 242 million words of virtual address space were given to each code (1.8 Gbytes).

[3]MA48 does not compute this statistic.

Table 7.2: Results

Matrix/ code	factor time (sec)	refac. time (sec)	solve time (sec)	solve, w/iter. (sec)	entries in LU (10^6)	mem. used (10^6)	min. mem. (10^6)	flop count (10^6)
AV41092								
`MA38`	1307.9	1618.3	9.88	55.18	39.38	89.58	59.60	57110
`UMFPACK V1.1`	1502.9	1689.0	10.64	-	41.33	102.53	58.49	70290
`MA41`	296.6	254.0	1.39	7.72	16.53	22.65	22.65	11380
`MA42`	4236.3	4232.6	32.74	-	135.35	148.66	10.10	235079
`MA48`	3335.7	1296.1	5.57	50.73	27.31	-	58.46	59040
`SuperLU`	3799.0	2962.8	14.04	136.43	39.95	52.56	52.56	64970
TWOTONE								
`MA38`	220.7	171.4	1.46	6.24	9.75	26.46	17.94	6988
`UMFPACK V1.1`	228.5	174.9	1.52	-	9.59	35.04	19.31	6493
`MA41`	817.2	741.1	2.85	13.56	26.44	42.97	42.97	38230
`MA42`	-	-	-	-	-	333.60	4.64	-
`MA48`	725.0	306.9	1.29	11.70	10.86	-	24.65	14680
`SuperLU`	758.0	697.6	8.23	147.22	24.73	37.77	37.77	12420
PSMIGR_1								
`MA38`	219.0	197.5	0.53	3.01	6.37	46.34	22.86	9412
`UMFPACK V1.1`	207.4	195.6	0.71	-	6.36	26.28	22.02	9428
`MA41`	191.7	188.3	0.53	2.14	6.28	20.36	20.36	9214
`MA42`	256.0	255.0	0.66	-	8.27	17.07	8.75	13856
`MA48`	206.1	178.7	0.51	5.25	6.44	-	14.01	10580
`SuperLU`	938.7	774.5	1.80	20.21	8.71	11.10	11.10	16630
LHR71C								
`MA38`	114.5	92.9	1.01	22.25	6.93	11.72	10.87	496
`UMFPACK V1.1`	444.3	422.3	1.96	-	8.35	13.26	12.55	732
`MA41`	1012.8	995.7	3.63	82.27	18.84	23.75	23.75	4683
`MA42`	340.6	337.4	1.41	-	12.83	16.15	2.97	1202
`MA48`	492.3	286.5	1.11	30.51	6.51	-	16.54	695
`SuperLU`	479.3	451.4	2.26	64.79	7.15	11.57	11.57	487
ONETONE1								
`MA38`	60.8	57.3	0.54	2.16	4.69	15.92	7.92	2148
`UMFPACK V1.1`	73.7	69.2	0.58	-	5.05	17.34	9.19	2855
`MA41`	194.7	189.1	0.82	3.88	9.50	12.97	12.97	8169
`MA42`	164.9	164.2	1.64	-	16.58	18.52	1.49	6012
`MA48`	323.4	109.7	0.56	4.87	5.11	-	11.29	4564
`SuperLU`	109.3	97.9	1.16	9.49	4.68	7.10	7.10	2540

Table 7.3: Results, continued

Matrix/ code	factor time (sec)	refac. time (sec)	solve time (sec)	solve, w/iter. (sec)	entries in LU (10^6)	mem. used (10^6)	min. mem. (10^6)	flop count (10^6)
ONETONE2								
`MA38`	$\underline{10.7}$	$\underline{5.0}$	0.29	$\underline{1.25}$	$\underline{1.27}$	4.09	2.54	159
`UMFPACK V1.1`	12.6	6.8	$\underline{0.22}$	-	1.46	4.82	3.33	217
`MA41`	19.1	14.9	0.29	1.61	2.68	3.90	3.90	601
`MA42`	35.2	34.8	0.64	-	7.10	8.04	$\underline{0.73}$	684
`MA48`	36.6	8.0	$\underline{0.21}$	2.02	$\underline{1.26}$	-	3.36	223
`SuperLU`	$\underline{9.2}$	6.6	0.44	3.77	$\underline{1.31}$	$\underline{2.70}$	2.70	$\underline{132}$
LHR14C								
`MA38`	$\underline{8.9}$	$\underline{4.8}$	$\underline{0.17}$	$\underline{1.19}$	$\underline{1.23}$	$\underline{2.21}$	2.01	$\underline{64}$
`UMFPACK V1.1`	11.4	5.7	0.20	-	1.62	3.40	2.70	122
`MA41`	28.9	27.1	0.32	1.58	3.14	4.54	4.54	518
`MA42`	19.2	18.6	0.28	-	2.57	3.43	$\underline{0.81}$	235
`MA48`	21.1	$\underline{4.7}$	$\underline{0.17}$	1.81	$\underline{1.22}$	-	3.15	95
`SuperLU`	12.0	6.7	0.35	3.81	1.37	$\underline{2.26}$	2.26	83
RW5151								
`MA38`	2.4	$\underline{1.4}$	$\underline{0.06}$	$\underline{0.14}$	$\underline{0.41}$	1.24	0.65	47
`UMFPACK V1.1`	3.1	2.2	$\underline{0.06}$	-	0.48	1.39	0.89	69
`MA41`	2.2	2.1	0.07	$\underline{0.15}$	0.67	0.89	0.89	91
`MA42`	$\underline{1.6}$	$\underline{1.6}$	0.07	-	0.79	0.89	$\underline{0.08}$	63
`MA48`	6.5	2.5	$\underline{0.05}$	0.71	0.47	-	1.04	126
`SuperLU`	$\underline{1.8}$	$\underline{1.6}$	0.10	0.73	$\underline{0.39}$	$\underline{0.65}$	0.65	$\underline{33}$
ORANI678								
`MA38`	$\underline{1.2}$	0.4	$\underline{0.02}$	$\underline{0.21}$	$\underline{0.11}$	0.74	0.46	$\underline{4}$
`UMFPACK V1.1`	$\underline{1.2}$	0.4	$\underline{0.02}$	-	$\underline{0.11}$	$\underline{0.67}$	0.48	$\underline{4}$
`MA41`	5.3	3.6	0.04	$\underline{0.21}$	0.36	2.79	2.79	70
`MA42`	8.6	8.5	0.08	-	1.00	2.78	1.82	267
`MA48`	$\underline{1.1}$	$\underline{0.2}$	$\underline{0.02}$	$\underline{0.20}$	$\underline{0.13}$	-	$\underline{0.39}$	10
`SuperLU`	39.7	1.6	0.16	1.70	0.80	1.00	1.00	31
RDIST1								
`MA38`	1.6	0.8	$\underline{0.04}$	$\underline{0.24}$	0.44	0.97	0.74	22
`UMFPACK V1.1`	2.0	1.1	0.06	-	0.54	1.39	0.97	31
`MA41`	$\underline{0.7}$	$\underline{0.4}$	$\underline{0.03}$	$\underline{0.20}$	$\underline{0.28}$	$\underline{0.49}$	0.49	$\underline{10}$
`MA42`	2.3	2.2	0.08	-	1.02	1.22	$\underline{0.20}$	90
`MA48`	7.4	1.3	0.06	0.53	0.41	-	1.04	25
`SuperLU`	1.6	0.8	0.08	0.94	0.39	0.60	0.60	17

7. Minimum memory requirements, in millions of words. For the unifrontal code `MA42`, this is the in-core memory required for out-of-core factorization (it also includes the memory required to hold the matrix **A**). For `MA38`, `UMFPACK V1.1`, and `MA48`, this is the minimum amount required to guarantee a successful factorization, and is the statistic used in Figure 6.6. This excludes fragmentation in the work arrays, which is removed via garbage collection whenever necessary. The partial pivoting code `SuperLU` and the sequential option of the symmetric-pattern multifrontal code `MA41` do not have fragmentation in their work arrays, and thus do not require garbage collection. For these two methods the minimum required memory and actual memory used are the same, although we needed to add another counter to the HSL version of `MA41` to compute this statistic.

8. Floating-point operation count in the factorization phase, in millions. The values given are obtained from the codes. Operations for the assembly phase are not included.

Results within 10% of the best for each statistic and matrix are underlined. We compared the solutions obtained from each code, and found that all codes that provide iterative refinement compute the solutions with comparable accuracy, in terms of the scaled residual and the relative error. When iterative refinement is not in use, `MA38` and `UMFPACK V1.1` produce relative errors with a loss of about two digits of accuracy for six of the matrices when compared with the other methods, and the same accuracy for the other four.

Over all the codes, `MA38` has the fastest factorization time for four out of the ten matrices, and is within 10% of the fastest time for two more matrices. Except for one matrix (AV41092) it never takes more than about twice the time of the fastest code. It also has the fastest refactorization time for five matrices and is within 10% of the fastest time for one more matrix. The solve phase of MA38 is the fastest for four matrices and within 10% of the fastest for two more. Although the time for MA38 solution with iterative refinement seems to rank even better, both the MA48 and SuperLU iterative refinement times include also the calculation of the matrix condition number and an estimate of the forward error. Typically this requires twice as much time as only computing the backward error (as in the iterative refinement runs for MA38 and MA41). The memory requirements of `MA38` are rarely the lowest but are usually comparable to the other in-core codes for most matrices. The floating-point operation count for `MA38` is often much less than its predecessor, `UMFPACK V1.1`.

8 Summary

We have demonstrated how the advantages of the unifrontal and multifrontal approaches can be combined. The resulting algorithm (`MA38`) performs well for unsymmetric matrices from a wide range of disciplines, and is an improvement over the previous unsymmetric-pattern multifrontal code (`UMFPACK V1.1`). Other differences between `UMFPACK V1.1` and `MA38` include an option of overwriting the matrix **A** with its LU factors, printing of input and output parameters, iterative refinement with sparse backward error analysis (Arioli et al. 1989), avoidance of an extra copy of the numerical values of **A** when iterative refinement is not in use, more use of Level 3 BLAS within the numerical refactorization routine, and a simpler calling interface. These features improve the robustness of the code and result in a modest decrease in memory use.

Since the codes being compared all offer quite different capabilities and are designed for different environments and different classes of matrices, the results should not be interpreted as a direct comparison between them. For example, `MA38` is designed for structurally unsymmetric matrices. A code like `MA41` would normally be expected to perform much better on matrices that are symmetrically structured or nearly so. We also note that our current discussion compares performance on only one machine and, as in shown in Duff and Scott (1996*a*), comparative behavior can be strongly influenced by the computing platform being used. However, what we would like to highlight is the improvement that our new technique used in `MA38` brings to the unsymmetric-pattern multifrontal method and that `MA38` is at least comparable in performance with other sparse matrix codes on our unsymmetric test set.

The combined unifrontal/multifrontal method is available as the Fortran 77 codes, `UMFPACK Version 2.2` in Netlib (Dongarra and Grosse 1987)[4], and `MA38` in Release 12 of the Harwell Subroutine Library (HSL 1996).[5]

9 Acknowledgments

We would like to thank Nick Gould, John Reid, and Jennifer Scott from the Rutherford Appleton Laboratory for their helpful comments.

References

Amestoy, P., Davis, T. A. and Duff, I. S. (1996), 'An approximate minimum degree ordering algorithm', *SIAM J. Matrix Anal. Applic.* **17**(4), 886–905.

Amestoy, P. R. and Duff, I. S. (1989), 'Vectorization of a multiprocessor multifrontal code', *Internat. J. Supercomputer Appl.* **3**(3), 41–59.

Amestoy, P. R., Daydé, M. and Duff, I. S. (1989), Use of computational kernels in the solution of full and sparse linear equations., *in* M. Cosnard, Y. Robert, P. Quinton and M. Raynal, eds, 'Parallel and Distributed Algorithms: Proceedings of the International Workshop on Parallel and Distributed Algorithms, Chateau de Bonas, Gers, France, 3–6 October, 1988', North Holland, Amsterdam, pp. 13–19.

Arioli, M., Demmel, J. W. and Duff, I. S. (1989), 'Solving sparse linear systems with sparse backward error', *SIAM J. Matrix Anal. Applic.* **10**, 165–190.

Bai, Z., Day, D., Demmel, J. and Dongarra, J. (1996), Test matrix collection (non-Hermitian eigenvalue problems), Release 1, Technical report, University of Kentucky. Available at ftp://ftp.ms.uky.edu/pub/misc/bai/Collection.

[4]Versions 1.0 through 2.2 of `UMFPACK` may only be used for research, education, or benchmarking by academic users and by the U.S. Government. For a copy, send email to `netlib@ornl.gov` with the message `send index from linalg`. Other users may use `UMFPACK` only for benchmarking purposes. Version 2.2 includes complex and complex*16 codes.

[5]The Harwell Subroutine Library is available from AEA Technology, Harwell; the contact is Dr. Scott Roberts, Harwell Subroutine Library Manager, B 552, AEA Technology Plc, Harwell, Didcot, Oxon OX11 0RA, telephone +44 1235 434988, fax +44 1235 434136, email Scott.Roberts@aeat.co.uk, who will provide details of price and conditions of use.

Chan, W. M. and George, A. (1980), 'A linear time implementation of the reverse Cuthill-Mckee algorithm', *BIT* **20**, 8–14.

Cuthill, E. and McKee, J. (1969), Reducing the bandwidth of sparse symmetric matrices, *in* 'Proceedings 24th National Conference of the Association for Computing Machinery', Brandon Press, New Jersey, pp. 157–172.

Davis, T. A. (1995), Users' guide to the unsymmetric-pattern multifrontal package (UMFPACK, version 1.1), Technical Report TR-95-004, CISE Dept., Univ. of Florida, Gainesville, FL.

Davis, T. A. (1997), 'University of Florida sparse matrix collection', Available at http://www.cise.ufl.edu/~davis and ftp://ftp.cise.ufl.edu/pub/faculty/davis.

Davis, T. A. and Duff, I. S. (1991), Unsymmetric-pattern multifrontal methods for parallel sparse LU factorization, Technical Report TR-91-023, CISE Dept., Univ. of Fl., Gainesville, FL.

Davis, T. A. and Duff, I. S. (1997), 'An unsymmetric-pattern multifrontal method for sparse LU factorization', *SIAM J. Matrix Anal. Applic.* **18**(1), 140–158.

Daydé, M. J. and Duff, I. S. (1996), A blocked implementation of Level 3 BLAS for RISC processors, Technical Report RAL-TR-96-014, Rutherford Appleton Laboratory. Also ENSEEIHT-IRIT Technical Report RT/APO/96/1 and CERFACS Report TR/PA/96/06.

Demmel, J. W., Eisenstat, S. C., Gilbert, J. R., Li, X. S. and Liu, J. W. H. (1995), A supernodal approach to sparse partial pivoting, Technical Report UCB//CSD-95-883, Computer Science Div., U.C. Berkeley, Berkeley, CA. (also Xerox PARC CSL-95-03, LAPACK Working Note #103).

Demmel, J. W., Gilbert, J. R. and Li, X. S. (1997), SuperLU users' guide, Technical report, Univ. of California, Berkeley, Berkeley, CA. (available from netlib).

Dongarra, J. J. and Grosse, E. (1987), 'Distribution of mathematical software via electronic mail', *Comm. ACM* **30**, 403–407.

Dongarra, J. J., Du Croz, J. J., Duff, I. S. and Hammarling, S. (1990), 'A set of Level 3 Basic Linear Algebra Subprograms', *ACM Trans. Math. Softw.* **16**, 1–17.

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

Duff, I. S. (1984*b*), The solution of nearly symmetric sparse linear systems, *in* R. Glowinski and J.-L. Lions, eds, 'Computing methods in applied sciences and engineering, VI', North Holland, Amsterdam New York and London, pp. 57–74.

Duff, I. S. and Reid, J. K. (1983), 'The multifrontal solution of indefinite sparse symmetric linear systems', *ACM Trans. Math. Softw.* **9**, 302–325.

Duff, I. S. and Reid, J. K. (1984), 'The multifrontal solution of unsymmetric sets of linear equations', *SIAM J. Sci. Comput.* **5**(3), 633–641.

Duff, I. S. and Reid, J. K. (1993), MA48, a Fortran code for direct solution of sparse unsymmetric linear systems of equations, Technical Report RAL-93-072, Rutherford Appleton Laboratory, Didcot, Oxon, England.

Duff, I. S. and Reid, J. K. (1996), 'The design of MA48, a code for the direct solution of sparse unsymmetric linear systems of equations', *ACM Trans. Math. Softw.* **22**(2), 187–226.

Duff, I. S. and Scott, J. A. (1993), MA42 – a new frontal code for solving sparse unsymmetric systems, Technical Report RAL 93-064, Rutherford Appleton Laboratory.

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

Duff, I. S. and Scott, J. A. (1996*a*), A comparison of frontal software with other sparse direct solvers, Technical Report RAL-TR-96-102, Rutherford Appleton Laboratory.

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

Duff, I. S., Grimes, R. G. and Lewis, J. G. (1989), 'Sparse matrix test problems', *ACM Trans. Math. Softw.* **15**, 1–14.

Feldmann, P., Melville, R. and Long, D. (1996), Efficient frequency domain analysis of large nonlinear analog circuits, *in* 'Proceedings of the IEEE Custom Integrated Circuits Conference', Santa Clara, CA.

George, A. and Liu, J. W. H. (1981), *Computer Solution of Large Sparse Positive Definite Systems*, Englewood Cliffs, New Jersey: Prentice-Hall.

George, A. and Liu, J. W. H. (1989), 'The evolution of the minimum degree ordering algorithm', *SIAM Review* **31**(1), 1–19.

HSL (1996), *Harwell Subroutine Library. A Catalogue of Subroutines (Release 12)*, AEA Technology, Harwell Laboratory, Oxfordshire, England. For information concerning HSL contact: Dr Scott Roberts, AEA Technology, 552 Harwell, Didcot, Oxon OX11 0RA, England (tel: +44-1235-434988, fax: +44-1235-434136, email: Scott.Roberts@aeat.co.uk).

Irons, B. M. (1970), 'A frontal solution program for finite element analysis', *Internat. J. Numer. Methods Eng.* **2**, 5–32.

Liu, J. W. H. (1992), 'The multifrontal method for sparse matrix solution: Theory and practice', *SIAM Review* **34**(1), 82–109.

Liu, J. W. H. and Sherman, A. H. (1976), 'Comparative analysis of the Cuthill-Mckee and the reverse Cuthill-Mckee ordering algorithms for sparse matrices', *SIAM Journal on Numerical Analysis* **13**(2), 198–213.

Saad, Y. (1994), SPARSKIT: a basic tool kit for sparse matrix computations, Version 2, (available via anonymous ftp to ftp.cs.umn.edu:pub/sparse), Computer Science Dept., Univ. of Minnesota.

Vavasis, S. A. (1993), Stable finite elements for problems with wild coefficients, Technical Report 93-1364, Dept. of Computer Science, Cornell Univ., Ithaca, NY. (To appear, SIAM J. Numerical Analysis).

Zitney, S. E. (1992), Sparse matrix methods for chemical process separation calculations on supercomputers, *in* 'Proc. Supercomputing '92', IEEE Computer Society Press, Minneapolis, MN, pp. 414–423.

Zitney, S. E. and Stadtherr, M. A. (1993), 'Supercomputing strategies for the design and analysis of complex separation systems', *Ind. Eng. Chem. Res.* **32**, 604–612.

Zitney, S. E., Mallya, J., Davis, T. A. and Stadtherr, M. A. (1996), 'Multifrontal vs. frontal techniques for chemical process simulation on supercomputers', *Comput. Chem. Eng.* **20**(6/7), 641–646.