RAL-TR-1998-054

Direct Methods[1]

Iain S. Duff[2]

ABSTRACT

We review current methods for the direct solution of sparse linear equations. We discuss basic concepts such as fill-in, sparsity orderings, indirect addressing and compare general sparse codes with codes for dense systems. We examine methods for greatly increasing the efficiency when the matrix is symmetric positive definite.

We consider frontal and multifrontal methods emphasizing how they can take advantage of vectorization, RISC architectures, and parallelism. Some comparisons are made with other techniques and the availability of software for the direct solution of sparse equations is discussed.

Keywords: sparse matrices, direct methods, indirect addressing, fill-in, cliques, graph theory, frontal methods, multifrontal methods, parallel computers, software.

AMS(MOS) subject classifications: 65F05, 65F50.

---

[1]This paper is a preprint of a Chapter of the book "Numerical Linear Algebra for High-Performance Computers" by Dongarra, Duff, Sorensen, and van der Vorst that will be published by SIAM Press.

[2]I.S.Duff@rl.ac.uk

Current reports available by anonymous ftp from matisa.cc.rl.ac.uk in the directory "pub/reports". This report is in file duffRAL98054.ps.gz. Also published as Technical Report TR/PA/98/28 from CERFACS.

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

July 30, 1998.

Contents

1 Direct Solution of Sparse Linear Systems **1**

2 Introduction to Direct Methods for Sparse Linear Systems **4**
2.1 Four Approaches 5
2.2 Description of Sparse Data Structure 6
2.3 Manipulation of Sparse Data Structures 7

3 General Sparse Matrix Methods **10**
3.1 Fill-in and Sparsity Ordering 10
3.2 Indirect Addressing ... Its Effect and How to Avoid It 13
3.3 Comparison with Dense Codes 15
3.4 Other Approaches 16

4 Methods for Symmetric Matrices and Band Systems **17**
4.1 The Clique Concept in Gaussian Elimination 19
4.2 Further Comments on Ordering Schemes 21

5 Frontal Methods **21**
5.1 Frontal Methods ... Link to Band Methods and Numerical Pivoting . 23
5.2 Vector Performance 24
5.3 Parallel Implementation of Frontal Schemes 25

6 Multifrontal Methods **27**
6.1 Performance on Vector Machines 30
6.2 Performance on RISC machines 34
6.3 Performance on Parallel Machines 34
6.4 Exploitation of Structure 38
6.5 Unsymmetric Multifrontal Methods 39

7 Other Approaches for Exploitation of Parallelism **40**

8 Software **42**

9 Brief Summary **43**

1 Direct Solution of Sparse Linear Systems

In this paper, we discuss the direct solution of linear systems

$$Ax = b, \tag{1.1}$$

where the coefficient matrix A is large and sparse. Sparse systems arise in very many application areas. We list just a few such areas in Table 1.1.

Table 1.1: **A list of some application areas for sparse matrices**

acoustic scattering	4	demography	3	network flow	1
air traffic control	1	economics	11	numerical analysis	4
astrophysics	2	electric power	18	oceanography	4
biochemical	2	electrical engineering	1	petroleum engineering	19
chemical eng.	16	finite elements	50	reactor modeling	3
chemical kinetics	14	fluid flow	6	statistics	1
circuit physics	1	laser optics	1	structural engineering	95
computer simulation	7	linear programming	16	survey data	11

This table, reproduced from Duff, Grimes and Lewis (1989*a*), shows the number of matrices from each discipline present in the Harwell-Boeing Sparse Matrix Collection. This standard set of test problems is currently being upgraded to a new Collection called the Rutherford-Boeing Sparse Matrix Collection (Duff, Grimes and Lewis 1997*a*) that will include far larger systems and matrices from an even wider range of disciplines. This new Collection will be available from `netlib` and the Matrix Market (`http://math.nist.gov/MatrixMarket`).

Some of the algorithms that we will describe may appear complicated, but it is important to remember that we are primarily concerned with methods based on Gaussian elimination. That is, most of our algorithms compute an LU factorization of a permutation of the coefficient matrix A, so that $PAQ = LU$, where P and Q are permutation matrices, and L and U are lower and upper triangular matrices, respectively. These factors are then used to solve the system (1.1) through the forward substitution $Ly = P^T b$ followed by the back substitution $U(Q^T x) = y$. When A is symmetric, this fact is reflected in the factors, and the decomposition becomes $PAP^T = LL^T$ (Cholesky factorization) or $PAP^T = LDL^T$ (needed for an indefinite matrix). This last decomposition is sometimes called root-free Cholesky. It is common to store the inverse of D rather than D itself in order to avoid divisions when using the factors to solve linear systems. Note that we have used the same symbol L in all three factorizations although they each represent a different lower triangular matrix.

The solution of (1.1) can be divided naturally into several phases. Although the exact subdivision will depend on the algorithm and software being used, a common subdivision is given by:

1. A preordering phase that exploits structure, for example a preordering to block triangular or bordered block diagonal form (see Duff, Erisman and Reid (1986), for example).

2. An analysis phase where the matrix structure is analyzed to produce a suitable ordering and data structures for efficient factorization.

3. A factorization phase where the numerical factorization is performed.

4. A solve phase where the factors are used to solve the system using forward and back substitution.

Some codes combine phases 2 and 3 so that numerical values are available when the ordering is being generated. Phase 3 (or the combined phase 2 and 3) usually requires the most computing time, while the solve phase is generally an order of magnitude faster. Note that the concept of a separate factorization phase, which may be performed on a different matrix to that originally analyzed, is peculiar to sparse systems. In the case of dense matrices, only the combined analysis and factorization phase exists and, of course, phase 1 does not apply.

As we have seen in the earlier chapters, the solution of (1.1) where A, of order n, is considered as a dense matrix requires $O(n^2)$ storage and $O(n^3)$ floating-point arithmetic operations. Since we will typically be solving sparse systems that are of order several thousand or even several tens or hundreds of thousands, dense algorithms quickly become infeasible on both grounds of work and storage. The aim of sparse matrix algorithms is to solve equations of the form (1.1) in time and space proportional to $O(n) + O(\tau)$, for a matrix of order n with τ nonzeros. It is for this reason that sparse codes can become very complicated. Although there are cases where this linear target cannot be met, the complexity of sparse linear algebra is far less than in the dense case.

The study of algorithms for effecting such solution schemes when the matrix A is large and sparse is important not only for the problem in its own right, but also because the type of computation required makes this an ideal paradigm for large-scale scientific computing in general. In other words, a study of direct methods for sparse systems encapsulates many issues that appear widely in computational science and that are not so tractable in the context of really large scientific codes.

The principal issues can be summarized as follows:

1. Much of the computation is integer.

2. The data-handling problem is significant.

3. Storage is often a limiting factor, and auxiliary storage is frequently used.

4. Although the innermost loops are usually well defined, often a significant amount of time is spent in computations in other parts of the code.

5. The innermost loops can sometimes be very complicated.

Issues 1–3 are related to the manipulation of sparse data structures. The efficient implementation of techniques for handling these is of crucial importance in the solution of sparse matrices, and we discuss this in Section 2. Similar issues arise when handling large amounts of data in other large-scale scientific computing problems. Issues 2 and 4 serve to indicate the sharp contrast between sparse and non-sparse linear algebra. In code for large dense systems, well over 98 percent of the time (on a serial machine) is typically spent in the innermost loops, whereas a substantially lower fraction is spent in the innermost loops of sparse codes. The lack of dominance of a single loop is also characteristic of a wide range of large-scale applications.

Specifically, the data handling nearly always involves indirect addressing (see Section 3.2). This clearly has efficiency implications particularly for vector or parallel architectures. Much of our discussion on suitable techniques for such platforms is concerned with avoiding indirect addressing in the innermost loops of the numerical computation.

Another way in which the solution of sparse systems acts as a paradigm for a wider range of scientific computation is that it exhibits a hierarchy of parallelism that is typical of that existing in the wider case. This hierarchy comprises three levels:

- *System level.* This involves the underlying problem which, for example, may be a partial differential equation (PDE) or a large structures problem. In these cases, it is natural (perhaps even before discretization) to subdivide the problem into smaller subproblems, solve these independently, and combine the independent solutions through a small (usually dense) interconnecting problem. In the PDE case, this is done through domain decomposition; in the structures case, it is called substructuring. An analogue in discrete linear algebra is partitioning and tearing.

- *Matrix level.* At this level, parallelism is present because of sparsity in the matrix. A simple example lies in the solution of tridiagonal systems. In a structural sense no (direct) connection exists between variable 1 in equation 1 and variable n in equation n (the system is assumed to have order n), and so Gaussian elimination can be applied to both of these "pivots" simultaneously. The elimination can proceed, pivoting on entry 2 and $n-1$, then 3 and $n-2$ simultaneously so that the resulting factorization (known in LINPACK as the BABE algorithm) has a parallelism of two. This is not very exciting, although the amount of arithmetic is unchanged from the normal sequential

factorization. However, sparsity allows us to pivot simultaneously on every other entry; and when this is continued in a nested fashion, the cyclic reduction (or nested dissection in this case also) algorithm results. Now we have only $\log n$ parallel steps, although the amount of arithmetic is about double that of the serial algorithm. This sparsity parallelism can be automatically exploited for any sparse matrix, as we discuss in Section 6.

- *Submatrix level.* This level is exploited in the same way as for dense matrices since we are here concerned with eliminations within dense submatrices of the overall sparse system. Thus, the techniques of the dense linear algebra case (e.g., Level 3 BLAS) can be used. The only problem is how to organize the sparse computation to yield operations on dense submatrices. This is easy with band, variable band, or frontal solvers (Section 5) but can also be extended, through multifrontal methods, to any sparse system (Section 6).

It is important to stress that the Basic Linear Algebra Subprograms (BLAS) that we envisage using at the submatrix level are just the dense matrix BLAS discussed in the earlier chapters of this book. There is much discussion of BLAS standards for sparse matrices, see for example Duff, Marrone, Radicati and Vittoli (1997*b*), but this is aimed at the iterative methods that we discuss in the next chapter rather than for use in algorithms or software for direct methods.

Throughout this chapter, we illustrate our points by the results of numerical experiments using computer codes, most commonly from the Harwell Subroutine Library (HSL) (HSL 1996). Most of the matrix codes in the HSL can be identified by the first two characters "MA" in the subroutine or package name. We sometimes use artificially generated test problems, but most are taken from the Harwell-Boeing Sparse Matrix Collection (Duff, Grimes and Lewis 1992). These problems can be obtained by anonymous ftp from the directory pub/harwell_boeing of the machine matisa.cc.rl.ac.uk (130.246.8.22) at the Rutherford Appleton Laboratory. We have supplemented this test set by some matrices collected by Tim Davis, available by anonymous ftp from `ftp://ftp.cise.ufl.edu/pub/faculty/davis/matrices`.

2 Introduction to Direct Methods for Sparse Linear Systems

The methods that we consider for the solution of sparse linear equations can be grouped into four main categories: general techniques, frontal methods, multifrontal approaches, and supernodal algorithms. In this section, we introduce the algorithms and approaches and examine some basic operations on sparse matrices. In particular, we wish to draw attention to the features that are important in exploiting vector and parallel architectures. For background on these techniques, we recommend the book by Duff et al. (1986).

The study of sparse matrix techniques is in great part empirical, so throughout we illustrate points by reference to runs of actual codes. We discuss the availability of codes in the penultimate section of this chapter.

2.1 Four Approaches

We first consider (Section 3) a general approach typified by the HSL code MA48 (Duff and Reid 1996*a*) or Y12M (Zlatev, Waśniewski and Schaumburg 1981). The principal features of this approach are that numerical and sparsity pivoting are performed at the same time (so that dynamic data structures are used in the initial factorization) and that sparse data structures are used throughout—even in the inner loops. As we shall see, these features must be considered drawbacks with respect to vectorization and parallelism. The strength of the general approach is that it will give a satisfactory performance over a wide range of structures and is often the method of choice for very sparse unstructured problems. Some gains and simplification can be obtained if the matrix is symmetric or banded. We discuss these methods and algorithms in Section 4.

Frontal schemes can be regarded as an extension of band or variable-band schemes and will perform well on systems whose bandwidth or profile is small. The efficiency of such methods for solving grid-based problems (for example, discretizations of partial differential equations) will depend crucially on the underlying geometry of the problem. One can, however, write frontal codes so that any system can be solved; sparsity preservation is obtained from an initial ordering, and numerical pivoting can be performed within this ordering. A characteristic of frontal methods is that no indirect addressing is required in the innermost loops and so dense matrix kernels can be used. We use the HSL code MA42 (Duff and Scott 1993, Duff and Scott 1996) to illustrate this approach in Section 5.

The class of techniques that we study in Section 6, is an extension of the frontal methods termed *multifrontal*. The extension permits efficiency for any matrix whose nonzero pattern is symmetric or nearly symmetric and allows any sparsity ordering techniques for symmetric systems to be used. The restriction to nearly symmetric patterns arises because the initial ordering is performed on the sparsity pattern of the Boolean sum of the patterns of A and A^T. The approach can, however, be used on any system. The first example of this was the HSL code MA37 (Duff and Reid 1984) which was the basis for the later development of a code that uses the Level 3 BLAS and is also designed for shared memory parallel computers. This new HSL code is called MA41 (Amestoy and Duff 1993). As in the frontal method, multifrontal methods use dense matrices in the innermost loops so that indirect addressing is avoided. There is, however, more data movement than in the frontal scheme, and the innermost loops are not so dominant. In addition to the use of direct addressing, multifrontal methods differ from the first class of methods because the sparsity pivoting is usually separated from the numerical pivoting. However, there

have been very recent adaptations of multifrontal methods for general unsymmetric systems (Davis and Duff 1997*b*) which combine the analysis and factorization phases.

Another way of avoiding or amortizing the cost of indirect addressing is to combine nodes into supernodes. This technique is discussed in Section 7.

2.2 Description of Sparse Data Structure

We continue this introductory section by describing the most common sparse data structure, which is the one used in most general-purpose codes. The structure for a row of the sparse matrix is illustrated in Figure 2.1. All rows are stored in the same way, with the real values and column indices in two arrays with a one-to-one correspondence between the arrays so that the real value in position k, say, is in the column indicated by the entry in position k of the column index array. A sparse matrix can then be stored as a collection of such sparse rows in two arrays; one integer, the other real. A third integer array is used to identify the location of the position in these two arrays of the data structure for each row. If the ith position in this third array held the position marked "pointer" in Figure 2.1, then entry a_{i,j_1} would have value ξ. Clearly, access to the entries in a row is straightforward, although indirect addressing is required to identify the column index of an entry.

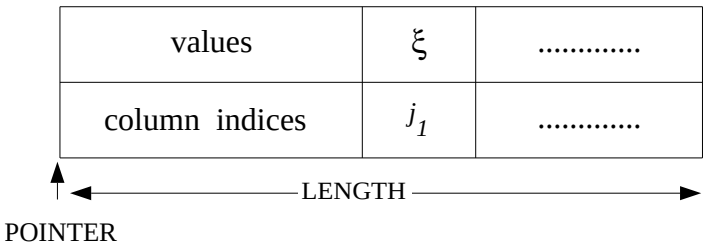

Figure 2.1: **Storage scheme for row of sparse matrix**

We illustrate this scheme in detail using the example in Figure 2.2. If we consider the matrix in Figure 2.2, we can hold each row as a packed sparse vector and the

$$\begin{pmatrix} 1 & 0 & 0 & 4 \\ -1 & 0 & 3 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & -2 & 0 & -4 \end{pmatrix}$$

Figure 2.2: **A** 4×4 **sparse matrix**

matrix as a collection of such vectors. For each member of the collection, we normally store an integer pointer to its start and the number of entries. Since we are thinking in terms of Fortran 77, a pointer is simply an array subscript indicating a position in an array. Thus, for example, the matrix of Figure 2.2 may be stored in Fortran arrays as shown in Table 2.1. Here LEN(i) contains the number of entries in row i, while IPTR(i) contains the location in arrays ICN and VALUE of the first entry in row i. For example, row 2 starts in position 3; referring to position 3 in ICN and VALUE, we find the (2,3) entry has value 3. Since LEN(2) = 2, the fourth position is also in row 2; specifically the (2,1) entry has value -1. Note that, with this scheme, the columns need not be held in order. This is important because, as we will see in Section 3, more entries will be added to the pattern of the matrix during the elimination process (called *fill-in*) and this will be facilitated by this scheme. A detailed discussion of this storage scheme is given by Duff et al. (1989*a*).

Table 2.1: **Matrix of Figure 2.2 stored as a collection of sparse row vectors**

Subscripts	1	2	3	4	5	6	7
LEN	2	2	1	2			
IPTR	1	3	5	6			
ICN	4	1	3	1	2	2	4
VALUE	4.	1.	3.	-1.	2.	-2.	-4.

Note that there is redundancy in holding both row lengths (LEN) and row pointers (IPTR) when the rows are held contiguously in order, as in Table 2.1. In general, however, operations on the matrix will result in the rows not being in order and thus both arrays are required.

2.3 Manipulation of Sparse Data Structures

To give a flavor of the issues involved in the design of sparse matrix algorithms, we now examine a manipulation of sparse data structures that occurs commonly in LU factorization. The particular manipulation we consider is the addition of a multiple of one row (the pivot row) of the matrix to other rows (the non-pivot rows) where the matrix is stored in the row pointer/column index scheme just described. Assume that an integer array, of length n, IQ say, containing all positive entries is available (for example, this may be a permutation array), and that there is sufficient space to hold temporarily a second copy of the pivot row in sparse format. Assume that the second copy has been made. Then a possible scheme is as follows:

1. Scan the pivot row, and for each entry determine its column index from the ICN value. Set the corresponding entry in IQ to the negation of the position

of that entry within the compact form of the pivot row. The original IQ entry is held in the ICN entry in the second copy of the pivot row.

For each non-pivot row, do steps 2 and 3:

2. Scan the non-pivot row. For each column index, check the corresponding entry in IQ. If it is positive, continue to the next entry in the non-pivot row. If the entry in IQ is negative, set it positive, and update the value of the corresponding entry in the non-pivot row (using the entry from the pivot row identified by the negation of the IQ entry).

3. Scan the unaltered copy of the pivot row. If the corresponding IQ value is positive, set it negative. If it is negative, then there is fill-in to the non-pivot row. The new entry (a multiple of the pivot row entry identified by the negated IQ value) is added to the end of the non-pivot row, and we continue to the next entry in the pivot row.

4. Finally, reset IQ to its original values using information from both copies of the pivot row.

Figure 2.3 illustrates the situation before each step. Note that it is not necessary to keep each row in column order when using this scheme. Conceptually simpler sparse vector additions can result when they are kept in order, but the extra work to keep the columns in order can lead to inefficiencies.

The important point about the rather complicated algorithm just described is that there are no scans of length n vectors. Thus, if this computation is performed at each major step of Gaussian elimination on a matrix of order n, there are (from this source) no $O(n^2)$ contributions to the overall work. Since our target in sparse calculations is to develop algorithms that are linear in the matrix order and number of nonzero entries, $O(n^2)$ calculations are a disaster and will dominate the computation if n is large enough.

In addition to avoiding such scans and permitting the indices to be held in any order, no real array of length n is needed. Indeed, the array IQ in Figure 2.3 can be used for any other purpose; our only assumption was that its entries were all positive. Another benefit is that we never check numerical values for zero (which might be the case if we expanded into a full-length real vector), so no confusion arises between explicitly held zeros and zeros not within the sparsity pattern. This point can be important when solving several systems with the same structure but different numerical values.

Before 1

IQ $\quad i_1 \quad i_2 \quad i_3 \quad i_4 \quad i_5 \quad i_6 \quad i_7 \quad \ldots\ldots \quad i_{n-1} \quad i_n$

Pivot row $\quad \begin{matrix} A \\ ICN \end{matrix} \quad \begin{matrix} \alpha_1 & \alpha_2 & \alpha_3 \\ j_1 & j_2 & j_3 \end{matrix} \quad + \quad$ second copy

Non-pivot row $\quad \begin{matrix} A \\ ICN \end{matrix} \quad \begin{matrix} \beta_1 & \beta_2 & \beta_3 & \beta_4 \\ j_2 & j_3 & j_4 & j_5 \end{matrix}$

Before 2

IQ $\quad -2 \quad -1 \quad i_3 \quad -3 \quad i_5 \quad i_6 \quad i_7 \quad \ldots\ldots \quad i_{n-1} \quad i_n$

Pivot row unchanged

Second copy of pivot row $\quad \begin{matrix} A \\ ICN \end{matrix} \quad \begin{matrix} \alpha_1 & \alpha_2 & \alpha_3 \\ i_2 & i_1 & i_4 \end{matrix}$

No change to non-pivot row

Before 3

IQ $\quad 2 \quad -1 \quad i_3 \quad 3 \quad i_5 \quad i_6 \quad i_7 \quad \ldots\ldots \quad i_{n-1} \quad i_n$

Pivot row unchanged

Non-pivot row $\quad \begin{matrix} A \\ ICN \end{matrix} \quad \begin{matrix} \beta_1 + \zeta\alpha_2 & \beta_2 + \zeta\alpha_3 & \beta_3 & \beta_4 \\ j_2 & j_3 & j_4 & j_5 \end{matrix}$

Before 4

IQ $\quad -2 \quad -1 \quad i_3 \quad -3 \quad i_5 \quad i_6 \quad i_7 \quad \ldots\ldots \quad i_{n-1} \quad i_n$

Pivot row unchanged

Non-pivot row $\quad \begin{matrix} A \\ ICN \end{matrix} \quad \begin{matrix} \beta_1 + \zeta\alpha_2 & \beta_2 + \zeta\alpha_3 & \beta_3 & \beta_4 & \zeta\alpha_1 \\ j_2 & j_3 & j_4 & j_5 & j_1 \end{matrix}$

Figure 2.3: **Sketch of steps involved when adding one sparse vector to another**

3 General Sparse Matrix Methods

We now discuss some fundamental aspects of sparse matrix manipulation and the use of sparsity exploiting techniques in Gaussian elimination on general sparse matrices.

3.1 Fill-in and Sparsity Ordering

A major concern when the matrix A is sparse is that the factors L and U will generally be denser than the original A.

After k steps of elimination on a matrix of order n, the reduced matrix is the lower $n-k$ by $n-k$ matrix modified from the original matrix according to the first k pivots steps. If we denote the entries of the original matrix by $a_{ij}^{(1)}$ and those of the reduced matrix after k stages of Gaussian elimination by $a_{ij}^{(k+1)}$, then fill-in is caused in Gaussian elimination if, in the basic operation

$$a_{ij}^{(k+1)} \leftarrow a_{ij}^{(k)} - a_{ik}^{(k)}[a_{kk}^{(k)}]^{-1}a_{kj}^{(k)}, \tag{3.1}$$

the entry in location (i,j) of the original A was zero. The ordering of the rows and columns of A can be important in preserving sparsity in the factors. Figure 3.1 gives an example of a case where ordering the rows and columns to preserve sparsity in Gaussian elimination is extremely effective. If pivots are chosen from the diagonal in the natural order, the reordered matrix preserves all zeros in the factorization, but the original order preserves none.

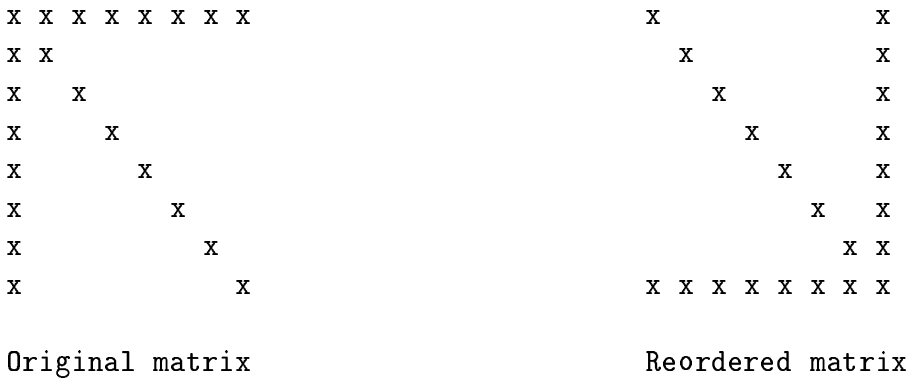

Figure 3.1: **Original and reordered matrix**

Table 3.1 illustrates the gains one can obtain in the sparse case by ignoring all or most of the zero entries both in the original matrix and in the ensuing calculation. The second row of the table gives figures for sparse elimination without any sparsity-exploiting reordering, while the third row indicates that further substantial gains can be made when sparsity-exploiting orderings are used. In larger problems, because

Table 3.1: **Benefits of Sparsity on Matrix of Order 2021 with 7353 entries**

Procedure	Total storage (Kwords)	Flops (10^6)	Time (secs) CRAY J90
Treating system as dense	4084	5503	34.5
Storing and operating only on nonzero entries	71	1073	3.4
Using sparsity pivoting	14	42	0.9

of the $O(n^3)$ and $O(n^2)$ complexity of dense codes for work and storage respectively, we might expect even more significant gains.

Another difference between dense and sparse systems arises when we consider the common case where a subsequent solution is required with a matrix of the same sparsity structure as a previously factored system. Whereas this has no relevance in the dense case, it does have a considerable influence in the sparse case since information from the first factorization can be used to simplify the second. Indeed, frequently most or all of the ordering and data organization can be done before any numerical factorization is performed.

Although it would not normally be sensible to use explicit inverses in the solution of dense systems of equations, it makes even less sense in the sparse case because (at least in the sense of the sparsity pattern) the computed inverse of an irreducible sparse matrix is always dense (Duff, Erisman, Gear and Reid 1988), whereas the factors can often be very sparse. Examples are a tridiagonal matrix and the arrowhead matrix shown in Figure 3.1. In short, it is particularly important that one does not use explicit inverses when dealing with large sparse matrices.

We illustrated the effect of ordering on sparsity preservation in Figure 3.1. A simple but effective strategy for maintaining sparsity is due to Markowitz (1957). At each stage of Gaussian elimination, he selects as a pivot the nonzero entry of the remaining reduced submatrix with the lowest product of number of other entries in its row and number of other entries in its column.

More precisely, before the kth major step of Gaussian elimination, let $r_i^{(k)}$ denote the number of entries in row i of the reduced $(n-k+1) \times (n-k+1)$ submatrix, Similarly let $c_j^{(k)}$ be the number of entries in column j. The Markowitz criterion chooses the entry $a_{ij}^{(k)}$ from the reduced submatrix to minimize the expression

$$(r_i^{(k)} - 1)(c_j^{(k)} - 1), \tag{3.2}$$

where $a_{ij}^{(k)}$ satisfies some numerical criteria also.

This strategy can be interpreted in several ways, for example, as choosing the pivot that modifies the least number of coefficients in the remaining submatrix. It may also be regarded as choosing the pivot that involves least multiplications and

divisions. Finally, we may think of (3.2) as a means of bounding the fill-in at this stage, because it would be equal to the fill-in if all $(r_i^{(k)}-1)(c_j^{(k)}-1)$ modified entries were previously zero.

In general, for the Markowitz ordering strategy in the unsymmetric case, we need to establish a suitable control for numerical stability. In particular, we restrict the Markowitz selection to those pivot candidates satisfying the inequality

$$|a_{kk}^{(k)}| \geq u\ |a_{ik}^{(k)}|, \quad i \geq k, \tag{3.3}$$

where u is a preset threshold parameter in the range $0 < u \leq 1$.

If we look back at equation (3.1), we see that the effect of u is to restrict the maximum possible growth in the numerical value of a matrix entry in a single step of Gaussian elimination to $(1 + 1/u)$. Since it is possible to relate the entries of the backward error matrix E (that is, the LU factors are exact for the matrix $A + E$) to such growth by the formula

$$|e_{ij}| \leq 3.01\ \epsilon\ n\ maxa$$

where ϵ is the machine precision and $maxa$ the modulus of the largest entry encountered in the Gaussian elimination process then, since the value of $maxa$ can be affected by the value of u, changing u can affect the stability of our factorization. We illustrate this effect in Table 3.2 and remark that, in practice, a value of u of 0.1 has been found to provide a good compromise between maintaining stability and having the freedom to reorder to preserve sparsity.

Table 3.2: **Effect of Variation in Threshold Parameter** u
(matrix of order 541 with 4285 entries)

u	Entries in Factors	Error in solution
1.0	16767	3×10^{-9}
0.25	14249	6×10^{-10}
0.10	13660	4×10^{-9}
0.01	15045	1×10^{-5}
10^{-4}	16198	1×10^{2}
10^{-10}	16553	3×10^{23}

The results in Table 3.2 are without any iterative refinement, but even with such a device no meaningful answer is obtained in the case with u equal to 10^{-10}. Note that, at very low values of the threshold, the number of entries in the factors increases with decreasing threshold. This is somewhat counter-intuitive but indicates the difficulty in choosing later pivots because of poor choices made earlier. If in the current

vogue for inexact factorizations, we consent to sacrifice accuracy of factorization for increase in sparsity, then this is done not through the threshold parameter u but rather through a drop tolerance parameter *tol*. Entries encountered during the factorization with a value less than *tol*, or less than *tol* times the largest in the row or column, are dropped from the structure, and an inexact or partial factorization of the matrix is obtained. We consider this use of partial factorizations as preconditioners in Duff and van der Vorst (1998).

3.2 Indirect Addressing ... Its Effect and How to Avoid It

Most recent vector processors have a facility for hardware indirect addressing, and one might be tempted to believe that all problems associated with indirect addressing have been overcome. For example, on the CRAY J90, the loop shown in Figure 3.2 (a sparse SAXPY) ran asymptotically at only 5.5 Mflop/s when hardware indirect addressing was inhibited but ran asymptotically at over 80 Mflop/s when it was not.

```
      DO 100 I=1,M
         A(ICN(I)) = A(ICN(I)) + AMULT * W(I)
100 CONTINUE
```

Figure 3.2: **Sparse SAXPY loop**

On the surface, the manufacturers' claims to have conquered the indirect addressing problem would seem vindicated, and we might believe that our sparse general codes would perform at about half the rate of a highly tuned dense matrix code. This reasoning has two flaws. The first lies in the $n_{1/2}$ value (Hockney and Jessup 1988) for the sparse loops (i.e., the length of the loop required to attain half the maximum performance). This measure is directly related to the startup time. For the loop shown in Figure 3.2, the $n_{1/2}$ value on the CRAY J90 is about 50, which—relative to the typical order of sparse matrices being solved by direct methods (greater than 10,000)—is insignificant. However, the loop length for sparse calculations depends not on the order of the system but rather on the number of entries in the pivot row. We have done an extensive empirical examination of this number using the MA48 code on a wide range of applications. We show a representative sample of our results in Table 3.3. Except for the example from structural analysis (the second matrix of order 1224 in the table), this length is very low and, even in this extreme case, is only about equal to the $n_{1/2}$ value mentioned above. Thus the typical performance rate for the sparse inner loop is far from the asymptotic performance.

It should be added that there are matrices where the number of entries in each row is very high, or becomes so after fill-in. This would be the case in large structures problems and for most discretizations of elliptic partial differential equations. For example, the factors of the five-diagonal matrix from a five-point star discretization

of the Laplace operator on a q by q grid could have about $6 \log_2 q$ entries in each row of the factors. Unfortunately, such matrices are very easy to generate and are thus sometimes overused in numerical experiments.

Table 3.3: **Statistics from MA48 on a CRAY J90 for various matrices from different disciplines**

Matrix		Application area	Av. Length of	% Time in
Order	Entries		Pivot Row	Inner Loops
680	2646	Atmospheric pollution	3.4	35
838	5424	Aerospace	24.5	49
1005	4813	Ship design	23.0	48
1107	5664	Computer simulation	19.2	53
1176	9874	Electronic circuit analysis	6.5	34
1224	9613	Oil reservoir modeling	12.7	49
1224	56126	Structural analysis	51.4	47
1374	8606	Nuclear engineering	16.7	49
1454	3377	Power systems networks	3.5	33
2021	7353	Chemical engineering	2.8	32
2205	14133	Oil reservoir modeling	4.0	52
2529	90158	Economic modeling	3.4	49

The second problem with the use of hardware indirect addressing in general sparse codes is that the amount of data manipulation in such a code means that a much lower proportion of the time is spent in the innermost loops than in code for dense matrices. Again we have performed an empirical study on MA48; we show these results in the last column of Table 3.3. The percentage given in that table is for the total time of three innermost loops in MA48, all at the same depth of nesting. We see that typically around 40-50 percent of the overall time on a CRAY J90 is spent in the innermost loops. Thus, even if these loops were made to run infinitely fast, a speedup of only at most a factor of two would be obtained: a good illustration of Amdahl's law.

The lack of locality of reference inherent in indirect addressing means that performance can also be handicapped or degraded on cache-based computers and parallel machines. One should mention, however, that more recent work effectively blocks the operations so that the indirect addressing is removed from the inner loops. If this approach (similar in many ways to the one we discuss in Section 6) is used, much higher computational rates can be achieved.

We conclude, therefore, that for general matrices vector indirect addressing is of limited assistance for today's general sparse codes. Even for general systems, however, advantage can be taken of vectorization by using a hybrid approach,

where a dense matrix routine is used when fill-in has caused the reduced matrix to become sufficiently dense. That is, at some stage, it is not worth paying attention to sparsity. At this point, the reduced matrix can be expanded as a full matrix, any remaining zero entries are held explicitly, and a dense matrix code can be used to effect the subsequent decomposition. The resultant hybrid code should combine the advantages of the reduction in operations resulting from sparsity in the early stages with the high computational rates of a dense linear algebra code in the latter. The point at which such a switch is best made will, of course, depend both on the vector computer characteristics and on the relative efficiency of the sparse and dense codes.

3.3 Comparison with Dense Codes

Every time there are improvements in algorithms or computer hardware that greatly improve the efficiency of dense matrix solvers, there are claims that sparse direct solvers are now obsolete and that these dense solvers should be used for sparse systems as well. There are two main reasons why this argument is severely flawed. The first is that, although the performance of dense codes is remarkable, even for fairly modest values of order n of the matrix, the $O(n^3)$ complexity of a dense solver makes such calculations prohibitively expensive, if the $O(n^2)$ storage requirements have not already made solution impossible. However, a far more telling rebuttal is provided by current sparse direct solvers like the HSL code MA48. We see from the results in Table 3.4 that MA48 performs much better than the LAPACK code SGESV on the CRAY Y-MP even for sparse matrices of fairly modest order. Clearly, the MA48 performance is dependent on the matrix structure, as is evident from the two runs on different matrices of order 1224, whereas the LAPACK code depends on the order and shows the expected $O(n^3)$ behavior, slightly tempered by the better performance of the Level 3 BLAS on larger matrices. However, even at its worst, MA48 comfortably outperforms SGESV. It should also be pointed out that subsequent factorizations of a matrix of the same pattern are much cheaper for the sparse code, for example the matrix BCSSTK27 can be refactorized by MA48 in 0.33 seconds, using the same pivot sequence as before.

The final nail in the coffin of dense matrix solvers lies in the fact that MA48 monitors the density of the reduced matrix as the elimination proceeds and switches to dense mode as indicated in Section 3.2. Thus, highly efficient dense solvers, can now be incorporated within the sparse code and so *a fortiori* the dense solvers can never beat the sparse code. This is not entirely true because there are some overheads associated with switching, but experiments with such an approach indicate that the switch-over density for overall time minimization can often be low (typically 20 percent dense) and that gains of a factor of over four can be obtained even with unsophisticated dense matrix code (Duff 1984*b*). Naturally, if we store the zeros of a reduced matrix, we might expect an increase in the storage requirements for the decomposition. Although the luxury of large memories on

some current computers gives us ample scope for allowing such an increase, it is interesting to note that the increase in storage for floating-point numbers is to some extent compensated for by the reduction in storage for the accompanying integer index information.

Table 3.4: **Comparison between MA48 and LAPACK (SGESV) on range of matrices from the Harwell-Boeing Sparse Matrix Collection.** Times for factorization and solution are in seconds on one processor of a CRAY Y-MP

Matrix	Order	Entries	MA48	SGESV
FS 680 3	680	2646	0.06	0.96
PORES 2	1224	9613	0.54	4.54
BCSSTK27	1224	56126	2.07	4.55
NNC1374	1374	8606	0.70	6.19
WEST2021	2021	7353	0.21	18.88
ORSREG 1	2205	14133	2.65	24.25
ORANI678	2529	90158	1.17	36.37

3.4 Other Approaches

In this section, we have concentrated on the approach typified by the codes MA48 (Duff and Reid 1996*a*) and Y12M (Zlatev et al. 1981). While these are possibly the most common codes used to solve general sparse systems arising in a wide range of application areas, there are other algorithmic approaches and codes for solving general unsymmetric systems. One technique is to preorder the rows, then to choose pivots from each row in turn, first updating the incoming row according to the previous pivot steps and then choosing the pivot by using a threshold criterion on the appropriate part of the updated row. If the columns are also preordered for sparsity, then an attempt is first made to see whether the diagonal entry in the reordered form is suitable. This approach is used by the codes NSPFAC and NSPIV of Sherman (1978). It is similar to the subsequent factorizations using the main approach of this section, and so is simple to code. It can, however, suffer badly from fill-in if a good initial ordering is not given or if numerical pivoting forbids keeping close to that ordering. Some of the multifrontal and supernodal approaches to sparse system solution also suffer in this way.

Another approach is to generate a data structure that, within a chosen column ordering, accommodates all possible pivot choices (George and Ng 1985). It is remarkable that this is sometimes not overly expensive in storage and has the benefit of good stability but within a subsequent static data structure. There are, of course, cases when the storage penalty is high.

Methods using a sparse QR factorization can be used for general unsymmetric systems. These are usually based on work of George and Heath (1980) and first obtain the structure of R through a symbolic factorization of the structure of the normal equations matrix A^TA. It is common not to keep Q but to solve the system using the semi-normal equations

$$R^TRx = A^Tb,$$

with iterative refinement usually used to avoid numerical problems. A QR factorization can, of course, be used for least-squares problems and implementations have been developed using ideas similar to those discussed later in Section 6 for Gaussian elimination. Further discussion of QR and least squares is outside the scope of this chapter.

Another approach particularly designed for unsymmetric matrices has been developed by Davis and Yew (1990). Here a set of pivots is chosen simultaneously using Markowitz and threshold criteria so that if the set is permuted to the upper part of the matrix, the corresponding block will be diagonal and all operations corresponding to these pivots can be performed simultaneously. Indeed, it is possible to design an algorithm to perform the pivot search in parallel also. Subsequent sets of independent pivots are chosen in a similar manner until the reduced matrix becomes dense enough to switch to dense code, as discussed in Section 3. Alaghband (1989) proposes a similar type of scheme.

One possibility for exploiting parallelism by general sparse direct methods is to use partitioning methods and, in particular, the bordered block triangular form (Duff et al. 1986). This approach is discussed by Arioli and Duff (1990), who indicate that reasonable speedups on machines with low levels of parallelism (4–8) are obtained fairly easily even on very difficult systems.

4 Methods for Symmetric Matrices and Band Systems

Elementary graph theory has been used as a powerful tool in the analysis and implementation of diagonal pivoting on symmetric matrices (see, for example, George and Liu (1981)). Although graph theory is sometimes overused in sparse Gaussian elimination, in certain instances it is a useful and powerful tool. We shall now explore one such area. For this illustration, we consider finite *undirected graphs*, G(V, E), which comprise a finite set of distinct *vertices* (or `nodes`) V and an *edge set* E consisting of unordered pairs of vertices. An edge $e \in$ E will be denoted by (u, v) for some $u, v \in$ V. The graph is *labeled* if the vertex set is in 1-1 correspondence with the integers $1, 2, ..., |\mathrm{V}|$, where $|\mathrm{V}|$ is the number of vertices in V. In this application of graph theory, the set E, by convention, does not include self-loops (edges of the form (u, u), $u \in$ V) or multiple edges. Thus, since the graph is undirected, edge

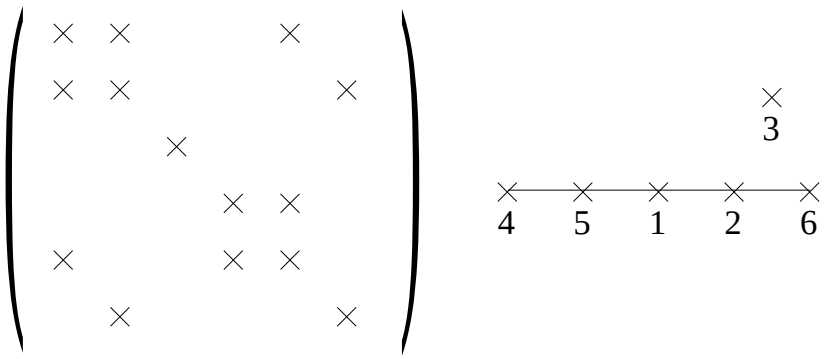

Figure 4.1: **Symmetric matrix and associated graph**

(u, v) is equal to edge (v, u), and only one is held. A subgraph H(U, F) of G(V, E) has vertex set U $\subseteq$ V, and edge set F $\subseteq$ E. H(U, F) is fully connected if $(u, v) \in$ F for all $u, v \in$ U. With any symmetric matrix, of order n say, we can associate a labeled graph with n vertices such that there is an edge between vertex i and vertex j (edge (i, j)) if and only if entry a_{ij} (and, by symmetry, a_{ji}) of the matrix is nonzero. We give an example of a matrix and its associated graph in Figure 4.1. If the pattern of A is symmetric and we can be sure that diagonal pivots produce a stable factorization (the most important example is when A is symmetric and positive definite), then two benefits occur. We do not have to carry numerical values to check for stability, and the search for the pivot is simplified to finding i such that

$$r_i^{(k)} = \min_t r_t^{(k)}$$

and using $a_{ii}^{(k)}$ as pivot, where $r_t^{(k)}$, is the number of entries in row t of the reduced matrix, as in Equation (3.2). This scheme was introduced by Tinney and Walker (1967) as their Scheme 2 and is normally termed the *minimum degree algorithm* because of its graph theoretic interpretation: in the graph associated with a symmetric sparse matrix, this strategy corresponds to choosing for the next elimination the vertex that has the fewest edges connected to it. Surprisingly (as we shall see later in this section), the algorithm can be implemented in the symmetric case without explicitly updating the sparsity pattern at each stage, a situation that greatly improves its performance.

The main benefits of using such a correspondence can be summarized as follows:

1. The structure of the graph is invariant under symmetric permutations of the matrix (they correspond merely to a relabeling of the vertices).

2. For mesh problems, there is usually a correspondence between the mesh and the graph of the resulting matrix. We can thus work more directly with the underlying structure.

3. We can represent *cliques* (fully connected subgraphs) in a graph by listing the vertices in a clique without storing all the interconnection edges.

4.1 The Clique Concept in Gaussian Elimination

To illustrate the importance of the clique concept in Gaussian elimination, we show in Figure 4.2 a matrix and its associated graph (also the underlying mesh, if, for example, the matrix represents the five-point discretization of the Laplacian operator). If the circled diagonal entry in the matrix were chosen as pivot (admittedly not a very sensible choice on sparsity grounds), then the resulting reduced matrix would have the dashed (pivot) row and column removed and have additional entries (fill-ins) in the checked positions and their symmetric counterparts. The corresponding changes to the graph cause the removal of the circled vertex and its adjacent edges and the addition of all the dashed edges shown.

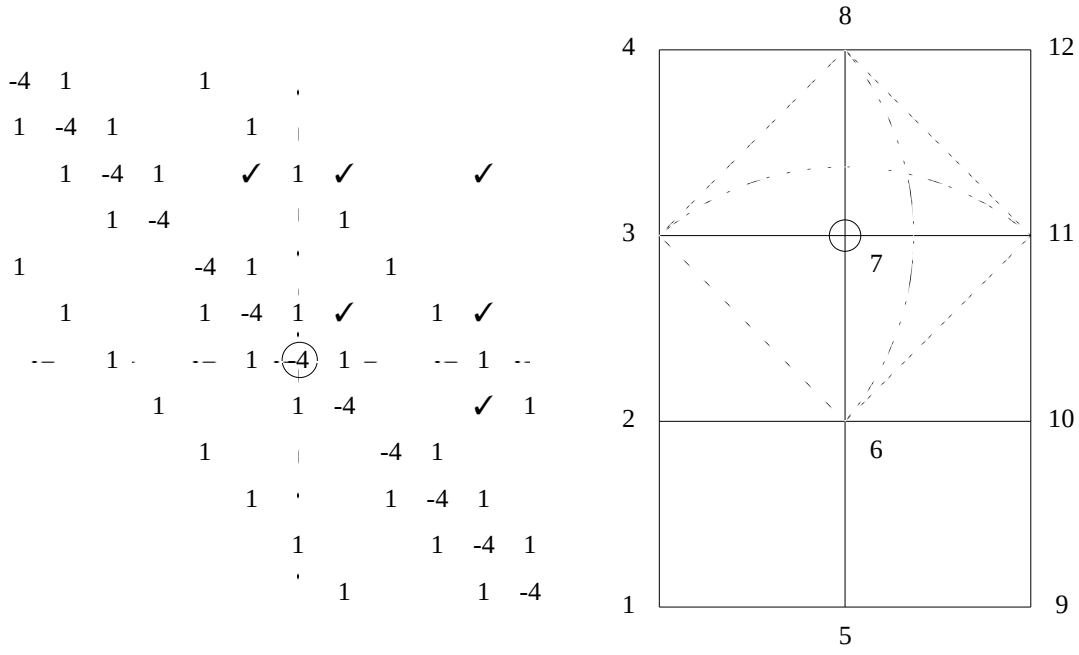

Figure 4.2: **Use of cliques in Gaussian elimination**

Thus, after the elimination of vertex 7, the vertices 3,6,8,11 form a clique because all vertices of the graph of the reduced matrix that were connected to the vertex associated with the pivot will become pairwise connected after that step of the elimination process. Although this clique formation has been observed for some time (Parter 1961), it was more recently that techniques exploiting clique formation have been used in ordering algorithms. Our example clique has 4 vertices and 6 interconnecting edges; but, as the elimination progresses, this difference will be

more marked, since a clique on q vertices has $q(q-1)/2$ edges corresponding to the off-diagonal entries in the associated dense submatrix.

If we are using the clique model for describing the elimination process then, when a vertex is selected as pivot, the cliques in which it lies are amalgamated. We illustrate clique amalgamation in Figure 4.3 where the circled element is being used as a pivot and the vertices in each rectangle are all pairwise connected. We do not show the edges internal to each rectangle, because we wish to reflect the storage and data manipulation scheme that will be used.

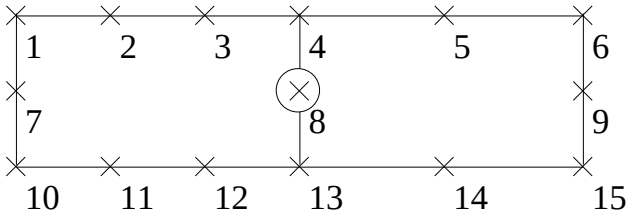

Figure 4.3: **Clique amalgamation**

The two cliques (sometimes called elements or generalized elements by analogy with elements in the finite-element method) are held only as lists of constituent vertices (1,2,3,4,7,8,10,11,12,13) and (4,5,6,8,9,13,14,15). The result of eliminating (that is pivoting on) vertex 8 is to amalgamate these two cliques. That is, after the elimination of vertex 8, the variables in both of these cliques will be pairwise connected to form a new clique given by the list (1,2,3,4,5,6,7,9,10,11,12,13,14,15). Not only is a list merge the only operation required, but the storage after elimination (for the single clique) will be less than before the elimination (for the two cliques). Since such a clique amalgamation can be performed in a time proportional to the number of vertices in the cliques concerned, both work and storage are linear rather than quadratic in the number of vertices in the cliques.

To summarize, two important aspects of this clique-based approach make it very significant and exciting. Earlier codes for computing minimum degree orderings mimicked the Gaussian elimination process. That is, they performed the elimination symbolically without doing any floating-point arithmetic. With the clique-based approach, we do not have to mimic the Gaussian elimination operations during this ordering phase, and finding the ordering can be significantly faster than the actual elimination. The second—and in many ways more important—aspect is that of storage, where the ordering can be implemented so that it requires only slightly more storage (a small multiple of n) over that for the matrix itself. Since the storage required for computing the ordering is independent of the fill-in and is known in advance, we can ensure that sufficient storage is available for successful completion of this phase. Thus, even for very large problems whose decomposition must necessarily be out-of-core, it may be possible to do an in-core ordering.

When computing the ordering we can determine the storage and work required for subsequent factorization. This forecast could be crucial if we wish to know whether the in-core solution by a direct method is feasible or whether an out-of-core solver or iterative method must be used.

The dramatic effect of using a clique amalgamation approach in the implementation of minimum degree can be seen very clearly in the performance of the two HSL codes MA17 and MA27. The former code, written before clique amalgamation techniques were developed, runs two orders of magnitude slower than the HSL code MA27 which uses clique amalgamation. The improvement in minimum degree codes over the decade 1970 to 1980 is shown by Duff et al. (1986).

4.2 Further Comments on Ordering Schemes

Researchers have added numerous twists to the minimum degree scheme. Many of these are implementation refinements; they are discussed in a review by George and Liu (1989). Others are developed to compromise a strict minimum degree so that some other property is enhanced—for example, so that the ordering produced retains the good reduction in work afforded by minimum degree while, at the same time, yields an ordering more amenable to parallel implementation than strict minimum degree (Duff, Gould, Lescrenier and Reid 1990, Liu 1988).

Additionally, techniques have been developed to use an approximate minimum degree to further reduce the computation times for finding an ordering. The approximate minimum degree (AMD) algorithm of Amestoy, Davis and Duff (1996) is often much faster than the best implementation of minimum degree and produces an ordering of comparable quality.

There has been much work very recently in developing dissection techniques that produce orderings resulting in lower fill-in and operation count for a subsequent Cholesky factorization (Rothberg 1996) over a wide range of problems. Much of this is motivated by finding good orderings for the elimination on parallel computers.

5 Frontal Methods

Frontal methods have their origins in the solution of finite-element problems from structural analysis. One of the earliest computer programs implementing the frontal method was that of Irons (1970). He considered only the case of symmetric positive-definite systems. The method can, however, be extended to unsymmetric systems (Hood 1976) and need not be restricted to finite-element applications (Duff 1981).

The usual way to describe the frontal method is to view its application to finite-element problems where the matrix A is expressed as a sum of contributions from

the elements of a finite-element discretization. That is,

$$A = \sum_{l=1}^{m} A^{[l]}, \tag{5.1}$$

where $A^{[l]}$ is nonzero only in those rows and columns that correspond to variables in the lth element. If a_{ij} and $a_{ij}^{[l]}$ denote the (i,j)th entry of A and $A^{[l]}$, respectively, the basic assembly operation when forming A is of the form

$$a_{ij} \Leftarrow a_{ij} + a_{ij}^{[l]}. \tag{5.2}$$

It is evident that the basic operation in Gaussian elimination

$$a_{ij} \Leftarrow a_{ij} - a_{ip}[a_{pp}]^{-1}a_{pj} \tag{5.3}$$

may be performed as soon as all the terms in the triple product (5.3) are *fully summed* (that is, are involved in no more sums of the form (5.2)). The assembly and Gaussian elimination processes can therefore be interleaved and the matrix A is never assembled explicitly. This allows all intermediate working to be performed in a dense matrix, termed the *frontal matrix*, whose rows and columns correspond to variables that have not yet been eliminated but occur in at least one of the elements that have been assembled.

For non-element problems, the rows of A (equations) are added into the frontal matrix one at a time. A variable is regarded as fully summed whenever the equation in which it last appears is assembled. The frontal matrix will, in this case, be rectangular. A full discussion of the equation input can be found in Duff (1984*a*).

We now describe the method for element input. After the assembly of an element, if the k fully summed variables are permuted to the first rows and columns of the frontal matrix, we can partition the frontal matrix F in the form

$$F = \begin{pmatrix} B & C \\ D & E \end{pmatrix}, \tag{5.4}$$

where B is a square matrix of order k and E is of order $r \times r$. Note that $k + r$ is equal to the current size of the frontal matrix, and $k \ll r$, in general. Typically, B might have order 10 to 20, while E is of order 200 to 500. The rows and columns of B, the rows of C, and the columns of D are fully summed; the variables in E are not yet fully summed. Pivots may be chosen from anywhere in B. For symmetric positive-definite systems, they can be taken from the diagonal in order but in the unsymmetric case, pivots must be chosen to satisfy a threshold criteria. This is discussed in Duff (1984*a*).

It is also possible to view frontal methods as an extension of band or variable-band schemes. We do this in the following section.

5.1 Frontal Methods ... Link to Band Methods and Numerical Pivoting

A common method of organizing the factorization of a band matrix of order n with semibandwidth b is to allocate storage for a dense $b \times 2b-1$ matrix, which we call the frontal matrix, and to use this as a window that runs down the band as the elimination progresses. Thus, at the beginning, the frontal matrix holds rows 1 to b of the band system. This configuration enables the first pivotal step to be performed (including pivoting if this is required); and, if the pivot row is then moved out of the frontal matrix, row $b+1$ of the band matrix can be accommodated in the frontal matrix. One can then perform the second pivotal step within the frontal matrix. Typically, a larger frontal matrix is used since greater efficiency can be obtained by moving blocks of rows at a time. It is then usually possible to perform several pivot steps within the one frontal matrix. The traditional reason for this implementation of a banded solver is for the solution of band systems by out-of-core methods since only the frontal matrix need be held in main storage. This use of auxiliary storage is also one of the principal features of a general frontal method.

This "windowing" method can be easily extended to variable-band matrices. In this case, the frontal matrix must have order at least $\max_{a_{ij}\neq 0}\{|i-j|\}$. Further extension to general matrices is possible by observing that any matrix can be viewed as a variable-band matrix. Here lies the main problem with this technique: for any arbitrary matrix with an arbitrary ordering, the required size for the frontal matrix may be very large. However, for discretizations of partial differential equations (whether by finite elements or finite differences), good orderings can usually be found (see, for example, Duff, Reid and Scott (1989*b*) and Sloan and Randolph (1983)). In fact, recent experiments by Duff and Scott (1997) show that there is a reasonably wide range of problems for which a frontal method could be the method of choice, particularly if the matrix is expressed as a sum of element matrices in unassembled form.

We limit our discussion here to the implementation of Gaussian elimination on the frontal matrix. The frontal matrix is of the form (5.4). The aim at this stage is to perform k steps of Gaussian elimination on the frontal matrix (choosing pivots from B), to store the factors $L_B U_B$ of B, DB^{-1}, and C on auxiliary storage, and to generate the Schur complement $E - DB^{-1}C$ for use at the next stage of the algorithm.

As we mentioned earlier, in the unsymmetric case, the pivots can be chosen from anywhere within B. In our approach (Duff 1984*a*), we use the standard sparse matrix technique of threshold pivoting, where $b_{ij} \in B$ is suitable as pivot only if

$$|b_{ij}| \geq u \max(\max_s |b_{sj}|, \max_s |d_{sj}|), \tag{5.5}$$

where u is a preset parameter in the range $0 < u \leq 1$.

Notice that, although we can only choose pivots from the block B, we must test

potential pivot candidates for stability by comparing their magnitude with entries from B and from D. This means that large entries in D can prevent the selection of some pivots from B. Should this be the case, $k_1 \leq k$ steps of Gaussian elimination will be performed, and the resulting Schur complement $E - DB_1^{-1}C$, where B_1 is a square submatrix of B of order k_1, will have order $r + k - k_1$. Although this can increase the amount of work and storage required by the algorithm, the extra cost is typically very low, and all pivotal steps will eventually be performed since the final frontal matrix has a null E block (that is, $r = 0$).

An important aspect of frontal schemes is that all the elimination operations are performed within a dense matrix, so that kernels and techniques (including those for exploiting vectorization or parallelism) can be used on dense systems. It is also important that k is usually greater than 1, in which case more than one elimination is performed on the frontal matrix and Level 2 and Level 3 BLAS can be used as the computational kernel. Indeed, on some architectures (for example the SGI Power Challenge), it can be beneficial to increase the size of the B block by doing extra assemblies before the elimination stage, even if more floating-pont operations are then required to effect the factorization (Cliffe, Duff and Scott 1998).

5.2 Vector Performance

We now illustrate some of the points just raised with numerical experiments. For the experimental runs, we use as our standard model problem an artificial finite-element problem designed to simulate those actually occurring in some CFD calculations. The elements are nine-node rectangular elements with nodes at the corners, mid-points of the sides, and center of the element. A parameter to the generation routine determines the number of variables per node which has been set to five for the runs in this chapter. The elements are arranged in a rectangular grid whose dimensions are given in the tables.

A code for frontal elimination, called MA32, was included in the Harwell Subroutine Library in 1981 (Duff 1981). This code was used extensively by people working with finite-elements particularly on the CRAY-2, then at Harwell. The MA32 code was substantially improved to produce the HSL code MA42 (Duff and Scott 1993) that has a very similar interface and functionality to its predecessor but makes more extensive use of higher Level BLAS than the earlier routine. On a machine with fast Level 3 BLAS, the performance can be impressive. We illustrate this point by showing the performance of our standard test problem on one processor of a CRAY Y-MP, whose peak performance is 333 Mflop/s and on which the Level 3 BLAS matrix-matrix multiply routine SGEMM runs at 313 Mflop/s on sufficiently large matrices. It is important to realize that the Megaflop rates given in Table 5.1 include all overheads for the out-of-core working used by MA42.

Although, as we see in Table 5.1, very high Megaflop rates can be achieved with the frontal scheme, this is often at the expense of more flops than are necessary

Table 5.1: **Performance (in Mflop/s) of MA42 on one processor of a CRAY Y-MP on our standard test problem**

Dimension of element grid	16 x 16	32 x 32	48 x 48	64 x 64	96 x 96
Max order frontal matrix	195	355	515	675	995
Total order of problem	5445	21125	47045	83205	186245
Mflop/s	145	208	242	256	272

to perform the sparse factorization. The usefulness of the frontal scheme is very dependent on this trade off between flops and Mflop/s. It may be possible to reorder the matrix to reduce the flop count and extend the applicability of frontal methods, using reordering algorithms similar to those for bandwidth reduction of assembled matrices (Duff et al. 1989*b*). As an example of the effect of reordering, the matrix LOCK3491 from the Harwell-Boeing Sparse Matrix Collection can be factorized by MA42 at a rate of 118 Mflop/s on a CRAY Y-MP. If the matrix is first reordered using the HSL subroutine MC43, the frontal code then runs at only 89 Mflop/s but the number of floating-point operations is reduced substantially so that only 1.07 seconds are now required for the factorization compared with 9.69 seconds for the unordered problem. This is a graphic illustration of the risk of placing too much emphasis on computational rates (Mflop/s) when evaluating the performance of algorithms. Additionally, although the effect of reordering can be quite dramatic, it is very similar to bandwidth reduction and so will not enable the efficient use of frontal schemes on general systems.

In addition to the problem that far more arithmetic can be done than is required for the numerical factorization, a second problem with the frontal scheme is that there is little scope for parallelism other than that which can be obtained within the higher level BLAS. Indeed, if we view the factorization in terms of a computational tree where nodes correspond to factorizations of frontal matrices of the form (5.4) and edges correspond to the transfer of the Schur complement data, then the computational tree of the method just described is a chain. Since all the data must be received from the child before work at the parent node can complete, parallelism cannot be exploited at the tree level. These deficiencies can be at least partially overcome through allowing the use of more than one front. This permits pivot orderings that are better at preserving sparsity and also gives more possibility for exploitation of parallelism through working simultaneously on different fronts.

5.3 Parallel Implementation of Frontal Schemes

This way of exploiting parallelism when using frontal methods is similar to domain decomposition, where we partition the underlying "domain" into subdomains,

perform a frontal decomposition on each subdomain separately (this can be done in parallel) and then factorize the matrix corresponding to the remaining "boundary" or "interface" variables (the Schur complement system), perhaps by also using a frontal scheme, as in Duff and Scott (1994), or by any suitable solver. This strategy corresponds to a bordered block diagonal ordering of the matrix and can be nested. More recently, a class of algorithms have been designed (Ashcraft and Liu 1996) that combine different ordering strategies on the subdomain and the interface variables.

Although the main motivation for using multiple fronts is usually to exploit parallelism, it is also important that the amount of work can be significantly changed by partitioning the domain, and in some cases can be much reduced. For example, suppose we have a 5-point finite-difference discretization on a $2N$ x $2N$ grid, resulting in a matrix of order $4N^2$ and semibandwidth $2N$. Then straightforward solution using a frontal scheme requires $32N^4 + O(N^3)$ floating-point operations whereas, if the domain is partitioned into four subdomains each of size N x N, the operation count can be reduced to $18.6N^4 + O(N^3)$. Note that the ordering within each subdomain is important in achieving this performance.

We illustrate this change in the number of floating-point operations by running a multiple-front code on our model problem on a 48 x 48 element grid on a single processor of a CRAY Y-MP (Duff and Scott 1994). With a single domain, the CPU time is 69.4 secs and there are 16970 million floating-point operations in the factorization, while the corresponding figures using four subdomains are 48.4 and 10350. If we partition further, the figures reduce to 40.3 and 8365, when using 8 subdomains.

We have examined the performance of this approach in two parallel environments: on an eight processor shared-memory CRAY Y-MP8I and on a network of five DEC Alpha workstations using PVM (Duff and Scott 1994). In each case, we divide the original problem into a number of subdomains equal to the number of processors being used. It is difficult to get meaningful times in either environment because we cannot guarantee to have a dedicated machine. The times in Table 5.2 are, however, for lightly loaded machines.

The results on the CRAY are encouraging and show good speedup. Those on the Alpha farm are quite comparable and indicate that the overheads of PVM and communication costs do not dominate and good speedups are possible. We should add that it is important that disk i/o is local to each processor. The default on our Alpha system was to write all files centrally and this increased data traffic considerably, gave poor performance, and varied greatly depending on system loading.

We also observe that the speedup obtained when increasing the number of subdomains from 2 to 4 is greater than 2. This apparent superlinear behavior is caused by the reduction in the number of floating-point operations for the four subdomain partitioning as discussed earlier.

Other work on the parallel implementation of a frontal elimination has been

Table 5.2: **Multiprocessor performance of MA42 on CRAY Y-MP and DEC Alpha farm on 48 x 48 grid for our standard test problem**

	CRAY Y-MP		DEC Alpha farm	
Number of subdomains	Wall clock time (secs)	Speedup	Wall clock time (secs)	Speedup
1	98.8	-	1460.3	-
2	64.6	1.5	1043.0	1.4
4	30.7	3.2	457.5	3.2
8	15.3	6.5	-	-

carried out by Benner, Montry and Weigand (1987) using large-grain parallelism on a CRAY X-MP and an ELXSI, and by Lucas, Blank and Tiemann (1987) on a hypercube.

6 Multifrontal Methods

If one takes the techniques discussed in the previous section to their logical conclusion, then many separate fronts can be developed simultaneously and can be chosen using a sparsity preserving ordering such as minimum degree. We call such methods "multifrontal" methods. The idea is to couple a sparsity ordering with the efficiency of a frontal matrix kernel so allowing good exploitation of high performance computers. Multifrontal methods are described in some detail by Duff et al. (1986), and their potential for parallelism is discussed by Duff (1986*a*,1986*b*,1989*a*).

In this section, we shall work through a small example to give a flavor of the important points and to introduce the notion of an elimination tree. An elimination tree, discussed in detail by Duff (1986*a*) and Liu (1990), is used to define a precedence order within the factorization. The factorization commences at the leaves of the tree, and data is passed towards the root along the edges in the tree. To complete the work associated with a node, all the data must have been obtained from the children of the node, otherwise work at different nodes is independent. We use the example in Figure 6.1 to illustrate both the multifrontal method and its interpretation in terms of an elimination tree.

The matrix shown in Figure 6.1 has a nonzero pattern that is symmetric. (Any system can be considered, however, if we are prepared to store explicit zeros.) The matrix is ordered so that pivots will be chosen down the diagonal in order. At the first step, we can perform the elimination corresponding to the pivot in position (1,1), first "assembling" row and column 1 to get the submatrix shown in Figure 6.2. By "assembling", we mean placing the entries of row and column 1 into a submatrix of order the number of entries in row and column 1. Thus the zero entries a_{12}

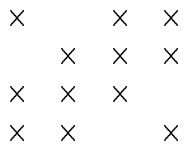

Figure 6.1: **Matrix used to illustrate multifrontal scheme**

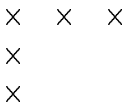

Figure 6.2: **Assembly of first pivot row and column**

and a_{21} cause row and column 2 to be omitted in Figure 6.2, and so an index vector is required to identify the rows and columns that are in the submatrix. The index vector for the submatrix in Figure 6.2 would have entries (1,3,4) for both the rows and the columns. Column 1 is then eliminated by using pivot (1,1) to give a reduced matrix of order two with associated row (and column) indices 3 and 4. In conventional Gaussian elimination, updating operations of the form

$$a_{ij} \Leftarrow a_{ij} - a_{i1}[a_{11}]^{-1}a_{1j} \tag{6.1}$$

would be performed immediately for all (i, j) such that $a_{i1}a_{1j} \neq 0$. However, in this formulation, the quantities

$$a_{i1}[a_{11}]^{-1}a_{1j} \tag{6.2}$$

are held in the reduced submatrix, and the corresponding updating operations are not performed immediately. These updates are not necessary until the corresponding entry is needed in a later pivot row or column. The reduced matrix can be stored until that time.

Row (and column) 2 is now assembled, the (2,2) entry is used as pivot to eliminate column 2, and the reduced matrix of order two—with associated row (and column) indices of 3 and 4—is stored. These submatrices are called frontal matrices. More than one frontal matrix generally is stored at any time (currently we have two stored). This is why the method is called "multifrontal". Now, before we can perform the pivot operations using entry (3,3), the updating operations from the first two eliminations (the two stored frontal matrices of order two) must be performed on the original row and column 3. This procedure is effected by summing or assembling the reduced matrices with the original row and column 3, using the index lists to control the summation. Note that this gives rise to an assembled submatrix of order 2 with indices (3,4) for rows and columns. The pivot operation that eliminates

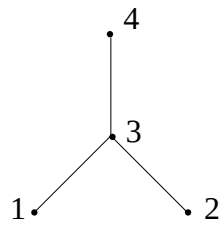

Figure 6.3: **Elimination tree for the matrix of Figure 6.1**

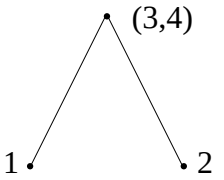

Figure 6.4: **Assembly tree for the matrix of Figure 6.1 after node amalgamation**

column 3 by using pivot (3,3) leaves a reduced matrix of order one with row (and column) index 4. The final step sums this matrix with the (4,4) entry of the original matrix. The sequence of major steps in the elimination is represented by the tree in Figure 6.3.

The same storage and arithmetic are needed if the (4,4) entry is assembled at the same time as the (3,3) entry, and in this case the two pivotal steps can be performed on the same submatrix. This procedure corresponds to collapsing or amalgamating nodes 3 and 4 in the tree of Figure 6.3 to yield the tree of Figure 6.4. On typical problems, node amalgamation produces a tree with about half as many nodes as the order of the matrix. We call this tree the *assembly* tree. Additional advantage can be taken of amalgamations which do not preserve the number of arithmetic operations, that is where there are variables in the child node that are not present in the parent. This is sometimes called "relaxed amalgamation". Duff and Reid (1984) employ node amalgamation to enhance the vectorization of a multifrontal approach, and much subsequent work has also used this strategy for better vectorization and exploitation of parallelism (for example, Ashcraft (1987) and Liu (1990)).

The computation at a node of the tree is simply the assembly of information concerning the node, together with the assembly of the reduced matrices from its children, followed by some steps of Gaussian elimination. Each node corresponds to the formation of a frontal matrix of the form (5.4) followed by some elimination steps, after which the Schur complement is passed on for assembly at the parent node.

Viewing the factorization using an assembly tree has several benefits. Since only a partial ordering is defined by the tree, the only requirement for a numerical

factorization with the same amount of arithmetic is that the calculations must be performed for all the children of a node before those at the parent node can complete. Thus, many different orderings with the same number of floating-point operations can be generated from the elimination tree. In particular, orderings can be chosen for economic use of storage, for efficiency in out-of-core working, or for parallel implementation (see Section 6.3). Additionally, small perturbations to the tree and the number of floating-point operations can accommodate asymmetry, numerical pivoting, or enhanced vectorization.

Liu (1990) presents a survey of the role of elimination trees in Gaussian elimination and discusses the efficient generation of trees (in time effectively linear in the number of entries in the original matrix) as well as the effect of different orderings of the tree and manipulations of the tree that preserve properties of the elimination. For example, the ordering of the tree can have a significant effect on the storage required by the intermediate frontal matrices generated during the elimination. Permissible manipulations on the tree include tree rotations, by means of which the tree can, for example, be made more suitable for driving a factorization that exploits parallelism better (Simon, Vu and Yang 1989).

The multifrontal method can be used on indefinite systems and on matrices that are symmetric in pattern but not in value. The HSL codes MA27 and MA47 are designed for symmetric indefinite systems (Duff and Reid 1982) and provide a stable factorization by using a combination of 1×1 and 2×2 pivots from the diagonal (Bunch, Kaufman and Parlett 1976). The HSL code MA41 (Amestoy and Duff 1989) will solve systems for which the matrix is unsymmetric but it bases its original analysis on the pattern formed from the Boolean summation of the matrix with its transpose. Although this strategy could be poor, it even works for systems whose pattern is quite unsymmetric, as we show in Table 6.7. For matrices which are very structurally unsymmetric (like those in the last two columns of Table 6.7), Davis and Duff (1997*b*) have developed an approach that generalizes the concept of elimination trees and performs a Markowitz style analysis on the unsymmetric pattern while retaining the efficiency of a multifrontal method. We will discuss this further in Section 6.5.

6.1 Performance on Vector Machines

Because multifrontal methods have identical kernels to the frontal methods, one might expect them to perform well on vector machines. However, the dense matrices involved are usually of smaller dimension. There is also a greater amount of data handling outside the kernel, and it is important to consider not only the performance of the kernel but also the assembly operations involved in generating the assembled frontal matrix.

Since a characteristic of frontal schemes is that, even for a matrix of irregular structure, the eliminations are performed using direct addressing and indirect

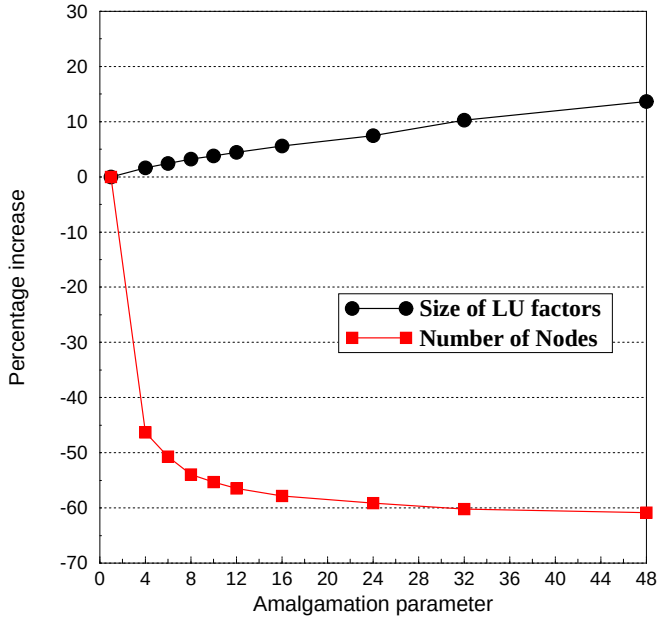

Figure 6.5: **Effect of amalgamation on size of factors and number of nodes**

addressing is only performed at the assembly stage, it is possible to affect the balance of these operations by performing more assemblies before performing eliminations. This node amalgamation was first suggested in the original paper of Duff and Reid (1983) where it was recommended as an aid to performance on vector machines. They parameterized this amalgamation by coalescing a parent node with its eldest child if the number of eliminations originally scheduled for the parent was less than a specified *amalgamation parameter*. Naturally, as this amalgamation parameter increases, the number of nodes will monotonically decrease although, because there is not a complete overlap of the variables in the parent and child, the operations for factorization and number of entries in the factors will increase. We show this effect in Figure 6.5 where the runs were performed by Amestoy using the MA41 code on test matrix BCSSTK15.

The intended effect of node amalgamation with respect to increased efficiency on vector machines is to reduce the number of costly indirect addressing operations albeit at the cost of an increase in the number of elimination operations. We show this effect in Figure 6.6 where we note that, as the amalgamation parameter is increased, there is a much greater reduction in indirect operations than the increase in direct operations so that we might expect amalgamation to be very beneficial on machines where indirect operations are more costly than directly addressed operations. This we illustrate in Figures 6.7 and 6.8. The effect is very marked on the CRAY C90 vector computer (Figure 6.7) but is also noticeable on RISC

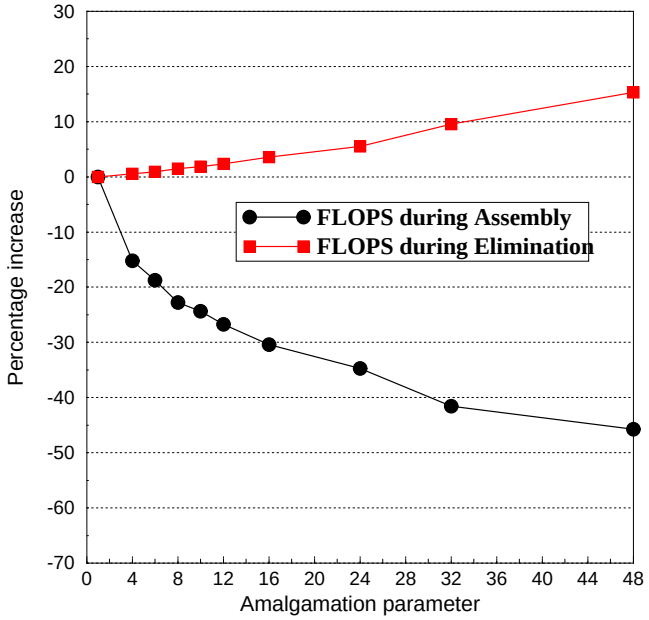

Figure 6.6: **Effect of amalgamation on number of indirect and direct operations**

workstations (Figure 6.8) where the use of indirect addressing causes problems for the management of the cache.

Amestoy and Duff (1989) have used BLAS kernels in both assembly and elimination operations to achieve over 0.75 Gflop/s on one processor of the CRAY C98 for the numerical factorization of problems from a range of applications. This high computational rate is achieved without incurring a significant increase in arithmetic operations. Ashcraft (1987) also reports on high computational rates for the vectorization of a multifrontal code for symmetric systems.

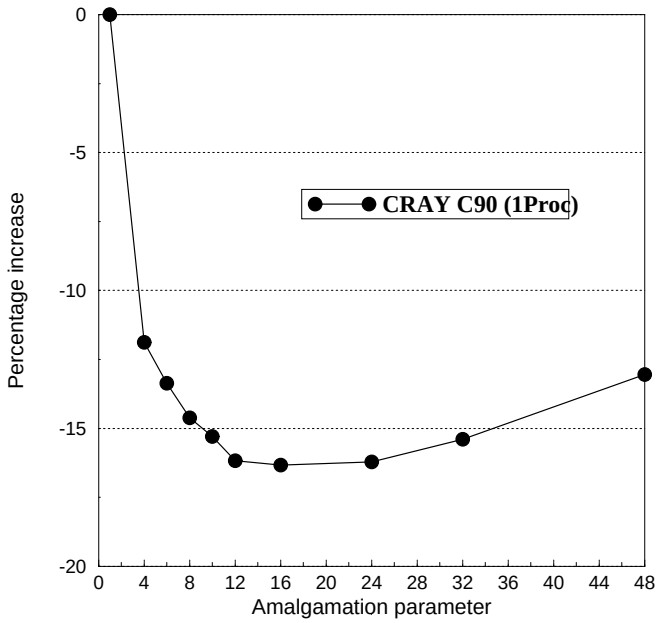

Figure 6.7: **Effect of amalgamation on factorization time on CRAY C90**

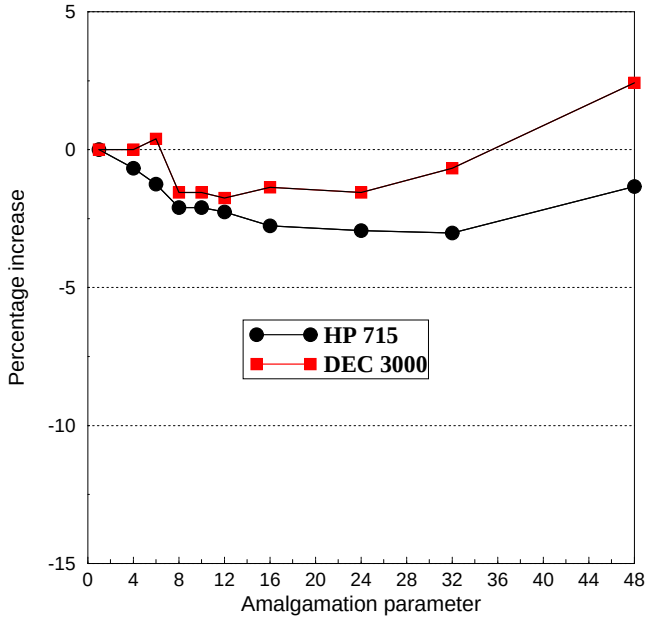

Figure 6.8: **Effect of amalgamation on factorization time on RISC workstations**

6.2 Performance on RISC machines

When we were discussing indirect addressing in Section 3, we remarked that its use in the innermost loops can be inefficient on machines with cache memory. Conversely, dense kernels allow good use of memory hierarchies of the kind found in the modern generation of RISC architecture based machines. We illustrate this with the results shown in Table 6.1.

Table 6.1: **Performance in Mflop/s of multifrontal code MA41 on matrix BCSSTK15 on range of RISC processors**

Computer	Peak	DGEMM	MA41
DEC 3000/400 AXP	133	49	34
HP 715/64	128	55	30
IBM RS6000/750	125	101	64
IBM SP2 (Thin node)	266	213	122
MEIKO CS2-HA	100	43	31

In this table, we show the performance of a tuned version of DGEMM on the machine. This is the Level 3 BLAS routine for (dense) matrix-by-matrix multiplication and is usually the fastest non-trivial kernel for any machine. We feel this is a more meaningful yardstick by which to judge performance than the peak speed, although we show this also in Table 6.1. The MA41 performance ranges from 50 to 70% of the DGEMM performance showing that good use is being made of the Level 3 BLAS.

6.3 Performance on Parallel Machines

Duff and Reid (1983) considered the implementation of multifrontal schemes on parallel computers with shared memory. In this implementation, there are really only three types of tasks, the assembly of information from the children, the selection of pivots, and the subsequent eliminations, although he also allows the eliminations to be blocked so that more than one processor can work on the elimination operations from a single pivot. He chooses to store all the tasks available for execution in a single queue with a label to identify the work corresponding to the task. When a processor is free, it goes to the head of the queue, selects the task there, interprets the label, and performs the appropriate operations. This process may in turn generate other tasks to be added to the end of the queue. This model was used by Duff in designing a prototype parallel code from the HSL code MA37 and we show, in Table 6.2, speedup figures that he obtained on the Alliant FX/8 (Duff 1989*b*). It was this prototype code that was later developed into the code MA41.

Table 6.2: **Speedup on Alliant FX/8 of Five-Point Discretization of Laplace on a 30×30 grid**

No. processors	Time	Speedup
1	2.59	-
2	1.36	1.9
4	.74	3.5
6	.57	4.5
8	.46	5.6

The crucial aspect of multifrontal methods that enables exploitation of parallelism is that work at different nodes of the assembly tree is independent and the only synchronization required is that data from the child nodes must be available before the computation at a node can be completed. The computation at any leaf node can proceed immediately and simultaneously. Although this provides much parallelism near the leaf nodes of the tree, there is less towards the root and, of course, for an irreducible problem there is only one root node. If the nodes of the elimination tree are regarded as atomic, then the level of parallelism reduces to one at the root and usually increases only slowly as we progress away from the root. If, however, we recognize that parallelism can be exploited within the calculations at each node (corresponding to one or a few steps of Gaussian elimination on a dense submatrix), much greater parallelism can be achieved. This loss of parallelism by regarding nodes as atomic is compounded by the fact that most of the floating-point arithmetic is performed near the root so that it is vital to exploit parallelism also within the computation at each node. We illustrate the amount of parallelism available from the assembly tree by the results in Table 6.3, where we show the size and number of tree nodes at the leaves and near the root of the tree. Thus we see that while there is much parallelism at the beginning of the factorization, there is much less near the root, if the tree nodes are regarded as atomic. Furthermore, about 75% of the work in the factorization is performed within these top three levels. However, the size of the frontal matrices are much larger near the root so we can exploit parallelism at this stage of the factorization by, for example, using parallel variants of the Level 3 BLAS for the elimination operations within a node. The effect of this is seen clearly in the results from Amestoy, Daydé, Duff and Morère (1995) shown in Table 6.4, where the increased speedups in columns (2) are due to exploiting parallelism within the node.

The code used in the experiments in Table 6.4 was the MA41 code (Amestoy and Duff 1993, Amestoy and Duff 1997) which was developed for shared memory parallel computers. It should be emphasized that, because of the portability provided through the use of the BLAS, the MA41 code is essentially identical for all shared

Table 6.3: **Statistics on front sizes in tree**

			Leaf nodes		Top 3 levels	
Matrix	Order	Tree nodes	Number	Av. size	Number	Av. size
BCSSTK15	3948	576	317	13	10	376
BCSSTK33	8738	545	198	5	10	711
BBMAT	38744	5716	3621	23	10	1463
GRE1107	1107	344	250	7	12	129
SAYLR4	3564	1341	1010	5	12	123
GEMAT11	4929	1300	973	10	112	148

Table 6.4: **Performance summary of the multifrontal factorization using MA41 on 7 processors of a CRAY C98.** In columns (1) we exploit only parallelism from the tree; in columns (2) we combine the two levels of parallelism

			(1)		(2)	
Matrix	order	entries	Mflop/s	(speedup)	Mflop/s	(speedup)
WANG3	26064	177168	1062	(1.42)	3718	(4.98)
WANG4	26068	177196	1262	(1.70)	3994	(5.39)
BBMAT	38744	1771722	2182	(3.15)	3777	(5.46)

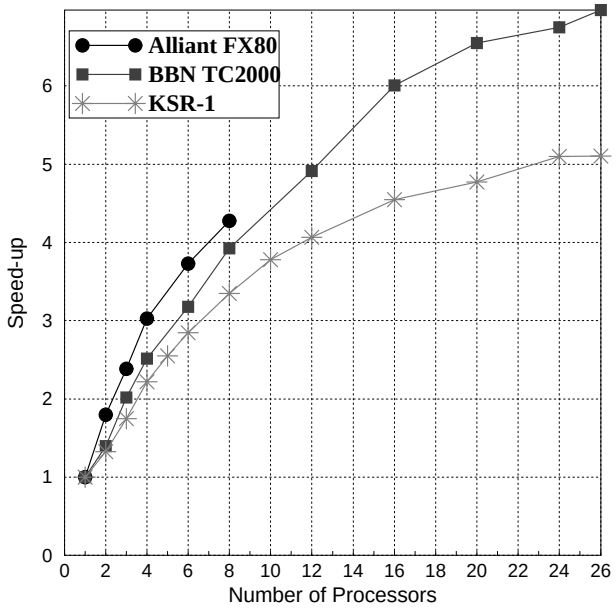

Figure 6.9: **Speedup graph**

memory machines. MA41 will also run with relatively minor changes on a virtual shared memory machine although it is beneficial to make more significant changes for efficiency. This is seen in Figure 6.9, where a significant modification to the virtual shared memory implementation was to perform some explicit copying of subarrays into the private memories and to use BLAS locally on each processor (Amestoy et al. 1995).

Multifrontal codes have also been developed for distributed memory machines. An efficient data parallel version of a multifrontal code has been implemented by Conroy, Kratzer and Lucas (1994). The most impressive times and speedups of which we are aware have been obtained by Gupta and Kumar (1994) for an implementation of a multifrontal code on a 1024-processor nCUBE-2 machine. We include some of their results for the matrix BCSSTK15 in Table 6.5. For larger matrices, they show even better performance with a speedup of over 350 on 1024 processors for matrix BCSSTK31 of order 35588 with about 6.4 million entries. In subsequent experiments, Gupta, Karypis and Kumar (1996) have implemented versions of their algorithms on a CRAY T3D, using SHMEM for message passing, and have obtained a high performance and good speedup on this machine also.

Clearly the parallelism available through the elimination tree depends on the ordering of the matrix. In general, short bushy trees are preferable to tall thin ones since the number of levels determines the inherent sequentiality of the computation. Two common ordering strategies for general sparse symmetric systems are minimum

Table 6.5: **Performance of code of Gupta and Kumar on nCUBE on matrix BCSSTK33.** Times in seconds. Results taken from Gupta and Kumar (1994)

Number of processors	1	4	16	64	256
Time	103.7	26.7	8.3	3.2	1.5
Speedup	1.00	3.9	12.5	32.4	67.8
Efficiency %	100	97	78	51	27
Load balance %	100	98	91	87	84

degree and nested dissection (see, for example, Duff et al. (1986) and George and Liu (1981)). Although these orderings are similar in behavior for the amount of arithmetic and storage, they give fairly different levels of parallelism when used to construct an elimination tree. We illustrate this point in Table 6.6, where the maximum speedup is computed from a simulation of the elimination as the ratio of the number of operations in the sequential algorithm to the number of sequential operations in the parallel version, with account taken of data movement as well as floating-point calculations (Duff and Johnsson 1989).

Table 6.6: **Comparison of two orderings for generating an elimination tree for multifrontal solution** (the problem is generated by a 5-point discretization of a 10×100 grid)

.

Ordering	Minimum degree	Nested dissection
Number of levels in tree	52	15
Number of pivots on longest path	232	61
Maximum speedup	9	47

6.4 Exploitation of Structure

The multifrontal methods that we have just described generate the assembly tree and corresponding guiding information for the numerical factorization without access to numerical values and assume that the pivot order thus selected will be numerically suitable. For matrices which are not positive definite, this is not necessarily the case, so the numerical factorization must be able to tolerate enforced changes to the predicted sequence by performing additional pivoting. For general symmetric systems, this is usually not much of a problem and the overheads of the additional

pivoting are fairly low. However, symmetric systems of the form

$$\begin{pmatrix} H & A \\ A^T & 0 \end{pmatrix}, \tag{6.3}$$

called augmented systems, occur frequently in many application areas (Duff 1994). It is necessary to exploit the structure during the symbolic analysis so that pivots are not chosen from the zero block and it is preserved during the factorization. The effect of taking this structure into account can be very dramatic. For example, on a matrix of the form (6.3) with H a diagonal matrix of order 1028 with 504 ones and 524 zeros and A the 1028 x 524 matrix FFFFF800 from the collection in Gay (1985), an earlier multifrontal code which does not exploit the zero-block structure (namely HSL code MA27) forecasts 1.5 million floating-point operations but requires 16.5 million. The HSL code MA47 (Duff and Reid 1995, Duff and Reid 1996*b*), however, forecasts and requires only 7,954 flops. Unfortunately, the new code is much more complicated and involves more data movement since frontal matrices are not necessarily absorbed by the parent and can percolate up the tree. Additionally, the penalty for straying from the forecast pivot sequence can be very severe.

6.5 Unsymmetric Multifrontal Methods

Although the multifrontal schemes that we have just described are designed for structurally symmetric matrices, structurally unsymmetric matrices can be handled by explicitly holding zero entries. It is more complicated to design an efficient multifrontal scheme for matrices that are asymmetric in structure. The main difference is that the elimination cannot be represented by an (undirected) tree but a directed acyclic graph (DAG) is required (Eisenstat and Liu 1992). The frontal matrices are, of course, no longer square and, as in the case discussed in the previous section, they are not necessarily absorbed at the parent node and can persist in the DAG. Finally the complication of *a posteriori* numerical pivoting is even more of a problem with this scheme so that the approach adopted is normally to take account of the real values when computing the DAG and the pivot order.

An unsymmetric multifrontal code by Davis and Duff, based on this approach, is included as subroutine MA38 in the Harwell Subroutine Library. We show a comparison, taken from Davis and Duff (1997*b*), of this code with the "symmetric" HSL code MA41 in Table 6.7.

The problems in Table 6.7 are arranged in order of increasing asymmetry, where the asymmetry index is defined as

$$\frac{\text{Number of pairs such that } a_{ij} = 0, a_{ji} \neq 0}{\text{Total number of off-diagonal entries}}, \tag{6.4}$$

so that a symmetric matrix has asymmetry index 0.0. These results show clearly the benefits of treating the asymmetry explicitly.

Table 6.7: **Comparison of "symmetric" (MA41) and "unsymmetric" (MA38) code**

Order	13535	62424	26064	22560	120750	36057
Entries	390607	1717792	177168	1014951	1224224	227628
Index of asymmetry (6.4)	0.00	0.00	0.00	0.36	0.76	0.89
Floating-point ops (10^9)						
MA41	0.3	2.3	10.4	2.9	38.2	0.6
MA38	3.8	5.3	62.2	9.0	7.0	0.2
Factorization time (*seconds on Sun ULTRA*)						
MA41	8	46	174	81	809	19
MA38	85	127	1255	226	220	11

We show a comparison of the HSL MA38 code with HSL codes MA41 and MA48 in Table 6.8, taken from Davis and Duff (1997*a*). These results show that the new code can be very competitive, sometimes outperforming the other codes.

Table 6.8: **Comparison of MA38 with MA48 and MA41 on a few test matrices.** Times in seconds on a Sun ULTRA-1 workstation

Matrix	Order	No entries	Factorization Time		
			MA38	MA48	MA41
AV41092	41092	1683902	1618	1296	254
PSMIGR_1	3140	543162	198	179	188
ONETONE1	36057	341088	57	110	189
LHR71	70304	1528092	93	287	996

7 Other Approaches for Exploitation of Parallelism

Although we feel that the multifrontal approach is very suitable for exploitation of parallelism, it is certainly not the only approach being pursued. Indeed, the Cholesky algorithm viewed as a left-looking algorithm can be implemented for sparse systems and can also be blocked by using a supernodal formulation similar to the node amalgamation that we discussed in Section 6. A code based on this approach attained very high performance on some structural analysis and artificially generated problems on a CRAY Y-MP (Simon et al. 1989). A variant of the standard column-oriented sparse Cholesky algorithm has also been implemented

on hypercubes (George, Heath, Liu and Ng 1988, George, Heath, Liu and Ng 1989). Highly efficient codes based on a supernodal factorization for MIMD machines, in particular for an INTEL Paragon, have been developed by Rothberg (1994).

The supernodal concept has recently been extended to unsymmetric systems by Demmel, Eisenstat, Gilbert, Li and Liu (1995). It is now not possible to use Level 3 BLAS efficiently. However, Demmel et al. (1995) have developed an implementation that performs a dense matrix multiplication of a block of vectors and, although these cannot be written as another dense matrix, they show that this Level 2.5 BLAS has most of the performance characteristics of Level 3 BLAS since the repeated use of the same dense matrix allows good use of cache and memory hierarchy. In Table 7.1 we compare their code, SuperLU, with the multifrontal approach on a range of examples. The multifrontal code MA41 was used if the asymmetry index of the matrix was less than 0.5 and the code MA38 was used otherwise. This seems to support the premise that the simpler multifrontal organization performs better although it is important to use the *very* unsymmetric code, MA38, when the matrix is very structurally asymmetric.

Table 7.1: **Comparison of multifrontal against supernodal approach.** Times in seconds on a Sun ULTRA-1 workstation

Matrix	Order	Entries	Analyze and Factorize Time (secs)		Entries in LU factors (10^6)	
			SuperLU	Multif	SuperLU	Multif
ONETONE2	36057	227628	9	11	1.3	1.3
TWOTONE	120750	1224224	758	221	24.7	9.8
WANG3	26024	177168	1512	174	27.0	11.4
VENKAT50	62424	1717792	172	46	18.0	11.9
RIM	22560	1014951	78	80	9.7	7.4
GARON2	13535	390607	60	8	5.1	2.4

A quite different approach to designing a parallel code is more related to the general approach discussed in Section 3. In this technique, when a pivot is chosen all rows with entries in the pivot column and all columns with entries in the pivot row are marked as ineligible and a subsequent pivot can only be chosen from the eligible rows and columns. In this way, a set of say k independent pivots is chosen. This set of pivots not affect each other and can be used in parallel. In other words, if the pivots were permuted to the beginning of the matrix, this pivot block would be diagonal. The resulting elimination operations are performed in parallel using a rank k update. This is similar to the scheme of Davis and Yew discussed in Section 3.4. We show some results of this approach taken from van der Stappen, Bisseling and van de Vorst (1993) in Table 7.2, where it is clear that good speedups can be obtained.

Table 7.2: **Results from van der Stappen, Bisseling and van de Vorst (1993) on a PARSYTEC SuperCluster FT-400.** Times in seconds for *LU* factorization.

Matrix	Order	Entries	Number of processors			
			1	16	100	400
SHERMAN 2	1080	23094	1592	108	32.7	15.6
LNS 3937	3937	25407	2111	168	37.9	23.7

8 Software

Although much software is available that implements direct methods for solving sparse linear systems, little is within the public domain. There are several reasons for this situation, the principal ones being that sparse software often is encapsulated within much larger packages (for example, for structural analysis) and that much work on developing sparse codes is funded commercially so that the fruits of this labor often require licenses. There are also several research codes that can be obtained from the authors but, since these usually lack full documentation and often require the support of the author, we do not discuss these here.

Among the public domain sparse software are some routines from the Collected Algorithms of ACM (available from `netlib`), mostly for matrix manipulation (for example, bandwidth reduction, ordering to block triangular form) rather than for equation solution, although the NSPIV code from Sherman (1978) is available as Algorithm 533.

Both Y12M and the HSL code MA28, referenced in this chapter, are available from `netlib`, although people obtaining MA28 in this way are still required to sign a license agreement, and use of the newer HSL code MA48 is recommended. A freely available research version of MA38, called UMFPACK, which includes a version for complex matrices, is available in `netlib` as is the C code SuperLU implementing the supernodal factorization of Demmel et al. (1995). There is also a skeleton sparse *LU* code from Banks and Smith in the *misc* collection in `netlib`, and Joseph Liu distributes his multiple minimum-degree code upon request.

Among the codes available under license are those from the Harwell Subroutine Library that were used to illustrate many of the points in this chapter, a subset of which is also marketed by NAG under the title of the Harwell Sparse Matrix Library. Contact details for these organizations can be found in Appendix A.2. The IMSL Library also has codes for the direct solution of sparse systems, and a sparse *LU* code is available in the current release of ESSL for the IBM RS/6000 and SP2 computers. Sparse linear equation codes are also available to users of Cray computers upon request to Cray Research Inc.

Other packages include the SPARSPAK package, primarily developed at the

University of Waterloo (George, Liu and Ng 1980), which solves both linear systems and least-squares problems, and routines in the PORT library from Bell Labs (Kaufman 1982), details of which can be obtained from `netlib`. Versions of the package YSMP, developed at Yale University (Eisenstat, Gursky, Schultz and Sherman 1982), can be obtained from Scientific Computing Associates at Yale, who also have several routines implementing iterative methods for sparse equations.

A public domain version of SuperLU for shared memory parallel computers, SuperLU_MT, is available from the Berkeley and the MA41 code from HSL that we have discussed in this paper, also has a version for shared memory computers. The only public domain software for distributed memory machines that we are aware of is the CAPSS code by Heath and Raghavan (1995,1997), which is included in the SCALAPACK package. Gupta is freely distributing a non-machine specific source code version of the WSSMP code (Gupta, Joshi and Kumar 1997) that is available under license for the IBM SP2. Koster and Bisseling (1994) plan a public release of their parallel Markowitz/threshold solver, SPLU, designed for message passing architectures, while the EU LTR Project PARASOL (http://192.129.37.12/parasol/) plans to develop a suite of direct and iterative sparse solvers for message passing architectures that are primarily targeted at finite-element applications.

It should be stressed that we have been referring to fully supported products. Many other codes are available that are either at the development stage or are research tools (for example, the SMMS package of Fernando Alvarado at Wisconsin (Alvarado 1989)) and the SPARSKIT package of Yousef Saad at Minneapolis (Saad 1994).

9 Brief Summary

We have discussed several approaches to the solution of sparse systems of equations, with particular reference to their suitability for the exploitation of vector and parallel architectures. We see considerable promise in both frontal and multifrontal methods on vector machines and reasonable possibilities for exploitation of parallelism by supernodal and multifrontal methods. A principal factor in attaining high performance is the use of dense matrix computational kernels, which have proved extremely effective in the dense case.

Finally, we have tried to keep the narrative flowing in this presentation by avoiding an excessive number of references. For such information, we recommend the recent review by Duff (1997), where 215 references are listed.

Acknowledgment

We would like to thank Jennifer Scott and John Reid of the Rutherford Appleton Laboratory for helpful comments on an earlier draft.

References

Alaghband, G. (1989), 'Parallel pivoting combined with parallel reduction and fill-in control', *Parallel Computing* **11**, 201–221.

Alvarado, F. L. (1989), 'Manipulation and visualisation of sparse matrices', *ORSA J. Computing* **2**, 186–207.

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

Amestoy, P. R. and Duff, I. S. (1993), 'Memory management issues in sparse multifrontal methods on multiprocessors', *Int J. Supercomputer Applications* **7**, 64–82.

Amestoy, P. R. and Duff, I. S. (1997), MA41: a parallel package for solving sparse unsymmetric sets of linear equations, Technical Report (to appear), CERFACS, Toulouse, France.

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

Amestoy, P. R., Daydé, M. J., Duff, I. S. and Morère, P. (1995), 'Linear algebra calculations on a virtual shared memory computer', *Int J. High Speed Computing* **7**(1), 21–43.

Arioli, M. and Duff, I. S. (1990), Experiments in tearing large sparse systems, *in* M. G. Cox and S. Hammarling, eds, 'Reliable Numerical Computation', Oxford University Press, Oxford, pp. 207–226.

Ashcraft, C. (1987), A vector implementation of the multifrontal method for large sparse, symmetric positive definite linear systems, Report ETA-TR-51, ETA.

Ashcraft, C. and Liu, J. W. H. (1996), Robust ordering of sparse matrices using multisection, Technical Report ISSTECH-96-002, Boeing Information and Support Services, Seattle. Also Report CS-96-01, Department of Computer Science, York University, Ontario, Canada.

Benner, R. E., Montry, G. R. and Weigand, G. G. (1987), 'Concurrent multifrontal methods: shared memory, cache, and frontwidth issues', *Int J. Supercomputer Applications* **1**, 26–44.

Bunch, J. R., Kaufman, L. and Parlett, B. N. (1976), 'Decomposition of a symmetric matrix', *Numerische Mathematik* **27**, 95–110.

Cliffe, K. A., Duff, I. S. and Scott, J. A. (1998), 'Performance issues for frontal schemes on a cache-based high performance computer', *Int J. Numerical Methods in Engineering* **42**, 127–143.

Conroy, J. M., Kratzer, S. G. and Lucas, R. F. (1994), Data-parallel sparse matrix factorization, *in* J. G. Lewis, ed., 'Proceedings 5th SIAM Conference on Linear Algebra', SIAM Press, Philadelphia, pp. 377–381.

Davis, T. A. and Duff, I. S. (1997*a*), A combined unifrontal/multifrontal method for unsymmetric sparse matrices, Technical Report RAL-TR-97-046, Rutherford Appleton Laboratory. Also appeared as CERFACS Report TR/PA/97/34 and CISE Dept. University of Florida Report TR-97-016. To appear in *ACM Trans. Math. Softw.*

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

Davis, T. A. and Yew, P. C. (1990), 'A nondeterministic parallel algorithm for general unsymmetric sparse LU factorization', *SIAM J. Matrix Analysis and Applications* **11**, 383–402.

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 Division, U. C. Berkeley, Berkeley, California.

Duff, I. S. (1981), MA32 – A package for solving sparse unsymmetric systems using the frontal method, Technical Report AERE R11009, Her Majesty's Stationery Office, London.

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

Duff, I. S. (1984*b*), The solution of sparse linear equations on the CRAY-1, *in* J. S. Kowalik, ed., 'Proceedings of the NATO Advanced Research Workshop on High-Speed Computation, held at Julich, Federal Republic of Germany, June 20-22, 1983', NATO ASI series. Series F, Computer and Systems Sciences; Vol. 7, Springer-Verlag, Berlin, pp. 293–309.

Duff, I. S. (1986*a*), 'Parallel implementation of multifrontal schemes', *Parallel Computing* **3**, 193–204.

Duff, I. S. (1986*b*), The parallel solution of sparse linear equations, *in* W. Handler, D. Haupt, R. Jeltsch, W. Juling and O. Lange, eds, 'CONPAR 86: Conference on Algorithms and Hardware for Parallel Processing, Aachen, September 17–19, 1986: Proceedings', Lecture Notes in Computer Science 237, Springer-Verlag, Berlin, pp. 18–24.

Duff, I. S. (1989*a*), 'Multiprocessing a sparse matrix code on the Alliant FX/8', *J. Comput. Appl. Math.* **27**, 229–239.

Duff, I. S. (1989*b*), Parallel algorithms for sparse matrix solution, *in* D. J. Evans and C. Sutti, eds, 'Parallel computing. Methods algorithms and applications', Adam Hilger Ltd., Bristol, pp. 73–82.

Duff, I. S. (1994), The solution of augmented systems, *in* D. F. Griffiths and G. A. Watson, eds, 'Numerical Analysis 1993, Proceedings of the 15th Dundee Conference, June-July 1993', Pitman Research Notes in Mathematics Series. **303**, Longman Scientific & Technical, Harlow, England, pp. 40–55.

Duff, I. S. (1997), Sparse numerical linear algebra: direct methods and preconditioning, *in* I. S. Duff and G. A. Watson, eds, 'The State of the Art in Numerical Analysis', Oxford University Press, Oxford, pp. 27–62.

Duff, I. S. and Johnsson, S. L. (1989), Node orderings and concurrency in structurally-symmetric sparse problems, *in* G. F. Carey, ed., 'Parallel Supercomputing: Methods, Algorithms and Applications', J. Wiley and Sons, New York, pp. 177–189.

Duff, I. S. and Reid, J. K. (1982), MA27 – A set of Fortran subroutines for solving sparse symmetric sets of linear equations, Technical Report AERE R10533, Her Majesty's Stationery Office, London.

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 systems', *SIAM J. Scientific and Statistical Computing* **5**, 633–641.

Duff, I. S. and Reid, J. K. (1995), MA47, a Fortran code for direct solution of indefinite sparse symmetric linear systems, Technical Report RAL 95-001, Rutherford Appleton Laboratory.

Duff, I. S. and Reid, J. K. (1996*a*), '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 Reid, J. K. (1996*b*), 'Exploiting zeros on the diagonal in the direct solution of indefinite sparse symmetric linear systems', *ACM Trans. Math. Softw.* **22**(2), 227–257.

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. G. 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), 'The design of a new frontal code for solving sparse unsymmetric systems', *ACM Trans. Math. Softw.* **22**(1), 30–45.

Duff, I. S. and Scott, J. A. (1997), A comparison of frontal software with other Harwell Subroutine Library sparse direct solvers, Technical Report RAL-TR-96-102 (Revised), Rutherford Appleton Laboratory. To appear in *High Performance Solution to Structured Matrix Problems*. Edited by Arbenz, Paprzycki, Sameh and Sarin, NOVA Science Publishers, Inc.

Duff, I. S. and van der Vorst, H. A. (1998), Preconditioning and parallel preconditioning, Technical Report RAL-TR-1998-052, Rutherford Appleton Laboratory.

Duff, I. S., Erisman, A. M. and Reid, J. K. (1986), *Direct Methods for Sparse Matrices*, Oxford University Press, Oxford, England.

Duff, I. S., Erisman, A. M., Gear, C. W. and Reid, J. K. (1988), 'Sparsity structure and Gaussian elimination', *SIGNUM Newsletter* **23**(2), 2–8.

Duff, I. S., Gould, N. I. M., Lescrenier, M. and Reid, J. K. (1990), The multifrontal method in a parallel environment, *in* M. G. Cox and S. Hammarling, eds, 'Reliable Numerical Computation', Oxford University Press, Oxford, pp. 93–111.

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

Duff, I. S., Grimes, R. G. and Lewis, J. G. (1992), Users' guide for the Harwell-Boeing sparse matrix collection (Release I), Technical Report RAL 92-086, Rutherford Appleton Laboratory.

Duff, I. S., Grimes, R. G. and Lewis, J. G. (1997*a*), The Rutherford-Boeing Sparse Matrix Collection, Technical Report RAL-TR-97-031, Rutherford Appleton Laboratory. Also Technical Report ISSTECH-97-017 from Boeing Information & Support Services and Report TR/PA/97/36 from CERFACS, Toulouse.

Duff, I. S., Marrone, M., Radicati, G. and Vittoli, C. (1997*b*), 'Level 3 Basic Linear Algebra Subprograms for sparse matrices: a user level interface', *ACM Trans. Math. Softw.* **23**(3), 379–401.

Duff, I. S., Reid, J. K. and Scott, J. A. (1989*b*), 'The use of profile reduction algorithms with a frontal code', *Int J. Numerical Methods in Engineering* **28**, 2555–2568.

Eisenstat, S. C. and Liu, J. W. H. (1992), 'Exploiting structural symmetry in unsymmetric sparse symbolic factorization', *SIAM J. Matrix Analysis and Applications* **13**, 202–211.

Eisenstat, S. C., Gursky, M. C., Schultz, M. H. and Sherman, A. H. (1982), 'Yale sparse matrix package, I: The symmetric codes', *Int J. Numerical Methods in Engineering* **18**, 1145–1151.

Gay, D. M. (1985), 'Electronic mail distribution of linear programming test problems', Mathematical Programming Society. COAL Newsletter.

George, A. and Heath, M. T. (1980), 'Solution of sparse linear least squares problems using Givens rotations', *Linear Algebra and its Applications* **34**, 69–83.

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.

George, A. and Ng, E. (1985), 'An implementation of Gaussian elimination with partial pivoting for sparse systems', *SIAM J. Scientific and Statistical Computing* **6**, 390–409.

George, A., Heath, M. T., Liu, J. W. H. and Ng, E. (1988), 'Sparse Cholesky factorization on a local-memory multiprocessor', *SIAM J. Scientific and Statistical Computing* **9**, 327–340.

George, A., Heath, M. T., Liu, J. W. H. and Ng, E. (1989), 'Solution of sparse positive definite systems on a hypercube', *J. Comput. Appl. Math.* **27**, 129–156.

George, A., Liu, J. W. H. and Ng, E. G. (1980), User's guide for SPARSPAK: Waterloo sparse linear equations package, Technical Report CS-78-30 (Revised), University of Waterloo, Canada.

Gupta, A. and Kumar, V. (1994), A scalable parallel algorithm for sparse matrix factorization, Technical Report TR-94-19, Department of Computer Science, University of Minnesota.

Gupta, A., Joshi, M. and Kumar, V. (1997), WSSMP: Watson Symmetric Sparse Matrix Package. Users Manual: Version 2.0β, Technical Report RC 20923 (92669), IBM T. J. Watson Research Centre, P. O. Box 218, Yorktown Heights, NY 10598.

Gupta, A., Karypis, G. and Kumar, V. (1996), Highly scalable parallel algorithms for sparse matrix factorization, Technical Report UMSI 96/97, University of Minnesota, Supercomputer Institute.

Heath, M. T. and Raghavan, P. (1995), 'A Cartesian parallel nested dissection algorithm', *SIAM J. Matrix Analysis and Applications* **16**(1), 235–253.

Heath, M. T. and Raghavan, P. (1997), 'Performance of a fully parallel sparse solver', *Int J. Supercomputer Applications* **11**(1), 49–64.

Hockney, R. and Jessup, C. (1988), *Parallel Computers 2: Architecture, Programming and Algorithms*, Adam Hilger/IOP Publishing, Bristol, United Kingdom.

Hood, P. (1976), 'Frontal solution program for unsymmetric matrices', *Int J. Numerical Methods in Engineering* **10**, 379–400.

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', *Int J. Numerical Methods in Engineering* **2**, 5–32.

Kaufman, L. (1982), Usage of the sparse matrix programs in the PORT library, Technical Report Report 105, Bell Laboratories, Murray Hill, New Jersey.

Koster, J. and Bisseling, R. H. (1994), 'Parallel sparse LU decomposition on a distributed-memory multiprocessor'. Submitted to *SIAM J. Scientific Computing*.

Liu, J. W. H. (1988), 'The minimum degree ordering with constraints', *SIAM J. Scientific and Statistical Computing* **10**, 1136–1145.

Liu, J. W. H. (1990), 'The role of elimination trees in sparse factorization', *SIAM J. Matrix Analysis and Applications* **11**, 134–172.

Lucas, R., Blank, T. and Tiemann, J. (1987), 'A parallel method for large sparse systems of equations', *IEEE Trans. on Computer-Aided Design* **CAD-6**, 981–991.

Markowitz, H. M. (1957), 'The elimination form of the inverse and its application to linear programming', *Management Science* **3**, 255–269.

Parter, S. V. (1961), 'The use of linear graphs in gaussian elimination', *SIAM Review* **3**, 119–130.

Rothberg, E. (1994), Efficient sparse Cholesky factorization on distributed-memory multiprocessors, *in* J. G. Lewis, ed., 'Proceedings 5th SIAM Conference on Linear Algebra', SIAM Press, Philadelphia, p. 141.

Rothberg, E. (1996), Exploring the tradeoff between imbalance and separator size in nested dissection ordering, Technical Report Unnumbered, Silicon Graphics Inc.

Saad, Y. (1994), SPARSKIT: a basic tool kit for sparse matrix computations. Version 2, Technical report, Computer Science Department, University of Minnesota.

Sherman, A. H. (1978), 'Algorithm 533. NSPIV, A Fortran subroutine for sparse Gaussian elimination with partial pivoting', *ACM Trans. Math. Softw.* **4**, 391–398.

Simon, H. D., Vu, P. and Yang, C. (1989), Performance of a supernodal general sparse solver on the CRAY Y-MP: 1.68 Gflops with autotasking, Report SCA-TR-117, Boeing Computer Services, Seattle.

Sloan, S. W. and Randolph, M. F. (1983), 'Automatic element reordering for finite-element analysis with frontal schemes', *Int J. Numerical Methods in Engineering* **19**, 1153–1181.

Tinney, W. F. and Walker, J. W. (1967), 'Direct solutions of sparse network equations by optimally ordered triangular factorization', *Proc. of the IEEE* **55**, 1801–1809.

van der Stappen, A. F., Bisseling, R. H. and van de Vorst, J. G. G. (1993), 'Parallel sparse LU decomposition on a mesh network of transputers', *SIAM J. Matrix Analysis and Applications* **14**, 853–879.

Zlatev, Z., Waśniewski, J. and Schaumburg, K. (1981), *Y12M - Solution of Large and Sparse Systems of Linear Algebraic Equations*, Vol. 121 of *Lecture Notes in Computer Science*, Springer-Verlag, New York.