NUMERICAL LINEAR ALGEBRA WITH APPLICATIONS
Numer. Linear Algebra Appl. (2012)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/nla.1810

An efficient analyse phase for element problems

Jonathan D. Hogg and Jennifer A. Scott*,†

Computational Science and Engineering Department, STFC Rutherford Appleton Laboratory, Harwell Oxford, Didcot OX11 0QX, UK

SUMMARY

The analyse phase of a sparse direct solver for symmetrically structured linear systems of equations is used to determine the sparsity pattern of the matrix factor. This allows the subsequent numerical factorisation and solve phases to be executed efficiently. Many direct solvers require the system matrix to be in assembled form. For problems arising from finite element applications, assembling and then using the system matrix can be costly in terms of both time and memory. This paper describes and implements a variant of the work of Gilbert, Ng and Peyton for matrices in elemental form. The proposed variant works with an *equivalent matrix* that avoids explicitly assembling the system matrix and exploits supervariables. Numerical experiments using problems from practical applications are used to demonstrate the significant advantages of working directly with the elemental form. Copyright © 2012 John Wiley & Sons, Ltd.

Received 24 November 2010; Revised 11 October 2011; Accepted 30 October 2011

KEY WORDS: sparse symmetric linear systems; direct solver; analyse phase; element problems; supervariables; supernodes

1. INTRODUCTION

The solution of sparse symmetric linear systems of equations

$$A\boldsymbol{x} = \boldsymbol{b}$$

using a direct method is a well-established and important problem. Most sparse direct solvers use a classical four-phase approach: first, a fill-reducing ordering is found; next, an analysis is performed using the sparsity pattern of A to establish the work flow and data structures for the numerical factorisation phase; then, the matrix is factorised; finally, the solve phase uses the computed factors to solve for one or more right-hand sides $\boldsymbol{b}$. We remark that some solvers optionally combine the ordering and analyse phase, but in this paper, we focus on the analyse phase.

Key objectives of the analyse phase are to identify supernodes (sets of columns of L with similar sparsity patterns) that will allow the exploitation of high-level Basic Linear Algebra Subroutines (BLAS) during the subsequent factorisation to determine the supernode index lists (i.e. the nonzero pattern of the factors) and to determine an assembly tree that will be used to guide the numerical factorisation. The determination of supernode index lists may be performed during the factorisation, in which case the number of nonzeros in each column of the factors is normally determined by the analyse phase. Other tasks the analyse phase may perform include modifying the ordering of the assembly tree to minimise memory requirements in a multifrontal method, reordering variables within supernodes to increase cache locality during the factorisation and handling errors in the user-supplied data (including out-of-range indices and duplicate entries).

*Correspondence to: Jennifer A. Scott, Computational Science and Engineering Department, STFC Rutherford Appleton Laboratory, Harwell Oxford, Didcot OX11 0QX, UK.

†E-mail: jennifer.scott@stfc.ac.uk

In this paper, our aim is to design and implement an efficient analyse phase that can be used in the development of modern sparse direct solvers. Our main interest is in element problems but we want to allow both matrices that are held in assembled form (input by columns) and in elemental form (input by elements). In each case, we want to exploit supervariables (columns of A with the same sparsity pattern) and we want to keep memory bandwidth to a minimum, while not compromising efficiency. Thus, our main contributions are the incorporation of supervariables into the Gilbert, Ng and Peyton [1] approach to the analyse phase for assembled problems and the design and implementation of an efficient analyse phase for elemental problems that avoids explicitly assembling the matrix pattern. Our new analyse phase is available within the HSL software library [2] as package `HSL_MC78`. We remark that, historically, the analyse phase was much faster than the factorisation phase. Considerable effort has gone into parallelizing the factorisation so that the gap between the times for the two phases has narrowed. It is therefore important that the analyse phase be implemented efficiently to prevent it from becoming a bottleneck.

This paper is organised as follows. In the rest of this introduction, we introduce the terms and notation that we will use throughout the paper and we present our test problems and our test environment. Then, in Section 2, we summarise and briefly discuss key algorithms used within the analyse phase. Section 3 focuses on the case when A is in assembled form and looks at efficiently identifying supervariables, incorporating supervariables into constructing the elimination tree and into the column count algorithm of Gilbert, Ng and Peyton [1]; numerical results illustrate the potential savings resulting from exploiting supervariables. In Section 4, we consider how to handle problems in elemental form without explicitly assembling the sparsity pattern of A. Results for problems arising from finite-element applications are presented. Timings for the analyse phase of one of our sparse direct solvers that incorporates our new analyse code are presented in Section 5, along with timings for other state-of-the-art packages. Finally, in Section 6, our findings are summarised.

1.1. Terms and notation

We assume a basic knowledge of the steps involved in sparse Cholesky factorisation and with the use of graphs in these algorithms (see, e.g. [3]). However, as the terminology used in the sparse matrix literature is not always consistent, in this section, we define the terms and introduce the notation we will employ throughout the remainder of the paper; we will use these when discussing both our own and others' work.

Given a sparse $n \times n$ symmetric matrix A with ne nonzero entries, we describe an analyse phase that seeks to find the sparsity pattern of the Cholesky factor L of the matrix PAP^T, where P is a user-supplied fill-reducing permutation. To simplify notation, we will assume throughout that $P = I$ (i.e. the pivot order is the natural order $1, 2, \ldots, n$), but all the algorithms can be written in terms of a general P, and in our numerical experiments, a fill-reducing permutation is used.

Sparse matrices are normally held using one of the two forms:

(1.) In *assembled* form $A = \{a_{ij}\}$ where only the nonzero entries a_{ij} are stored using, for example, coordinate format or compressed sparse row or column storage (see, for instance, [4]).
(2.) As a sum of $nelt$ element matrices

$$A = \sum_{k}^{nelt} A^{(k)},$$

where $A^{(k)}$ is nonzero only in those rows and columns that correspond to variables in the $k-$th element. We refer to this as *elemental* form. For each k, an integer list $\mathcal{E}_k$ of length $nvar_k$ specifies which columns of A are associated with $A^{(k)}$, and an $nvar_k \times nvar_k$ array is used to hold $A^{(k)}$.

We define $\mathcal{A}_j$ to be the set of row indices of the entries on or below the diagonal of column j of A,

$$\mathcal{A}_j = \{i : i \geqslant j, a_{ij} \neq 0\}.$$

DOI: 10.1002/nla

Following the standard approach, we will use the concept of an *elimination tree*. Consider the assembled matrix A and associate a node with each column labelled with the column index. The *parent* $\pi(j)$ of node j is the first nonzero below the diagonal in column j of the factor L, that is

$$\pi(j) = \min\{i : i > j,\ l_{ij} \neq 0\}.$$

If this set is empty, j has no parent and is a *root* of the elimination tree. For a root, we set $\pi(j) = n + 1$. If $\pi(j) = i$, i is a *child* of j. A *leaf* node is one with no children. The tree may be a forest with more than one root, but it is convenient to still call it an elimination tree. An extensive theoretical survey and treatment of elimination trees and associated structures in sparse matrix factorisations is given by Liu [5]. A key result stated in that paper (see also [6]) is that any postordering of the elimination tree (that is any ordering where the nodes within every subtree of the elimination tree are numbered consecutively, with the root of the subtree numbered last in the subtree) will not alter the number of floating-point operations or amount of fill-in associated with the factorisation.

The performance of most algorithms used in the analyse phase can be enhanced by identifying sets of columns with the same (or similar) sparsity patterns. The set of variables that correspond to such a set of columns in A is called a *supervariable*. In the elemental case, they are normally identified as a set of variables that belong to the same set of elements, and in problems arising from finite element applications, they occur frequently as a result of each node of the finite element mesh having multiple degrees of freedom associated with it. Under the assumption that diagonals are always present in the factor L, supervariables cannot exist in L because L is lower triangular. Instead, we use the concept of a supernode. Let $\mathcal{L}_j$ denote the sparsity pattern of column j of L

$$\mathcal{L}_j = \{i : l_{ij} \neq 0\}.$$

The *column count* $cc(j)$ for node j is the number of entries in $\mathcal{L}_j$. A *supernode* is a set of nodes corresponding to a set of contiguously numbered columns of L, say $i, i+1, \ldots, i+r-1$, such that for any successive pair $j-1$ and j in the set, $j-1$ is the only child of j, and $\mathcal{L}_j = \mathcal{L}_{j-1} \backslash \{j\}$. That is, the set of nodes in a supernode forms a chain in the elimination tree, and the corresponding columns of L have identical nonzero patterns, excluding entries in the upper triangular part (so that $cc(j) = cc(j-1) - 1$ for $j = i+1, \ldots, i+r-1$). If we regard all nodes that are not part of a supernode as supernodes of size 1, we can define the *assembly tree* to be the reduction of the elimination tree that contains only supernodes (this is sometimes referred to as the supernodal elimination tree).

Supernodes can be exploited in the factorisation phase to facilitate the use of highly efficient dense linear algebra kernels and, in particular, Level-3 BLAS kernels. These can offer such a large performance increase that it is often advantageous to merge supernodes that have similar (but not exactly the same) nonzero patterns, despite this, increasing the fill-in and operation count. This process is termed *supernode amalgamation*, and the resultant nodes are often referred to as *relaxed supernodes* (see, e.g. [6–8]). A survey of available heuristics for supernode amalgamation is outside the scope of this paper; we will employ supernode amalgamation but will not distinguish between supernodes and relaxed supernodes in our discussions.

Throughout this paper, we assume that compressed sparse column storage is used for assembled matrices.

1.2. Test environment

In this paper, we employ two test sets. The first is a set of 12 large-scale assembled matrices taken from the University of Florida Sparse Matrix Collection [9]. These were selected to represent the different subcollections of symmetric problems within the Collection and because they have non-trivial supervariables. The second set comprises 18 elemental matrices; they are available at `ftp://ftp.numerical.rl.ac.uk/pub/matrices/`. Note that the assembled form of these matrices is included in the University of Florida Sparse Matrix Collection, whereas we obtained the elemental versions from Christian Damhaug of DNV Software. The two sets are listed in Tables I and II.

DOI: 10.1002/nla

Table I. Assembled test problems. *nsvar* is number of supervariables, and k is the average number of variables in each supervariable. The storage is the integer storage for holding the sparsity pattern of A in its original form and in its condensed form (see Section 3).

Problem	n (10^3)	ne (10^6)	*nsvar* (10^3)	k	Storage (Mbytes) Original	Condensed
Boeing/bcsstk39	47	2.06	10	4.61	15	<1
TKK/s4dkt3m2	90	3.75	15	5.93	28	<1
Rothberg/gearbox	154	9.08	56	2.74	69	11
Boeing/pwtk	218	11.52	42	5.25	88	3
DNVS/fullb	199	11.71	33	5.96	89	2
Chen/pkustk14	152	14.84	34	4.45	113	6
INPRO/msdoor	416	19.17	61	6.82	154	3
Koutsovasilis/F1	344	26.84	120	2.85	204	25
GHS_psdef/ldoor	952	42.49	138	6.92	354	7
Schenk_AFE/af_shell10	1508	52.26	302	5.00	401	16
Oberwolfach/bone010	987	47.85	328	3.01	546	60
GHS_psdef/audikw_1	944	77.65	314	3.00	592	65

Table II. Elemental test problems. *nsvar* is the number of supervariables, and k is the average number of variables in each supervariable.

Problem	n (10^3)	*nelt*	*nsvar* (10^3)	k
trdheim	22	813	3	7.71
opt1	15	977	4	4.06
tsyl201	21	960	3	7.18
crplat2	18	3152	3	6.00
thread	30	2176	9	3.36
ship_001	35	3431	6	5.98
srb1	55	9240	9	6.00
m_t1	98	5328	17	5.72
x104	108	26019	17	6.28
shipsec8	115	32580	19	5.88
shipsec1	141	41037	23	6.00
scondp2	202	35836	34	5.95
ship_003	122	45464	20	6.00
troll	213	41084	48	4.41
shipsec5	180	52272	30	5.91
fullb	199	59738	33	5.97
halfb	225	70211	78	2.86

The numerical results reported in this paper were performed on a single thread of a 2-way quadcore Harpertown machine. The Intel 11.1 compiler with options -g -static -xSSE4.1 -O3 -no-prec-div -ipo was used.

2. THE GILBERT, NG AND PEYTON APPROACH

A basic tree-based algorithm for computing the pattern of L and parent pointers is shown as Algorithm 1. The sparsity pattern $\mathcal{L}_{col}$ of each column *col* of L is determined in turn and is the union of the sparsity pattern $\mathcal{A}_{col}$ of column *col* of A with the pattern of the children i of *col* in the elimination tree. The elimination tree is built node-by-node, by definition, the parent of any node corresponds to a column that is later in the pivot sequence.

The operation of computing the union of a set of column sparsity patterns is fundamental to algorithms that identify the supernode index lists. However, it is expensive, executing in time proportional to the number of entries in L. The sparsity pattern of every column of L is not needed,

DOI: 10.1002/nla

Algorithm 1 Basic tree-based algorithm

Input: Sparsity pattern of the assembled $n \times n$ matrix A.
Output: Sparsity pattern of L and elimination tree π.
Initialise: $\pi(:) = n + 1$
for $col = 1, n$ **do**
$\mathcal{L}_{col} = \mathcal{A}_{col} \cup \{col\} \cup \left(\bigcup \{\mathcal{L}_i \backslash \{i\} : \pi(i) = col\} \right)$
$\pi(col) = \min\{j : j \in \mathcal{L}_{col}, j \neq col\}$ *! $\pi(col)$ is the parent of col*
end for

only that of the first column of each supernode. If supernodes can be found without the need to find the index lists, the amount of merging can be substantially reduced. This was realised by Gilbert, Ng and Peyton [1], who describe an algorithm for determining the column counts that is almost linear in the number of entries in A. They also propose a scheme for determining supernodes that takes the column counts and elimination tree as inputs. An efficient analyse algorithm that uses this approach is summarised in Algorithm 2.

Algorithm 2 Analysis algorithm exploiting supernodes

Find the elimination tree of A.
Postorder the elimination tree.
Determine the column counts and supernodes.
Perform supernode amalgamation and determine the assembly tree.
Perform a symbolic factorization using the assembly tree.

In a postordering, the d_i descendants of each node i in the elimination tree are numbered from $i - d_i$ to $i - 1$. The postordering ensures that a node with a single child c is always numbered $c + 1$. Iteration over the tree in postorder corresponds to a depth-first search, and the reordering has no effect on the number of entries in L.

This analyse algorithm was incorporated by Ng and Peyton into their sparse Cholesky solver SPRSBLKLLT (details of this package are given in [10]) and their incomplete Cholesky factorisation preconditioner [11]. In the mid 1990s, it was employed by Damhaug and Reid in the analyse phase of the HSL [2] package `MA46` for the direct solution of sparse unsymmetric linear systems of equations from finite-element applications [12]. A notable example of more recent use is the CHOLMOD package of Davis [8, 13]. An unsymmetric variant was developed by Gilbert, Li, Ng and Peyton [14] for use in QR and LU codes. In particular, the unsymmetric variant is used in the sparse solver SuperLU [15, 16].

In Algorithm 3, we outline the approach of Liu [5] for computing the elimination tree. The algorithm exploits the characterisation of the elimination tree as the first nonzero below the diagonal in the column of L. One node at a time is added to the tree. At the start of stage i, the partially built tree has $i - 1$ nodes and node i is the next to be added. Suppose the entries in row i of A to the left of the diagonal are in columns $j_1, j_2, \ldots, j_k$. For each nonzero j_p, we find *jroot*, the root of the partial elimination tree that contains j_p. Node i is added to the tree by setting $\pi(jroot) = i$. Once all nonzeros in row i have been processed, i is (temporarily) the new root of the tree containing i, so $\pi(i)$ is set to $n + 1$. Path compression is used to improve the efficiency of finding *jroot* and employs a *virtual tree* $\bar{\pi}(:)$ that is only used for this purpose. After the traversal, the new root of the tree will be i and thus future traversals may be accelerated by setting $\bar{\pi}(k) = i$ for each node k visited. Note that the entries of the lower triangular part of A are accessed by rows. This is straightforward if both the lower and upper triangles are stored, but this roughly doubles the amount of data to be read, which may negatively impact performance.

DOI: 10.1002/nla

Algorithm 3 Find the elimination tree of an $n \times n$ assembled matrix (from Liu [5])

Input: Sparsity pattern of the assembled $n \times n$ matrix A.
Output: Elimination tree π.
Initialise virtual tree $\bar{\pi}(:) = n + 1$
for $i = 1, n$ **do**
  ! for row i, loop over entries to left of diagonal
  for $j : j < i, a_{ij} \neq 0$ **do**
    ! perform traversal to find jroot, the root of the partial tree containing j
    $jroot = j$
    while $\bar{\pi}(jroot) < i$ **do**
      $next = \bar{\pi}(jroot)$
      $\bar{\pi}(jroot) = i$ *! path compression to accelerate future traversals*
      $jroot = next$
    end while
    if $\bar{\pi}(jroot) = i$ **cycle**
    $\pi(jroot) = i$ *! make i the parent of jroot*
    $\bar{\pi}(jroot) = i$
  end for
  $\pi(i) = n + 1$ *! i is the root of the partial tree containing node i*
end for

The algorithm we use for computing column counts of L is a modification of the original algorithm of Gilbert, Ng and Peyton [1] for assembled problems (the original algorithm also found row counts). Our variant is given in Algorithm 4. To cope with supervariables that have different numbers of variables, it allows variables to be weighted. The use of these weights will be described in Section 3. We note that, in their work on union-find algorithms, Patwary *et al.* [17] recently described advances on the Gilbert, Ng and Peyton algorithm. However, their interleaved algorithms are not suitable for use in Algorithm 4.

Key to Algorithm 4 is the determination of the number of indices shared by the children of a node. This is accomplished by storing the last column at which an index was encountered (*last_p*) and a partial elimination tree ($\bar{\pi}$). The partial elimination tree consists of all nodes processed so far, plus their parents. As the tree is postordered, the current root of the partial tree containing node j is the least common ancestor of node j and the current node. Observe that the first time an index will be double counted is at the least common ancestor of the current node and the last node where it was encountered. By storing the first descendant and last neighbour of each node (in *first* and *last_nbr*, respectively), redundant work can be avoided if an index has already been encountered at a descendant.

3. IDENTIFYING AND USING SUPERVARIABLES: ASSEMBLED CASE

Supervariables are identified and used within the analyse phase of a number of sparse solvers including, for example, the HSL codes `MA47` [18] and `HSL_MA77` [19]. Gilbert, Ng and Peyton did not incorporate supervariables within their original description of their analyse algorithm. For assembled problems, the system matrix can be condensed so that we are dealing with supervariables rather than variables. If the average number of variables in each supervariable is k, this reduces the amount of integer data read during the analyse phase by a factor of about k^2. The final two columns of Table I illustrate the storage savings. These savings can substantially reduce memory traffic and may allow the problem to reside in cache.

We can now see how the weights of Algorithm 4 may be used. Rather than feeding the original matrix with weights of 1, instead the condensed matrix can be supplied with weights equal to the number of variables in each supervariable to achieve the same answer with less memory traffic.

DOI: 10.1002/nla

Algorithm 4 Column count algorithm with weights

Input: Assembled matrix A, postordered elimination tree π, column weights wt
Output: Column counts cc
Algorithm:

```
For each i, set first(i) to be lowest numbered descendant of node i.
Set cc(i) = wt(i) if node i is a leaf and cc(i) = 0 otherwise.
Initialise π̄(:) = 0 such that every node is its own tree.
Initialise last_p(:) = 0, last_nbr(:) = 0.
for each column j of A do
  for i : j < i, a_ij ≠ 0 do
    if first(j) > last_nbr(i) then
      ! index j has not been encountered at a descendant
      cc(j) = cc(j) + wt(i) ! new entry in current column
      pp = last_p(i)
      if pp ≠ 0 then
        ! i has been encountered before at node pp
        Find lca, the least common ancestor of j and pp.
        cc(lca) = cc(lca) − wt(i)
      end if
      last_p(i) = j
    end if
    last_nbr(i) = j
  end for
  π̄(j) = π(j) ! add parent of j to tree containing j
  cc(π(j)) = cc(π(j)) + cc(j) − wt(j) ! Pass all uneliminated variables up tree to parent
end for
```

With careful choice of data structures, the process of identifying supervariables can be made to execute in $\mathcal{O}(n + ne)$ time. The algorithm we use is outlined in Algorithm 5; it is a variant of that of Duff and Reid [18]. The algorithm works progressively so that after j steps the supervariable structure for the submatrix comprising the first j columns has been constructed, the algorithm commences with all the variables in a single supervariable $\mathcal{S}_1$ (for the submatrix with no columns). It then splits this supervariable into two, according to which rows do or do not have an entry in column 1. These are then split according to the entries in column 2, and so on. The splitting is performed by moving the variables one at a time to the new supervariable. At each stage, $seen(sv) < j$ indicates that supervariable $\mathcal{S}_{sv}$ has not yet occurred in the current column j. When it does occur in the current column, $seen(sv)$ is set to j. Should a second variable from supervariable $\mathcal{S}_{sv}$ be encountered, it is placed into the same supervariable as the first; this information is stored as $map(sv)$. Once supervariables have been identified, the pivot order is modified so that all variables in a supervariable occur consecutively in the location of the first such occurrence. Note that this will not increase the size of the factors L.

Key features of our supervariable algorithm are that the special case of trivial supervariables is handled efficiently and a stack is used to ensure that new supervariables are established using space from those that have recently become empty, exploiting cache locality.

Further discussion together with timing comparisons for other supervariable identification algorithms from the HSL library may be found in the report by Hogg and Scott [20]. This report also examines combining identification of supervariables with constructing the elimination tree. Hogg and Scott found that, for problems with non-trivial supervariables, this approach was slower than first identifying the supervariables, condensing the matrix and running the elimination tree algorithm on the condensed matrix.

DOI: 10.1002/nla

Algorithm 5 Determine supervariables of an $n \times n$ assembled matrix

Input: Sparsity pattern of the assembled $n \times n$ matrix A.
Output: Number of supervariables and the number of variables belonging to each supervariable.

```
Initialise S_1 = {1, 2, ... n}.
Place n empty supervariables on free supervariable stack.
Initialise seen(:) = 0
for j = 1, n do
  for i : a_ij ≠ 0 do
    Let S_sv be the supervariable to which i belongs.
    if |S_sv| = 1 then ! S_sv contains a single variable
      if seen(sv) < j cycle
      Move i from S_sv to S_map(sv).
      Place empty S_sv on top of free stack.
    else
      if seen(sv) < j then ! First occurrence of sv in column j
        Take a new supervariable S_new from top of free stack.
        map(sv) = new
        seen(sv) = j ! Flag S_sv as seen in column j
      end if
      Move i from S_sv to S_map(sv).
    end if
  end for
end for
```

Table III. The performance of our analyse code with and without supervariables. k is the average number of variables in each supervariable. Times are given in seconds.

Problem	k	With	Without	Ratio
Boeing/bcsstk39	4.61	0.0222	0.0419	1.89
TKK/s4dkt3m2	5.93	0.0399	0.0799	2.00
Rothberg/gearbox	2.74	0.1358	0.1732	1.27
Boeing/pwtk	5.25	0.1263	0.2236	1.77
DNVS/fullb	5.96	0.1212	0.2212	1.82
Chen/pkustk14	4.45	0.1606	0.2339	1.46
INPRO/msdoor	6.82	0.2140	0.4277	2.00
Koutsovasilis/F1	2.85	0.4127	0.5192	1.26
GHS_psdef/ldoor	6.92	0.5118	1.0183	1.99
Schenk_AFE/af_shell10	5.00	0.7250	1.3473	1.86
Oberwolfach/bone010	3.01	0.9795	1.3782	1.41
GHS_psdef/audikw_1	3.00	1.0794	1.4443	1.34

In Table III, we present run times for our implementation of the analyse code applied to assembled problems. Times are given both with and without the exploitation of supervariables. We see that the time taken to identify supervariables is more than offset by the time saved in the remainder of the analyse phase by working with the condensed matrix, with gains of close to a factor of 2 for the test problems with more than five variables in each supervariable. Recall that all our test problems have non-trivial supervariables. For problems with almost n supervariables, using the supervariable option adds an overhead. Experiments on such problems show that if our analyse code is run both with and without the supervariable option, the latter is about 20% faster (full details are given in Hogg and Scott [20]).

DOI: 10.1002/nla

4. ANALYSE PHASE FOR ELEMENTAL PROBLEMS

4.1. Avoiding explicit assembly of elemental problems

A comparison of columns 2, 3 and 5 headed 'Elemental' and 'Assembled' in Table IV illustrate that for each of our elemental test problems, it is more memory efficient to hold A in elemental format than to assemble it explicitly. Thus, when performing the memory-bound analyse phase on modern computers, we want to avoid assembling A. With the exception of the work of Damhaug and Reid, all other references and software that we are aware of relate exclusively to the use of the Gilbert, Ng and Peyton algorithm within the analyse phase for assembled matrices. The code `MA46` of Damhaug and Reid is designed for problems in elemental form; it avoids holding the sparsity pattern of the assembled matrix by using an *implicit adjacency structure* that represents the nodal structure of the coefficient matrix. For each variable, a list of the associated elements is held. This facilitates iterating over all entries in a column by iterating over the variables belonging to all associated elements. However, this approach is inefficient because it is equivalent to assembling the column of A (and it must be repeated each time the column is needed). Thus, we seek an alternative; one is provided by the following lemma.

Lemma 1
The pattern of the Cholesky factor L of the matrix $A = \sum_k A^{(k)}$, where each $A^{(k)}$ is nonzero in the rows and columns corresponding to the set $\mathcal{E}_k$, is the same as that of the Cholesky factor $\hat{L}$ of the matrix $\hat{A} = \sum_k \hat{A}^{(k)}$, where the first row and column of $\hat{A}^{(k)}$ have the same sparsity pattern as the first row and column of $A^{(k)}$, and all other entries are zero.

We observe that this lemma follows straightforwardly from fill in during the Cholesky factorisation. Consider the element matrices in turn. Each missing entry in $\hat{A}^{(k)}$ is replaced by fill in caused by its first row and column. It follows that the pattern of the factors is identical. We now formalise this proof.

Proof of Lemma 1
Because $\hat{\mathcal{A}}_1 = \mathcal{A}_1$, the first column of L has the same sparsity pattern as the first column $\hat{L}$ (i.e. $\hat{\mathcal{L}}_1 = \mathcal{L}_1$).
Proceed by induction: assume that all columns i with $i < j$ satisfy $\mathcal{L}_i = \hat{\mathcal{L}}_i$.

Table IV. A comparison of the storage (in Mbytes) for the elemental, assembled and equivalent forms.

		Lower triangle only		Upper + lower triangles		Condensed lower triangle	
Problem	Elemental	Assembled	Equivalent	Assembled	Equivalent	Assembled	Equivalent
trdheim	0.34	7.64	0.50	14.9	1.01	0.31	0.17
opt1	0.41	7.54	0.47	14.8	0.94	0.91	0.12
tsyl201	0.45	9.60	0.53	18.9	1.06	0.36	0.16
crplat2	0.57	3.87	0.49	7.47	0.99	0.23	0.14
thread	0.98	17.4	1.01	34.3	2.01	3.17	0.23
ship_001	1.21	18.2	1.23	35.7	2.46	1.03	0.27
srb1	1.75	11.9	1.49	23.0	2.97	0.70	0.42
m_t1	2.08	38.3	2.53	75.2	5.06	2.54	0.74
x104	2.25	40.0	2.80	78.4	5.60	2.15	0.83
shipsec8	5.43	26.7	3.77	51.6	7.53	1.61	0.88
shipsec1	6.30	31.4	4.54	60.7	9.08	1.83	1.07
fcondp2	6.81	45.4	5.66	87.7	11.3	2.68	1.54
ship_003	7.08	32.2	4.58	62.6	9.15	1.87	0.93
troll	7.61	48.2	6.35	93.1	12.7	5.78	1.63
shipsec5	8.09	40.6	5.80	78.5	11.6	2.42	1.37
fullb	9.38	46.9	6.55	90.8	13.1	2.76	1.52
halfb	10.4	49.8	7.25	96.2	14.5	3.04	1.71

DOI: 10.1002/nla

For each nonzero $p \in \mathcal{L}_j$, one of the following holds:

1. $p \in \mathcal{A}_j \cap \hat{\mathcal{A}}_j$. Because p belongs to $\hat{\mathcal{A}}_j$, it must also belong to $\hat{\mathcal{L}}_j$.
2. $p \in \mathcal{A}_j \backslash \hat{\mathcal{A}}_j$. There exists k such that $j, p \in \mathcal{E}_k$. Let i be the smallest index in $\mathcal{E}_k$. Then, $i < j$ and $j, p \in \hat{\mathcal{L}}_i$. Because $j \leqslant p$, $p \in \hat{\mathcal{L}}_j$.
3. $p \in \mathcal{L}_j \backslash \mathcal{A}_j$. By induction, any fill in $\mathcal{L}_j$ must also be in $\hat{\mathcal{L}}_j$.

Hence, $\mathcal{L}_j \subseteq \hat{\mathcal{L}}_j$ and because $\hat{\mathcal{A}}_j \subseteq \mathcal{A}_j$, we conclude that $\hat{\mathcal{L}}_j = \mathcal{L}_j$. □

We will refer to the matrix $\hat{A}$ as the *equivalent matrix*. We hold it as an assembled matrix; the storage it requires is reported in Table IV. We see that for most of our problems, it is the best. Although comparing columns 2 and 4, for some very sparse problems (including m_t1, thread and tsyl201), the overhead of storing the column pointers that are needed for the equivalent matrix means that the original elemental storage requires less memory.

4.2. Identification of supervariables: elemental case

As in the assembled case, the use of supervariables will reduce storage requirements further. In the elemental case, we seek variables that are present in the same set of elements, rather than the same set of columns. A simple modification of Algorithm 5 replacing the loop over columns with a loop over elements is sufficient to identify the sets of elements. This may mean we miss some supervariables that would be found from the columns of the assembled matrix. However, for our problems arising from finite element applications, our experience is that we obtain the majority of them. Indeed, for all but four of our elemental test examples, all the supervariables were found, except for only one problem (opt1) where the number of missed supervariables was more than 1%. In the final two columns of Table IV, we report the storage for the condensed assembled and equivalent forms. In each case, the condensed equivalent form requires significantly less storage than the condensed assembled form and less storage than the original elemental form.

4.3. Building the elimination tree: elemental case

Motivated by the storage savings offered by the equivalent matrix, we now compare two possible approaches to constructing the elimination tree in the elemental case. The first is a purely element-based algorithm, whereas the second applies the Liu algorithm [5] (Algorithm 3) to the equivalent matrix. For the former, we first characterise the elimination tree in element terms. For each element variable list $\mathcal{E}_k$, we build a simple tree with each node representing an entry $j \in \mathcal{E}_k$. The parent of j is the next variable of $\mathcal{E}_k$ in elimination order. The elimination tree is obtained by merging these simple trees and finding the transitive reduction. This leads to Algorithm 6, where variable lists are merged into the tree one at a time. Each ancestral relationship is added individually, with all changes propagated before the next is added. An example of adding a variable list is shown in Figure 1. Note how the variable *ancestor* is used to temporarily store swapped out ancestors as the change is propagated.

Algorithm 6 may be applied to the original elemental form or to the condensed elemental form; the times for our test problems are given in columns 2 and 4 of Table V, respectively. The time to identify supervariables and obtain the condensed elemental form is given in column 3 (headed 'find svs'), whereas column 5 reports the total time for the condensed elemental approach to construct the elimination tree. A comparison of columns 2 and 5 shows that exploiting supervariables when using the elemental approach for constructing the elimination tree results in substantial savings.

For efficiency of the equivalent matrix approach using supervariables, we need to minimise the number of passes through the data. Recall that Algorithm 3 requires access to the lower triangle of L by rows (or equivalently, access to the upper triangle by columns). Hogg and Scott [20] investigated a number of alternatives and found that the most efficient approach uses the first pass to determine the supervariables and builds a lower triangular supervariable condensed equivalent matrix on the second pass, which also determines column counts needed in finding the upper triangle. A pass through the condensed lower form is sufficient to place entries of the upper triangle in their final

DOI: 10.1002/nla

Algorithm 6 Determine the elimination tree of an $n \times n$ elemental problem

Input: Integer lists $\mathcal{E}_k$ for each element matrix and the elimination order.
Output: Elimination tree π.
Initialize $\pi(:) = n + 1$.
for each element k **do**
  Copy $\mathcal{E}_k$ in elimination order into $work(:)$.
  $next = work(1)$
  for $i = 2, \ldots, nvar_k$ **do**
    $child = next$ *! start at variable* $i - 1$ *of* $\mathcal{E}_k$
    $ancestor = work(i)$;
    $next = ancestor$ *! store for next iteration*
    ! Ascend tree, placing entry $i \in \mathcal{E}_k$ *into correct position and modifying tree as required*
    while $child \neq ancestor$ and $child \leq n$ **do**
      if $ancestor < \pi(child)$ **then** swap $ancestor$ and $\pi(child)$.
      $child = \pi(child)$
    end while
  end for
end for

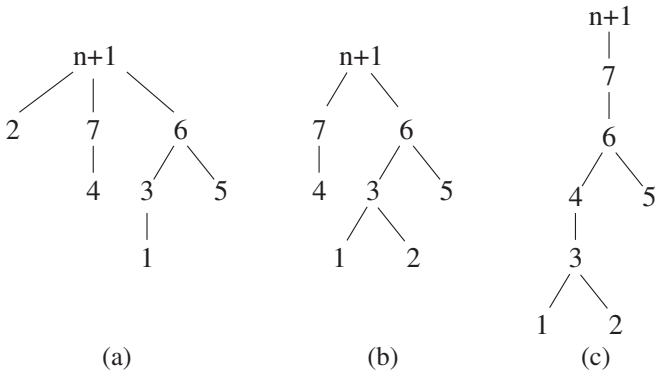

Figure 1. Example showing the merger of element variable list $\{2, 3, 4\}$ into the partially constructed elimination tree (a). Incorporating the relation that 3 is an ancestor of 2 gives (b). Adding the relation that 4 is an ancestor of 3 results in (c).

locations, enabling Algorithm 3 to then be used to determine the elimination tree. Timings for the equivalent matrix approach, with and without supervariables, are included in Table V. The times in column 9 are for Algorithm 3 applied to the condensed equivalent matrix, whereas those in column 10 are the total times for constructing the elimination tree by using the condensed equivalent matrix. Comparing the total times in columns 5 and 10, we conclude that using the condensed equivalent matrix is faster than using the condensed elemental approach and thus we adopt the former. In Figure 2, columns 2, 5 and 8 are compared with column 10.

5. COMPARISON OF ANALYSE PHASE TIMINGS

One of our main aims was to design and implement an efficient analyse phase for elemental problems, without explicitly assembling the system matrix A. We have already shown that the condensed equivalent form for elemental problems leads to significant storage savings and to savings in the time for constructing the elimination tree. To assess how successful we have been in terms of the analyse time, in Figure 3, we compare the performance of our implementation `HSL_MC78` of the analyse

DOI: 10.1002/nla

Table V. Comparison of the performance of the elemental and equivalent matrix approaches for constructing the elimination tree of a problem in elemental form. 'Total' denotes the total time to construct the elimination tree. Times are given in seconds.

	Elemental approach				Equivalent matrix approach				
	Original	Condensed			Original			Condensed	
Problem	Alg. 6	Find svs	Alg. 6	Total	Build	Alg. 3	Total	Alg. 3	Total
trdheim	0.0060	0.0006	0.0003	0.0010	0.0006	0.0005	0.0011	0.0001	0.0008
opt1	0.0135	0.0009	0.0015	0.0024	0.0008	0.0007	0.0015	0.0003	0.0011
tsyl201	0.0256	0.0008	0.0008	0.0016	0.0007	0.0007	0.0014	0.0001	0.0010
crplat2	0.0127	0.0009	0.0007	0.0016	0.0009	0.0008	0.0017	0.0001	0.0011
thread	0.0820	0.0018	0.0096	0.0114	0.0019	0.0019	0.0038	0.0006	0.0026
ship_001	0.0367	0.0019	0.0022	0.0041	0.0024	0.0019	0.0043	0.0004	0.0025
srb1	0.0611	0.0030	0.0030	0.0060	0.0033	0.0024	0.0057	0.0005	0.0036
m_t1	0.0852	0.0041	0.0046	0.0088	0.0047	0.0038	0.0084	0.0009	0.0056
x104	0.2002	0.0043	0.0079	0.0122	0.0047	0.0040	0.0088	0.0009	0.0057
shipsec8	0.1895	0.0099	0.0114	0.0213	0.0145	0.0091	0.0235	0.0014	0.0128
shipsec1	0.2262	0.0108	0.0118	0.0226	0.0171	0.0112	0.0282	0.0016	0.0155
fcondp2	0.4800	0.0131	0.0218	0.0349	0.0133	0.0113	0.0246	0.0019	0.0182
ship_003	0.2432	0.0118	0.0161	0.0279	0.0172	0.0123	0.0295	0.0018	0.0166
troll	0.6562	0.0155	0.0569	0.0724	0.0178	0.0154	0.0331	0.0034	0.0240
shipsec5	0.3343	0.0148	0.0182	0.0331	0.0251	0.0147	0.0398	0.0021	0.0213
fullb	0.4395	0.0166	0.0229	0.0395	0.0250	0.0168	0.0418	0.0024	0.0241
halfb	0.3185	0.0183	0.0192	0.0375	0.0277	0.0191	0.0468	0.0027	0.0272

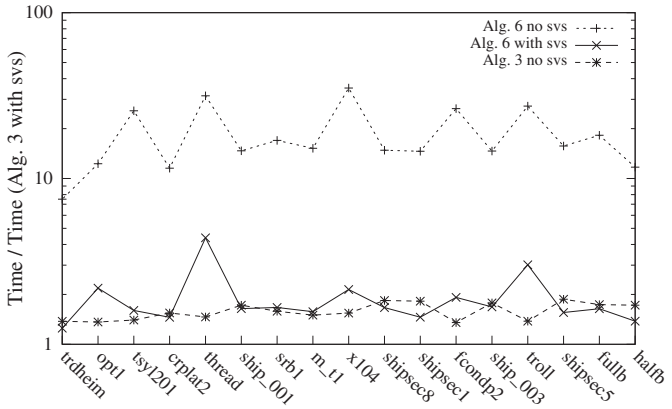

Figure 2. Comparison of the total time to find the elimination tree to the time for the best approach (Algorithm 3 applied to the equivalent matrix using supervariables).

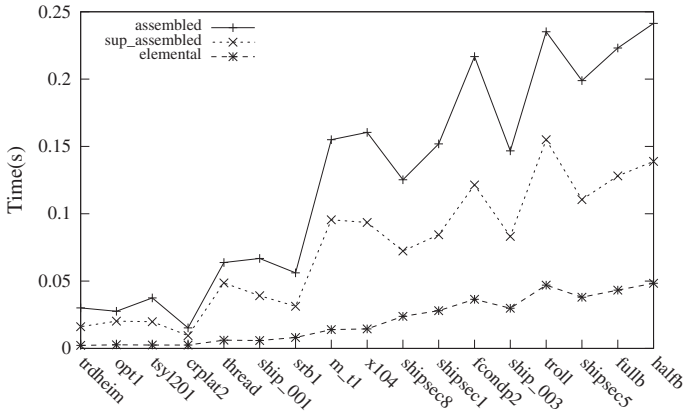

Figure 3. Graph of the performance of the analyse code `HSL_MC78` using the elemental and assembled mode with and without supervariables.

DOI: 10.1002/nla

phase run in both elemental and assembled modes. For the latter, we do not include the time to assemble A and report only the run time for `HSL_MC78`. The elemental mode exploits supervariables, and the assembled mode is run with and without using supervariables. We see that working with the elemental form is significantly faster for all our problems and, once assembled, substantial savings can be achieved by exploiting supervariables.

Finally, we present timings to illustrate that incorporating our new analyse phase into our prototype symmetric multifrontal solver `RAL_SYMF` (an in-core version of `HSL_MA77` [19]) leads to this solver having an analyse phase whose performance compares favourably with that of other state-of-the-art sparse direct solvers. The solvers we use in our experiments are listed in Table VI.

Table VI. Sparse direct solvers used in our numerical experiments.

Code	Date/version	Authors/website
CHOLMOD [13]	3.2009/ v1.7.1	T. Davis `http://www.cise.ufl.edu/research/sparse/cholmod/`
`MA57` [21]	11.2009/ v3.4.0	I.S. Duff, HSL `http://www.hsl.rl.ac.uk/`
PARDISO [22]	10.2009/ v4.0.0	O. Schenk and K. Gärtner `http://www.pardiso-project.org`
WSMP [23, 24]	06.2010/ v10.5.26	A. Gupta, IBM `http://www-users.cs.umn.edu/~agupta/wsmp.html`

Table VII. Times (in seconds) for the analyse phase of a range of solvers.

Problem	`MA57`	`RAL_SYMF`	PARDISO	WSMP	CHOLMOD
Boeing/bcsstk39	0.1061	0.0336	0.2286	0.1029	0.0821
TKK/s4dkt3m2	0.2221	0.0587	0.4156	0.1881	0.1538
Rothberg/gearbox	0.5005	0.1874	1.0595	0.4675	0.3679
Boeing/pwtk	0.6336	0.1869	1.2692	0.5693	0.4658
DNVS/fullb	0.8407	0.1827	1.3890	0.6116	0.4704
Chen/pkustk14	1.0654	0.2358	1.7716	0.7291	0.5655
INPRO/msdoor	1.0717	0.3229	2.2412	1.1063	0.8375
Koutsovasilis/F1	2.5116	0.5429	3.6837	1.7772	1.1028
GHS_psdef/ldoor	3.2226	0.7689	5.8729	2.5869	1.9596
Schenk_AFE/af_shell10	5.5576	1.2044	6.5910	2.8010	2.4545
Oberwolfach/bone010	9.9569	1.3642	10.889	3.9215	2.9509
GHS_psdef/audikw_1	11.727	1.5352	12.799	4.7763	3.2384
crplat2	0.0368	0.0054	0.0928	0.0523	0.0319
fcondp2	0.8640	0.0914	1.2544	0.7341	0.4436
fullb	1.1186	0.1049	1.3930	0.8063	0.4666
halfb	1.0557	0.1133	1.4371	0.8376	0.4966
m_t1	0.5892	0.0339	0.9917	0.5837	0.3455
opt1	0.0705	0.0053	0.1910	0.1107	0.0658
ramage02	0.1399	0.0070	0.3135	0.1778	0.1002
ship_001	0.2076	0.0133	0.4666	0.2816	0.1608
ship_003	0.6155	0.0709	0.9852	0.5326	0.3178
shipsec1	0.4715	0.0629	0.8714	0.5000	0.3167
shipsec5	0.8228	0.0881	1.1515	0.6545	0.4057
shipsec8	0.4199	0.0525	0.7593	0.4310	0.2652
srb1	0.1492	0.0175	0.3065	0.1856	0.1148
thread	0.2673	0.0125	0.5195	0.2901	0.1608
trdheim	0.0564	0.0050	0.1724	0.1071	0.0675
troll	0.9852	0.1085	1.4103	0.7948	0.4759
tsyl201	0.0986	0.0057	0.2532	0.1383	0.0858
x104	0.5093	0.0354	0.9929	0.6007	0.3593

DOI: 10.1002/nla

RAL_SYMF accepts problems in either assembled or elemental form and then employs the corresponding mode of HSL_MC78. We remark that it is beyond the scope of this paper to attempt to describe and review the algorithms implemented by the analyse phase of each of the packages but observe that the analyse phase of the HSL multifrontal code MA57 is built on the original work of Duff and Reid in the early 1980s that was initially used in developing the well-known package MA27 [6, 25]. We note also that all the solvers generate different computational data during their analyse phase. In particular, PARDISO and WSMP are designed to be run in parallel and so the analyse phase of each of these codes includes the setting up of data structures for parallel working, which incurs additional overheads; we are not able to separate out the times for the different stages within the analyse phase and are only able to report total analyse times.

Timings are presented in Table VII and plotted in Figure 4. In each test, the same pivot order is supplied to all the solvers and this is generated using the METIS graph partitioning package [26, 27]. Otherwise, default settings are used for all control parameters (for RAL_SYMF the option to exploit supervariables is selected). For problems in the lower half of the table, RAL_SYMF is run in elemental mode. RAL_SYMF is the only package tested that is able to accept problems in elemental form; for the other solvers, we assemble the element matrices but omit the time for this. The time reported for RAL_SYMF includes work to set up its multifrontal data structures and prepare for the numerical factorisation in addition to the execution of HSL_MC78. From Table VII and Figure 4, we conclude that the performance of the analyse phase of RAL_SYMF compares favourably with that of other solvers, and in particular, these results demonstrate that it is beneficial both to exploit supervariables and to use the elemental form.

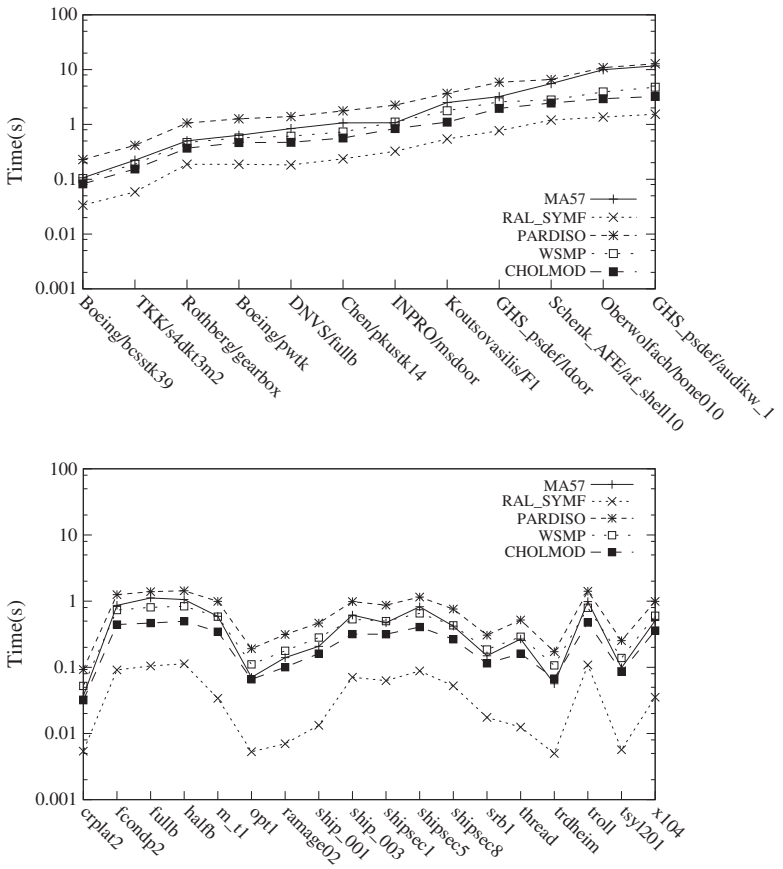

Figure 4. Comparison of the time taken for the analyse phase of a range of solvers.

DOI: 10.1002/nla

6. CONCLUDING REMARKS

In this paper, we have considered the key steps within the analyse phase of a modern sparse direct solver and, in particular, we have focused on designing and implementing an efficient analyse phase for problems in elemental form. Our starting point was the algorithm of Gilbert, Ng and Peyton for determining the column counts of the matrix factor L. We have incorporated supervariables and have shown that, for problems with a significant number of non-trivial supervariables, worthwhile savings in terms of memory and time can be achieved. For problems in elemental form, we have shown how the introduction of an equivalent matrix can avoid explicit assembly of the matrix and can lead to very fast analyse times.

Our implementation of the analyse phase is included as a separate package `HSL_MC78` within the HSL mathematical software library and is available without charge for academic purposes. The performance of `HSL_MC78` within our sparse multifrontal solver `RAL_SYMF` has been shown to compare favourably with that of other state-of-the-art packages. In particular, the efficient performance of `RAL_SYMF` on elemental problems has confirmed our view that, where available, the elemental form should be used in preference to the assembled form.

ACKNOWLEDGEMENTS

We are grateful to our colleagues Iain Duff and John Reid for commenting on a draft of this paper and to the authors of the codes run in Section 5 for access to their software. Thanks, as ever, to Tim Davis and Yifan Hu for the University of Florida Sparse Matrix Collection. Finally, we would like to thank two anonymous referees for their comments. Contact grant sponsor: EPSRC; contact grant number: EP/E053351/1

REFERENCES

1. Gilbert JR, Ng EG, Peyton BW. An efficient algorithm to compute row and column counts for sparse Cholesky factorization. *SIAM Journal on Matrix Analysis and Applications* 1994; **15**(4):1075–1091.
2. HSL. A collection of Fortran codes for large-scale scientific computation, 2011. http://www.hsl.rl.ac.uk.
3. George A, Liu JWH. *Computer Solution of Large Sparse Positive Definite Systems*. Prrentice-Hall Inc: Englewood Cliffs, New Jersey, 1981.
4. Duff IS, Erisman AM, Reid JK. *Direct Methods for Sparse Matrices*. Oxford University Press: Oxford, 1986.
5. Liu JWH. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications* 1990; **11**(1):134–172.
6. Duff IS, Reid JK. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software* 1983; **9**:302–325.
7. Ashcraft C, Grimes R. The influence of relaxed supernode partitions on the multifrontal method. *ACM Transactions on Mathematical Software* 1999; **15**:291–309.
8. Davis TA, Hager WW. Dynamic supernodes in sparse Cholesky update/downdate and triangular solves. *ACM Transactions Mathematical Software* 2009; **35**. Article 27.
9. Davis TA, Hu Y. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software* 2011; **28**. Article 1, 25 pages.
10. Ng EG, Peyton BW. Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM Journal on Scientific Computing* 1993; **14**(5):1034–1056.
11. Ng EG, Peyton BW, Raghavan P. A blocked incomplete Cholesky preconditioner for hierarchical-memory computers. In *Proceedings of the Fourth IMACS International Symposium on Iterative Methods in Scientific Computation*, Kincaid DR, Elster AC (eds), 1999; 211–222.
12. Damhaug AC, Reid JK. MA46, a FORTRAN code for the direct solution of sparse unsymmetric linear systems of equations from finite-element applications. *Technical Report RAL-TR-96-010*, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England, 1996.
13. Chen Y, Davis TA, Hager WW, Rajamanickam S. Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software* 2008; **35**. Article 22 (14 pages).
14. Gilbert JR, Li XS, Ng EG, Peyton BW. Computing row and column counts for sparse QR and LU factorization. *BIT* 2001; **41**(4):693–710.
15. Demmel JW, Eisenstat SC, Gilbert JR, Li XS, Liu JWH. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications* 1999; **20**(3):720–755.
16. Demmel JW, Gilbert JR, Li XS. SuperLU users' guide, 2010. LBNL-44289, Lawrence Berkeley National Laboratory.
17. Patwary M, Blair J, Manne F. Experiments on union-find algorithms for the disjoint-set data structure. *Experimental Algorithms* 2010; **6049**:411–423.

DOI: 10.1002/nla

18. Duff IS, Reid JK. Exploiting zeros on the diagonal in the direct solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software* 1996; **22**(2):227–257.
19. Reid JK, Scott JA. An out-of-core sparse Cholesky solver. *ACM Transactions on Mathematical Software* 2009; **36**(2). Article 9, 33 pages.
20. Hogg JD, Scott JA. A modern analyse phase for sparse tree-based direct methods. *Technical Report RAL-TR-2010-031*, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England, 2010.
21. Duff IS. MA57– a new code for the solution of sparse symmetric definite and indefinite systems. *ACM Transactions on Mathematical Software* 2004; **30**:118–154.
22. Schenk O, Gärtner K. Solving unsymmetric sparse systems of linear equations with PARDISO. *Journal of Future Generation Computer Systems* 2004; **20**:475–487.
23. Gupta A, Joshi M, Kumar V. WSMP: a high-performance serial and parallel sparse linear solver. *Technical Report RC 22038 (98932)*, IBM T. J. Watson Research Center, 2001. http://www.cs.umn.edu/~agupta/doc/wssmp-paper.ps.
24. Gupta A. WSMP: Watson sparse matrix package (Part-I: direct solution of symmetric sparse systems). *Technical Report RC 21886*, IBM T. J. Watson Research Center, Yorktown Heights, NY, November 2000. http://www.cs.umn.edu/~agupta/wsmp.
25. Duff IS, Reid JK. MA27 - a set of Fortran subroutines for solving sparse symmetric sets of linear equations. *Technical Report AERE-R 10533*, Harwell Laboratory, 1982.
26. Karypis G, Kumar V. METIS - family of multilevel partitioning algorithms, 1998. http://glaros.dtc.umn.edu/gkhome/views/metis.
27. Karypis G, Kumar V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing* 1999; **20**:359–392.

DOI: 10.1002/nla