Science & Technology Facilities Council

Technical Report
RAL-TR-2010-031

A modern analyse phase for sparse tree-based direct methods

J Hogg, J Scott

November 2010

ISSN 1358-6254

A modern analyse phase for sparse tree-based direct methods

Jonathan D. Hogg and Jennifer A. Scott[1]

ABSTRACT

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 factorization and solve phases to be executed efficiently. The analyse phase typically involves identifying supernodes, amalgamating supernodes to form relaxed supernodes and computing their variable lists, and determining the assembly tree.

Two main approaches have been used. The first, based on the initial work of Duff and Reid in the early 1980s, originates from their development of the multifrontal method. This approach emphasises the identification of supervariables of A and, in later versions, handles pre-specified 2×2 block pivots; it has been successfully used in both out-of-core and parallel solvers. The second approach, following the work of Gilbert, Ng and Peyton a decade or so later, adopts a graph theoretic view of assembled matrices, exploiting this to determine column counts for the matrix factor without finding the explicit pattern of the factor. This allows supernodes to be amalgamated before a symbolic factorization is performed, leading to significant savings in the analyse time.

The aims of this paper are two-fold: to incorporate supervariables into the Gilbert, Ng and Peyton approach and to describe an adaptation for matrices in elemental form (such as arise in finite-element applications), without explicitly assembling the system matrix. Various implementation details designed to enhance performance are described. Modifications to support block pivots are introduced. Numerical experiments using problems from practical applications are used throughout and demonstrate that it is advantageous, in terms of both memory and time, to work directly with the elemental form.

Keywords: sparse symmetric linear systems, direct solver, analyse phase, element problems, supervariables, supernodes.

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

---

[1] Computational Science and Engineering Department, Rutherford Appleton Laboratory, Chilton, Oxfordshire, OX11 0QX, UK.
Email: jonathan.hogg@stfc.ac.uk and jennifer.scott@stfc.ac.uk
Work supported by EPSRC grant EP/E053351/1.
Current reports available from `http://www.numerical.rl.ac.uk/reports/reports.html`.

November 23, 2010

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 factorization phase; then the matrix is factorized; 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 BLAS during the subsequent factorization; to determine the supernode index lists (that is, the nonzero pattern of the factors); and to determine an assembly tree that will be used to guide the numerical factorization. The determination of supernode index lists may be done during the factorization, 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 minimize memory requirements in a multifrontal method; reordering variables within supernodes to increase cache locality during the factorization; ensuring block pivots (a user-identified group of pivots) remain together and are numbered sequentially; and handling errors in the user-supplied data (including out-of-range indices and duplicate entries).

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. We want to allow both for matrices that are held in assembled form (input by columns) and matrices in elemental form (input by elements). In each case, we want to allow supervariables (columns of A with the same sparsity pattern) to be exploited 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 [15] 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 [20] as package `HSL_MC78`. We remark that, historically, the analyse phase was much faster than the factorization phase. Considerable effort has gone into parallelising the factorization 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 summarize 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 [15]; numerical results illustrate the savings resulting from exploiting supervariables. In Section 4, we consider how to handle problems in elemental form (such as arise from a finite-element application) without explicitly assembling the sparsity pattern of the matrix. Adapting the analyse phase to allow block pivots is discussed in Section 5. Timings for the analyse phase of two of our recent sparse direct solvers that incorporate our new analyse code are presented in Section 6, along with timings for older HSL solvers and other state-of-the-art packages. Finally, in Section 7, our findings are summarised.

1.1 Terms and notation

We assume a basic knowledge of the steps involved in sparse Cholesky factorization and with the use of graphs in these algorithms (see, for example, [12]). 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 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. This factor has the same pattern as the matrix L in the indefinite factorization $PAP^T = LDL^T$ if the same pivot sequence is used and there are no block pivots that affect the fill-in (here D is block diagonal). To simplify notation, we will assume throughout that $P = I$ (that is, the pivot order is the natural order $1, 2, ..., n$) 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 two forms:

(i) 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, [7]).

(ii) 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,

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

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. 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 factorizations is given by Liu [23]. A key result stated in that paper (see also [9]) 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 factorization.

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 since 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 contiguously numbered nodes, say $i, i{+}1, ..., i{+}r{-}1$, that form a path in the elimination tree such that $cc(j) = cc(j{-}1){-}1$ for $j = i{+}1, ..., i{+}r{-}1$. That is, they represent a contiguous set of r columns of L whose sparsity patterns are related thus: $\mathcal{L}_j = \mathcal{L}_{j-1} \backslash \{j\}$ for $j = i + 1, ..., i + r - 1$. 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 factorization 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, for example, [1, 5, 9]). In this paper, we will not distinguish between supernodes and relaxed supernodes.

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 24 large-scale assembled matrices taken from the University of Florida Sparse Matrix Collection [4]. These were selected to represent the different sub-collections of symmetric problems within the Collection and include problems that our previous studies into sparse matrix algorithms have highlighted as being tough problems. 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; we obtained the elemental versions from Christian Damhaug of DNV Software. The two sets are listed in Tables 1.1 and 1.2.

Table 1.1: Assembled test problems.

Problem	n (10^3)	ne (10^6)	Description/application area
Cunningham/qa8fk	66	1.66	3D acoustic FE stiffness matrix
Lin/Lin	256	1.77	Large sparse eigenvalue problem
Wissgott/parabolic_fem	526	3.67	Parabolic FEM. Diffusion-convection reaction
CEMW/tmt_sym	727	5.08	Electromagnetics
McRae/ecology2	1000	5.00	Circuit theory applied to animal/gene flow
AMD/G3_circuit	1585	7.66	Circuit simulation
TSOPF/TSOPF_FS_b300_c2	568	8.77	Transient optimal power flow
Gupta/gupta3	17	9.32	Linear programming
Andrianov/mip1	66	10.35	Unknown
PARSEC/Ga41As41H72	268	18.49	Quantum chemistry: density functional theory calculations
ND/nd24k	72	28.72	3D mesh problem
Schenk/nlpkkt240	27994	760.6	Symmetric indefinite KKT matrix
Boeing/bcsstk39	47	2.06	Structural engineering: shuttle solid rocket booster
TKK/s4dkt3m2	90	3.75	Structural mechanics:, cylindrial shell
Rothberg/gearbox	154	9.08	Structural engineering: ZF aircraft flap actuator
Boeing/pwtk	218	11.52	Structural engineering: pressurised wind tunnel
DNVS/fullb	199	11.71	Structural engineering: full-breadth barge
Chen/pkustk14	152	14.84	Structural engineering: tall building
INPRO/msdoor	416	19.17	Structural engineering: medium size door
Koutsovasilis/F1	344	26.84	Symmetric indefinite matrix
GHS_psdef/ldoor	952	42.49	Structural engineering
Schenk_AFE/af_shell10	1508	52.26	Sheet metal forming indefinite matrix
Oberwolfach/bone010	987	47.85	Model reduction: 3D trabecular bone
GHS_psdef/audikw_1	944	77.65	Structural engineering

The numerical results reported in this paper were performed on a single thread of a 2-way quadcore

Table 1.2: Elemental test problems.

Problem	n (10^3)	*nelt*	Description/application area
trdheim	22.10	813	CFD simulation; mesh of Trondheim fjord
opt1	15.45	977	Part of condeep cylinder
tsyl201	20.68	960	Part of condeep cylinder
crplat2	18.01	3152	Corrugated plate field
thread	29.74	2176	Threaded connector
ship_001	34.92	3431	Ship structure - predesign
srb1	54.92	9240	Space shuttle rocket booster
m_t1	97.58	5328	Tubular joint
x104	108.4	26019	Beam joint
shipsec8	114.9	32580	Section of a ship
shipsec1	140.9	41037	Section of a ship
fcondp2	201.8	35836	Oil production platform
ship_003	121.7	45464	Ship structure - production
troll	213.4	41084	Structural analysis
shipsec5	179.9	52272	Section of a ship
fullb	199.2	59738	Full-breadth barge
halfb	224.6	70211	Half-breadth barge

Harpertown machine. The Intel 11.1 compiler with options -g -static -xSSE4.1 -O3 -no-prec-div -ipo was used. With the exception of the largest problem Schenk/nlpkkt240, each test was run 5 times; the worst time was discarded and the reported time is the average of the remaining 4.

2 Background and previous work

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 j 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.

Algorithm 1 Basic tree-based algorithm

```
**Input:** Sparsity pattern of the assembled $n \times n$ matrix $A$.
**Initialise:** $\pi(1:n) = n+1$
**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**
```

In the elemental case, it is not necessary to explicitly assemble the system matrix A; instead we can proceed as in Algorithm 2. The pattern $\mathcal{L}_{col}$ of each column *col* of L is again determined in turn, and is the union of the elements k that have minimum pivot *col* with the patterns of the children i of *col* in the elimination tree.

The operation of computing the union of a set of column sparsity patterns is fundamental to all algorithms that identify the supernode index lists. Algorithms to perform this merge were used by the

Algorithm 2 Basic tree-based elemental analysis

```
**Input:** Sparsity pattern of the $n \times n$ matrix $A$ in elemental form with $nelt$ elements.
**Initialise:** $\pi(1:n) = n+1$
**for** $k = 1, nelt$ **do**
    $mp(k) = \min\{j : j \in \mathcal{E}_k\}$    ! lowest numbered pivot in $k$
**end for**
**for** $col = 1, n$ **do**
    $\mathcal{L}_{col} = \left( \bigcup \{\mathcal{E}_k : mp(k) = col\} \right) \cup \{col\} \cup \left( \bigcup \{\mathcal{L}_i \backslash \{i\} : \pi(i) = col\} \right)$
    $\pi(col) = \min\{j : j \in \mathcal{L}_{col}, j \neq col\}$
**end for**
```

analyse phase within sparse Cholesky packages from the early 1980s (notably SPARSPAK [13], YSMP [11] and `MA27` [8, 9]). A simple but inefficient way to merge index lists is to use an array of size n to hold the indices, and to then compress this by removing unused entries (and order if required) once the merge is complete. The technique used in `MA27` is to scan each list to be merged in turn and to use a dense vector of length n as a flag array to avoid duplicates in the merged list (see also [27] for a variant that uses a boolean array of size n and a stack for the merged list). The entries may be ordered after the merge is complete. An alternative approach if the desired lists are to be ordered is to order the individual lists and then to merge two or more of these lists to form another ordered list without duplicates.

Regardless of the algorithm used to perform such merging, the operation is expensive, executing in time proportional to the number of entries in L. The sparsity pattern of every column of L is not needed, only that of the first column of each supernode. If supernodes can be identified without the need to find the index lists, the amount of merging can be substantially reduced. This was realised by Gilbert, Ng and Peyton [15], 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 input. An efficient analyse algorithm that uses this approach is summarised in Algorithm 3.

Algorithm 3 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.
```

This algorithm was incorporated by Ng and Peyton into their sparse Cholesky solver SPRSBLKLLT (details of this package are given in [24]) and their incomplete Cholesky factorization preconditioner [25]. In the mid 1990s, it was employed by Damhaug and Reid in the analyse phase of the HSL [20] package `MA46` for the direct solution of sparse unsymmetric linear systems of equations from finite-element applications [3]. A notable example of more recent use is the CHOLMOD package of Davis [2, 5]. An unsymmetric variant was developed by Gilbert, Li, Ng and Peyton [14] for use in QR and LU codes.

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 Algorithm 3 within the analyse phase for assembled matrices. `MA46` 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. This is equivalent to assembling the column of A (and it must be done each time it is needed). `MA46` requires that supervariables are explicitly identified by the user on input as lists of variables associated with a node of the finite-element mesh, avoiding any specific effort that would otherwise be needed to exploit their presence.

The supernode amalgamation algorithm used will influence the effectiveness of Algorithm 3. A full survey of available heuristics for supernode amalgamation is outside the scope of this paper. In our study and resulting software, we use a simple variant of the algorithm of Reid and Scott [28]. Given a user-defined parameter `nemin` ≥ 1, Reid and Scott pass through the assembly tree in natural order. In our implementation, the tree is postordered and so we use natural depth-first search order. A child is merged with its parent if either both parent and child have fewer than `nemin` variables that are eliminated or merging parent and child generates no additional nonzeros in L. The choice of `nemin` determines the level of supernode amalgamation, with a value in the range 8 to 32 typically recommended as providing a good balance between sparsity and efficiency in factorize and solve phases (see, for example, [18, 30]).

We end this section by outlining in Algorithm 4 the approach of Liu [23] for computing the elimination tree. 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. The algorithm exploits the characterization of the elimination tree as the first non-zero below the diagonal in the column of L. When scanning row i, the elimination tree for the $(i-1)\times(i-1)$ top-left submatrix of A is known. For a given non-zero, either it is the first non-zero below the diagonal in the column (it is a root in the elimination tree of the $(i-1)\times(i-1)$ submatrix) and may be set as the parent of j in the elimination tree, or it is not. In this case, it may cause fill-in in columns corresponding to a path from j to the node at the root of its elimination tree in the current submatrix. The same is true for each non-zero on this path, so only the root is of interest. As the root must be the first non-zero in its column, the parent of the root can be set to i. A virtual forest, denoted $\bar{\pi}(:)$ is stored that contains pointers to the current (or recent) root of a tree, enabling the root to be found in a small number of operations. The use of the virtual forest for path compression is essential to the complexity of this algorithm — without it, $\mathcal{O}(ne(L))$ operations would be executed rather than $\mathcal{O}(ne(A))$.

Algorithm 4 Find the elimination tree of an $n \times n$ assembled matrix (from Liu [23])

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

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` [10] and `HSL_MA77` [30]. Gilbert, Ng and Peyton did not incorporate supervariables within their original description of their analyse algorithm. For assembled problems, we can exploit the fact that the columns associated with the same supervariable are treated as having the same sparsity pattern by holding a single index list for these columns. We will refer to this as *column compression*. An alternative approach is to condense the matrix so that we are dealing with supervariables, rather than variables. If the average number of variables in each supervariable is k, column compression will reduce the amount of integer data read during the analyse phase by a factor of about k, while the condensed storage will reduce it by a factor of about k^2. Table 3.1 illustrates these savings. The problems in the top half of the table have only trivial supervariables ($k < 1.5$) so that (almost) no compression is possible. For those in the lower half of the table for which the number of supervariables is significantly less than the number of variables, we anticipate that the reduction in data for holding A will lead to improved efficiency in the rest of the analyse phase since we will effectively be working with a smaller problem and that, if the supervariables are sufficiently large, this will more than offset to cost of identifying supervariables. This is confirmed in Table 3.4 in Section 3.4.

Table 3.1: The storage savings resulting from using supervariables. The storage is the integer storage for holding the sparsity pattern of A with no compression, with column compression, and using the condensed form. *nsvar* denotes the number of supervariables; k is the average number of variables in each supervariable and *std* denotes the standard deviation.

					Storage (Mbytes)		
Problem	n (10^3)	*nsvar* (10^3)	k	*std*	No comp.	Column comp.	Condensed
Cunningham/qa8fk	66	66	1.00	0.01	12	12	12
Lin/Lin	256	256	1.00	0.00	13	13	13
Wissgott/parabolic_fem	526	526	1.00	0.00	28	28	28
CEMW/tmt_sym	727	727	1.00	0.00	38	38	38
McRae/ecology2	1000	1000	1.00	0.00	38	38	38
AMD/G3_circuit	1585	1585	1.00	0.00	58	58	58
TSOPF/TSOPF_FS_b300_c2	57	56	1.01	0.08	66	66	66
Gupta/gupta3	17	17	1.01	0.32	71	70	70
Andrianov/mip1	66	65	1.02	1.41	78	77	77
PARSEC/Ga41As41H72	268	268	1.00	0.00	141	141	141
ND/nd24k	72	72	1.00	0.00	219	219	219
Schenk/nlpkkt240	27994	27994	1.00	0.00	5908	5908	5908
Boeing/bcsstk39	47	10	4.61	1.18	15	3	<1
TKK/s4dkt3m2	90	15	5.93	0.45	28	4	<1
Rothberg/gearbox	154	56	2.74	1.31	69	25	11
Boeing/pwtk	218	42	5.25	1.70	88	16	3
DNVS/fullb	199	33	5.96	0.33	89	14	2
Chen/pkustk14	152	34	4.45	1.62	113	26	6
INPRO/msdoor	416	61	6.82	0.77	154	22	3
Koutsovasilis/F1	344	120	2.85	0.62	204	72	25
GHS_psdef/ldoor	952	138	6.92	0.52	354	51	7
Schenk_AFE/af_shell10	1508	302	5.00	0.00	401	80	16
Oberwolfach/bone010	987	328	3.01	0.18	546	182	60
GHS_psdef/audikw_1	944	314	3.00	0.09	592	197	65

3.1 Identification of supervariables

Having demonstrated that for some problems the use of supervariables can potentially lead to savings, we consider how we can efficiently identify supervariables. With careful choice of data structures, the whole process can be made to execute in $\mathcal{O}(n + ne)$ time. As already noted, supervariables are widely used and within the HSL library [20] there are a number of packages that implement different algorithms for identifying them. The algorithm used by the sparse direct solver `MA47` is described in [10]. `MC60` [31] is designed to reduce the bandwidth and profile of a sparse symmetric matrix. It is more efficient to do this for a condensed matrix; the algorithm it uses is outlined in Algorithm 5. The `MC60` implementation avoids complications involved in reusing empty supervariables by using an extra pass over the data to reuse them immediately. In numerical experiments we found that this extra pass can adversely effect performance, so instead we propose Algorithm 6, which is designed to combine the best of both the `MA47` and `MC60` implementations with a reordering of the if statements to allow better compiler optimization. Key features of Algorithm 6 are that the special case of trivial supervariables is handled efficiently and a stack is used to ensure new supervariables are established using space from those that have recently become empty, exploiting cache locality.

Algorithm 5 `MC60` supervariable determination for an $n \times n$ assembled matrix

```
Initialise $\mathcal{S}_1 = \{1, 2, \dots n\}$. $sv = 1$.
Initialise $next\_sv = 2$, $seen(:) = 0$.
for $j = 1, n$ do
  $count(sv) = |\mathcal{S}_{sv}|$ for $sv = 1$ to $next\_sv - 1$
  for $i : a_{ij} \neq 0$ do
    Let $sv$ be the supervariable to which $i$ belongs.
    $count(sv) = count(sv) - 1$
  end for
  for $i : a_{ij} \neq 0$ do
    Let $sv$ be the supervariable to which $i$ belongs.
    if $seen(sv) < j$ then
      if $(count(sv) = 0)$ cycle
      $map(sv) = next\_sv$
      $next\_sv = next\_sv + 1$
      $seen(sv) = j$
    end if
    Move $i$ from $\mathcal{S}_{sv}$ to $\mathcal{S}_{map(sv)}$
  end for
end for
```

Table 3.2 compares the runtime of Algorithm 6 with that of the `MA47` and `MC60` implementations. We see that, with the exception of problem McRae/ecology2, Algorithm 6 either matches or exceeds the performance of the other variants and is thus incorporated into our new analyse code.

3.2 Combining identifying supervariables and constructing elimination tree

If a problem has only trivial supervariables, the time spent searching for supervariables is wasted. One way to try to reduce this overhead is to identify supervariables at the same time as constructing the elimination tree. Table 3.3 shows the effect of combining the identification of supervariables with the determination of the elimination tree such that the matrix data is only read once. Column 2 reports the time for the elimination tree algorithm (Algorithm 4, no identification of supervariables), the next four columns are the times to identify supervariables using Algorithm 6 and rearrange the pivot order (denoted by Alg. 6), to use them to obtain a condensed matrix, to run the elimination tree algorithm on

Algorithm 6 Determine supervariables of an $n \times n$ assembled matrix

```
Initialise $\mathcal{S}_1 = \{1, 2, \dots n\}$.
Place $n$ empty supervariables on free supervariable stack.
Initialise $seen(:) = 0$
for $j = 1, n$ do
  for $i : a_{ij} \neq 0$ do
    Let $sv$ be the supervariable to which $i$ belongs.
    if $|\mathcal{S}_{sv}| = 1$ then ! $\mathcal{S}_{sv}$ contains a single variable
      if $seen(sv) < j$ cycle
      Move $i$ from $\mathcal{S}_{sv}$ to $\mathcal{S}_{map(sv)}$.
      Place empty $\mathcal{S}_{sv}$ to top of free stack.
    else
      if $seen(sv) < j$ then
        Take a new supervariable $\mathcal{S}_{new}$ from top of free stack.
        $map(sv) = new$
        $seen(sv) = j$
      end if
      Move $i$ from $\mathcal{S}_{sv}$ to $\mathcal{S}_{map(sv)}$.
    end if
  end for
end for
```

Table 3.2: Performance of Algorithm 6 and the `MC60` and `MA47` algorithms. Times are in seconds for identifying supervariables.

Problem	Alg. 6	`MC60`	`MA47`
Cunningham/qa8fk	0.01	0.01	0.01
Lin/Lin	0.01	0.02	0.01
Wissgott/parabolic_fem	0.03	0.05	0.05
CEMW/tmt_sym	0.05	0.06	0.05
McRae/ecology2	0.05	0.05	0.04
AMD/G3_circuit	0.08	0.09	0.08
TSOPF/TSOPF_FS_b300_c2	0.04	0.05	0.04
Gupta/gupta3	0.04	0.06	0.04
Andrianov/mip1	0.04	0.11	0.04
PARSEC/Ga41As41H72	0.08	0.14	0.09
ND/nd24k	0.10	0.19	0.10
Schenk/nlpkkt240	3.33	6.06	3.66
Boeing/bcsstk39	0.01	0.01	0.02
TKK/s4dkt3m2	0.02	0.02	0.03
Rothberg/gearbox	0.06	0.06	0.09
Boeing/pwtk	0.07	0.07	0.11
DNVS/fullb	0.07	0.08	0.11
Chen/pkustk14	0.10	0.10	0.14
INPRO/msdoor	0.13	0.14	0.20
Koutsovasilis/F1	0.21	0.21	0.32
GHS_psdef/ldoor	0.31	0.33	0.47
Schenk_AFE/af_shell10	0.35	0.35	0.48
Oberwolfach/bone010	0.43	0.45	0.60
GHS_psdef/audikw_1	0.50	0.55	0.72

this condensed matrix (denoted by Alg. 4c), and the total of these times. Finally, column 7 is the total time for running Algorithm 6 followed Algorithm 4 (this is the time without condensing the matrix), and column 8 is the time to combine finding the supervariables and the elimination tree. For problems with non-trivial supervariables, the latter is generally slower than the total time given in column 6, whereas if there are only trivial supervariables, the column 7 time (Alg. 6 + Alg. 4) is generally less than that in column 8. Clearly, finding the elimination tree without identifying supervariables gives the fastest times, however the condensed form will accelerate the subsequent steps of the analyse phase for problems with non-trivial supervariables, leading to savings overall (see Table 3.4). Note that for problems with trivial supervariables, the Alg. 4c times are slightly less than the Alg. 4 times because the data has been reordered and the diagonal entries dropped.

Table 3.3: The effect of combining the identification of supervariables and the construction of the elimination tree. Times are given in seconds.

		Find tree of condensed form					Combined
Problem	Alg. 4	Alg. 6	Condense	Alg. 4c	total	Alg. 6+Alg. 4	approach
Cunningham/qa8fk	0.0170	0.0084	0.0139	0.0140	0.0363	0.0254	0.0269
Lin/Lin	0.0276	0.0172	0.0218	0.0217	0.0608	0.0448	0.0522
Wissgott/parabolic_fem	0.0638	0.0454	0.0516	0.0486	0.1456	0.1092	0.1175
CEMW/tmt_sym	0.0821	0.0674	0.0684	0.0651	0.2008	0.1495	0.1458
McRae/ecology2	0.0722	0.0730	0.0725	0.0544	0.2000	0.1453	0.1455
AMD/G3_circuit	0.1245	0.1077	0.1119	0.0995	0.3191	0.2322	0.2570
TSOPF/TSOPF_FS_b300_c2	0.0340	0.0376	0.0566	0.0278	0.1219	0.0715	0.0627
Gupta/gupta3	0.0272	0.0347	0.0528	0.0270	0.1145	0.0619	0.0533
Andrianov/mip1	0.0480	0.0429	0.0610	0.0445	0.1484	0.0909	0.0853
PARSEC/Ga41As41H72	0.1856	0.0862	0.1603	0.1656	0.4121	0.2719	0.2811
ND/nd24k	0.1354	0.1015	0.1719	0.1282	0.4016	0.2369	0.2112
Schenk/nlpkkt240	7.2577	4.4085	8.4874	5.1427	18.039	11.666	12.641
Boeing/bcsstk39	0.0130	0.0130	0.0029	0.0009	0.0168	0.0261	0.0231
TKK/s4dkt3m2	0.0239	0.0231	0.0051	0.0012	0.0294	0.0470	0.0419
Rothberg/gearbox	0.0563	0.0668	0.0268	0.0110	0.1045	0.1230	0.1080
Boeing/pwtk	0.0695	0.0736	0.0176	0.0045	0.0957	0.1431	0.1231
DNVS/fullb	0.0694	0.0775	0.0152	0.0035	0.0962	0.1468	0.1235
Chen/pkustk14	0.0786	0.1078	0.0220	0.0074	0.1372	0.1863	0.1578
INPRO/msdoor	0.1291	0.1389	0.0272	0.0048	0.1708	0.2680	0.2293
Koutsovasilis/F1	0.1699	0.2103	0.0673	0.0292	0.3067	0.3802	0.3251
GHS_psdef/ldoor	0.3005	0.3252	0.0640	0.0109	0.4002	0.6258	0.5226
Schenk_AFE/af_shell10	0.3716	0.3805	0.1113	0.0257	0.5175	0.7521	0.6505
Oberwolfach/bone010	0.4358	0.4461	0.1635	0.0660	0.6755	0.8819	0.8189
GHS_psdef/audikw_1	0.4669	0.5161	0.1643	0.0775	0.7579	0.9830	0.8768

3.3 Column count algorithm incorporating supervariables

The algorithm we employ for computing column counts of L is a modification of the original algorithm of Gilbert, Ng and Peyton [15] (which also found row counts). Our variant is given in Algorithm 7. To cope with supervariables that consist of differing numbers of variables, it allows for variables to be weighted. We note that, in their work on union-find algorithms, Patwary et al.[26] recently described advances on the Gilbert, Ng and Peyton algorithm. However, their interleaved algorithms are not suitable for use in Algorithm 7.

Algorithm 7 Column count algorithm with weights

```
Input: Assembled matrix $A$, postordered elimination tree $\pi$, 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 virtual forest $\bar{\pi}(:)$ such that every node is its own tree.
  Initialise last_p(:) = 0, last_nbr(:) = 0.
  for each column $j$ of $A$ do
    for each $\{a_{ij} : j < i\}$ do
      ! non-zero in row j has not been inherited from descendant
      if first($j$) > last_nbr($i$) then
        cc($j$) = cc($j$) + wt($i$) ! new non-zero in current column
        $pp$ = last_p($i$)
        if $pp \neq 0$ 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($u$) = $j$
      end if
      last_nbr($u$) = $j$
    end for
    $\bar{\pi}(j) = \pi(j)$ ! add parent of j to tree containing j
    cc($\pi(j)$) = cc($\pi(j)$) + cc($j$) − wt($j$) ! Pass all uneliminated variables up tree to parent
  end for
```

3.4 Performance of the analyse phase with and without supervariables

Having looked at incorporating supervariables into the algorithms that are used to build the analyse phase, we end this section on the assembled case by presenting, in Table 3.4, times for running our new analyse code `HSL_MC78` both with and without the exploitation of supervariables. As expected, for the problems in the top half of the table that have only trivial supervariables, using the supervariable option adds an overhead. This is generally less than 20 per cent. For the problems in the lower half of the table, worthwhile gains can be achieved by using supervariables. For example, for problems DNVS/fullb and GHS_psdef/ldoor (for which the average number of variables per supervariable is 5.96 and 6.92, respectively) the analyse time is reduced by close to 50 per cent through the use of supervariables. Within `HSL_MC78` the use of supervariables is controlled by a parameter that may be set the user. The default for assembled problems is not to use supervariables.

Table 3.4: The performance of `HSL_MC78` with and without supervariables. Times are given in seconds.

Problem	With	Without
Cunningham/qa8fk	0.0612	0.0530
Lin/Lin	0.1467	0.1289
Wissgott/parabolic_fem	0.3336	0.2843
CEMW/tmt_sym	0.4453	0.3775
McRae/ecology2	0.5063	0.4366
AMD/G3_circuit	0.8535	0.7353
PARSEC/Ga41As41H72	0.7747	0.6742
TSOPF/TSOPF_FS_b300_c2	0.1592	0.1203
Gupta/gupta3	0.1095	0.0720
Andrianov/mip1	0.1850	0.1402
ND/nd24k	0.4770	0.3670
Schenk/nlpkkt240	36.970	32.410
Boeing/bcsstk39	0.0222	0.0419
TKK/s4dkt3m2	0.0399	0.0799
Rothberg/gearbox	0.1358	0.1732
Boeing/pwtk	0.1263	0.2236
DNVS/fullb	0.1212	0.2212
Chen/pkustk14	0.1606	0.2339
INPRO/msdoor	0.2140	0.4277
Koutsovasilis/F1	0.4127	0.5192
GHS_psdef/ldoor	0.5118	1.0183
Schenk_AFE/af_shell10	0.7250	1.3473
Oberwolfach/bone010	0.9795	1.3782
GHS_psdef/audikw_1	1.0794	1.4443

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 4.1, illustrate that 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. We have already observed that the solver `MA46` avoids assembling A by using an implicit adjacency structure. However, this approach is inefficient since 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.

Table 4.1: A comparison of the integer storage (in Mbytes) for the elemental, assembled and equivalent forms.

		Lower triangle only		Upper and lower triangles	
Problem	elemental	assembled	equivalent	assembled	equivalent
trdheim	0.34	7.64	0.50	14.9	1.01
opt1	0.41	7.54	0.47	14.8	0.94
tsyl201	0.45	9.60	0.53	18.9	1.06
crplat2	0.57	3.87	0.49	7.47	0.99
thread	0.98	17.4	1.01	34.3	2.01
ship_001	1.21	18.2	1.23	35.7	2.46
srb1	1.75	11.9	1.49	23.0	2.97
m_t1	2.08	38.3	2.53	75.2	5.06
x104	2.25	40.0	2.80	78.4	5.60
shipsec8	5.43	26.7	3.77	51.6	7.53
shipsec1	6.30	31.4	4.54	60.7	9.08
fcondp2	6.81	45.4	5.66	87.7	11.3
ship_003	7.08	32.2	4.58	62.6	9.15
troll	7.61	48.2	6.35	93.1	12.7
shipsec5	8.09	40.6	5.80	78.5	11.6
fullb	9.38	46.9	6.55	90.8	13.1
halfb	10.4	49.8	7.25	96.2	14.5

Lemma 1 *The pattern of the Cholesky factor L of the matrix $A = \sum_k A^{(k)}$, where each $A^{(k)}$ is non-zero 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 factorization. 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
Since $\hat{\mathcal{A}}_1 = \mathcal{A}_1$, the first column of L has the same sparsity pattern as the first column $\hat{L}$ (that is, $\hat{\mathcal{L}}_1 = \mathcal{L}_1$).
Proceed by induction: assume that all columns i with $i < j$ satisfy $\mathcal{L}_i = \hat{\mathcal{L}}_i$.
For each non-zero $p \in \mathcal{L}_j$ one of the following holds:

1. $p \in \mathcal{A}_j \cap \hat{\mathcal{A}}_j$**.** Since 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 $j, p \in \hat{\mathcal{L}}_i$. Since $j \leq p$, $p \in \hat{\mathcal{L}}_j$.

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 since $\hat{\mathcal{A}}_j \subseteq \mathcal{A}_j$, we conclude $\hat{\mathcal{L}}_j = \mathcal{L}_j$. □

From this lemma, it follows that, for each element k, we can remove all the entries from the sparsity pattern of the assembled matrix apart for those in the row and column corresponding to the variable in the element index list $\mathcal{E}_k$ that is first in the pivot sequence, and still obtain the same pattern for the factors after the symbolic factorization. We call the resulting matrix the *equivalent matrix*. We hold the equivalent matrix as an assembled matrix; the storage its requires is reported in Table 4.1. 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 is 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 6 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 4 of our elemental test examples, all the supervariables were found and for only one problem (opt1) was the number of missed supervariables more than 1 per cent. In Table 4.2 we report the number of supervariables together with 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.

Table 4.2: A comparison of the integer storage required for the condensed assembled and condensed equivalent forms (lower triangle only). *nsvar* denotes the number of supervariables.

	n	$nsvar$	Storage (Mbytes)	
Problem	(10^3)	(10^3)	assembled	equivalent
trdheim	22.10	2.868	0.31	0.17
opt1	15.45	3.802	0.91	0.12
tsyl201	20.68	2.881	0.36	0.16
crplat2	18.01	3.004	0.23	0.14
thread	29.74	8.838	3.17	0.23
ship_001	34.92	5.843	1.03	0.27
srb1	54.92	9.154	0.70	0.42
m_t1	97.58	17.04	2.54	0.74
x104	108.4	17.26	2.15	0.83
shipsec8	114.9	19.53	1.61	0.88
shipsec1	140.9	23.48	1.83	1.07
fcondp2	201.8	33.91	2.68	1.54
ship_003	121.7	20.29	1.87	0.93
troll	213.4	48.43	5.78	1.63
shipsec5	179.9	30.43	2.42	1.37
fullb	199.2	33.44	2.76	1.52
halfb	224.6	78.56	3.04	1.71

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 while the second applies the Liu algorithm [23] (Algorithm 4) to the equivalent matrix. For the former, we first characterize 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 8, where lists are merged into the tree one at a time.

Algorithm 8 may be applied to the original elemental form or to the supervariable elemental form; the times for our test problems are given in columns 2 and 4 of Table 4.3, respectively. The time to identify supervariables and obtain the condensed elemental matrix is given in column 3 (headed 'find svs'), while column 5 reports the total time for the supervariable elemental approach. 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.

Algorithm 8 Determine the elimination tree of an $n \times n$ elemental problem

```
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
        $current = next$ ! start at variable $i - 1$ of $\mathcal{E}_k$
        $next = work(i)$
        ! Ascend tree, placing entry $i \in \mathcal{E}_k$ into correct position and modifing tree as required
        while $current \neq work(i)$ and $current \leq n$ do
            if $work(i) < \pi(current)$ then swap $work(i)$ and $\pi(current)$
            $current = \pi(current)$
        end while
    end for
end for
```

Table 4.3 also reports times for the equivalent matrix approach, both with and without supervariables. In column 9, 'find svs' denotes the time to identity the supervariables, obtain the condensed matrix and determine the equivalent matrix. To build the equivalent matrix requires two passes of the element data. For each pivot p the first pass builds a linked list of the elements k for which p is the smallest index in $\mathcal{E}_k$. The second pass builds the equivalent matrix column-by-column, eliminating duplicates as they are encountered. Comparing column 2 with column 7 and column 4 with column 10, we see that Algorithm 4 applied to the equivalent matrix is faster than Algorithm 8. However, once the overheads involved in identifying supervariables and converting to the equivalent form are included, the fastest time for computing the elimination tree for approximately half the problems is the supervariable elemental algorithm time (column 5) whilst the equivalent matrix algorithm without supervariables (column 8) is fastest for the others. This motivated us to try and improve the efficiency of using supervariables by reducing the number of passes of the data required for the preprocessing.

Recalling that Algorithm 4 requires access to the lower triangle of L by rows (or equivalently, access to the upper triangle by columns), we have implemented three variants:

Variant 1 determines the column counts for an uncondensed equivalent matrix in the same pass as identifying the supervariables. A second pass places entries of the equivalent matrix in their final locations. This is then condensed using the supervariable and a modified version of Algorithm 4 is used to determine the elimination tree.

Variant 2 determines the supervariables on the first pass through the data and builds a lower triangular supervariable condensed equivalent matrix on the second pass, which also determines column counts needed in finding the upper triangular form. A pass through the condensed lower form is sufficient to place entries of the upper triangular matrix in their final locations. Algorithm 4 is then used to determine the elimination tree.

Variant 3 determines the column counts for an uncondensed equivalent matrix in the same pass as identifying the supervariables. These are then converted into upper limits on the column sizes of the condensed equivalent upper triangular matrix. A second pass through the elemental data builds both lower and upper triangular matrices, but requires an additional array of length equal to the number of supervariables to hold the lengths of each column in the upper triangular form. A version of Algorithm 4 that operates on this modified data structure is used to determine the elimination tree.

Timing results are given in Table 4.4. They show that, while each variant is faster than the time reported in the final column of Table 4.3, Variant 2 is consistently the fastest. Furthermore, Variant 2

Table 4.3: Comparison of the performance of the elemental and equivalent matrix approaches for constructing the elimination tree of a problem in elemental form. Times are given in seconds; for each problem, the fastest time for constructing the tree is in bold. 'find svs' denotes the time to identity the supervariables, condense the matrix and, in the equivalent approach, to build the equivalent matrix. 'total' denotes the total time to construct the elimination tree.

	elemental approach				equivalent matrix approach					
	original	condensed			original			condensed		
Problem	Alg. 8	find svs	Alg. 8	total	build	Alg. 4	total	find svs	Alg. 4	total
trdheim	0.0060	0.0006	0.0003	**0.0010**	0.0006	0.0005	0.0011	0.0007	0.0001	0.0015
opt1	0.0135	0.0009	0.0015	0.0024	0.0008	0.0007	**0.0015**	0.0010	0.0003	0.0021
tsyl201	0.0256	0.0008	0.0008	0.0016	0.0007	0.0007	**0.0014**	0.0009	0.0001	0.0018
crplat2	0.0127	0.0009	0.0007	**0.0016**	0.0009	0.0008	0.0017	0.0011	0.0001	0.0022
thread	0.0820	0.0018	0.0096	0.0114	0.0019	0.0019	**0.0038**	0.0023	0.0006	0.0048
ship_001	0.0367	0.0019	0.0022	**0.0041**	0.0024	0.0019	0.0043	0.0023	0.0004	0.0051
srb1	0.0611	0.0030	0.0030	0.0060	0.0033	0.0024	**0.0057**	0.0038	0.0005	0.0076
m_t1	0.0852	0.0041	0.0046	0.0088	0.0047	0.0038	**0.0084**	0.0056	0.0009	0.0112
x104	0.2002	0.0043	0.0079	0.0122	0.0047	0.0040	**0.0088**	0.0062	0.0009	0.0118
shipsec8	0.1895	0.0099	0.0114	**0.0213**	0.0145	0.0091	0.0235	0.0130	0.0014	0.0288
shipsec1	0.2262	0.0108	0.0118	**0.0226**	0.0171	0.0112	0.0282	0.0148	0.0016	0.0335
fcondp2	0.4800	0.0131	0.0218	0.0349	0.0133	0.0113	**0.0246**	0.0183	0.0019	0.0335
ship_003	0.2432	0.0118	0.0161	**0.0279**	0.0172	0.0123	0.0295	0.0155	0.0018	0.0345
troll	0.6562	0.0155	0.0569	0.0724	0.0178	0.0154	**0.0331**	0.0220	0.0034	0.0432
shipsec5	0.3343	0.0148	0.0182	**0.0331**	0.0251	0.0147	0.0398	0.0198	0.0021	0.0470
fullb	0.4395	0.0166	0.0229	**0.0395**	0.0250	0.0168	0.0418	0.0220	0.0024	0.0494
halfb	0.3185	0.0183	0.0192	**0.0375**	0.0277	0.0191	0.0468	0.0247	0.0027	0.0551

Table 4.4: Comparison of the total times (in seconds) for our three variants for computing a supervariable condensed lower triangular form and elimination tree for elemental matrices.

Problem	Variant 1	Variant 2	Variant 3
trdheim	0.0013	0.0008	0.0009
opt1	0.0017	0.0011	0.0012
tsyl201	0.0015	0.0010	0.0011
crplat2	0.0020	0.0011	0.0013
thread	0.0039	0.0026	0.0026
ship_001	0.0044	0.0025	0.0028
srb1	0.0065	0.0036	0.0044
m_t1	0.0099	0.0056	0.0066
x104	0.0103	0.0057	0.0070
shipsec8	0.0269	0.0128	0.0157
shipsec1	0.0319	0.0155	0.0188
fcondp2	0.0324	0.0182	0.0217
ship_003	0.0334	0.0166	0.0198
troll	0.0425	0.0240	0.0275
shipsec5	0.0459	0.0213	0.0254
fullb	0.0483	0.0241	0.0285
halfb	0.0538	0.0272	0.0322

is faster than the elemental algorithm. Since the difference between the Variant 2 timings and the time for Algorithm 8 on the condensed elemental form (column 4 in Table 4.3) is small, applying similar improvements to the latter is unlikely to yield a faster algorithm. Within our analyse code `HSL_MC78`, we employ Variant 2.

4.4 Comparison of using assembled and elemental forms

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 the condensed equivalent form for element 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 Table 4.5 we compare the performance of `HSL_MC78` 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.

Table 4.5: The performance of the analyse code `HSL_MC78` using the elemental mode and the assembled mode with and without supervariables. Times are given in seconds.

Problem	elemental	assembled	sup_assembled
trdheim	0.0022	0.0301	0.0161
opt1	0.0027	0.0276	0.0203
tsyl201	0.0025	0.0375	0.0198
crplat2	0.0025	0.0153	0.0095
thread	0.0061	0.0638	0.0486
ship_001	0.0059	0.0668	0.0391
srb1	0.0081	0.0561	0.0313
m_t1	0.0140	0.1550	0.0955
x104	0.0145	0.1604	0.0935
shipsec8	0.0238	0.1253	0.0723
shipsec1	0.0280	0.1519	0.0844
fcondp2	0.0365	0.2167	0.1214
ship_003	0.0298	0.1467	0.0832
troll	0.0470	0.2352	0.1550
shipsec5	0.0380	0.1989	0.1105
fullb	0.0433	0.2231	0.1281
halfb	0.0484	0.2414	0.1389

5 Block pivots

It is sometimes useful to specify block pivots a priori based on existing knowledge of the numerical properties of A. For example, for indefinite systems where A is of the form

$$\begin{pmatrix} H & C^T \\ C & 0 \end{pmatrix}, \tag{5.1}$$

it is well-known that it can be advantageous to allow 2×2 pivots to be included within the pivot sequence (and particularly in the case $H = 0$) to reduce the modifications that must be made to during the factorization phase. As far as we are aware, the only sparse symmetric solvers that currently allow the user to supply 2×2 pivots as part of the pivot sequence are the HSL packages `MA47` [10] and `HSL_MA77`

[29], although other codes (including PARDSIO [33]) exploit the structure of A when selecting a pivot sequence as part of the analyse phase. The user of these HSL packages specifies consecutive pivots in the elimination order to be part of a block pivot. These pivots then remain consecutive in the final elimination order that is passed to the factorization, and will share the same non-zero pattern in the factor L, making them part of the same supernode.

The incorporation of block pivots (of order 2 or greater) into the analyse phase can achieved through three modifications:

- The pivots are forced into a parent-child relation in the elimination tree by adding explicit entries on the subdiagonal of the block pivot (if not already present).
- Any permutations applied to the elimination order are also applied to a vector describing the block pivots to ensure the user can identify members of a block pivot after the analyse phase has completed.
- At the supernode amalgamation stage, all members of a block pivot are combined into the same supernode before any other nodes are allowed to amalgamate.

Difficulties arise with supernodes. The first consideration is which has priority when reordering a matrix. Consider the following 4×4 example

$$\begin{pmatrix} & x & & x \\ x & & x & \\ & x & & x \\ x & & x & \end{pmatrix},$$

If block pivots (1,2) and (3,4) are desired, but supervariables (1,3) and (2,4) are identified, only one of these matchings can be chosen such that the pairs are contiguous in the pivot order. We choose to honour the block pivots as these are more likely to be important for stability and speed in the factorization phase: supervariables are only a way of speeding up the symbolic manipulations during the analyse phase. Thus if the user specifies block pivots, supervariables are not exploited.

We observe that we could pre-merge block pivots into the same supervariable, before detecting other supervariables. The potential pitfall of this approach is demonstrated by the example given in Figure 5.1. If 1×1 pivots are used, there is no fill in the original sparsity pattern. Selecting 2×2 pivots (1,2), (3,4) and (5,6) introduces an additional three non-zeros. However, if block pivots are replaced with supervariables, the factor becomes dense. This is because the supervariables introduce fill to the left of each pivot that is not actually there. It is for this reason we do not utilise this approach.

Figure 5.1: Showing fill in patterns without block pivots or supervariables (x), with block pivots $\bullet$ and treating block pivots as supervariables $\circ$.

$$\begin{pmatrix} x & & & & & \\ x & x & & & & \\ \circ & \circ & x & & & \\ \bullet & x & x & x & & \\ \bullet & x & \bullet & x & x & \\ \circ & \circ & \circ & \circ & x & x \end{pmatrix}$$

To illustrate the benefits of including 2×2 pivots, we consider the (assembled) problems listed in Table 5.1. Again, they are taken from the University of Florida Sparse Matrix Collection. Each is of the form (5.1) with $H = 0$. A pivot sequence containing 2×2 pivots can be generated using the analyse phase of the HSL direct solver MA47, which was designed explicitly for the solution of indefinite problems with zeros on the diagonal (note that this is one of the orderings offered by the ordering package HSL_MC68). In

Table 5.1, we report the analyse and factorize times together with the number of delayed pivots (pivots that, because of stability considerations, are used in the factorization later than in the pivot sequence set up by the analyse phase) when our multifrontal solver `RAL_SYMF` [19] is run using the METIS ordering (1×1 pivots only) and the `MA47` ordering. `RAL_SYMF` incorporates `HSL_MC78` within its analyse phase and allows the user to supply 1×1 and/or 2×2 pivots. We see that there are no delayed pivots for the `MA47` ordering (the factorization phase is able to use the pivot sequence unmodified) but there are a large number for the METIS ordering, which takes no account of the zeros on the diagonal (note the number of delayed pivots can exceed n since if a pivot is delayed at more than one node of the assembly tree, it is counted at each such node). This in turn results in the factorization time for the METIS ordering being significantly greater than for the `MA47` ordering. This confirms our view that, in some instances, it is advantageous to allow a pivot sequence containing block pivots.

Table 5.1: Comparison of the performance of `RAL_SYMF` using the `MA47` and METIS pivot orderings. *ndelay* denotes the number of delayed pivots. Times for the analyse and factorize phases are in seconds.

Problem	n	Analyse		Factorize		*ndelay*	
		`MA47`	METIS	`MA47`	METIS	`MA47`	METIS
GHS_indef/aug2d	29008	0.0140	0.0119	0.2459	1.120	0	67801
GHS_indef/aug3d	24300	0.0403	0.0162	1.7007	21.58	0	310300
GHS_indef/dtoc	24993	0.0074	0.0066	0.0241	4.689	0	45282

6 Comparison of analyse phase timings

In this section, we present timings to show that `HSL_MC78` provides an efficient implementation of the algorithms used within a modern analyse phase and that incorporating it into two of our recent sparse direct solvers leads to these solvers having analyse phases whose performance compare very favourably with that of other state-of-the-art sparse direct solvers. The two solvers we have incorporated `HSL_MC78` into are `RAL_SYMF` and `HSL_MA87` [18]. The latter is designed for solving sparse symmetric systems (in assembled form) on multicore architectures using a DAG-based approach. In Table 6.1 we compare the performance of `RAL_SYMF` and `HSL_MA87` with the older and well-known HSL multifrontal codes `MA27` [8, 9] and `MA57` [6]. For the problems in the lower half of the table, `HSL_MC78`, `RAL_SYMF` and `HSL_MA87` are run with supervariable detection switched on (the other codes do not offer such an option). In all the experiments reported on in this section, in each test the same pivot order is supplied to all the solvers and this is generated using the METIS graph partitioning package [21, 22]. Otherwise, default settings are used for all control parameters.

Comparing the times in columns 2 and 3 and columns 2 and 4 of Table 6.1 we see that `HSL_MC78` generally accounts for a significant proportion of the time taken by the analyse phase. A notable exception is `HSL_MA87` run on problem PARSEC/Ga41As41H72. In this case, the time taken for `HSL_MA87` to set up the block data structures for the factorization phase takes most of the analyse time. It is clear from Table 6.1 that the employment of the supervariable variant of the Gilbert, Ng and Peyton algorithm results in `RAL_SYMF` and `HSL_MA87` being substantially faster than `MA27` and `MA57`. The benefits are most significant for the largest problems.

The other solvers we use in our experiments are listed in Table 6.2. 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. 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.

Table 6.1: Times (in seconds) for the new analyse code `HSL_MA78` and for the analyse phase of `HSL_MA87`, `RAL_SYMF`, `MA27` and `MA57`. † denotes did not complete within 3000 seconds.

Problem	`MC78`	`MA87`	`RAL_SYMF`	`MA27`	`MA57`
Cunningham/qa8fk	0.0530	0.0683	0.0699	0.1850	0.2065
Lin/Lin	0.1289	0.3139	0.2584	0.9039	1.1074
Wissgott/parabolic_fem	0.2843	0.3667	0.4261	0.5515	0.7051
CEMW/tmt_sym	0.3775	0.4878	0.5623	0.5474	0.7450
McRae/ecology2	0.4366	0.5866	0.6973	0.6102	0.8623
AMD/G3_circuit	0.7353	1.0995	1.2402	1.4038	1.9861
PARSEC/Ga41As41H72	0.6742	20.4627	2.4968	17.0149	18.1183
TSOPF/TSOPF_FS_b300_c2	0.1203	0.1867	0.1803	0.4753	0.3226
Gupta/gupta3	0.0720	0.0804	0.0828	0.2445	0.2124
Andrianov/mip1	0.1402	0.1593	0.1962	0.3824	0.3200
ND/nd24k	0.3670	0.8393	0.5809	3.8180	2.8939
Schenk/nlpkkt240	32.410	846.4	95.448	†	†
Boeing/bcsstk39	0.0222	0.0303	0.0336	0.0956	0.1061
TKK/s4dkt3m2	0.0399	0.0545	0.0587	0.1893	0.2221
Rothberg/gearbox	0.1358	0.1659	0.1874	0.4735	0.5005
Boeing/pwtk	0.1263	0.1676	0.1869	0.7262	0.6336
DNVS/fullb	0.1212	0.1835	0.1827	1.2702	0.8507
Chen/pkustk14	0.1606	0.2339	0.2358	1.2601	1.0654
INPRO/msdoor	0.2140	0.2993	0.3229	1.6663	1.0717
Koutsovasilis/F1	0.4127	0.5510	0.5429	4.0369	2.5116
GHS_psdef/ldoor	0.5118	0.7342	0.7689	4.8267	3.2226
Schenk_AFE/af_shell10	0.7250	1.1284	1.2044	3.7086	5.5576
Oberwolfach/bone010	0.9795	1.4384	1.3642	8.9039	9.9569
GHS_psdef/audikw_1	1.0794	1.7921	1.5352	18.634	11.727

Table 6.2: Sparse direct solvers used in our numerical experiments.

Code	Date/version	Authors/website
CHOLMOD [2]	3.2009/ v1.7.1	T. Davis `http://www.cise.ufl.edu/research/sparse/cholmod/`
PARDISO [32]	10.2009/ v4.0.0	O. Schenk and K. Gärtner `http://www.pardiso-project.org`
WSMP [17, 16]	06.2010/ v10.5.26	A. Gupta, IBM `http://www-users.cs.umn.edu/∼agupta/wsmp.html`

The results for these codes are presented in Tables 6.3 and 6.4. For problems in Table 6.4, 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. We conclude that the performance of the analyse phase of our new solvers 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.

Table 6.3: Times (in seconds) for the analyse phase of HSL_MA87, RAL_SYMF, PARDISO, WSMP and CHOLMOD. † denotes failed to complete.

Problem	MA87	RAL_SYMF	PARDISO	WSMP	CHOLMOD
Cunningham/qa8fk	0.0683	0.0699	0.3045	0.1083	0.0829
Lin/Lin	0.3139	0.2584	1.3036	0.2557	0.2028
Wissgott/parabolic_fem	0.3667	0.4261	0.8701	0.4403	0.4130
CEMW/tmt_sym	0.4878	0.5623	1.1865	0.5821	0.5640
McRae/ecology2	0.5866	0.6973	1.2506	0.6827	0.6638
AMD/G3_circuit	1.0995	1.2402	2.3494	1.2314	1.1570
PARSEC/Ga41As41H72	20.463	2.4968	14.361	2.8172	†
TSOPF/TSOPF_FS_b300_c2	0.2278	0.2214	0.9479	0.4550	0.3407
Gupta/gupta3	0.1218	0.1182	0.9686	0.4971	0.4235
Andrianov/mip1	0.2070	0.2357	1.2893	0.5337	0.3679
ND/nd24k	0.8393	0.5809	4.5038	1.7815	1.0483
Schenk/nlpkkt240	846.4	95.448	†	†	†
Boeing/bcsstk39	0.0303	0.0336	0.2286	0.1029	0.0821
TKK/s4dkt3m2	0.0545	0.0587	0.4156	0.1881	0.1538
Rothberg/gearbox	0.1659	0.1874	1.0595	0.4675	0.3679
Boeing/pwtk	0.1676	0.1869	1.2692	0.5693	0.4658
DNVS/fullb	0.1835	0.1827	1.3890	0.6116	0.4704
Chen/pkustk14	0.2339	0.2358	1.7716	0.7291	0.5655
INPRO/msdoor	0.2993	0.3229	2.2412	1.1063	0.8375
Koutsovasilis/F1	0.5510	0.5429	3.6837	1.7772	1.1028
GHS_psdef/ldoor	0.7342	0.7689	5.8729	2.5869	1.9596
Schenk_AFE/af_shell10	1.1284	1.2044	6.5910	2.8010	2.4545
Oberwolfach/bone010	1.4384	1.3642	10.889	3.9215	2.9509
GHS_psdef/audikw_1	1.7921	1.5352	12.799	4.7763	3.2384

7 Concluding remarks

In this paper, we have considered the key steps within the analyse phase of a modern sparse direct solver. Our starting point was the algorithm of Gilbert, Ng and Peyton for determining the column counts of the matrix factor L. This algorithm is more efficient than the earlier approach of Duff and Reid. We have incorporated supervariables into the Gilbert, Ng and Peyton algorithm and have shown that, for problems with a significant number of with non-trivial supervariables, worthwhile savings in terms of memory and time can be achieved. We have also considered problems in elemental form and have shown how the introduction of an equivalent matrix can avoid explicit assembly of the matrix and can lead to very fast analyse times. Finally, we have considered the incorporation of block pivots within the analyse phase and illustrated the potential benefits of this. One of our future directions of work will be to develop new algorithms for choosing pivot sequences containing block pivots that are efficient in that they lead to sparse factors but also require little modification during the factorization.

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. HSL_MC78 has been

Table 6.4: Times (in seconds) for the analyse phase of a range of solvers run on our set of element test problems.

Problem	MA57	RAL_SYMF	PARDISO	WSMP	CHOLMOD
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
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

incorporated into two of our recent sparse direct solvers (`RAL_SYMF` and `HSL_MA87`); the performances of the analyse phases of these solvers have been shown to compare favourably with those 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 their comments and to the authors of the codes run in Section 6 for access to their software. Thanks, as ever, to Tim Davis and Yifan Hu for the University of Florida Sparse Matrix Collection.

References

[1] C. Ashcraft and R. Grimes. The influence of relaxed supernode partitions on the multifrontal method. *ACM Trans. Math. Softw.*, 15:291–309, 1999.

[2] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam. Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Trans. Math. Softw.*, 35, 2008. Article 22 (14 pages).

[3] A. C. Damhaug and J. K. Reid. 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.

[4] T. A. Davis. The University of Florida sparse matrix collection. Technical Report, University of Florida, 2007. `http://www.cise.ufl.edu/~davis/techreports/matrices.pdf`.

[5] T. A. Davis and W. W. Hager. Dynamic supernodes in sparse Cholesky update/downdate and triangular solves. *ACM Trans. Math. Softw.*, 35, 2009. Article 27.

[6] I. S. Duff. MA57– a new code for the solution of sparse symmetric definite and indefinite systems. *ACM Trans. Math. Softw.*, 30:118–154, 2004.

[7] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, 1986.

[8] I. S. Duff and J. K. Reid. MA27 - a set of Fortran subroutines for solving sparse symmetric sets of linear equations. Technical Report AERE-R 10533, Harwell Laboratory, 1982.

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

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

[11] S. C Eisenstat, M. C. Gursky, M. H Schultz, and A. H Sherman. Yale sparse matrix package, i: The symmetric codes. *Intl. J. Numer. Methods Eng*, 18:1145–1151, 1982.

[12] A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prrentice-Hall Inc., Englewood Cliffs, New Jersey, 1981.

[13] A. George, J. W. H. Liu, and E. Ng. User guide for SPARSPAK: Waterloo sparse linear equations package. Technical Report CS-78-30 (Rev. Jan. 1980), Dept. of Computer Science, University of Waterloo, 1980.

[14] J. R. Gilbert, X. S. Li, E. G. Ng, and B. W. Peyton. Computing row and column counts for sparse QR and LU factorization. *BIT*, 41(4):693–710, 2001.

[15] J. R. Gilbert, E. G. Ng, and B. W. Peyton. An efficient algorithm to compute row and column counts for sparse Cholesky factorization. *SIAM J. Matrix Anal. Appl*, 15(4):1075–1091, 1994.

[16] A. Gupta. 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`.

[17] A. Gupta, M. Joshi, and V. Kumar. 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`.

[18] J. D. Hogg, J. K. Reid, and J. A. Scott. Design of a multicore sparse Cholesky factorization using DAGs. Technical Report RAL-TR-2009-027, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England, 2009. To appear in *SIAM J. Scientific Computing*.

[19] J. D. Hogg and J. A. Scott. `RAL_SYMF`: a multifrontal code for sparse symmetric systems. Technical Report RAL-TR-2010-0xx, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England, 2010.

[20] HSL. A collection of Fortran codes for large-scale scientific computation, 2007. See `http://www.cse.stfc.ac.uk/nag/hsl/`.

[21] G. Karypis and V. Kumar. METIS - family of multilevel partitioning algorithms, 1998. See `http://glaros.dtc.umn.edu/gkhome/views/metis`.

[22] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Scientific Computing*, 20:359–392, 1999.

[23] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Appl.*, 11(1):134–172, 1990.

[24] E. G. Ng and B. W. Peyton. Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM J. Sci. Comp.*, 14(5):1034–1056, 1993.

[25] E. G. Ng, B. W. Peyton, and P. Raghavan. A blocked incomplete Cholesky preconditioner for hierarchical-memory computers. In *Proceedings of the Fourth IMACS International Symposium on Iterative Methods in Scientific Computation. D.R Kincaid and A.C Elster, eds*, pages 211–222, 1999.

[26] M. Patwary, J. Blair, and F. Manne. Experiments on union-find algorithms for the disjoint-set data structure. *Experimental Algorithms*, pages 411–423, 2010.

[27] A Pothen and S. Toledo. Elimination structures in scientific computing. In *Handbook on Data Structures and Applications*. Chapman and Hall, 2001. Chapter 59, 29 pages.

[28] J. K. Reid and J. A. Scott. An out-of-core sparse Cholesky solver. Technical Report RAL-TR-2006-013, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England, 2006. Revised Nov. 2007.

[29] J. K. Reid and J. A. Scott. An efficient out-of-core sparse symmetric indefinite direct solver. Technical Report RAL-TR-2008-024, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England, 2008.

[30] J. K. Reid and J. A. Scott. An out-of-core sparse Cholesky solver. *ACM Trans. Math. Softw.*, 36(2), 2009. Article 9, 33 pages.

[31] J.K. Reid and J.A. Scott. Ordering symmetric sparse matrices for small profile and wavefront. *Intl. J. Numer. Methods Engrng.*, 45:1737–1755, 1999.

[32] O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Journal of Future Generation Computer Systems*, 20:475–487, 2004.

[33] O. Schenk and K. Gärtner. On fast factorization pivoting methods for symmetric indefinite systems. *Elec. Trans. Numer. Anal*, 23:158–179, 2006.