Algorithms **2013**, *6*, 702-725; doi:10.3390/a6040702

OPEN ACCESS

algorithms

ISSN 1999-4893

www.mdpi.com/journal/algorithms

Article

New Parallel Sparse Direct Solvers for Multicore Architectures

Jonathan Hogg and Jennifer Scott *

Scientific Computing Department, STFC Rutherford Appleton Laboratory, Harwell Oxford, OX11 0QX, UK; E-Mails: jonathan.hogg@stfc.ac.uk (J.H.); jennifer.scott@stfc.ac.uk (J.S.)

* Author to whom correspondence should be addressed; E-Mail: jennifer.scott@stfc.ac.uk; Tel.: +44-0-1235-445131.

Received: 9 August 2013; in revised form: 16 October 2013 / Accepted: 21 October 2013 / Published: 1 November 2013

Abstract: At the heart of many computations in science and engineering lies the need to efficiently and accurately solve large sparse linear systems of equations. Direct methods are frequently the method of choice because of their robustness, accuracy and potential for use as black-box solvers. In the last few years, there have been many new developments, and a number of new modern parallel general-purpose sparse solvers have been written for inclusion within the HSL mathematical software library. In this paper, we introduce and briefly review these solvers for symmetric sparse systems. We describe the algorithms used, highlight key features (including bit-compatibility and out-of-core working) and then, using problems arising from a range of practical applications, we illustrate and compare their performances. We demonstrate that modern direct solvers are able to accurately solve systems of order 10^6 in less than 3 minutes on a 16-core machine.

Keywords: sparse matrices; sparse linear systems; symmetric systems; direct solvers; multifrontal; supernodal; bit-compatibility; Fortran 95; OpenMP; parallel

---

1. Background and Motivation

The efficient solution of linear systems of equations is the cornerstone of a wide range of problems in computational science and engineering. In many cases, particularly when discretizing continuous problems, the system is large and the associated matrix A is sparse. Furthermore, for many applications, the matrix is symmetrically structured; sometimes, such as in some finite-element applications, A is positive definite, while in other cases, it is indefinite.

Consider the sparse linear system

$$Ax = b$$

where the $n \times n$ matrix A and the right-hand side b are given and it is required to find the solution x. This is normally done using either a direct method or an iterative method, with the use of the latter being generally dependent upon the existence of a suitable preconditioner for the particular problem being solved. Most sparse direct methods are variants of Gaussian elimination and involve the explicit factorization of A (or, more usually, a permutation of A) into the product of lower and upper triangular matrices, L and U. In the symmetric case, for positive-definite problems $U = L^T$ (Cholesky factorization) or, more generally, $U = DL^T$, where D is a (block) diagonal matrix. Forward elimination

$$Ly = b$$

followed by backward substitution

$$Ux = y$$

completes the solution process for each given right-hand side b. Such methods are important because of their generality and robustness. Indeed, black-box direct solvers are frequently the method of choice because finding and implementing a good preconditioner for an iterative method can be expensive in terms of developer time, with no guarantee that it will significantly outperform a direct method (but where a high-quality preconditioner is available for a given application, an iterative method can be the best option). For the "tough" linear systems arising from some applications, direct methods are currently the only feasible solution methods as no reliable preconditioner is available. However, for some problems, especially large three-dimensional applications, iterative methods have to be employed because of the memory demands of direct methods, which generally grow rapidly with problem size.

The first general-purpose sparse direct solvers were developed in the late 1960s and early 1970s. Since then, as computer architectures have changed and evolved, there has been constant interest in the design and development of new and efficient direct algorithms and accompanying software packages. The HSL mathematical software library [1] (formerly the Harwell Subroutine Library) has a particularly rich history of providing sparse direct solvers and, in recent years, a number of new shared memory parallel solvers have been added to HSL, the majority designed for sparse symmetric systems. As well as being more efficient than existing HSL codes for solving large problems, they address a number of additional issues. One is an out-of-core solver designed to deal with very large problems where the factors are unable to fit in memory. Another implements a new task-based parallel paradigm that allows high performance on multicore machines. Finally, the latest solver focuses on achieving bit-compatible results when run on any number of threads (see Section 2.5). The aim of this paper is to describe the key features of our new symmetric solvers and to illustrate and compare how they perform on problems from engineering applications on commodity desktop hardware. It is assumed throughout that the matrix A is symmetric.

For ease of reference, in Table 1 and below we summarise the new solvers from the HSL library that are used in this study. Each of these solvers can be run in parallel using OpenMP.

Table 1. Key features of the new HSL sparse direct solvers. Here, the year is when the solver was first included within HSL.

Package	Year	Real/ Complex	Positive definite	Indefinite	Out-of-core	Bit-compatible	Notes on parallelism
HSL_MA77	2006	Real	✓	✓	✓	✓	Node-level parallelism only.
HSL_MA86	2011	Both		✓			Fast task-based parallelism.
HSL_MA87	2009	Both	✓				Fast task-based parallelism.
HSL_MA97	2011	Both	✓	✓		✓	Constrained tree and node-level parallelism.

HSL_MA77 [2]: Solves very large sparse symmetric positive-definite and indefinite systems using a multifrontal algorithm; a key feature is out-of-core working, and there is an option to input the matrix A as a sum of (unassembled) finite elements.

HSL_MA86 [3]: Solves sparse symmetric indefinite systems using a task-based algorithm; designed for multicore architectures.

HSL_MA87 [4]: Solves sparse symmetric positive-definite systems using a task-based algorithm; designed for multicore architectures.

HSL_MA97 [5]: Solves sparse symmetric positive-definite and indefinite systems using a multifrontal algorithm, optionally using OpenMP for parallel computation; a key feature is bit-compatibility.

The remainder of this paper is organised as follows. In Section 2, we provide a brief overview of sparse direct methods (for further details, the reader is referred to the books [6,7] and the references therein). We explain the different phases that are common to all modern sparse direct solvers (not just those used in this study). There are two main approaches to performing the numerical factorization: supernodal and multifrontal. The codes HSL_MA86 and HSL_MA87 employ a supernodal approach, and this is explained in Section 3, while HSL_MA77 and HSL_MA97 implement a multifrontal algorithm and this is discussed in Section 4. For both approaches, we highlight how parallelism is exploited. Numerical results are presented in Section 5. Finally, concluding remarks are made in Section 6.

The main contribution of this paper is to present a brief overview of sparse direct methods, aimed at those who have an interest and need to solve such systems efficiently and accurately. Furthermore, through numerical experiments on a range of problems from different applications, we are able to demonstrate and compare the performance of different algorithms in a multicore environment. Note that it is beyond the scope of this paper to present a comprehensive and detailed study of all available direct solvers (see [8] for a comparison of serial codes that was carried out a few years ago).

2. Sparse Direct Algorithms

Sparse direct methods solve systems of linear equations by factorizing the matrix A, generally employing graph models to limit both the storage needed and work performed. Sparse direct solvers have a number of distinct phases. Although the exact subdivision depends on the algorithm and software being used, a common subdivision is as follows.

1. An ordering phase that exploits the sparsity (non-zero) structure of A to determine a pivot sequence (that is, the order in which the Gaussian elimination operations will be performed). The choice of pivot sequence significantly influences both memory requirements and the number of floating-point operations required to carry out the factorization.
2. An analyse phase that uses the pivot sequence to establish the work flow and data structures for the factorization. This phase generally works only with the sparsity pattern of A (this is the case for all the solvers in this study).
3. A factorization phase that performs the numerical factorization. Following the analyse phase, more than one matrix with the same sparsity pattern may be factorized.
4. A solve phase that performs forward elimination followed by back substitution using the stored factors. The solvers in this study all allow the solve phase to solve for a single right-hand side or for multiple right-hand sides on one call. Repeated calls to the solve phase may follow the factorization phase. This is typically used to implement iterative refinement (see, for example, [9]) to improve the accuracy of the computed solution.

2.1. Selecting an Ordering

During the past 30 years, considerable research has gone into the development of algorithms that generate good pivot sequences. An important class of ordering methods is based upon the minimum degree algorithm, first proposed in 1967 by Tinney and Walker [10]. Minimum degree and variants, including approximate minimum degree (AMD) [11,12] and multiple minimum degree [13], perform well on many small and medium-sized problems (typically, those of order less than 50,000). However, nested dissection (a term introduced by George and Liu [14]) has been found to work better for very large problems, particularly those from three-dimensional discretizations (see, for example, the results in [15]). Furthermore, a parallel solver is generally better able to exploit parallelism if a nested dissection ordering is used. Many direct solvers now offer a choice of orderings, including either their own implementation of nested dissection or an interface to a standard implementation, such as those provided by METIS [16,17], SCOTCH [18] or Zoltan [19]. Note that a nested dissection algorithm recursively bisects the matrix graph and then, when the subgraph is sufficiently small, switches to using a variant of minimum degree.

HSL offers a number of routines that may be used to compute a suitable ordering. Additionally, `HSL_MA97` implements the heuristic described in [20] for automatically choosing an ordering based on the size and sparsity of A.

2.2. The Analyse Phase

The aim of the analyse phase of our sparse symmetric direct solvers is to determine the pattern of the sparse Cholesky factor L such that

$$PAP^T = LL^T$$

where the permutation matrix P holds the elimination (pivot) order. 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 problems arising from finite-element applications, supervariables occur frequently as a result of each node of the finite-element mesh having multiple degrees of freedom associated with it.

After (optionally) determining supervariables, the analyse phase proceeds by building an *elimination tree*. This is a graph that describes the structure of the Cholesky factor in terms of data dependence between pivotal columns. This permits permutations of the elimination order that do not affect the number of entries in L to be identified and allows fast algorithms to be used in determining the exact structure of L. An extensive theoretical survey and treatment of elimination trees and associated structures in sparse matrix factorizations is given in [21].

A *supernode* is a set of contiguous columns of L with the same sparsity structure below a dense (or nearly dense) triangular submatrix. This trapezoidal matrix has zero rows corresponding to variables that are eliminated later in the pivot sequence at supernodes that are not ancestors in the elimination tree. This matrix can be compressed by holding only the nonzero rows, each with an index held in an integer. The resulting dense trapezoidal matrix is referred to as the *nodal matrix*. The condensed version of the elimination tree consisting of supernodes is referred to as the *assembly tree*. Supernodes are important as they can be exploited in the factorization phase to facilitate the use of efficient dense linear algebra kernels and, in particular, Level-3 Basic Linear Algebra Subroutines (BLAS kernels) [22]. These can offer such a large performance increase that it is often advantageous to amalgamate supernodes that have similar (but not exactly the same) nonzero patterns, despite this increasing the fill in L and the operation count [23]. Within the HSL solvers, node amalgamation is controlled by a user-defined parameter (which has a default setting chosen on the basis of extensive experimentation).

Determining supervariables, constructing an assembly tree, amalgamating supernodes and finding row lists (that is, lists of the variables that belong to each supernode) are common tasks that are performed by most modern sparse direct solvers. For efficiency in terms of both development and software maintenance, all the new HSL solvers, with the exception of `HSL_MA77`, use the same code to perform these tasks [24]. This common code is not suitable for use in `HSL_MA77`, since it requires that the matrix A is held in main memory: `HSL_MA77` is designed for very large systems and allows A to be held in files on disk (see Section 4.2). Thus, separate analyse routines were developed for `HSL_MA77` that read the entries of A as they are required into main memory.

2.3. An Introduction to the Factorize Phase

The factorization can be performed in many different ways, depending on the order in which matrix entries are accessed and/or updated. Possible variants include left-looking (fan-in), right-looking (fan-out) and multifrontal algorithms. The (supernodal) right-looking variant computes a (block) row and column of L at each step and uses them to immediately update all rows and columns in the part of the matrix that has not yet been factored. In the (supernodal) left-looking variant, the updates are not applied immediately; instead, before a (block) column k is eliminated, all updates from previous columns of L are applied together to the (block) column k of A. In the multifrontal method [25,26], the updates are accumulated; they are propagated from a descendant column j to an ancestor column k via all intermediate nodes on the elimination tree path from j to k. Two of the packages in this study (`HSL_MA77` and `HSL_MA97`) implement multifrontal methods, while `HSL_MA86` for indefinite systems

and HSL_MA87 for positive-definite systems use a supernodal left-right looking approach. In Sections 3 and 4, we discuss the two approaches in more detail and, in particular, consider how parallelism can be exploited.

For symmetric matrices that are positive definite, the factorization phase can use the chosen pivot order without modification. Moreover, the data structures determined by the analyse phase can be static throughout the factorization phase. For symmetric indefinite problems, using the pivot order from the analyse phase may be unstable or impossible because of (near) zero diagonal entries. Thus, in general, it is necessary to modify the pivot sequence to ensure numerical stability. If symmetry is to be maintained, 1×1 and 2×2 pivoting must be performed, and the resulting factorization takes the form

$$PAP^T = LDL^T$$

where D is a block diagonal matrix with 1×1 and 2×2 blocks. The stability of the factorization of symmetric indefinite systems was considered in detail in the paper by Ashcraft, Grimes and Lewis [27]. They showed that bounding the size of the entries of L, together with a backward stable scheme for solving 2×2 linear systems, suffices to show backward stability for the entire solution process. They found that the widely used strategy of Bunch and Kaufmann [28] does not have this property, whereas the threshold pivoting technique first used by Duff and Reid [25] in their original multifrontal solver does.

Duff and Reid choose the pivots one-by-one, with the aim of limiting the size of the entries $l_{i,j}$ in L

$$|l_{i,j}| < u^{-1} \tag{1}$$

where the threshold u is a user-set value in the range $0 < u \leq 1.0$. Suppose that q denotes the number of rows and columns of D found so far. Let $a_{i,j}$, with $i > q$ and $j > q$, denote an entry of the matrix after it has been updated by all the permutations and pivot operations so far. For a 1×1 pivot in column $j = q + 1$, the requirement for inequality (1) corresponds to the stability threshold test

$$|a_{q+1,q+1}| > u \max_{i>q+1} |a_{i,q+1}|. \tag{2}$$

However, it is not always possible to select a valid 1×1 pivot. It is sufficient to use 2×2 pivots in these cases (see, for example, Section 4.4 of [9]). Following Duff *et al*. [29], the corresponding stability test for a 2×2 pivot is

$$\left| \begin{pmatrix} a_{q+1,q+1} & a_{q+1,q+2} \\ a_{q+1,q+2} & a_{q+2,q+2} \end{pmatrix}^{-1} \right| \begin{pmatrix} \max_{i>q+2} |a_{i,q+1}| \\ \max_{i>q+2} |a_{i,q+2}| \end{pmatrix} < \begin{pmatrix} u^{-1} \\ u^{-1} \end{pmatrix} \tag{3}$$

where the absolute value notation for a matrix refers to the matrix of corresponding absolute values. The choice of u controls the balance between stability and sparsity: in general, the smaller u is, the sparser L will be, while the larger u is, the more stable the factorization will be. Tests (2) and (3) with the default value of 0.01 (selected after extensive numerical experimentation) are used by each of the sparse indefinite HSL solvers included in this study.

Should no suitable 1×1 or 2×2 pivot be available within a supernode, all remaining pivot candidates are passed up the assembly tree to the parent. These are known as *delayed pivots*. Delaying a pivot candidate results in additional fill-in and more work and so is undesirable. In the extreme case where

all pivots are delayed to the root of the assembly tree, the factorization becomes equivalent to a dense factorization of an $n \times n$ matrix. Delayed pivots can have serious consequences in terms of the time, as well as the memory and flops, required for the factorization. A number of strategies have been proposed to reduce the number of delayed pivots; these are reviewed and compared in [30] (see also [31]). Some of these strategies are offered as options by our HSL solvers. However, since they weaken the stability tests, the factorization and hence the computed solution may be less accurate, and we do not, therefore, employ them by default (and they are not used in the numerical experiments reported in Section 5).

2.4. The Importance of Scaling

Scaling the problem is a key part of solving large sparse linear systems. In addition to reducing the residual, in the indefinite case, a good scaling can help from a purely computational standpoint by reducing the number of delayed pivots and, hence, the size of the computed factors and overall solution time. In this case, the factorization of the scaled matrix $\overline{A} = SAS$ is computed, where S is a diagonal scaling matrix. How to find a good S is an open question. A number of options for scaling are included within HSL and these are discussed in [32,33]. The indefinite HSL solvers in this study allow the user to specify scaling factors on the call to the factorization. In addition, `HSL_MA86` and `HSL_MA97` will compute scaling factors using either an iterative method based on matrix equilibration or a weighted bipartite matching. The latter may optionally combine the scaling and the ordering, and this can be beneficial in reducing delayed pivots for tough indefinite problems [30]. For very large matrices that will not fit into main memory, `HSL_MA77` offers the option of scaling the matrix using an out-of-core iterative algorithm that aims to compute S so that the infinity norm or one-norm of each row and column of $\overline{A}$ is approximately equal to one. Since each iteration involves reading the stored matrix data from disk, this is expensive and is only recommended if the user is unable to temporarily hold the matrix in main memory and call one of the HSL scaling packages.

2.5. The Issue of Bit Compatibility

In some applications, users want bit compatibility (reproducibility) in the sense that two runs of the solver on the same machine with identical input data should produce identical output. Not only is this an important aid in debugging and correctness checking, some industries (for example, nuclear and finance) require reproducible results to satisfy regulatory requirements. For sequential solvers, achieving bit compatibility is not a problem. However, enforcing bit compatibility can limit dynamic parallelism, and when designing a parallel sparse direct solver, the goal of efficiency potentially conflicts with that of bit compatibility.

The critical issue is the way in which N numbers (or, more generally, matrices) are assembled

$$sum = \sum_{j=1}^{N} S_j$$

where the S_j are computed using one or more processors. The assembly is commutative but because of the potential rounding of the intermediate results, is not associative, so that the result sum depends on the order in which the S_j are assembled. A straightforward approach to achieving bit compatibility

is to enforce a defined order on each assembly operation, independent of the number of processors. As discussed in [34], this is a viable approach for multifrontal codes and is the approach used by HSL_MA97. Our supernodal codes, HSL_MA86 and HSL_MA87, were designed with performance as a key objective and do not compute bit compatible results.

Note that for bit compatibility, the BLAS must produce bit compatible results. This can be achieved by choosing a BLAS library that guarantees this. Often the serial variant will do so, but our experiments with parallel variants show that while the Intel MKL BLAS are bit compatible, some other common BLAS libraries are not.

3. Supernodal Task-Based Approach

*3.1. Positive-Definite Case (*HSL_MA87*)*

We consider first the case when A is positive definite and briefly discuss the approach used by our supernodal sparse Cholesky solver HSL_MA87. This code is designed for use on multicore processors and is described in detail in [4]. Motivated by the work of Buttari *et al.* [35,36] on efficiently solving dense linear systems of equations on multicore processors, the factorization is divided into tasks, each of which alters a single block of the factor L. The block size nb is a user-controlled parameter. The columns of each nodal matrix are split into blocks of nb columns (the last block column may have fewer than nb columns) and then each block column is split into blocks, each with nb rows (again, the last block may have fewer than nb rows). The tasks to be performed on each block are partially ordered and the dependencies between them implicitly represented by a directed acyclic graph (DAG), with a vertex for each task and an edge for each dependency. A task is ready for execution if and only if all tasks with incoming edges to it are completed. While the order of the tasks must obey the DAG, there remains much freedom for exploitation of parallelism. At the start of the computation, there is one task ready for each leaf of the assembly tree; the final task will be associated with the factorization of a root of the assembly tree.

The DAG-based approach was adopted for HSL_MA87 since it offers significant improvements over utilizing more traditional fork-join parallelism by block columns. It avoids requiring all threads to finish their tasks for a block column before any thread can move on to the next block column. It also allows easy dynamic work sharing when another user or an asymmetric system load causes some threads to become slower than others. Such asymmetric loading can be common on multicore systems, caused either by operating system scheduling of other processes or by unbalanced triggering of hardware interrupts.

The tasks within a sparse supernodal Cholesky factorization algorithm are as follows:

factorize_block(L_{diag}) This computes the traditional dense Cholesky factor L_{diag} of the triangular part of a block that is on the diagonal. If the block is trapezoidal, this is followed by a triangular solve of its rectangular part

$$L_{rect} \Leftarrow L_{rect} L_{diag}^{-T}.$$

solve_block(L_{dest}) This performs a triangular solve of an off-diagonal block by the Cholesky factor L_{diag} of the block on its diagonal, *i.e.*,

$$L_{dest} \Leftarrow L_{dest} L_{diag}^{-T}.$$

update_internal(L_{dest}, $scol$) This performs the update of the block L_{dest} by the block column $scol$ belonging to the same nodal matrix, *i.e.*,

$$L_{dest} \Leftarrow L_{dest} - L_r L_c^T$$

where L_r is a block within $scol$ and L_c is a submatrix in the same block column.

update_between(L_{dest}, $snode$, $scol$) This performs the update of the block L_{dest} by the block column $scol$ of a descendant supernode $snode$ *i.e.*,

$$L_{dest} \Leftarrow L_{dest} - L_r L_c^T$$

where L_r and L_c are submatrices of contiguous rows of $scol$ that correspond to the rows and columns of L_{dest}, respectively.

The dense Cholesky factorization within the **factorize_block** task may be performed using subroutine `_potrf` from the LAPACK library [37] (a software library for numerical linear algebra that is built using the BLAS routines to effectively exploit the caches on cache-based architectures). The other tasks may be performed using BLAS-3 subroutines.

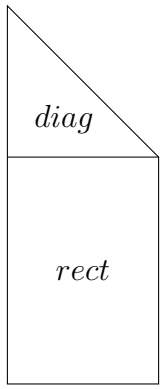
There are some restrictions on the order in which the tasks are performed; for example, the updating of a block of a nodal matrix from a block column of L that is associated with one of the supernode's descendants has to wait for all the rows of the block column that it needs to become available. At a moment during the factorize phase, some tasks will be executing while others will be ready for execution. Within `HSL_MA87` each task that is ready is held either in a local stack (one for each cache) or, if the stack is full, a global task pool.

Although the task dependencies can be represented by a DAG, in `HSL_MA87`, the whole DAG is not computed and stored explicitly. Instead, an implicit representation is used to determine if tasks are ready, and explicit lists of such tasks are maintained. As the order of the execution of tasks is not pre-defined, but can vary with the (essentially random) load on the machine, the order in which operations are performed can differ (but be equally valid) on different runs of the code on the same problem. Since floating-point addition is not associative, this leads to slightly different computed factors on each run and thus to solutions that are not bit-compatible.

3.2. Indefinite Case (`HSL_MA86`)

As already observed in Section 2.3, the main difference between the positive-definite and the indefinite cases is that, in the latter, it is necessary to include pivoting to ensure numerical stability. Consider the block column shown in Figure 1. In the indefinite case, large entries in the off-diagonal block, $rect$, may cause stability problems, unless they are taken into account when factorizing the diagonal block, $diag$. To be able to test for large entries, all the entries in $rect$ must be fully updated before $diag$ is factorized. To ensure this is the case, the indefinite solver `HSL_MA86` combines the **factorize_block** task and all the **solve_block** tasks for a block column into a single **factorize_column** task. Thus, the parallelism is less fine-grained and, for a matrix with the same sparsity pattern and the same block size nb, there are fewer tasks than in the positive-definite case

Figure 1. Trapezoidal block column, consisting of a square diagonal block, $diag$, and a rectangular off-diagonal block, $rect$.



If a pivot candidate is delayed because it is unstable (that is, it fails the tests (2) and (3)), some columns may be moved to different block columns and/or different nodes. For some degenerate problems in which many pivots are delayed, this can cause load balance issues resulting in a slow down of the solver.

Additional details of `HSL_MA86` and the differences between the positive-definite and indefinite DAG-based algorithms are given in [3].

4. Multifrontal Approach

4.1. Supernodal Multifrontal (`HSL_MA97`*)*

In a multifrontal method, the factorization of A proceeds using a succession of assembly operations into small dense matrices (the so-called *frontal matrices*), interleaved with partial factorizations of these matrices. For each pivot in turn, the multifrontal method first assembles all the rows that contain the pivot. This involves setting up a frontal matrix and adding the rows into it. A row of A that has been added to the frontal matrix is said to be *assembled*; rows that have not yet been assembled are referred to as *unassembled*. A partial factorization of the frontal matrix is performed (that is, the pivot and any other variables that are only involved in the assembled rows are eliminated). The computed columns of the matrix factor L are not needed again until the solve phase and so can be stored, while the rest of the frontal matrix (the *generated element* or *contribution block*), together with a list of the variables involved, is stored separately using a stack. At the next and subsequent stages, not only must unassembled rows of A that contain the pivot be assembled into the frontal matrix, but so too must any generated elements that contain the pivot. The method generalizes to a supernodal approach by working with blocks of pivots.

At each stage, the $m \times m$ frontal matrix can be expressed in the form

$$F = \begin{pmatrix} F_{11} & F_{21}^T \\ F_{21} & F_{22} \end{pmatrix} \tag{4}$$

where F_{11} and F_{21} are *fully summed*, that is, all the entries in the corresponding rows and columns of A have been assembled, while F_{22} is not yet fully summed. If F_{11} has order p and q pivots can be chosen stably from F_{11} (if A is positive definite, p pivots can be chosen in order down the diagonal, but in the

indefinite case, it may only be possible to select $q < p$ pivots stably), the partial factorization of F takes the form

$$F = Q \begin{pmatrix} L_1 & 0 \\ L_2 & I \end{pmatrix} \begin{pmatrix} D_1 & 0 \\ 0 & F_S \end{pmatrix} \begin{pmatrix} L_1^T & L_2^T \\ 0 & I \end{pmatrix} Q^T \tag{5}$$

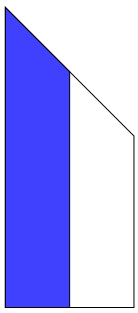
where Q is a permutation matrix of the form

$$Q = \begin{pmatrix} Q_1 & 0 \\ 0 & I \end{pmatrix}$$

with Q_1 of order p. If A is positive definite, L_1 is lower triangular, and $D_1 = I$, the identity matrix; if A is indefinite, L_1 is a unit lower triangular matrix of order q and D_1 is a block diagonal matrix of order q. Q_1, L_1 and D_1 are stored until the solve phase, while the Schur complement F_S is the generated element and is stacked.

For the multifrontal solver `HSL_MA97` we have developed dense linear algebra kernels to perform the partial factorization of the frontal matrices. To simplify the implementation of these, full storage of the (symmetric) frontal matrix is used, enabling blocking to be implemented within a recursive factorization scheme. Given an $m \times p$ fully summed block $\begin{pmatrix} F_{11} \\ F_{21} \end{pmatrix}$ to factorize, if p is small ($p \leq 16$), a *factorization kernel* is called. Otherwise, the block is divided in half, as shown in Figure 2. The factorization routine is called on the left half, and the right half is updated using the computed factors. The factorization routine is then called on the remaining fully summed columns. In the indefinite case, columns corresponding to delayed pivots are swapped to the end (of the right half) and are included in the factorization routine call with the remaining pivot candidates. As these calls to the factorization routine are recursive, multiple levels of division in both the left and right halves occur in practice.

Figure 2. The recursive dense factorization.



Once the factorization of the fully summed block is complete, update operations are performed on the $(2, 2)$ block of the frontal matrix (F_{22}) to compute the generated element.

Parallelism can be exploited in two ways in the multifrontal method:

Tree-level parallelism that performs assembly and factorization work associated with different frontal matrices on different threads.

Node-level parallelism that uses traditional dense linear algebra techniques to speed up the factorization of individual frontal matrices.

HSL_MA97 implements both types of parallelism and, as already remarked in Section 2.5, is designed to be bit compatible by enforcing a defined order on the assembly of the contributions to the frontal matrix F at each stage of the factorization. Note that if a nested dissection ordering is used, at most stages, only one or two contribution blocks must be assembled into F and thus the overhead from enforcing an ordering is not prohibitive.

*4.2. Out-of-Core Working (*HSL_MA77*)*

As the problem size grows, the computed factors and working space required by direct solvers generally increases significantly. This can lead to there being insufficient physical memory to store the data, particularly when the linear systems arise from discretizations of three-dimensional problems. In such cases, it may be possible to use a direct solver that is able to hold its data structures on disk, that is, an out-of-core solver. HSL_MA77 is designed as an out-of-core multifrontal solver.

The multifrontal method needs data structures for the original matrix A, the frontal matrix F, the stack of generated elements, and the matrix factor L. As already observed, computed columns of L are not needed again until the solve phase, so an out-of-core method writes the columns of L to disk as they are computed. If A, the stack and F are held in main memory and only the factor written to disk, the minimum possible input/output for an out-of-core method is performed: it writes the factor data to disk once and reads it once for the forward elimination and once for the back substitution. However, for very large problems, it may be necessary to hold further data on disk. HSL_MA77 is designed to also allow the stack and the original matrix data to be stored on disk, leaving only F to be held in main memory.

Separate dense kernels were designed for use within HSL_MA77 [2,38]. These employ block algorithms and, as in our other sparse solvers, the computation is performed using BLAS routines. A key feature that is important when attempting to minimise memory requirements, but does lead to complications in the coding and some copying overheads, is that storage for only the lower triangular part of F is required.

Writing the data to disk is done efficiently within HSL_MA77 through the use of a purpose-written system for virtual memory management. By exploiting information known by the programmer about when data will next be required, it is possible to significantly outperform the operating system's virtual memory manager. The virtual memory management system used by HSL_MA77 is described by Reid and Scott [39] (see [2] for further details).

HSL_MA77 was designed primarily as a serial code. However, an option is offered to exploit node-level parallelism. The requirement to use multiple stacks when exploiting tree-level parallelism, and the lack of appropriate locking within the virtual memory routines means that the exploitation of tree-level parallelism within HSL_MA77 would be difficult.

We note that HSL_MA77 allows the matrix A to be input both by square symmetric elements and by rows. Furthermore, the elements or rows may be entered one at a time, on separate calls to an input routine. This form of input reduces main memory requirements by avoiding the need to assemble A and is particularly convenient for many large-scale engineering applications that employ a finite-element approach. All the other solvers in this study require A to be input by rows in a single call.

5. Numerical Experiments

A range of test problems from different application areas have been chosen. Most are taken from the University of Florida Sparse Matrix Collection [40]; some additional large problems are from Anshul Gupta of IBM. All are supplied in assembled form and so we do not report on the element entry option offered by HSL_MA77. The problems are divided into those that are positive definite (Test Set 1) and those that are indefinite (Test Set 2). We report the order n and number of entries $nz(A)$ in A. We also give the expected number of entries $nz(L)$ and the expected number of floating-point operations $nflops$ to compute L when HSL_MA97 is used with its default settings (that is, the number of entries in L and the number of floating-point operations if the supplied pivot sequence is used without modification). Note that within each test set, the problems are listed in increasing order of their operation counts.

All tests are run in double precision on an 16-core machine with 2 Intel Xeon E5-2687W processors with 64 GB of memory. The MKL BLAS (version 11.0 update 1) and ifort compiler (version 12.1.0 with flags -g -fast -openmp) are used. All runs are performed on 16 cores. The solvers use their default settings, except that a METIS nested dissection ordering is always used, a scaling is supplied and, in the case of HSL_MA77, support for 64-bit addressing is enabled and the file size for the virtual memory system is increased to 512 MB per file. Solvers are limited to using at most 64 GB of virtual memory to avoid paging.

The supplied scaling is calculated using the HSL package MC77 run for one iteration in the infinity-norm, followed by up to three iterations in the one-norm, as recommended by Ruiz and Uçar in [41].

In each test, the right-hand side is constructed to correspond to the solution $x = 1$. The runs incorporate up to 5 iterations of iterative refinement with the following termination condition on the scaled backwards error

$$\frac{\|Ax - b\|_\infty}{\|A\|_\infty \|x\|_\infty + \|b\|_\infty} \leq 10^{-14}. \tag{6}$$

For most problems, each of the solvers took the same number of iterations to reach the required accuracy. The solvers all failed to achieve the required accuracy for problem 44 (GHS_indef/bmw3_2), which is singular. HSL_MA86 and HSL_MA97 only just fail to achieve the desired accuracy on the non-singular problem 45 (Oberwolfach/t3dh), achieving an accuracy of 1.1×10^{-14}. HSL_MA77 was the only code to successfully factorize problem 49 (Zaoui/kkt_power) but this problem is also singular, and the required accuracy was not obtained.

Test Set 1: Positive-definite problems.
(*) indicates the problem was supplied by Anshul Gupta.

Index	**Name**	n (10^3)	$nz(A)$ (10^6)	$nz(L)$ (10^6)	$nflops$ (10^9)	**Description**
1.	GHS_psdef/vanbody	47.0	2.32	6.35	1.40	Structural
2.	GHS_psdef/oilpan	73.8	2.15	7.00	2.90	Structural
3.	GHS_psdef/s3dkq4m2	90.4	4.43	18.9	7.33	Structural
4.	Wissgott/parabolic_fem	525.8	3.67	31.0	7.46	CFD
5.	Schmid/thermal2	1,228	8.58	63.0	15.1	Steady state thermal
6.	Boeing/pwtk	217.9	11.5	50.8	22.9	Structural: wind tunnel
7.	GHS_psdef/crankseg_1	52.8	10.6	34.0	32.5	Structural
8.	Rothberg/cfd2	123.4	3.09	40.0	33.0	CFD
9.	DNVS/shipsec1	140.9	3.57	40.5	38.3	Structural: ship section
10.	DNVS/shipsec5	179.9	4.60	55.3	57.7	Structural: ship section
11.	AMD/G3_circuit	1,585	7.66	118.8	58.7	Circuit simulation
12.	GHS_psdef/bmwcra_1	148.8	10.6	71.8	61.5	Structural
13.	Schenk_AFE/af_5_k101	503.6	17.6	103.6	61.6	Structural: sheet metal forming
14.	Um/2cubes_sphere	101.4	1.65	46.5	75.2	Electromagnetics: 2 cubes in a sphere
15.	GHS_psdef/ldoor	952.2	42.5	154.7	79.9	Structural
16.	DNVS/ship_003	121.7	3.78	62.0	81.9	Structural: ship structure
17.	Um/offshore	259.8	4.24	88.4	106.3	Electromagnetics: transient field diffusion
18.	GHS_psdef/inline_1	503.7	36.8	179.6	146.1	Structural
19.	GHS_psdef/apache2	715.2	4.82	148.6	176.0	Structural
20.	ND/nd24k	72.0	28.7	321.7	2,057	2D/3D
21.	Gupta/nastran-b (*)	1,508	56.6	1,071	3,174	Structural
22.	Janna/Flan_1565	1,565	114.2	1,501	3,868	Structural: steel flange
23.	Oberwolfach/bone010	983.7	47.9	1,092	3,882	Model reduction: trabecular bone
24.	Janna/StocF-1465	1,465	21.0	1,149	4,391	CFD: flow with stochastic permeabilities
25.	GHS_psdef/audikw_1	943.7	77.7	1,259	5,811	Structural
26.	Janna/Fault_639	638.8	27.2	1,156	8,289	Structural: faulted gas reservoir
27.	Gupta/sgi_1M (*)	1,522	63.6	2,049	9,017	Structural
28.	Janna/Geo_1438	1,438	60.2	2,492	18,067	Structural: geo mechanical deformation model
29.	Gupta/ten-b (*)	1,371	54.7	3,298	33,095	3D metal forming
30.	Gupta/algor-big (*)	1,074	42.7	3,001	39,920	Stress analysis

For comparison purposes, we also include results for an older, but very widely used and well-known, HSL code `MA57` [42]. This is a multifrontal solver that is primarily designed for the solution of symmetric indefinite systems. It was not written as a parallel code, but parallel performance can be achieved by using multithreaded BLAS. To make the comparisons as fair as possible, we do not use the default scaling offered by `MA57`, but since `MA57` does not allow scaling factors to be input, we prescale A using the same scaling strategy as used by the other solvers.

Tables 2 and 3 show the execution times for the complete solution of $Ax = b$, including the time for ordering, analysis, scaling, factorization and iterative refinement. We see that all the problems that fit within the 64 GB of available physical memory (that is, all except problems 49, 54 and 55) complete in under 3 minutes using the fastest solver.

Test Set 2: Indefinite problems.

Index	**Name**	n (10^3)	$nz(A)$ (10^6)	$nz(L)$ (10^6)	$nflops$ (10^9)	**Description**
31.	GHS_indef/dixmaanl	60.0	0.30	0.61	0.007	Optimization
32.	Marini/eurqsa	7.3	0.007	0.29	0.03	Time series
33.	IPSO/HTC_336_4438	226.3	0.78	2.98	0.12	Power network
34.	TSOPF/TSOPF_FS_b39_c19	76.2	1.98	4.40	0.29	Transient optimal power flow
35.	GHS_indef/stokes128	49.7	0.56	2.98	0.37	CFD
36.	GHS_indef/mario002	389.9	2.10	8.09	0.55	2D/3D
37.	Boeing/bcsstk39	46.8	2.06	7.92	2.20	Structural: solid state rocket booster
38.	GHS_indef/cont-300	180.9	0.99	11.7	2.96	Optimization
39.	GHS_indef/turon_m	189.9	1.69	13.7	4.23	2D/3D: mine model
40.	GHS_indef/bratu3d	27.8	0.17	6.28	4.42	Optimization
41.	GHS_indef/d_pretok	182.7	1.64	14.6	5.06	2D/3D: mine model
42.	GHS_indef/copter2	55.5	0.76	10.4	5.49	CFD: rotor blade
43.	Cunningham/qa8fk	66.1	1.66	24.3	21.3	Acoustics
44.	GHS_indef/bmw3_2	227.4	11.3	49.1	29.8	Structural
45.	Oberwolfach/t3dh	79.2	4.35	48.1	69.1	Model reduction: micropyros thruster
46.	Dziekonski/gsm_106857	589.5	21.8	137.1	82.6	Electromagnetics
47.	Schenk_IBMNA/c-big	345.2	2.34	52.0	115	Optimization
48.	Schenk_AFE/af_shell10	1,508	52.3	368	393	Structural: sheet metal forming
49.	Zaoui/kkt_power	2,063	12.8	217	562	Optimal power flow
50.	Dziekonski/dielFilterV2real	1,157	48.5	607	1,296	Electromagnetics: dielectric resonator
51.	PARSEC/Si34H36	97.6	5.16	486	4,267	Quantum chemistry
52.	PARSEC/SiO2	155.3	11.3	1037	13,249	Quantum chemistry
53.	PARSEC/Si41Ge41H72	185.6	15.0	1,411	20,147	Quantum chemistry
54.	Schenk/nlpkkt80	1,062	28.2	2,282	29,265	Optimization
55.	Schenk/nlpkkt120	3,542	95.1	13,684	143,600	Optimization

It is clear that the task-based code `HSL_MA87` is the fastest solver for the positive-definite problems, although the multifrontal solver `HSL_MA97` is competitive for the smallest problems in Test Set 1. The compromises that had to be made in adapting the task-based design to incorporate pivoting in the indefinite case incur overheads. These are significant for the smaller problems in Test Set 2, and for these problems, `HSL_MA97` generally outperforms `HSL_MA86`. By comparing the columns for `MA57` and `HSL_MA97`, we see that the new multifrontal code offers significant advantages over the older code. We also see that the out-of-core code `HSL_MA77` is competitive with `MA57` and, in particular, significantly outperforms it on larger problems. As `MA57` is an older code, it was written without thought to using 64-bit integers for internal addressing of its arrays and this means that it cannot tackle the large problems 27–30 that the other solvers can.

Table 2. Execution time in seconds for the complete solution of $Ax = b$ for positive-definite problems (Test Set 1). Times within 5% of the fastest are in bold. + indicates `MA57` was unable to complete the factorization, as it is internally restricted to using 32-bit integer addressing.

Problem	MA57	HSL_MA77	HSL_MA87	HSL_MA97
1.	0.51	0.56	**0.24**	**0.25**
2.	0.71	0.72	**0.24**	**0.24**
3.	1.27	1.12	**0.33**	0.37
4.	3.90	5.47	**2.66**	**2.74**
5.	9.55	13.0	**7.11**	**7.24**
6.	3.66	3.17	**0.94**	0.99
7.	3.22	2.46	**0.83**	1.06
8.	3.87	3.54	**1.48**	1.64
9.	3.38	2.58	**0.73**	0.94
10.	4.99	3.46	**0.95**	1.68
11.	14.4	18.5	**8.16**	**8.34**
12.	6.23	5.00	**1.53**	1.71
13.	7.55	6.88	**1.75**	1.98
14.	5.40	4.21	**1.37**	1.71
15.	11.8	11.7	**3.69**	4.12
16.	6.49	3.95	**1.08**	1.47
17.	9.61	8.82	**3.02**	3.74
18.	16.4	13.0	**5.20**	5.68
19.	15.7	16.0	**5.07**	5.72
20.	120	78.3	**15.0**	40.0
21.	155	97.8	**30.0**	40.0
22.	164	184	**29.2**	40.0
23.	144	118	**24.2**	33.7
24.	177	155	**33.2**	43.1
25.	194	159	**33.0**	49.7
26.	268	158	**36.4**	64.1
27.	+	186	**55.9**	76.4
28.	+	244	**78.6**	120
29.	+	398	**137**	223
30.	+	452	**162**	270

Table 3. Execution time in seconds for the complete solution of $Ax = b$ for indefinite problems (Test Set 2). Times within 5% of the fastest are in bold. - indicates insufficient memory to perform factorization; † indicates requested accuracy not achieved.

Problem	MA57	HSL_MA77	HSL_MA86	HSL_MA97
31.	**0.13**	0.21	0.15	0.15
32.	-	0.11	0.53	**0.09**
33.	**1.30**	1.67	1.41	**1.29**
34.	1.42	1.05	0.83	**0.66**
35.	0.55	0.44	**0.29**	**0.29**
36.	2.28	2.54	1.82	**1.56**
37.	0.58	0.52	**0.23**	**0.24**
38.	4.33	2.54	1.29	**1.16**
39.	1.76	1.93	**1.19**	**1.16**
40.	5.77	1.75	**0.79**	1.01
41.	2.71	1.99	**1.20**	**1.16**
42.	1.11	1.15	**0.51**	0.57
43.	2.38	1.96	**0.97**	1.16
44.	†6.94	†6.31	†**2.52**	†**2.59**
45.	6.53	4.91	†3.68	†**3.28**
46.	9.73	16.1	**7.18**	**7.49**
47.	7.01	11.4	**4.29**	8.09
48.	23.9	22.4	**8.00**	9.16
49.	-	†**4,891**	-	-
50.	1,522	266	-	**73.8**
51.	265	178	**31.4**	81.2
52.	685	397	**84.6**	196
53.	2,508	600	**118**	301
54.	-	**2,168**	-	-
55.	-	**44,649**	-	-

Previous testing on machines with less memory has shown that of the in-core codes, the supernodal codes (HSL_MA86 and HSL_MA87) are able to solve larger problems than the multifrontal codes (MA57 and HSL_MA97). This is because they do not maintain significant storage, other than that needed for the factors, whereas the multifrontal method additionally requires a stack.

All the problems are successfully factorized by the out-of-core solver HSL_MA77. Theoretically, the only restrictions on the size of problem HSL_MA77 can solve is the amount of disk space available to store the factors, the fact that 64-bit integers are used to address the virtual memory used and the size of front that can be stored in main memory. In practice, it is the latter restriction that is the limiting factor. It would be possible to adapt the dense factorization kernel used by HSL_MA77 to overcome this by holding only part of the frontal matrix in memory at any one time. However, while such an adaptation is conceptually straightforward, it would be time-consuming to implement and is beyond the scope of this paper. An alternative strategy to try and limit the amount of main memory required would be to weaken the stability criterion by using a smaller pivot threshold u, with the aim of reducing the number of delayed pivots (the large front is because there is a large number of delayed pivots). The consequences

of this would be a less accurate factorization, and more steps of iterative refinement (which is costly in the out-of-core case) would be needed to try and restore accuracy. A mixed precision approach could also be used in which the factorization is computed in single precision (requiring less memory) and then double precision accuracy recovered using refinement. This is discussed further in [43].

Figures 3 and 4 illustrate the slow down of HSL_MA97 (bit-compatible) and HSL_MA77 (out-of-core) compared to HSL_MA86/7. This provides some indication of the penalty incurred by imposing bit compatibility. The out-of-core code (which is also bit compatible) is typically 2–4 times slower than the in-core code, but note that this overhead is not only because of working out-of-core; it is also the result of the fact that HSL_MA77 only exploits node-level parallelism.

Finally, Figures 5 and 6 show the proportion of time spent in each phase of the multifrontal solver HSL_MA97 as a percentage of the total time (the time for the factorize phase includes the time taken for scaling). Ordering represents a significant portion (over 50% in a number of cases) for all except the largest problems (the highest numbered problems in each test set), where the factorize phase dominates. This is due in part to Amdahl's Law, because the ordering phase has not been parallelized. While parallel implementations of nested dissection exist, they have historically only been used when the matrix A cannot be stored in the memory of a single node, as the quality of ordering produced (and thus the number of operations and time for the subsequent factorize phase) is poorer than the serial implementations. It may be time to re-evaluate this conventional wisdom, at least for problems of medium size.

Figure 3. Comparative slow down of the bit-compatible code (HSL_MA97) and the out-of-core code (HSL_MA77) compared with HSL_MA87. Positive-definite problems.

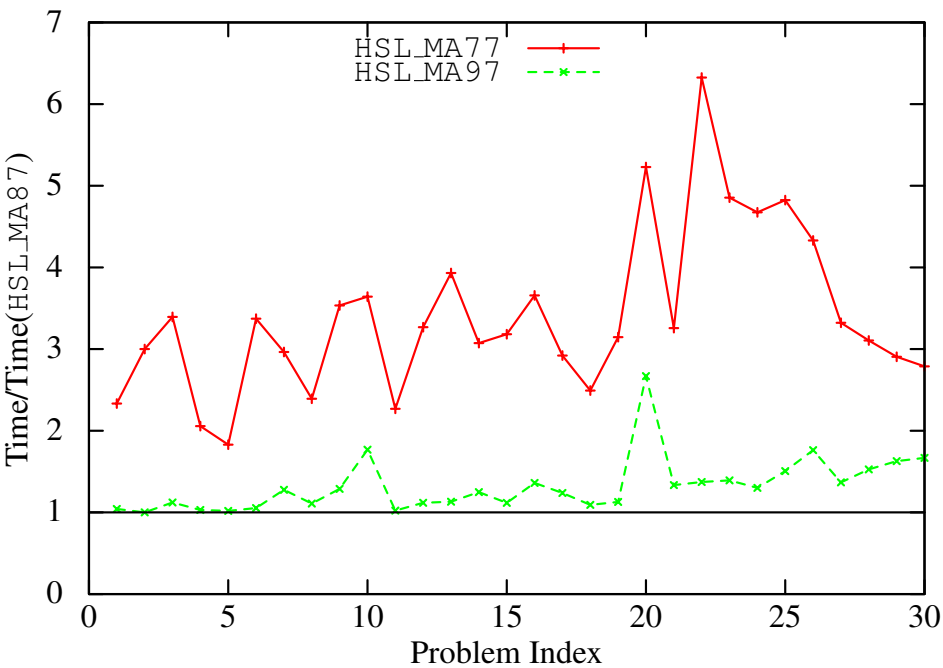


Figure 4. Comparative slow down of the bit-compatible code (HSL_MA97) and the out-of-core code (HSL_MA77) compared with HSL_MA86. Points below the line represent speedup. Indefinite problems.

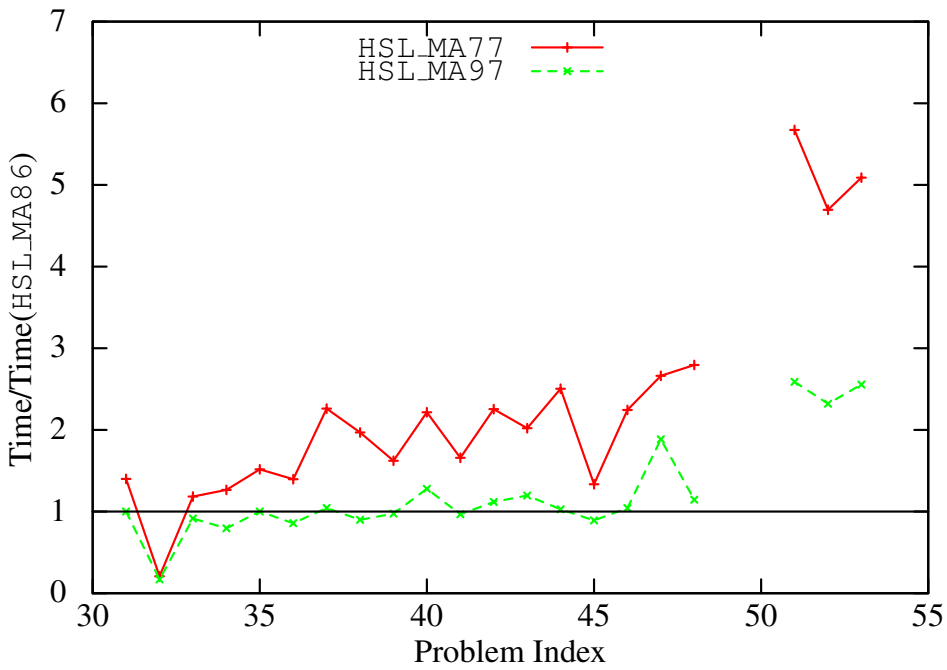


Figure 5. Cumulative proportions of the time for different phases of HSL_MA97 (gaps between curves represent the proportion of the total time spent in that phase). Positive-definite problems.

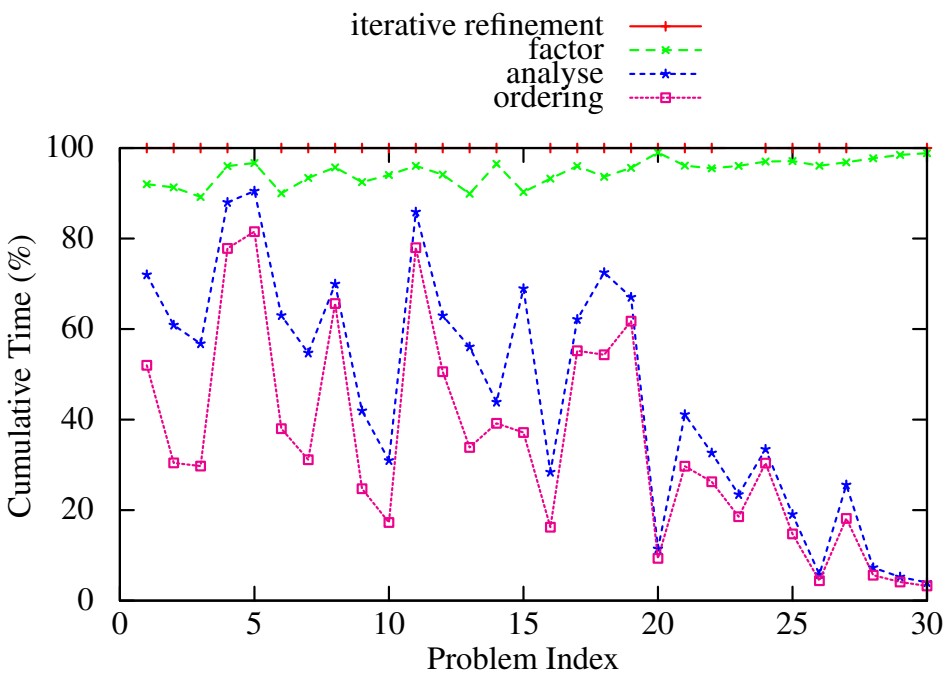
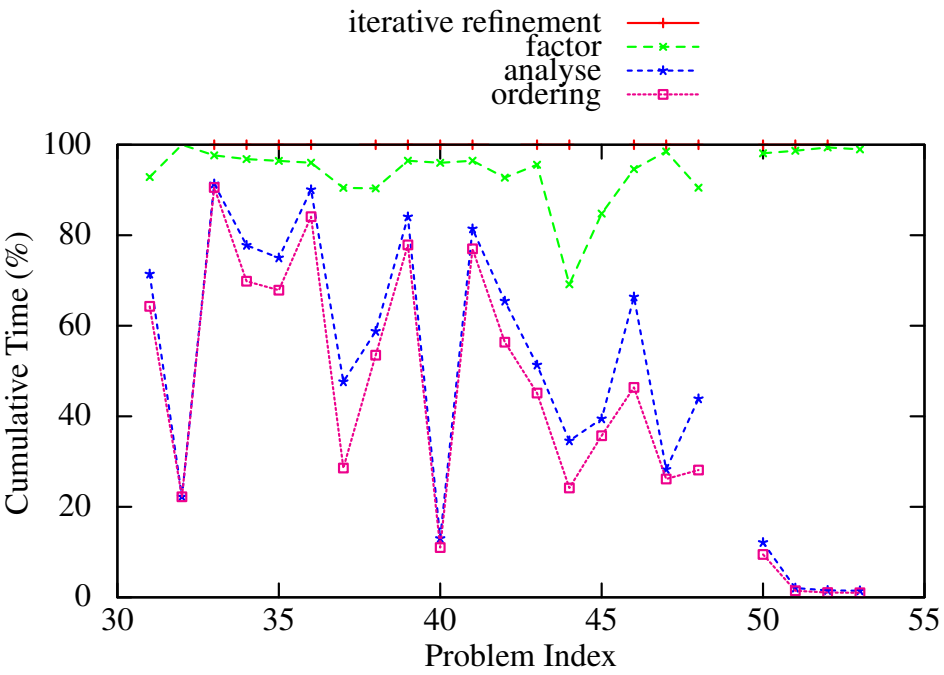


Figure 6. Cumulative proportions of the time for different phases of `HSL_MA97` (gaps between curves represent the proportion of the total time spent in that phase). Indefinite problems.



6. Concluding Remarks

This paper has provided a brief overview of modern sparse direct methods for the efficient solution of large symmetric linear systems of equations (both positive-definite and indefinite systems) and, in particular, has looked at the algorithms implemented within the shared memory parallel solvers that have been developed for the HSL mathematical software library. Supernodal and multifrontal solvers have been considered and their performance illustrated using problems from a range of practical applications. We conclude that the limiting factor on the use of modern sparse direct methods is the available memory for storing the matrix factors. Out-of-core techniques can significantly extend the range of problems that can be tackled and, in our tests, imposed a relatively modest time penalty. If bit-compatible answers are desired, these can be achieved and our results suggest that, for a multifrontal approach, the penalty is not prohibitive.

Finally, we note that a number of other (non-HSL) parallel sparse solvers have been developed in recent years. For general symmetric systems, the most well-known are PARDISO [44,45], MUMPS [46], PaStiX [47] and WSMP [48]. The algorithms implemented within these codes are essentially those that we have discussed in this paper (PARDISO and PaStiX are supernodal packages, while MUMPS and WSMP are based on multifrontal techniques), although each varies in the fine details (such as the choice of pivoting and scaling strategies). While it is beyond the scope of this paper to compare the HSL solvers with these packages, results presented in [5] demonstrate that the performance of the HSL solvers is competitive.

Code Availability

The HSL sparse direct solvers discussed in this paper are written in Fortran 95 and employ OpenMP for parallel implementation. They adhere to the Fortran 95 standard, except that they use allocatable structure components and dummy arguments (which are part Fortran 2003 and supported by all modern Fortran compilers).

Each of the solvers is available as part of the 2013 release of HSL. As such, they are fully documented, tested and maintained. All use of HSL requires a licence; licenses are available to academics without charge for individual research and teaching purposes. Details of how to obtain a licence and the solvers are available at http://www.hsl.rl.ac.uk/ or by email to hsl@stfc.ac.uk.

Acknowledgements

This work was supported by EPSRC grants EP/E053351/1 and EP/I013067/1.

We would like to thank our colleagues, Iain Duff and John Reid, of the Rutherford Appleton Laboratory for many helpful discussions on sparse solvers and software development and John, in particular, for his role as a co-author of `HSL_MA77` and `HSL_MA87` and for invaluable advice on Fortran. We gratefully thank Anshul Gupta of IBM for some of the large test matrices. Our thanks also to two anonymous reviewers for carefully reading our manuscript and making constructive suggestions for its improvement.

Conflicts of Interest

The authors declare no conflict of interest.

References

1. HSL. A collection of Fortran codes for large-scale scientific computation, 2013. Available online: http://www.hsl.rl.ac.uk/ (accessed on 28 October 2013).
2. Reid, J.; Scott, J. An out-of-core sparse Cholesky solver. *ACM Trans. Math. Softw.* **2009**, *36*, doi:10.1145/1499096.1499098.
3. Hogg, J.; Scott, J. *An Indefinite Sparse Direct Solver for Large Problems on Multicore Machines*; Technical Report RAL-TR-2010-011; Rutherford Appleton Laboratory: Didcot, UK, 2010.
4. Hogg, J.; Reid, J.; Scott, J. Design of a multicore sparse Cholesky factorization using DAGs. *SIAM J. Sci. Comput.* **2010**, *32*, 3627–3649.
5. Hogg, J.; Scott, J. *HSL_MA97: A Bit-Compatible Multifrontal Code for Sparse Symmetric Systems*; Technical Report RAL-TR-2011-024; Rutherford Appleton Laboratory: Didcot, UK, 2011.
6. Davis, T. *Direct Methods for Sparse Linear Systems*; SIAM: Philadelphia, PA, USA, 2006.
7. Duff, I.; Erisman, A.; Reid, J. *Direct Methods for Sparse Matrices*; Oxford University Press: Oxford, UK, 1989.
8. Gould, N.; Hu, Y.; Scott, J. A numerical evaluation of sparse direct symmetric solvers for the solution of large sparse, symmetric linear systems of equations. *ACM Trans. Math. Softw.* **2007**, *33*, doi:10.1145/1236463.1236465.

9. Golub, G.; van Loan, C. *Matrix Computations*, 3rd ed.; The Johns Hopkins University Press: Baltimore, MD, USA, 1996.
10. Tinney, W.; Walker, J. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proc. IEEE* **1967**, *55*, 1801–1809.
11. Amestoy, P.; Davis, T.; Duff, I. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl.* **1996**, *17*, 886–905.
12. Amestoy, P.; Davis, T.; Duff, I. Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw.* **2004**, *30*, 381–388.
13. Liu, J. Modification of the minimum-degree algorithm by multiple elimination. *ACM Trans. Math. Softw.* **1985**, *11*, 141–153.
14. George, A. Nested dissection of a regular finite-element mesh. *SIAM J. Numer. Anal.* **1973**, *10*, 345–363.
15. Gould, N.; Scott, J. A numerical evaluation of HSL packages for the direct solution of large sparse, symmetric linear systems of equations. *ACM Trans. Math. Softw.* **2004**, *30*, 300–325.
16. Karypis, G.; Kumar, V. METIS: A software package for partitioning unstructured graphs, partitioning meshes and computing fill-reducing orderings of sparse matrices—Version 4.0, 1998. Available online: http://www-users.cs.umn.edu/ karypis/metis/ (accessed on 28 October 2013).
17. Karypis, G.; Kumar, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.* **1999**, *20*, 359–392.
18. Pellegrini, F. *SCOTCH 5.1 User's Guide*; Technical Report; LaBRI: Universit/'e Bordeaux I, Frace, 2008. Available online: http://www.labri.fr/perso/pelegrin/scotch/ (accessed on 28 October 2013).
19. Boman, E.; Devine, K.; Fisk, L.A.; Heaphy, R.; Hendrickson, B.; Vaughan, C.; Catalyurek, U.; Bozdag, D.; Mitchell, W.; Teresco, J. *Zoltan 3.0: Parallel Partitioning, Load-balancing, and Data Management Services; User's Guide*; Technical Report SAND2007-4748W; SANDIA National Laboratory: Albuquerque, NM, USA, 2007. Available online: http://www.cs.sandia.gov/Zoltan/ug_html/ug.html (accessed on 28 October 2013).
20. Duff, I.; Scott, J. *Towards an Automatic Ordering for a Symmetric Sparse Direct Solver*; Technical Report RAL-TR-2006-001; Rutherford Appleton Laboratory: Didcot, UK, 2005.
21. Liu, J. The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Appl.* **1990**, *11*, 134–172.
22. Dongarra, J.; Croz, J.D.; Hammarling, S.; Duff, I.S. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.* **1990**, *16*, 1–17.
23. Ashcraft, C.; Grimes, R.G. The influence of relaxed supernode partitions on the multifrontal method. *ACM Trans. Math. Softw.* **1989**, *15*, 291–309.
24. Hogg, J.; Scott, J. *A Modern Analyse Phase for Sparse Tree-Based Direct Methods*; Technical Report RAL-TR-2010-031; Rutherford Appleton Laboratory: Didcot, UK, 2010.
25. Duff, I.; Reid, J. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Trans. Math. Softw.* **1983**, *9*, 302–325.
26. Liu, J. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Rev.* **1992**, *34*, 82–109.

27. Ashcraft, C.; Grimes, R.G.; Lewis, J.G. Accurate symmetric indefinite linear equation solvers. *SIAM J. Matrix Anal. Appl.* **1999**, *20*, 513–561.
28. Bunch, J.R.; Kaufman, L. Some stable methods for calculating inertia and solving symmetric linear systems. *Math. Comput.* **1977**, *31*, 163–179.
29. Duff, I.S.; Gould, N.I.M.; Reid, J.K.; Scott, J.A.; Turner, K. Factorization of sparse symmetric indefinite matrices. *IMA J. Numer. Anal.* **1991**, *11*, 181–204.
30. Hogg, J.; Scott, J. Pivoting strategies for tough sparse indefinite systems. *ACM Trans. Math. Softw.* **2014**, *40*, to be published.
31. Hogg, J.; Scott, J. *Compressed Threshold Pivoting for Sparse Symmetric Indefinite Systems*; Technical Report RAL-P-2013-007; Rutherford Appleton Laboratory: Didcot, UK, 2013.
32. Hogg, J.; Scott, J. *The Effects of Scalings on the Performance of a Sparse Symmetric Indefinite Solver*; Technical Report RAL-TR-2008-007; Rutherford Appleton Laboratory: Didcot, UK, 2008.
33. Hogg, J.; Scott, J. Optimal weighted matchings for rank-deficient sparse matrices. *SIAM J. Matrix Anal. Appl.* **2013**, *34*, 1431–1447.
34. Hogg, J.; Scott, J. *Achieving Bit Compatibility in Sparse Direct Solvers*; Technical Report RAL-P-2012-005; Rutherford Appleton Laboratory: Didcot, UK, 2012.
35. Buttari, A.; Dongarra, J.; Kurzak, J.; Langou, J.; Luszczek, P.; Tomov, S. The Impact of Multicore on Math Software. In Proceedings of Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA'06), Umeå, Sweden, 18–21 July 2006.
36. Buttari, A.; Langou, J.; Kurzak, J.; Dongarra, J. *A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures*; Technical Report UT-CS-07-600; Also LAPACK Working Note 191; ICL, University of Tennessee: Knoxville, TN, USA, 2007.
37. Anderson, E.; Bai, Z.; Bischof, C.; Blackford, S.; Demmel, J.; Dongarra, J.; Du Croz, J.; Greenbaum, A.; Hammarling, S.; McKenney, A.; *et al.* *LAPACK Users' Guide*, 3rd ed.; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 1999.
38. Reid, J.; Scott, J. Partial Factorization of a Dense Symmetric Indefinite Matrix. *ACM Trans. Math. Softw.* **2011**, *38*, doi:10.1145/2049673.2049674.
39. Reid, J.; Scott, J. Algorithm 891: A Fortran virtual memory system. *ACM Trans. Math. Softw.* **2009**, *36*, 1–12.
40. Davis, T.; Hu, Y. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.* **2011**, *38*, doi:10.1145/2049662.2049663.
41. Ruiz, D.; Uçar, B. *A Symmetry Preserving Algorithm of Matrix Scaling;* Technical Report RR-7552; INRIA: Paris, France, 2011.
42. Duff, I.S. MA57—a new code for the solution of sparse symmetric definite and indefinite systems. *ACM Trans. Math. Softw.* **2004**, *30*, 118–154.
43. Hogg, J.; Scott, J. A fast and robust mixed precision solver for the solution of sparse symmetric linear systems. *ACM Trans. Math. Softw.* **2010**, *37*, doi:10.1145/1731022.1731027.
44. Schenk, O.; Gärtner, K. Solving unsymmetric sparse systems of linear equations with PARDISO. *J. Future Generation Comput. Syst.* **2004**, *20*, 475–487.
45. Schenk, O.; Gärtner, K. On fast factorization pivoting methods for symmetric indefinite systems. *Electron. Trans. Numer. Anal.* **2006**, *23*, 158–179.

46. Amestoy, P.; Duff, I.; L'Excellent, J.Y.; Koster, J. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Anal. Appl.* **2001**, *23*, 15–41.
47. Hénon, P.; Ramet, P.; Roman, J. PaStiX: A high-performance parallel direct solver for sparse symmetric definite systems. *Parallel Comput.* **2002**, *28*, 301–321.
48. 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. Available online: http://www.cs.umn.edu/∼agupta/doc/wssmp-paper.ps (accessed on 28 October 2013).