Science & Technology Facilities Council

Technical Report

# An indefinite sparse direct solver for large problems on multicore machines

J. D. Hogg and J. A. Scott

April 21, 2010

RAL-TR-2010-011

Library and Information Services
SFTC Rutherford Appleton Laboratory
Harwell Science and Innovation Campus
Didcot
OX11 0QX
UK
Tel: +44 (0)1235 445384
Fax: +44(0)1235 446403
Email: library@rl.ac.uk



**ISSN 1358-6254**

# An indefinite sparse direct solver for large problems on multicore machines

J. D. Hogg and J. A. Scott[1]

The sparse direct solver HSL_MA87 uses a DAG-based algorithm to obtain fine-grain parallel execution of the factorization phase on multicore architectures. The first release of the package performed the Cholesky factorization of symmetric positive-definite systems. In this report, we discuss the changes we have made to HSL_MA87 to accommodate symmetric indefinite systems. The main changes stem from the inclusion of threshold partial pivoting for numerical stability. In particular, we use a new combined factorize_column task that replaces the separate tasks of the Cholesky factorization. Numerical results for a range of practical problems are given to illustrate the effectiveness of the DAG-based approach for solving indefinite linear systems on multicore machines. We find that speedups in excess of 6 can be achieved for some of the largest problems on our 8-core machine.

**Keywords:** sparse symmetric indefinite linear systems, direct solver, DAG-based, parallel, multicore, Fortran 95, OpenMP.

**AMS(MOS) subject classifications:** 65F05, 65F50, 65Y05

---

[1] Computational Science and Engineering Department, Rutherford Appleton Laboratory, Chilton, Oxfordshire, OX11 0QX, UK.
Emails: 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.

April 21, 2010

# 1 Introduction

Many problems require the efficient and accurate solution of linear systems

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

where $A$ is a large, sparse, symmetric matrix of order $n$. For the case where $A$ is positive definite, we recently developed a sparse Cholesky solver for multicore machines [10]. $A$ is factorized into the product $LL^T$, where the sparse factor $L$ is lower triangular; the solution process is completed by performing forward substitution ($Ly = b$) and then back substitution ($L^T x = y$). In this report, we discuss how we have extended the solver to indefinite systems. Such systems arise in numerous areas (including fluid flow problems, augmented systems arising in linear and nonlinear optimization, electromagnetic scattering and eigenvalue problems) but are frequently hard to solve since the direct solver needs to employ pivoting techniques to ensure numerical stability whilst also retaining sparsity. In the indefinite case, the aim is to compute a sparse factorization

$$A = (PL)D(PL)^T \tag{1.2}$$

where $P$ is a permutation matrix, $L$ is unit lower triangular, and $D$ is block diagonal with blocks of size $1 \times 1$ and $2 \times 2$. Our solver is called `HSL_MA87` and is available as part of the HSL mathematical software library [12]. The code is written in Fortran 95 with the widely available extension of allocatable components of structures, part of Fortran 2003. To provide a portable approach that allows the exploitation of shared caches, `HSL_MA87` uses OpenMP, and achieves speedups of over 6 on an 8 core machine for some of our largest test problems.

The outline of this report is as follows. In Section 2, we provide a brief overview of `HSL_MA87` in the positive-definite case; we explain the data structures it uses and its use of tasks and a directed acyclic graph. The modifications that are required for the indefinite case are discussed in Section 3. In particular, we consider pivoting and how this affects the tasks and show how we deal with delayed pivots. Numerical results are presented in Section 4. These include comparisons with sparse other direct solvers in serial and in parallel. Finally, in Section 5, we outline some of our future plans for `HSL_MA87` that aim to improve performance further for indefinite systems.

For clarity of notation, we assume throughout that $A$ has been preordered (using, for example, a nested dissection or minimum degree algorithm) so that the pivot order that is passed to the solver is the natural order $1, 2, \ldots, n$.

# 2 Brief overview of the sparse Cholesky case

In this section, we briefly review the approach used by `HSL_MA87` to factorize $A$ in the sparse positive-definite case. Motivated by the work of Buttari et al. [2, 3] 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 $L$. These tasks 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. The first vertex corresponds to the factorization of the first diagonal block of $A$ and the final vertex corresponds to the factorization of the final diagonal block of $A$. While the order of the tasks must obey the DAG, there remains much freedom for exploitation of parallelism.

In common with other sparse direct solvers, `HSL_MA87` has a number of separate phases.

**Analyse** takes the sparsity pattern of $A$ and prepares the data structures for the numerical factorization.

**Factorize** uses the data structures set up by the analyse phase to compute a sparse factorization.

**Solve** uses the computed factors generated by the factorize phase to solve one or more systems $Ax = b$.

We now explain the data structures set up by the analyse phase and describe how the factorize phase proceeds using a DAG.

## 2.1 Data structures

The analyse phase starts by computing the assembly tree using the sparsity pattern of $A$. A node of the assembly tree represents a set of contiguous columns of $L$ with the same (or nearly 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 nodes that are not ancestors. We compress this matrix in the traditional manner by holding only the nonzero rows, each with an index held in an integer. We refer to this dense trapezoidal matrix as the *nodal matrix* and store it using a row hybrid blocked structure, with "full" storage for the blocks on the diagonal (to allow efficient BLAS and LAPACK routines to be exploited). If the number of columns in the nodal matrix is large, we use the block size $nb$ (specified through a user-controlled parameter) and the blocks will be of size $nb \times nb$, except in the trailing rows and columns where they may be smaller. For example, if the block size was 3, a node with 5 columns and 8 rows would be stored as

| | | | | |
|---|---|---|---|---|
| 1 | | | | |
| 4 | 5 | | | |
| 7 | 8 | 9 | | |
| 10 | 11 | 12 | 25 | |
| 13 | 14 | 15 | 27 | 28 |
| 16 | 17 | 18 | 29 | 30 |
| 19 | 20 | 21 | 31 | 32 |
| 22 | 23 | 24 | 33 | 34 |

## 2.2 Tasks

We divide the sparse Cholesky factorization into the following tasks:

**factorize_block($L_{diag}$)** Computes the traditional dense Cholesky factor $L_{diag}$ of the triangular part of a block that is on the diagonal using the LAPACK subroutine `_potrf`. If the block is trapezoidal, this is followed by a triangular solve of its rectangular part

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

using the BLAS subroutine `_trsm`.

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

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

using the BLAS subroutine `_trsm`.

**update_internal** ($L_{dest}$, *scol*) Performs the update

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

where $L_r$ is a block of the block column *scol* and $L_c$ is a submatrix of this block column. This task is accomplished using the BLAS routines `_syrk` and `_gemm`.

**update_between** ($L_{dest}$, *snode*, *scol*) Performs the update

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

where $L_r$ and $L_c$ are submatrices of contiguous rows of the block column *scol* of the node *snode* that correspond to the rows and columns of $L_{dest}$, respectively. This task exploits the fact that row boundaries in the nodal data structure are artificial and can be ignored to perform a direct outer product into a buffer. This buffer is then expanded out into the destination node using a sparse mapping that is calculated on the fly.

The tasks are partially ordered; for example, the updating of a block of a nodal matrix from a block column of $L$ that is associated with one of the node'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, we will be executing some tasks while others will be ready for execution. We store the tasks that are ready in local stacks, one for each cache, and a global task pool. Initially, the task pool is given a factorize_block task from each leaf node of the assembly tree.

### 2.3 The factorize phase

Although the task dependencies can be represented by a DAG, in `HSL_MA87` we do not compute and store the whole DAG explicitly. During the analyse phase, we calculate a count for each block of $L$. If the block is on the diagonal, the count is the number of updates (update_internal or update_between) that will be applied to it; for all other blocks, the count is one more than the number of updates that will be applied to it. During the factorize phase, we decrement the block's count by one after the completion of each update for it. When the count for a block on the diagonal reaches zero, a factorize_block task for it is stored. When a factorize_block task completes, we decrement the count of all the blocks in its block column. This ensures that when the count for an off-diagonal block reaches zero, all its updates are complete and the factorize_block for its block column is also complete, which means that its solve_block task is ready and may be stored.

When a factorize_block or solve_block task completes, we decrement its count to flag this event with a negative value. A column lock is set and we store each update task that is now ready. Once this has been done, the lock is released. The column lock and the counts ensure that each update task is added exactly once. Note that the negative count value is needed for a trapezoidal block on the diagonal since its factorize_block task includes a triangular solve.

Further details of the positive-definite case, as well as numerical results, are given in [10].

## 3 Modifications for the indefinite case

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. In this section, we discuss the pivoting strategy and its implications within `HSL_MA87`.

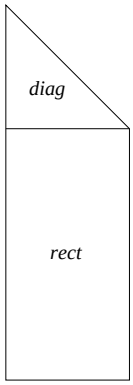
### 3.1 Combined factorize_column tasks with numerical pivoting

As we described in Section 2.2, in the positive-definite case, a block on the diagonal is first factorized using a dense Cholesky factorization routine and then triangular solves are performed on the rest of the block column. Consider the block column shown in Figure 3.1 In the indefinite case, large entries in *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 off-diagonal entries in a block column must be fully updated (so that the dependency counts for each of the off-diagonal blocks has been reduced to 1) before the block on the diagonal is factorized. We thus combine the factorize_block task and all the solve_block tasks for a block column into a single factorize_column task.

We want to factorize the diagonal block *diag* into the product

$$diag = (PL_{diag})D(PL_{diag})^T$$

Figure 3.1: Trapezoidal block column, consisting of a square diagonal block *diag* and a rectangular off-diagonal block *rect*.




where $P$ is a permutation matrix and $D$ is block diagonal with blocks of size $1 \times 1$ and $2 \times 2$. Pivots are chosen one or two at a time. Let $q$ denote the number of rows and columns of $D$ found so far (that is, the number of 1×1 pivots plus twice the number of 2×2 pivots). We use the notation $a_{ij}$, with $i > q$ and $j > q$, to denote an entry of the block column after it has been updated by all the permutations and pivot operations so far. Our stability test for a 1×1 pivot in column $m$ of the block column is the usual threshold test

$$|a_{mm}| > u \max_{i \neq m,\, i > q} |a_{im}|, \tag{3.1}$$

where the relative pivot tolerance $u$ is a user-set value in the range $0 \leq u \leq 1.0$. This is equivalent to the test

$$|a_{mm}|^{-1} \max_{i \neq m,\, i > q} |a_{im}| < u^{-1}. \tag{3.2}$$

In the case where $u$ is zero, this is interpreted as requiring that the pivot be nonzero. This was generalised in [6] to the test

$$\left| \begin{pmatrix} a_{mm} & a_{ml} \\ a_{ml} & a_{ll} \end{pmatrix}^{-1} \right| \begin{pmatrix} \max_{i \neq m,l; i > q} |a_{im}| \\ \max_{i \neq m,l; i > q} |a_{il}| \end{pmatrix} < \begin{pmatrix} u^{-1} \\ u^{-1} \end{pmatrix}, \tag{3.3}$$

for a 2×2 pivot in rows and columns $l$ and $m$, where the absolute value notation for a matrix refers to the matrix of corresponding absolute values and $u$ is the same user-set threshold value. In the case where $u$ is zero, this is interpreted as requiring that the pivot be nonsingular. We use these tests with the default value 0.01 for $u$.

If $a_{mm}$ is accepted as a 1×1 pivot, it becomes the next diagonal block of $D$ and row and column $m$ are permuted (if necessary) to the next pivotal position, $q$+1. The corresponding diagonal entry of $L_{diag}$ is 1 and inequality (3.2) tells us that the off-diagonal entries of this column of $L_{diag}$ are bounded in modulus by $u^{-1}$. If $\begin{pmatrix} a_{mm} & a_{ml} \\ a_{ml} & a_{ll} \end{pmatrix}$ is accepted as a 2×2 pivot, it becomes the next diagonal block of $D$ and rows and columns $l$ and $m$ are permuted (if necessary) to the next two pivotal positions, $q + 1$ and $q + 2$. The corresponding diagonal block of $L_{diag}$ is the identity matrix of order 2 and inequality (3.3) tells us that the off-diagonal entries of these columns of $L_{diag}$ are bounded in modulus by $u^{-1}$. Thus, all the diagonal entries of $L_{diag}$ are 1 and all the off-diagonal entries of $L_{diag}$ are bounded in modulus by $u^{-1}$.

If *diag* is of order $p$ and $q < p$ pivots can be found that satisfy the threshold tests (3.2), (3.3), $p - q$ pivots must be delayed until alternatives are available or they are safer to use. That is, the variables that have not been pivoted on must be passed up the assembly tree to the parent node and the columns of the block column corresponding to these variables must be added to those of the nodal matrix at the parent node. The delayed columns are retested at the parent node and, if necessary, passed further up the tree (at the root node, a full set of $p$ pivots can be chosen provided $u \leq 0.5$).

The implementation of the pivoting strategy within a factorize_column task is extracted from that used by Reid and Scott [15] in the dense indefinite partial factorization code HSL_MA64. The main modification stems from the fact that HSL_MA64 was primarily designed for use within the sparse multifrontal solver HSL_MA77 [13, 15] and hence requires a square matrix to be input, with pivots chosen from the first $p$ columns and the trailing submatrix updated (that is, the Schur complement formed) once these have been chosen. In our supernodal method, the trailing submatrix update is handled as a separate set of tasks. After excising this functionality, we were able to further simplify the code we had extracted from HSL_MA64 to avoid reordering to block column form internally, and we specialised it to deal only with a single block column. After some numerical testing, we decided to preserve the inner blocking functionality used within HSL_MA64 and controlled by the parameter nbi, as this gave a significant performance gain (although in our tests our experience was that there was little sensitivity to the precise value of nbi, provided it was at least 8).

When developing HSL_MA87 for the positive-definite case, we chose to use row-wise storage for $L$ as this facilitates updates between nodes by removing any discontinuities at row block boundaries. However, the factorize_column task of the indefinite case requires repeatedly scanning and updating individual columns, an access pattern better served by column-wise storage of $L$. We experimented with using both row-wise and column-wise storage. Our experience was that a hybrid variant that uses row-wise storage for the update tasks but converts to and from column-wise storage for the factorize_column task gives the best performance.

For the update tasks, we need both $L$ and $DL$ since there is no BLAS subroutine for the matrix multiplication $L_r D L_c^T$. We tested both storing $L_r D$ and calculating it from $L_r$ and $D$ on the fly. The first approach requires both $L$ and $DL$ to be read from main memory into cache, while the latter only reads $L$ at the cost of additional operations. The explicit $L$ and $DL$ approach performed best in serial, while the on-the-fly approach was better on our multicore test machine when memory bandwidth was more limited. HSL_MA87 uses the calculation approach because we are primarily interested in parallel performance.

Combining the factorize_block and solve_block tasks into a factorize_column task affords some simplifications in the part of the code that adds update tasks. Previously different solve_block tasks from within a column could be handled by different threads. As update_internal and update_between tasks are dependant on a *pair* of solve_block tasks, careful synchronisations and dependency checking was necessary to add each task exactly once. In the modified code, the entire block column is handled by a single thread, and all update operations from that column may be added as soon as the factorize_column task is complete. Unfortunately, the parallelism is now less fine-grained and the combined factorize_column tasks and update tasks are usually heavily imbalanced. To ensure sufficient tasks are available when running in parallel, we use a smaller block size than in the positive-definite case (see Table 4.3 in Section 4.1), but can still suffer from a long tail of (near) serial execution towards the end of the factorization phase. In our experiments, this had a particularly adverse effect on the performance of HSL_MA87 on some of the smaller problems in our test set (see Section 4).

To monitor when a factorize_column task is ready, a dependency count for each block column is computed at the start of the factorization phase. This is equal to the sum of the dependency counts of the blocks within the block column. Where in the positive-definite case a block dependency count is decremented, in the indefinite case the corresponding block column count is decremented; once it reaches zero, a factorize_column task is stored.

In the positive-definite case, the factorize phase starts by initialising the numeric representation of the factors to zero and then copying the entries of $A$ into the correct locations in $L$. In the indefinite case, each block column of $L$ is initialised to zero the first time a block within it is accessed and the appropriate entries of $A$ are added to it at the start of a factorize_column task. We found that this modification improved cache behaviour and resulted in a small reduction in the total factorize time.

## 3.2 Coping with delayed pivots

Storage for each of the block columns $L_{col}$ of $L$ is allocated at the start of the factorize phase. When a factorize_column task is ready to be performed, we must check whether there are any columns corresponding to delayed pivots to be accommodated. If there are $k > 0$ such columns and $L_{col}$ is the first block column in the node, we allocate a new index list and a new block column $L_{col}^{new}$ that is large enough to hold these $k$ columns and $L_{col}$. The first $k$ rows and columns of $L_{col}^{new}$ are initialised to zero before the columns corresponding to the delayed pivots from each of the child nodes are expanded into these columns. $L_{col}$ is then copied into the remaining columns and, finally, the storage used by $L_{col}$ is deallocated.

The factorize_column task proceeds by cyclically searching columns of $L_{col}^{new}$ (or $L_{col}$ if no delayed pivots were passed from the child nodes). Note that, since the columns corresponding to delayed pivots are likely to be rejected again, we permute them to the last $k$ columns of $L_{col}^{new}$ before we starting the search for pivots. If $L_{col}^{new}$ has $p$ columns but only $q < p$ pivots can be found that satisfy the stability criteria, the remaining $p-q$ delayed columns are permuted to be the last columns of $L_{col}^{new}$. When considering the next block column in the node, a new block column must be allocated to accommodate the delayed columns coming from the previous block column ($L_{col}^{new}$). For each block column, a flag is set to indicate whether $L_{col}^{new}$ or $L_{col}$ is to be used for the remainder of the factorize phase and by the subsequent solve phase.

## 3.3 Pivoting options

By default, `HSL_MA87` uses the threshold partial pivoting described in Section 3.1. But, as already noted, this can lead to pivots being delayed. We therefore include options for relaxed and static pivoting, that is, forcing pivots that do not satisfy the stability criteria to be chosen, perhaps after modification. The resulting factorization may be less accurate and a number of steps of iterative refinement (or other refinement process) may be required to restore full accuracy. The strategy offered by `HSL_MA87` follows that used within `HSL_MA77` (and `HSL_MA64`) (see also [7]). Relaxed pivoting may be requested by providing a lower bound *umin* for the threshold tolerance $u$. If no 1×1 or 2×2 candidate pivot satisfies the test (3.1) or (3.3) but the pivot that is nearest to satisfying the test would satisfy it with $u = v \geq umin$, the pivot is accepted and, on the thread that is implementing the factorize_column task, $u$ is reduced to $v$. On that thread, the new value of $u$ is employed thereafter. Note that this may mean that different values of the stability threshold are used on the different threads for part of the computation; at the end of the factorize phase, the smallest value used is returned to the user. Our default setting is $u = umin = 0.01$. At a root node, we do not allow the value of *umin* to exceed 0.5 in order to ensure that a complete set of pivots is chosen.

If static pivoting is requested and no 1×1 or 2×2 candidate pivot satisfies the test (3.1) or (3.3) even after relaxing the value of $u$, the 1×1 pivot that is nearest to satisfying the test is accepted. If its absolute value is less than the user-set threshold *static*, it is given the value that has the same sign but absolute value *static*. We remark that in some applications (including a number arising from optimization problems) the inertia of the matrix is required; using static pivoting may lead to the computed inertia being inaccurate.

Our philosophy has always been to produce robust solvers (sometimes at the expense of speed). Consequently, in common with other HSL sparse direct packages, the default settings within `HSL_MA87` switch off the relaxed and static pivoting options.

## 3.4 Factorization in the singular case

So far, we have assumed that $A$ is nonsingular, but consistent systems of linear equations with a singular matrix occur quite frequently in practice and we wish to accommodate them. We follow the same approach as is used in `HSL_MA77`. When a column is searched for a pivot, if its largest entry is found to be less than a user-set tolerance *small*, the row and column are set to zero, the diagonal entry is accepted as a zero 1×1 pivot, and no corresponding pivotal operations are applied to the rest of the matrix. This is equivalent

to perturbing the corresponding entries of $A$ by at most *small* to make our factorization be of a nearby nonsingular matrix. To allow for this case, we hold the inverse of $D$, and set the entry corresponding to the zero pivot to zero. This avoids the need for special action in BLAS calls later in the factorization and during the solve phase. It leads to a correct result with a reasonable norm when the given set of equations is consistent and avoids a solution having a large norm if the equations are not consistent.

# 4 Numerical experiments

The numerical experiments reported here were performed on our multicore test machine `fox`, details of which are given in Table 4.1. For timing, we used wall-clock times in seconds on a lightly loaded machine. As in [10], in each experiment, for each problem the reported times are averaged over ten runs (or three runs for the largest problems). All the problems used in our experiments are systems arising from practical

Table 4.1: Specifications of our test machine `fox`.

| | 2-way quad Harpertown (`fox`) |
|---|---|
| Architecture | Intel(R) Xeon(R) CPU E5420 |
| Clock | 2.50 GHz |
| Cores | $2 \times 4$ |
| Theoretical peak (1/8 cores) | 10 / 80 Gflop/s |
| `dgemm` peaks (1/8 cores) | 9.3 / 72.8 Gflop/s |
| Memory | 8 GB |
| Compiler | Intel 11.0 with option -fast |
| BLAS and LAPACK | Intel MKL 10.1 |

applications and are taken from the University of Florida Sparse Matrix Collection [4]. The subset chosen is that used in chapter 8 of [9] together with some larger problems. The problems in the tables of results are in increasing order of the time taken by `HSL_MA87` with its default settings. In each test, we check the scaled residual

$$\|b - Ax\|_\infty / (\|A\|_\infty \|x\|_\infty + \|b\|_\infty).$$

## 4.1 Parameter selection

`HSL_MA87` uses a number of control parameters, including a node amalgamation parameter *nemin* and blocking parameter *nb*. These may be tuned by the user to improve the performance for a particular problem or machine; alternatively, the default settings may be used. In this section, we report the results of our experiments that were used to choose the default values.

Node amalgamation is a well established means of improving factorization speed at the expense of the number of entries in $L$ and the operation counts during the factorize and solve phases. We use the version described in [14]. Based on our experiences in the positive-definite case [10], we tested *nemin* in the range 8 to 40; all other control parameters used by `HSL_MA87` were given their default settings. Factorization times on 1 and 8 cores are presented in Table 4.2. We see that there is no setting that gives the best results in all cases. Based on these timings, we have selected $nemin = 32$ as the default value (which is consistent with that used in the positive-definite case) but highly recommend that the user experiments with a range of values, particularly if several matrices with the same (or similar) sparsity pattern are to be factorized or a number of calls to the solve phase follows the factorization. In the latter case, a smaller value may be better (see [11]).

In Table 4.3, factorization times are given for a range of values of the blocking parameter *nb*. The inner block size *nbi* was taken to be the default value of 16 (again, chosen on the basis of experiments). We see that, on 8 cores, the best value is usually 128 or 192. Since the larger block size appears to be beneficial on the largest problem, we have chosen the default to be 192.

Table 4.2: Wall-clock times (in seconds) and speedups for the factorization phase of `HSL_MA87` on 1 and 8 cores with a range of values of the node amalgamation parameter *nemin*. Values within 5% of the best are given in bold. s denotes the speedup on 8 cores.

| Problem | *nemin* = 8 | | | *nemin* = 16 | | | *nemin* = 24 | | | *nemin* = 32 | | | *nemin* = 40 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 8 | s | 1 | 8 | s | 1 | 8 | s | 1 | 8 | s | 1 | 8 | s |
| Boeing/bcsstk38 | 0.083 | 0.166 | 0.50 | 0.081 | **0.110** | 0.74 | 0.084 | **0.114** | 0.74 | 0.084 | 0.144 | 0.60 | 0.090 | 0.171 | 0.53 |
| Schenk_IBMNA/c-56 | 0.174 | 0.171 | 1.02 | 0.158 | 0.178 | 0.880 | 0.155 | 0.160 | 0.97 | 0.163 | 0.152 | 1.07 | 0.185 | **0.099** | 1.87 |
| Simon/olafu | 0.232 | **0.156** | 1.48 | 0.232 | 0.226 | 1.02 | 0.232 | 0.165 | 1.41 | 0.244 | **0.152** | 1.60 | 0.248 | 0.185 | 1.34 |
| GHS_indef/stokes128 | 0.278 | 0.266 | 1.05 | 0.253 | 0.227 | 1.11 | 0.255 | 0.252 | 1.01 | 0.268 | 0.232 | 1.15 | 0.288 | **0.189** | 1.53 |
| Boeing/crystk03 | 1.25 | **0.356** | 3.52 | 1.25 | 0.402 | 3.12 | 1.26 | **0.347** | 3.62 | 1.29 | **0.359** | 3.58 | 1.33 | **0.351** | 3.79 |
| GHS_psdef/s3dkq4m2 | 1.86 | 0.495 | 3.75 | 1.86 | **0.427** | 4.35 | 1.90 | 0.506 | 3.75 | 1.93 | **0.440** | 4.40 | 1.98 | **0.437** | 4.53 |
| Koutsovasilis/F2 | 2.55 | 0.598 | 4.26 | 2.50 | **0.553** | 4.53 | 2.52 | **0.538** | 4.68 | 2.57 | 0.571 | 4.51 | 2.63 | **0.545** | 4.83 |
| Cunningham/qa8fk | 4.19 | **0.883** | 4.74 | 4.17 | **0.874** | 4.77 | 4.18 | **0.914** | 4.57 | 4.23 | **0.882** | 4.79 | 4.31 | 0.928 | 4.64 |
| Schenk_IBMNA/c-62 | 9.30 | 5.17 | 1.80 | 9.26 | 5.12 | 1.81 | 8.93 | **4.85** | 1.84 | 9.07 | **4.93** | 1.84 | 9.10 | **4.94** | 1.84 |
| Oberwolfach/t3dh | 12.1 | **2.26** | 5.33 | 12.0 | **2.17** | 5.54 | 12.0 | **2.24** | 5.39 | 12.1 | **2.17** | 5.58 | 12.2 | 2.29 | 5.34 |
| ND/nd6k | 24.0 | 4.83 | 4.98 | 21.9 | 4.23 | 5.18 | 21.2 | 4.08 | 5.19 | 20.6 | **3.94** | 5.23 | 20.2 | **3.87** | 5.24 |
| GHS_indef/aug3d | 58.7 | 42.8 | 1.37 | 67.4 | 53.5 | 1.26 | 47.0 | 34.7 | 1.35 | 36.5 | **25.9** | 1.41 | 37.8 | 28.1 | 1.35 |
| Schenk_AFE/af_shell10 | 72.4 | **11.9** | 6.06 | 72.1 | **11.7** | 6.16 | 72.2 | **11.7** | 6.16 | 72.8 | **11.7** | 6.22 | 73.3 | **11.8** | 6.19 |
| ND/nd12k | 99.6 | 18.2 | 5.46 | 93.5 | 16.6 | 5.64 | 90.8 | 15.9 | 5.71 | 88.5 | **15.2** | 5.83 | 87.1 | **14.8** | 5.87 |
| GHS_indef/sparsine | 359 | 71.7 | 5.01 | 291 | 54.5 | 5.34 | 265 | 48.0 | 5.52 | 250 | **44.3** | 5.65 | 242 | **42.3** | 5.73 |
| PARSEC/GaAsH6 | 314 | 58.6 | 5.36 | 284 | 50.7 | 5.61 | 270 | 46.9 | 5.77 | 264 | **45.3** | 5.83 | 258 | **43.8** | 5.89 |

Table 4.3: Wall-clock times (in seconds) and speedups for the factorization phase of `HSL_MA87` on 1 and 8 cores with a range of values of the blocking parameter $nb$. Values within 5% of the best are given in bold. s denotes the speedup on 8 cores.

| Problem | $nb = 64$ | | | $nb = 128$ | | | $nb = 192$ | | | $nb = 256$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 8 | s | 1 | 8 | s | 1 | 8 | s | 1 | 8 | s |
| Boeing/bcsstk38 | 0.091 | 0.181 | 0.50 | 0.085 | **0.120** | 0.71 | 0.087 | 0.144 | 0.60 | 0.087 | 0.172 | 0.50 |
| Schenk_IBMNA/c-56 | 0.178 | 0.171 | 1.04 | 0.165 | 0.163 | 1.01 | 0.163 | 0.152 | 1.07 | 0.164 | **0.114** | 1.43 |
| Simon/olafu | 0.267 | 0.170 | 1.57 | 0.243 | **0.155** | 1.57 | 0.244 | **0.152** | 1.60 | 0.247 | 0.220 | 1.12 |
| GHS_indef/stokes128 | 0.283 | **0.194** | 1.45 | 0.268 | 0.230 | 1.16 | 0.268 | 0.232 | 1.15 | 0.269 | 0.234 | 1.15 |
| Boeing/crystk03 | 1.47 | 0.423 | 3.49 | 1.31 | **0.348** | 3.77 | 1.29 | **0.359** | 3.58 | 1.32 | 0.405 | 3.25 |
| GHS_psdef/s3dkq4m2 | 2.23 | 0.501 | 4.44 | 1.97 | 0.481 | 4.09 | 1.93 | **0.440** | 4.40 | 1.95 | 0.516 | 3.78 |
| Koutsovasilis/F2 | 2.97 | 0.616 | 4.82 | 2.61 | **0.561** | 4.66 | 2.57 | **0.571** | 4.51 | 2.61 | 0.610 | 4.28 |
| Cunningham/qa8fk | 5.07 | 1.09 | 4.66 | 4.35 | 0.935 | 4.65 | 4.23 | **0.882** | 4.79 | 4.33 | 1.01 | 4.27 |
| Schenk_IBMNA/c-62 | 11.9 | 6.58 | 1.81 | 9.63 | 5.32 | 1.81 | 9.07 | 4.93 | 1.84 | 8.57 | **4.54** | 1.89 |
| Oberwolfach/t3dh | 14.9 | 2.72 | 5.47 | 12.5 | **2.25** | 5.54 | 12.1 | **2.17** | 5.58 | 12.5 | 2.71 | 4.62 |
| ND/nd6k | 26.0 | 4.66 | 5.58 | 21.5 | **3.87** | 5.55 | 20.6 | **3.94** | 5.23 | 20.9 | 4.39 | 4.76 |
| GHS_indef/aug3d | 37.1 | 26.5 | 1.40 | 35.2 | **24.6** | 1.43 | 36.5 | 25.9 | 1.41 | 37.9 | 27.1 | 1.40 |
| Schenk_AFE/af_shell10 | 87.5 | 14.6 | 5.98 | 75.0 | **12.0** | 6.23 | 72.8 | **11.7** | 6.22 | 76.9 | 13.6 | 5.67 |
| ND/nd12k | 118 | 19.7 | 6.00 | 95.3 | 16.2 | 5.88 | 88.5 | **15.2** | 5.83 | 90.2 | 17.6 | 5.11 |
| GHS_indef/sparsine | 340 | 57.6 | 5.91 | 277 | 50.17 | 5.52 | 250 | **44.3** | 5.65 | 254 | 52.0 | 4.90 |
| PARSEC/GaAsH6 | 368 | 61.0 | 6.03 | 292 | 51.5 | 5.67 | 264 | **45.3** | 5.83 | 268 | 54.4 | 4.93 |

## 4.2 Serial results

HSL_MA87 has been developed for multicore machines. Nevertheless, we are interested in its performance in serial. In Table 4.4, we compare its performance on a single core with that of the HSL direct solvers MA57 [5] and HSL_MA77 [13, 14]. Both implement multifrontal algorithms; two of the key differences between them are that HSL_MA77 is primarily designed for very large problems, allowing the matrix factors and the main work arrays to be held in files on disk, and it uses the dense symmetric factorization kernel HSL_MA64. Note that although HSL_MA77 is designed as an out-of-core solver, here we use the option of working in-core. Each code is supplied with the same ordering (computed using the analyse phase of MA57) and each problem is scaled using MC77 [16] (and the internal scaling offered by MA57 is switched off). Otherwise, default settings for the control parameters are used (in particular, all three codes use the threshold pivoting parameter $u = 0.01$), with all the problems treated as indefinite (even if known to be positive definite).

Table 4.4: Wall-clock times (in seconds) for the factorization phase of the HSL codes MA57, HSL_MA77 and HSL_MA87 on a single core. OOM indicates out of memory. * indicates a scaled residual of $10^{-14}$ or greater. The fastest times (and those within 5% of the fastest) are in bold.

| Problem | MA57 | HSL_MA77 | HSL_MA87 |
|---|---|---|---|
| Boeing/bcsstk38 | 0.152 | **0.076** | 0.087 |
| Schenk_IBMNA/c-56 | 0.404 | **0.130** | 0.163 |
| Simon/olafu | 0.559 | **0.234** | **0.244** |
| GHS_indef/stokes128 | 0.713* | **0.250*** | 0.268* |
| Boeing/crystk03 | 1.68 | **1.24** | **1.29** |
| GHS_psdef/s3dkq4m2 | 2.94 | **1.80** | 1.94 |
| Koutsovasilis/F2 | 4.48 | **2.42** | 2.57 |
| Cunningham/qa8fk | 7.00 | **4.13** | **4.23** |
| Schenk_IBMNA/c-62 | 19.7 | **7.26*** | 9.07 |
| Oberwolfach/t3dh | 20.2 | **11.7** | **12.1** |
| ND/nd6k | 40.5 | 24.5 | **20.6** |
| GHS_indef/aug3d | OOM | **38.2** | **36.5** |
| Schenk_AFE/af_shell10 | 100 | **76.2** | **72.8** |
| ND/nd12k | 164 | 105 | **88.5** |
| GHS_indef/sparsine | 537* | 376* | **250*** |
| PARSEC/GaAsH6 | 483* | 326* | **264*** |
| Oberwolfach/bone010 | 877 | 637 | **590** |
| PARSEC/Ga41As41H72 | OOM | 9241* | **7290*** |

We see that the factorization phase of HSL_MA77 and HSL_MA87 consistently outperform that of MA57. This is because, as already noted, both use the efficient dense factorization kernel HSL_MA64 (modified for HSL_MA87). In serial, for many of the smaller problems HSL_MA77 is faster than HSL_MA87 and for some problems (including Schenk_IBMNA/c-62) it is significantly faster. However, HSL_MA87 has the fastest factorization times on the largest problems. This behaviour is due substantially to the fact that HSL_MA77 is a multifrontal code that has been tuned for serial factorization while HSL_MA87 is a supernodal code tuned for parallel factorization. On the smaller problems, the additional overheads incurred by HSL_MA87 (including task handling and on-the-fly calculation of $DL$) result in it being slightly slower than HSL_MA77. On the larger problems, the HSL_MA87 overheads account for a much smaller proportion of the total factorization time, while HSL_MA77 has been designed for out-of-core working and, when run in-core, performs additional copying to keep its working set contiguous. It is thus unsurprising that it is overtaken by HSL_MA87's ability to exploit the random access nature of the computer memory. An additional factor in HSL_MA87's favour may be a better ability to reuse data in level 2 cache.

## 4.3 Results on a multicore machine

In Table 4.5, we report on the performance of HSL_MA87 on 1, 2, 4 and 8 cores of our multicore machine fox. For each problem, we report num_delay, which is the number of eliminations that were delayed, that is, the total number of variables that were passed from a child node to its parent node because of stability considerations. If a variable is passed further up the assembly tree, it will be counted again. We see that, provided num_delay is small, we achieve speedups that improve with the problem size. In particular, for some of the largest problems (including PARSEC/GaAsH6, Oberwolfach/bone010 and PARSEC/Ga41As41H72) we achieve speedups on 8 cores in excess of 6. However, if there are a lot of delayed pivots, the much larger factorize_column tasks that result seriously reduce the level of parallelism.

Table 4.5 also includes results for the well-known direct solver PARDISO 4.0 [17, 18] on 1 and 8 cores. Two different settings are used for PARDISO. In the first (labelled PARDISO no matching), PARDISO is supplied with the same pivot order as HSL_MA87, with default values used for its remaining control parameters and iterative refinement disabled. In the second (labelled PARDISO with matching), PARDISO is permitted to do its own ordering and, additionally, to perform its own scaling, perform an "advanced matching", and then do iterative refinement — the recommended settings for tackling numerically challenging problems. PARDISO is a non-multifrontal supernodal linear solver designed for shared-memory multiprocessors. It optionally uses preprocessing based on symmetric weighted matchings [19] followed by Bunch-Kaufmann pivoting [1] on the dense diagonal blocks that correspond to supernodes and, if a zero (or nearly zero) pivot occurs, it is perturbed so that pivots are not delayed beyond the current block. The preprocessing is designed to try and ensure good pivots are available within the supernodal block but numerical stability is not guaranteed. However, because there is no searching outside the diagonal blocks or dynamic reordering during the factorization, the PARDISO static pivoting strategy can have a substantial performance advantage over potentially more robust approaches such as that used by the HSL solvers. Results that illustrate this in serial are given in [8] and are confirmed here, with PARDISO producing much faster factorization times compared to HSL_MA87 for problems GHS_indef/stokes128, Schenk_IBMNA/c-62 and GHS_indef/aug3d for which HSL_MA87 experiences a large number of delayed pivots.

In our tests, for most of the small and mid-sized problems, HSL_MA87 is slower than PARDISO. However, for the larger problems with no (or only a few) delayed pivots (including the PARSEC problems and Oberwolfach/bone), HSL_MA87 is significantly faster and achieves better speedups. This is consistent with our earlier findings in the positive-definite case [10]. Our experience for some of the toughest of our test problems is that PARDISO is more likely to produce a large scaled residual than HSL_MA87 (although this is improved by using the matching option). The problems marked with a * in Table 4.5 have a scaled residual of $10^{-14}$ or greater, and we were unable to recover full precision using iterative refinement.

In Table 4.6, we report the number of Gigaflops (Gflops) required by the factorization phase of HSL_MA87 together with the Gflop rates on 8 cores. For some of the large problems with few delayed pivots, we achieve speeds in the range 40 to 60 per cent of the peak performance of gemm. These results again highlight that HSL_MA87 is suited to solving large problems.

Table 4.5: Wall-clock times (in seconds) for the factorization phase of HSL_MA87 on 1, 2, 4 and 8 cores (with threshold parameter $u = 0.01$) and PARDISO on 1 and 8 cores with and without matchings enabled. num_delay is the number of delayed pivots and the figures in parentheses are speedups. * denotes a scaled residual of $10^{-14}$ or greater. The fastest times (and those within 5% of the fastest) on 8 cores are in bold.

| | HSL_MA87 | | | | | | | | PARDISO no matching | | | PARDISO with matching | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Problem | num_delay | 1 | 2 | | 4 | | 8 | | 1 | 8 | | 1 | 8 | |
| Boeing/bcsstk38 | 38 | 0.087 | 0.056 | (1.55) | 0.125 | (0.70) | 0.144 | (0.60) | 0.089 | 0.110 | (0.81) | 0.088 | **0.065** | (1.34) |
| Schenk_IBMNA/c-56 | 235 | 0.163 | 0.116 | (1.41) | 0.150 | (1.09) | 0.152 | (1.07) | 0.072 | 0.083 | (0.87) | 0.194 | **0.055** | (3.54) |
| Simon/olafu | 3 | 0.244 | 0.162 | (1.51) | 0.180 | (1.35) | 0.152 | (1.60) | 0.287 | 0.110 | (2.62) | 0.307 | **0.099** | (3.10) |
| GHS_indef/stokes128 | 2138 | 0.268 | 0.163 | (1.64) | 0.196 | (1.37) | 0.232* | (1.15) | 0.225 | **0.109** | (2.06) | 0.535 | 0.125* | (4.28) |
| Boeing/crystk03 | 0 | 1.29 | 0.757 | (1.70) | 0.501 | (2.57) | 0.359 | (3.58) | 1.41 | **0.248*** | (5.68) | 1.43 | 0.269* | (5.31) |
| GHS_psdef/s3dkq4m2 | 0 | 1.94 | 1.10 | (1.76) | 0.666 | (2.90) | 0.440 | (4.40) | 2.12 | **0.377** | (5.63) | 2.18 | **0.389** | (5.60) |
| Koutsovasilis/F2 | 0 | 2.57 | 1.45 | (1.77) | 0.873 | (2.95) | 0.571 | (4.51) | 2.88 | **0.533*** | (5.40) | 3.20 | 0.583* | (5.49) |
| Cunningham/qa8fk | 0 | 4.23 | 2.40 | (1.76) | 1.40 | (3.02) | 0.882 | (4.79) | 4.68 | 0.848* | (5.52) | 4.00 | **0.704*** | (5.68) |
| Schenk_IBMNA/c-62 | 28728 | 9.07 | 7.26 | (1.25) | 5.74 | (1.58) | 4.93 | (1.84) | 1.77 | **0.414*** | (4.27) | 2.78 | 0.581 | (4.79) |
| Oberwolfach/t3dh | 0 | 12.1 | 6.80 | (1.78) | 3.85 | (3.14) | **2.17** | (5.58) | 13.5 | 2.54 | (5.29) | 12.6 | 2.43 | (5.19) |
| ND/nd6k | 0 | 20.6 | 11.6 | (1.78) | 6.81 | (3.03) | **3.94** | (5.23) | 25.5 | 5.61 | (4.54) | 26.7 | 6.00 | (4.46) |
| GHS_indef/aug3d | 144955 | 36.5 | 31.0 | (1.18) | 28.2 | (1.29) | 25.9 | (1.41) | 0.105 | **0.074** | (1.41) | 0.367 | 0.092 | (4.00) |
| ND/nd12k | 0 | 88.5 | 48.7 | (1.82) | 27.8 | (3.18) | **15.2** | (5.83) | 110 | 28.2 | (3.89) | 106 | 26.5 | (4.02) |
| GHS_indef/sparsine | 16 | 250 | 138 | (1.81) | 80.7 | (3.10) | **44.4*** | (5.65) | 500 | 174* | (2.87) | 494 | 159 | (3.11) |
| PARSEC/GaAsH6 | 0 | 264 | 145 | (1.82) | 83.0 | (3.18) | **45.3*** | (5.83) | 375.8 | 111* | (3.39) | 328 | 89.7 | (3.65) |
| Schenk_AFE/af_shell10 | 0 | 72.8 | 39.8 | (1.83) | 21.6 | (3.38) | **11.7** | (6.22) | 78.4 | 13.9 | (5.65) | 78.5 | 13.2 | (5.97) |
| Oberwolfach/bone010 | 0 | 590 | 317 | (1.86) | 170 | (3.46) | **88.3** | (6.68) | 668 | 148 | (4.52) | 646 | 174.1 | (3.71) |
| PARSEC/Ga41As41H72 | 0 | 7290 | 3918 | (1.86) | 2176 | (3.35) | **1141*** | (6.39) | 11493 | 3658* | (3.14) | 10160 | 3020 | (3.36) |

Table 4.6: The Gflops required by the factorization and the Gflop/s rates achieved by HSL_MA87 on 8 cores. $n$ denotes the order of $A$.

| Problem | $n$ | Gflops | Gflop/s |
|---|---|---|---|
| Boeing/bcsstk38 | 8032 | 0.24 | 1.7 |
| Schenk_IBMNA/c-56 | 35910 | 0.21 | 1.4 |
| Simon/olafu | 16146 | 0.91 | 6.0 |
| GHS_indef/stokes128 | 49666 | 0.57 | 2.5 |
| Boeing/crystk03 | 24696 | 6.19 | 17.3 |
| GHS_psdef/s3dkq4m2 | 90449 | 8.20 | 18.6 |
| Koutsovasilis/F2 | 71505 | 11.2 | 19.7 |
| Cunningham/qa8fk | 66127 | 22.2 | 25.1 |
| Schenk_IBMNA/c-62 | 41731 | 25.0 | 5.1 |
| Oberwolfach/t3dh | 79171 | 70.1 | 32.3 |
| GHS_indef/aug3d | 24300 | 109 | 4.2 |
| ND/nd6k | 18000 | 113 | 28.8 |
| Schenk_AFE/af_shell10 | 1508065 | 405 | 34.5 |
| ND/nd12k | 36000 | 514 | 33.9 |
| GHS_indef/sparsine | 50000 | 1391 | 31.4 |
| PARSEC/GaAsH6 | 61349 | 1574 | 34.8 |
| Oberwolfach/bone010 | 986703 | 3905 | 44.2 |
| PARSEC/Ga41As41H72 | 268096 | 47387 | 41.5 |

## 4.4 Performance on positive-definite problems

One way of assessing how well the indefinite version of HSL_MA87 is performing is to run it on positive-definite problems and compare the performance when run with the input parameter *matrix_type* set to 1 (positive definite) and set to 2 (indefinite). Results are given in Table 4.7 for a subset of our test problems that are positive definite when run on 1 and 8 cores. In each case, we use the default settings. Thus

Table 4.7: The factorize phase times with *matrix_type* = 1 and 2 on 1 and 8 cores.

| | *matrix_type* = 1 | | *matrix_type* = 2 | |
|---|---|---|---|---|
| Problem | 1 | 8 | 1 | 8 |
| Boeing/bcsstk38 | 0.069 | 0.168 | 0.087 | 0.144 |
| Simon/olafu | 0.202 | 0.174 | 0.244 | 0.152 |
| GHS_psdef/s3dkq4m2 | 1.61 | 0.446 | 1.94 | 0.440 |
| ND/nd6k | 18.3 | 3.02 | 20.6 | 3.94 |
| ND/nd12k | 80.0 | 12.3 | 88.5 | 15.2 |

for the indefinite runs, $nbi = 16$ and $nb = 192$, while for the positive-definite case $nb = 256$ (there is no inner block size $nbi$ in this case). With these settings, on each of the test examples, the positive-definite time on a single core is, as we would expect, less than the indefinite time; the indefinite overhead is generally 25 per cent or less. On 8 cores, on small problems (including Boeing/bcsstk38) we found that the indefinite factorization can outperform the positive-definite one. This is because the smaller block size $nb$ was a better choice for these problems. For the larger problems on 8 cores the indefinite overhead is more significant (for example, it exceeds 40 per cent for ND/nd12k).

# 5 Concluding remarks and future directions

In this report, we have discussed how we have extended the code HSL_MA87 to solve large sparse symmetric indefinite linear systems on multicore machines. The main change from the positive-definite case involved accommodating threshold pivoting and this required the factorize_block and solve_block tasks to be combined into a single factorize_column task. This was implemented within HSL_MA87 using a modified version of the dense indefinite solver HSL_MA64, allowing the use of Level 3 BLAS. However, if many pivots are delayed because they fail to satisfy the threshold criteria, the much larger factorize_column tasks that result seriously reduce the level of parallelism. Thus, although HSL_MA87 is performing well and on our 8-core test machine achieves impressive speedups on some of our largest test problems, we would like to improve its performance further.

We have a number of ideas for redesigning the algorithms used and their implementation for indefinite systems. We will investigate new block-orientated pivoting techniques with an option to backtrack and utilise more stringent pivoting if unacceptably large growth is encountered. We will also explore new ways of limiting the number of delayed pivots without compromising stability or inertia. These will include novel initial ordering strategies that take better account of zero diagonal entries in the system matrix and partially rescaling the matrix during the factorization.

# 6 Acknowledgements

We are very grateful to our colleague John Reid for many helpful discussions. Our thanks also to John and to Iain Duff for commenting on a draft of this report.

# References

[1] J. Bunch and L. Kaufmann, *Some stable methods for calculating inertia and solving symmetric linear systmes*, Math. Comp., 31 (1977), pp. 163–179.

[2] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov, *The impact of multicore on math software*, in Proceedings of Workshop on State-of-the-art in Scientific and Parallel Computing (PARA06), 2006.

[3] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, *A class of parallel tiled linear algebra algorithms for multicore architectures*, Technical Report UT-CS-07-600, ICL, 2007. Also LAPACK Working Note 191.

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

[6] I. S. Duff, N. I. M. Gould, J. K. Reid, J. A. Scott, and K. Turner, *Factorization of sparse symmetric indefinite matrices*, IMA J. Numer. Anal., 11 (1991), pp. 181–2044.

[7] I. S. Duff and S. Pralet, *Strategies for scaling and pivoting for sparse symmetric indefinite problems*, SIAM J. Matrix Anal. Appl., 27 (2005), pp. 313–340.

[8] N. I. M. Gould, J. A. Scott, and Y. Hu, *A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations*, ACM Trans. Math. Softw., 33 (2007).

[9] J. D. Hogg, *High Performance Cholesky and Symmetric Indefinite Factorizations with Applications*, PhD thesis, University of Edinburgh, 2010.

[10] 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.

[11] J. D. Hogg and J. A. Scott, *A note on the solve phase of a multicore solver*, Technical Report RAL-TR-2010-007, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England, 2010.

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

[13] 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.

[14] ——, *An out-of-core sparse Cholesky solver*, ACM Trans. Math. Softw., 36 (2009). Article 9, 33 pages.

[15] ——, *Partial factorization of a dense symmetric indefinite matrix*, Technical Report RAL-TR-2009-015, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England, 2009.

[16] D. Ruiz, *A scaling algorithm to equilibrate both rows and columns norms in matrices*, Technical Report RAL-TR-2001-034, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England, 2001.

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

[18] ——, *On fast factorization pivoting methods for symmetric indefinite systems*, Elec. Trans. Numer. Anal, 23 (2006), pp. 158–179.

[19] O. Schenk, A. Waechter, and M. Hagemann, *Matching-based preprocessing algorithms to the solution of saddle-point problems in large-scale nonconvex interior-point optimization*, J. Comp. Opt. Applic., 36 (2007), pp. 321–341.