Science & Technology Facilities Council

Technical Report

A DAG-based Sparse Cholesky Solver for Multicore Architectures

J. D. Hogg, J. K. Reid and J. A. Scott

April 27, 2009

RAL-TR-2009-004

© **Science and Technology Facilities Council**

Enquires about copyright, reproduction and requests for additional copies of this report should be addressed to:

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

The STFC ePublication archive (epubs), recording the scientific output of the Chilbolton, Daresbury, and Rutherford Appleton Laboratories is available online at: http://epubs.cclrc.ac.uk/

ISSN 1358-6254

Neither the Council nor the Laboratory accept any responsibility for loss or damage arising from the use of information contained in any of their reports or in any communication about their tests or investigation

RAL-TR-2009-004

A DAG-based Sparse Cholesky Solver for Multicore Architectures

J. D. Hogg[1] [2] , J. K. Reid[1] and J. A. Scott[1]

ABSTRACT

In this paper, we describe the design and development of a new code for the solution of sparse symmetric positive-definite linear systems aimed primarily at multicore architectures. Our new Fortran 95/OpenMP code, `HSL_MA87`, is available as part of the software library HSL and extends to the sparse case the task DAG-based approach that is becoming popular in the design and development of dense linear algebra kernels.

Comparisons are made with existing parallel solvers, using problems arising from a range of practical applications. We demonstrate that `HSL_MA87` obtains good serial and parallel times on an 8-core machine.

Keywords: Cholesky factorization, sparse symmetric linear systems, 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.
Email: john.reid@stfc.ac.uk and jennifer.scott@stfc.ac.uk
Work supported by EPSRC grants EP/F006535/1 and EP/E053351/1.
Current reports available from `http://www.numerical.rl.ac.uk/reports/reports.html`.

[2] School of Mathematics, University of Edinburgh, JCMB, King's Buildings, Edinburgh, EH9 3JZ, UK.
Email: J.Hogg@ed.ac.uk
Work partially supported by EPSRC grant EP/F006535/1.

April 27, 2009

1 Introduction

Many problems require the efficient and accurate solution of linear systems

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

where A is a large, sparse, real and symmetric matrix of order n. In recent years a number of direct solvers have been developed for this problem, including the serial codes `MA57` [8] and `HSL_MA77` [19] from the HSL software library [15] and CHOLMOD [6] as well as the parallel codes MUMPS [2], PARDISO [20], PaStiX [12] and WSMP [11]. A detailed comparison of serial codes is provided by Gould, Hu, and Scott [10]. The emergence of multicore machines has led to the need to design new solvers that are able to effectively exploit these architectures. In this paper, we describe how we have extended the directed acyclic graph (DAG) approach used by recent dense linear solvers [4, 5, 13] to the solution of sparse positive-definite linear systems on multicore architectures.

Multicore machines are shared memory systems featuring a complex hierarchy of shared caches, and are normally exploited through one of the following APIs:

OpenMP: Initially developed to allow easy exploitation of loop based parallelism, OpenMP works through pragmas in the program source code, enabling serial and parallel versions of a code to coexist. Modern versions allow non-loop based parallelism to be implemented in a standard cross-platform way in either C or Fortran.

MPI: The message passing framework is designed for distributed memory programming but it can also be implemented on shared memory systems. MPI is supported from both C and Fortran.

Pthreads: POSIX Threads are available on any reasonably POSIX compliant operating system (such as Linux), and allow a low-level parallel implementation that can be used directly from C.

HSL is a Fortran library and so our new sparse Cholesky solver `HSL_MA87` 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.

The outline of this paper is as follows. In Section 2, we provide a short description of the recent developments for dense linear systems that we will adapt to the sparse case. Section 3 presents a brief outline of the design of a traditional sparse direct solver and explains the changes required for our sparse DAG approach. Section 4 describes the implementation of our DAG-based Cholesky solver `HSL_MA87` and its user interface. Results of experiments with `HSL_MA87` on an 8-core machine and comparisons with a number of other modern direct solvers on selected problems from practical applications are given in Sections 5 and 6. Finally, in Sections 7 and 8, we comment on future work and code availability.

2 Dense DAG-based approach

Recent research by Buttari et al. [4, 5] and Hogg [13] into efficiently solving dense linear systems of equations on multicore architectures has shown that directed acyclic graphs (DAGs) can be used to obtain significant parallel speedups. Hogg reports near perfect speedups for sufficiently large problems. A blocked Cholesky factorization of a dense matrix A divides A into square blocks A_{ij} of order nb and then divides the work into a number of tasks:

- Factorize a block on the diagonal, $A_{kk} = L_{kk}L_{kk}^T$, where L_{kk} is lower triangular.
- Perform a triangular solve $L_{ik} = A_{ik}L_{kk}^{-T}$ to obtain an off-diagonal block L_{ik} of the Cholesky factor.
- Update a block of the remaining submatrix $A_{ij} \Leftarrow A_{ij} - L_{ik}L_{jk}^T$, $i \geq j$.

A more comprehensive definition of these tasks is given in Section 4.2 (see also [13]). While a partial ordering of these tasks must be followed, there exists much freedom in their scheduling — particularly of the update tasks, and this has led to a number of variants (for example, left- and right-looking algorithms).

In a DAG-based approach, the dependencies between the tasks are represented using a graph. Each task is represented by a node, with dependencies represented by directed edges. Tasks must be executed in conformance with this task DAG. Use of an appropriate strategy for scheduling the tasks allows the efficient exploitation of the inherent parallelism.

The following aims should be kept in mind when designing a DAG-based algorithm:

- Sufficient tasks need to be available so that it is rare for a thread to idle because of lack of available work.
- Reloading of data into caches is undesirable so data should be reused immediately if possible, and transfer of tasks between caches (which would cause an additional load) should be avoided when possible. This aim is complicated by the existence of shared caches in multicore machines.

Hogg has recently implemented a DAG-based dense Cholesky factorization as the HSL code `HSL_MP54`. This package uses OpenMP and has been shown to perform well on multicore machines. Full details, including performance results, are given in [13]. The main mechanism used within `HSL_MP54` is a single task pool from which all threads draw tasks to execute and in which new tasks are placed when the data they need become available. Immediate reuse of data is encouraged by the priorities used when selecting a task in the pool for execution, but no other mechanism is implemented to avoid transfer of data between caches.

The DAG-based approach offers significant improvements over utilising more traditional fork-join parallelism by block columns. It avoids the time wasted waiting for 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 worksharing to cope with the case where execution by another user or an asymmetric system load causes some threads to perform significantly slower than others. Such asymmetric loading can be common on multicore systems, caused either by operating system scheduling of other processes on a core that is also executing the application or by unbalanced triggering of hardware interrupts.

The sparse DAG-based factorization code described in this paper is based on a substantial rewrite and improvement of the dense case implemented within `HSL_MP54`.

3 General sparse direct solver framework

Sparse direct solvers for symmetric systems 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 accepts the sparsity structure of the matrix A, without any numerical values. The aim is to find an ordering that limits the fill in the matrix factor L (that is, the number of additional entries created in L during the factorization). A good pivot sequence significantly reduces both memory requirements and the number of floating-point operations required. The ordering usually relies on algorithms based on heuristics such as variants of the approximate minimum degree algorithm [1] or nested dissection [9].

2. An analyse phase (which is sometimes referred to as the symbolic factorization step) that accepts the sparsity structure and a pivot sequence and sets up data structures for efficient factorization. Dense subproblems are identified to speed the factorization. This phase usually replaces the given pivot sequence by a more efficient one that would produce the same result in exact arithmetic. The new pivot sequence is defined by a permutation matrix P.

3. A factorize phase that accepts the numerical values of the matrix A and uses the output from the analyse phase to factorize the matrix. In the positive-definite case, the lower triangular Cholesky factor L is computed, such that $A = PL(PL)^T$.

4. A solve phase that performs forward elimination followed by back substitution using the stored factorization. The factorization may be used for more than one solution.

In a serial implementation, factorize is usually the most time-consuming of the different phases, while the solve phase is generally significantly faster. In many software packages, the first two phases are combined into a single user-callable subprogram. Other packages, including HSL_MA77 [19], do not include the first phase and instead ask the user to supply an ordering. We do the same in our new code HSL_MA87.

3.1 Analyse

The analyse phase of HSL_MA87 is a modification of that of HSL_MA77. The main purpose of the latter is to take the user-supplied pivot sequence and use it to determine the assembly tree, from which it is possible to compute the memory requirements and dependency information necessary for the subsequent factorization.

The *elimination tree* of a symmetric sparse matrix A is defined to be the tree formed as follows:

- Each column j is represented as a node in the tree.
- The parent of node j is node i if l_{ij} is the first entry below the diagonal in column j of the Cholesky factor L.

Before a given column of A can be eliminated, we must first process all columns represented by its descendants in the tree. Thus the elimination tree represents a partial ordering of the elimination of the columns.

In order to employ dense linear algebra kernels (in particular, Level 3 BLAS) that are tuned to effectively exploit machine caches, each adjacent pair of nodes $i-1$ and i whose columns have the same sparsity structure in L from row i onwards are combined. The condensed form of the elimination tree thus obtained is known as the *assembly tree*. Since improved performance can generally be obtained from the BLAS by working with large nodes, nodes whose columns have similar structures may be further amalgamated to form larger nodes; the penalty is that zeros are held as entries, increasing the storage required for L as well as the number of floating-point operations needed in the factorize and solve phases.

Our existing code HSL_MA77 is an out-of-core solver and, by default, it stores the matrix data and the assembly tree in direct-access files. For our new DAG-based solver HSL_MA87, we have modified the analyse phase of HSL_MA77 to work in-core and to use the new data structures described in Section 4.1 when setting up the storage for L.

3.2 Factorize

The factorize phase of a sparse direct solver performs the actual numerical factorization. Current parallel approaches (see, for example, the MUMPS package [2]) normally rely on exploiting two levels of parallelism:

Tree-level parallelism exploits the fact that the assembly tree specifies only a partial ordering between operations associated with different nodes. The only requirement is that each parent node is processed after its children. Independent subtrees can therefore be processed in parallel.

Node-level parallelism exploits parallelism within the operations performed for a node. This is normally used near the root (node with no parent) where the nodes are large and little tree-level parallelism is available.

We have recently developed a separate sparse parallel solver that uses these levels of parallelism (see [14]). Our experience of implementing this approach on multicore machines is that shared caches provide a bottleneck because of premature cache eviction. We found that the speedups achieved were less than ideal and this motivated us to consider developing a sparse DAG-based code.

3.3 Solve

The solve phase takes one or more right-hand sides b and performs forward and back substitutions using the permutation P and factor L to solve the systems (1.1). This phase is normally limited by how fast the factor data can be streamed through the caches.

Our new code `HSL_MA87` currently implements this phase in serial mode, although parallelisation could be achieved along similar lines to those used in the dense code of Hogg [13]. Our intention is that this will be implemented in a future release.

4 DAG-based sparse direct solver

In this section, we look at adapting the use of task DAGs within the sparse factorize phase. At a high level, our modification to the dense DAG-based algorithm is merely the addition of a new type of task that performs a sparse update operation.

4.1 Nodal data structures

Since 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, we can hold it in memory as a dense trapezoidal matrix, as illustrated in Figure 4.1(a). We refer to this matrix as the *nodal matrix*. We store this matrix using the row hybrid blocked structure of Anderson et al. [3] with the modification that "full" storage is used for the blocks on the diagonal rather than storing only the actual entries (thus a rectangular array is used to store the trapezoidal matrix). Using the row hybrid scheme rather than the column hybrid scheme facilitates updates between nodes by removing any discontinuities at row block boundaries. Storing the blocks on the diagonal in full storage allows us to exploit efficient BLAS and LAPACK routines. This structure is illustrated in Figure 4.1(b). Note that the final block on the diagonal is often trapezoidal.

If the number of columns in the nodal matrix is large, we use the block size nb (which may be specified by the user) and most of the blocks will be of size $nb \times nb$. We divide the computation into tasks in which a single block is revised (details in Section 4.2). These tasks correspond to the vertices of our implicitly-held DAG.

If the number of columns nc in the nodal matrix is less than nb but the number of rows is large, using the block size nb can lead to small tasks and inefficient execution. We therefore attempt to balance the number of entries in the blocks by basing the block size on the value nb^2/nc. We round the size up to a multiple of 8 since our experience is that this enhances performance. Having block sizes that differ from node to node is unusual but we found that it sometimes greatly improves the performance. Of course, it will not help if most of the nodes have large numbers of columns.

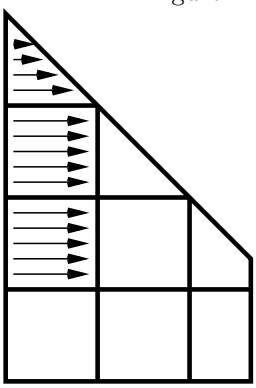
The tasks are partially ordered; for example, the updating of a block of a nodal matrix from a block column of L that is associated with one of the node's descendants has to wait for all the rows of the block column that is needs becoming available. As soon as a thread determines that all the data needed for a task is available, it places the task in a small stack of tasks for execution by itself or a thread with which it shares its cache. If this stack becomes full, its bottom half is moved to a task pool for execution by any thread, see Section 4.3.

4.2 Tasks

Following the design of our dense DAG-based code [13], we split the work involved in the sparse factorization of A into a number of separate tasks, which we categorise as follows (and illustrate graphically in Figure 4.2):

factorize(`diag`) Computes the traditional dense Cholesky factor L_{triang} of the triangular part of a block `diag` that is on the diagonal using the LAPACK subroutine `_potrf`. If the block is trapezoidal, this

Figure 4.1: Row hybrid block structure for a nodal matrix.



(a) Graphical view

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

(b) Indices of entries

is followed by a triangular solve of its rectangular part

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

using the BLAS subroutine `_trsm`.

solve(`dest`, `diag`) Performs a triangular solve of the off-diagonal block `dest` by the Cholesky factor L_{triang} of the block `diag` on its diagonal. i.e.

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

using the BLAS subroutine `_trsm`.

update_internal(`dest`, `rsrc`, `csrc`) Within a nodal matrix, performs the update

$$L_{dest} \Leftarrow L_{dest} - L_{rsrc} L_{csrc}^{T}$$

where L_{dest} is the matrix of the block `dest`, L_{rsrc} is the matrix of the off-diagonal block `rsrc` with rows that correspond to the rows of L_{dest}, and L_{csrc} is the matrix of those rows of the off-diagonal block `csrc` that correspond to the columns of L_{dest}, see Figure 4.2(c). If `dest` is an off-diagonal block, we use the BLAS 3 kernel `_gemm` for this. If `dest` is a block on the diagonal, we use the BLAS 3 kernel `_syrk` for the triangular part and `_gemm` for the rectangular part, if any.

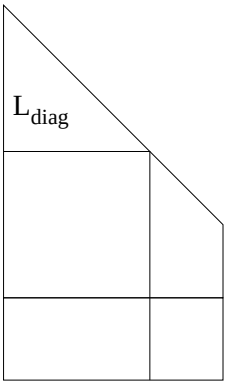
update_between(`dest`, `snode`, `scol`) Performs the update

$$L_{dest} \Leftarrow L_{dest} - L_{rsrc} L_{csrc}^{T}$$

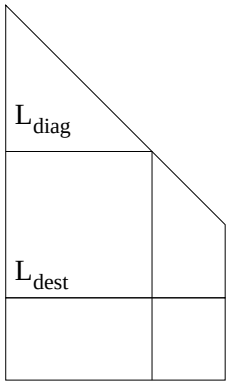
where L_{dest} is a submatrix of the block `dest` of an ancestor of the node `snode` and L_{rsrc} and L_{csrc} are submatrices of contiguous rows of the block column `scol` of the node `snode`. The first row of L_{rsrc} is the first row of `scol` that corresponds to a row in the block `dest` and the last row of L_{rsrc} is the last row of `scol` that corresponds to a row in the block `dest`. Similarly, the first/last row of L_{csrc} is the first/last row of `scol` that corresponds to a column in the block `dest`. The set of rows and columns of `dest` thus determine which two sets of contiguous rows in `scol` are involved. Unless the number of entries updated is very small, we exploit the BLAS 3 kernel `_gemm` (and/or `_syrk` for a block that is on the diagonal) by placing its result in a buffer from which we add the update into the appropriate entries of the destination block `dest`, see Figure 4.2(d).

This final update could have been cast in a number of different ways. We have chosen the above left-looking form because:

Figure 4.2: Graphical interpretations of sparse DAG tasks

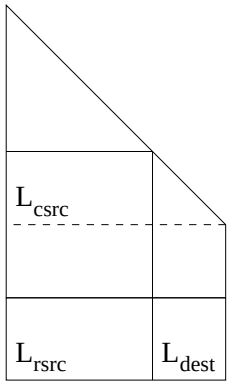


(a) factorize(`diag`)



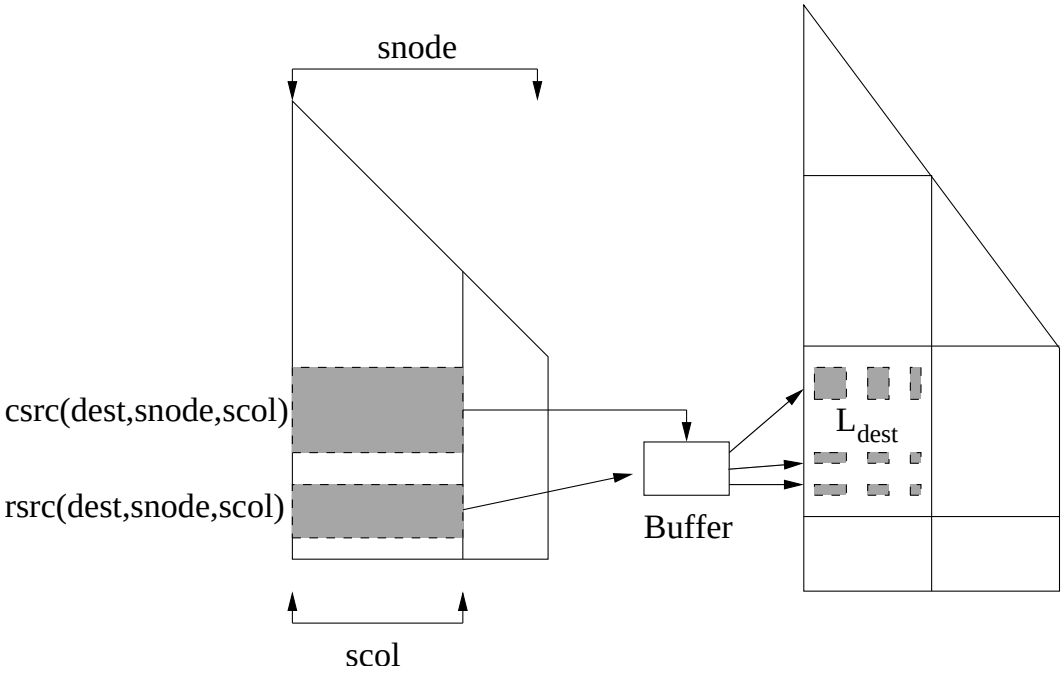
(b) solve(`dest`, `diag`)

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



(c) update_internal(`dest`, `rsrc`, `csrc`)

$L_{dest} \Leftarrow L_{dest} - L_{rsrc} L_{csrc}^{T}$



(d) update_between(`dest`, `snode`, `scol`)

1. Form outer product $L_{rsrc} L_{csrc}^{T}$ into Buffer.
2. Distribute the results into the destination block L_{dest}.

- The blocks of the hybrid block structure are stored contiguously and by rows (see Figure 4.1(b)), so the boundaries between them can be ignored and we can perform operations involving arbitrary row ranges.
- We could have cast this as an operation from a pair of blocks, each writing to multiple blocks. This would often cause the same destination block to be updated more than once from the same block column. As we have it, a destination block is updated exactly once for each block column of a descendant node. This is desirable since contested writes cause more cache misses than contested reads (as a write may invalidate a cache line in another cache but a read cannot).
- As we are updating a single block, the number of operations is bounded by $2nb^3$, so we are not generating a large amount of work per task, though we do risk generating very little computation.

During the analyse phase, we calculate the number of tasks that need to be performed for each block of L. For a block on the diagonal, this is the number of updates on it. For an off-diagonal block, it is one more than the number of updates since we need to include the factorization of the block on the diagonal in its block column.

During the factorization, a running count of outstanding tasks is kept for each block. This is initialized to the value calculated during the analyse phase and decremented by one as each task is performed for it. When the count reaches zero for a block on the diagonal, a factorize task for it is stored. When a factorize task is completed, its count is given the special value -2 and the count of each off-diagonal block in its block column is decremented by one. When the count reaches zero for an off-diagonal block, a solve task for it is stored.

When a solve task is completed, the count of its block is given the special value -2. If the block overlaps the triangular part of the nodal matrix, an update_internal task with its overlapping rows as L_{csrc} is stored for each block in the block column that has count -2, including itself, as L_{rsrc}. If the block does not overlap the triangular part, an update_internal task is stored with it as L_{rsrc} for each block of the block column that does overlap it and has a count of -2 with the overlapping rows as L_{csrc}. If the block overlaps the rectangular part, all blocks in ancestor nodes that have tasks depending on the block column are checked. If such a task depends only on rows in blocks with a count of -2 and depends on at least one row of the block for which the solve task was just completed, the task is stored.

4.3 Task dispatch engine

We have improved on the task dispatch engine of the dense DAG code `HSL_MP54` by using thread-specific stacks to make the code more cache aware. In the dense case [13], we found that complex prioritisation schemes that take into account critical paths offer very limited benefit. As a result, in `HSL_MA87` we have opted for a simpler prioritisation scheme, and instead favour cache awareness using a system of local task stacks together with a single task pool.

For each cache, there is a small stack of tasks that are intended for use by the threads sharing this cache. During the factorization, tasks are added to or drawn from the top of this local stack. If the local stack becomes full, a global lock is acquired and the bottom half of the local stack (that is, the tasks that have been in the stack the longest so that their data are unlikely still to be in the local cache) is moved to the task pool. The tasks in the task pool are given the following priorities:

1.	**factorize**	Highest priority
2.	**solve**	
3.	**update_internal**	
4.	**update_between**	Lowest priority

Tasks are prioritised in this way to try and ensure that a task is selected that, once performed, will result in as many new tasks as possible becoming available. This can lead to the task pool becoming very large. The default size of the task pool is 100,000. This was chosen on the basis of numerical experimentation (note

that the maximum size of the task pool for the test problems reported on in Section 5 was approximately 110,000 and for many of our test problems the task pool size did not exceed 10,000).

When a thread requires a task, it first attempts to draw a task from the top of its local stack. At this stage, the task priorities are not used. If there is a factorize task available it will always be on the top of the stack and so will be selected first. Once a solve task is completed, any update tasks that are spawned are placed on the stack, above any remaining stacked solve tasks. It is advantageous to perform these updates before another solve task since the data they need are likely to be in the local cache.

If the local task stack is empty, the thread tries to take a task from the task pool. Should this also be empty, the thread searches for the largest local stack belonging to another cache. If found, the tasks in the bottom half of this local stack are moved to the task pool (workstealing). The thread then takes the task of highest priority from the pool as its next task.

The task pool uses a separate stack for each priority level, each implemented as a singly linked list. The stacks share the same workarray. An additional linked list of free entries is maintained to facilitate this. If the size of the workarray is found to be too small, its size is doubled.

4.4 User interface

In this section, we briefly discuss the user interface to our new sparse Cholesky solver `HSL_MA87` and the options it offers its users. Further details, together with a simple example to illustrate how the package is called, are given in the user documentation that is supplied with the package.

Following the normal sparse direct solver design, `HSL_MA87` has separate subroutines that the user must call to perform the analyse, factorize and solve phases. An optional subroutine `MA87_input` may be called before the analyse phase if the user would like his or her matrix data to be checked for errors, duplicates, out-of-range entries etc. The matrix must be held in compressed sparse column format. Three derived types are used:

- `MA87_info`: its components are used to provide information about the execution of each of the user-callable subroutines. After the analyse phase, this includes information on the assembly tree (its depth and the number of nodes) and, after the factorization is complete, the maximum size of the task pool workarray. In addition, the component `flag` is an error flag that is set to a negative value if a fatal error is encountered (causing the computation to terminate prematurely) and to a value greater than 0 if a warning has been detected (for example, the matrix data on the call to `MA87_input` contained out-of-range indices that have been removed).

- `MA87_control`: its components control the action. There are components that control diagnostic printing, node amalgamation (and thus the size of the nodes of the assembly tree), the block size nb and initial size of the task pool workarray. These controls are automatically given default values in the definition of the type but one or more may be reset by the user.

- `MA87_keep`: used to hold data about the problem being solved. This includes the assembly tree and the computed factor, as well as data relating to the task DAG. It must be passed unchanged between the subroutines.

We remark that the use of derived types significantly simplifies the user interface and allows us to have short argument lists, which helps reduce the possibility of the user introducing errors.

More than one call to `MA87_factorize` may follow a call to `MA87_analyse` and more than one call to `MA87_solve` may follow a call to `MA87_factorize`. Note, however, that it is more efficient to specify multiple right-hand sides on a call to `MA87_solve` than to make repeated calls with a single right-hand side. This is because, with multiple right-hand sides, advantage is taken of high-level BLAS. An optional argument to `MA87_solve` allows a partial solve (that is, only forward substitution or only back substitution is performed).

5 Numerical results

5.1 Test environment

The experiments we report on in this paper were all performed on our multicore test machine `fox`, details of which are given in Table 5.1. The sparse matrices used are listed in Table 5.2. This set comprises 30 examples that arise from a range of practical applications. In selecting the test set, our aim was to choose a wide variety of large-scale problems. Each problem is available from the University of Florida Sparse Matrix Collection [7].

Table 5.1: Specifications of our test machine `fox`.

	2-way quad Harpertown (`fox`)
Architecture	Intel(R) Xeon(R) CPU E5420
Clock	2.50 GHz
Cores	2×4
Theoretical peak (1/8 cores)	10 / 80 Gflop/s
`DGEMM` peak (1/8 cores[1])	9.3 / 72.8 Gflop/s
Memory	8 GB
Compiler	Intel 11.0 with option -fast
BLAS	Intel MKL 10.1

[1] Measured by using MPI to run independent matrix-matrix multiplies on each core

For those matrices that are only available as a sparsity pattern, reproducible pseudo-random off-diagonal entries in the range $(0, 1)$ were generated using the HSL package `HSL_FA14`, while the i-th diagonal entry, $1 \leq i \leq n$, is set to $\max(100, 10\rho_i)$, where ρ_i is the number of off-diagonal entries in row i of the matrix, thus ensuring that the generated matrix is positive definite. For all tests, the right-hand side b is generated so that the required solution x is the vector of ones.

Unless stated otherwise, runs are performed using all 8 cores on our test machine and all control parameters used by the solvers are given their default settings. All times are elapsed times, in seconds, measured using the Fortran subroutine `system_clock`.

The analyse phase of the HSL sparse direct solver `MA57` [8] is used to compute the pivot sequence for `HSL_MA87`. `MA57` automatically chooses between an approximate minimum degree and a nested dissection ordering; in fact, for all our test problems, it selects a nested dissection ordering that is computed using `METIS_NodeND` [16, 17]. In Table 5.2, we include the number of millions of entries in the matrix factor (denoted by $nz(L)$) when this pivot sequence is used by `HSL_MA87` with the node amalgamation parameter `nemin` set to 16 (see Section 5.3).

5.2 Dense comparison

We can trivially represent a dense problem as a sparse one with a single node that contains all n variables. This allows us to compare the efficiency of the dense tasks within `HSL_MA87` with other dense Cholesky factorization implementations (although it should be noted that `HSL_MA87` accepts a different data format and hence avoids the reordering that they may require).

In Table 5.3 comparisons are given for randomly generated dense matrices of order up to 20000. Results for the following codes are presented:

`HSL_MA87(a)`: `HSL_MA87` **with** processor affinity enabled. Processor affinity refers to whether threads are tied to specific processors, or whether the operating system is allowed to move them. For our cache management techniques to reflect the actual physical circumstance, processor affinity needs to be

Table 5.2: Test matrices and their characteristics. n denotes the order of A in thousands; $nz(A)$ and $nz(L)$ are the number of entries in the lower triangular part of A and in L, respectively, in millions; * indicates only the sparsity pattern is provided.

Identifier	n	$nz(A)$	$nz(L)$	Application/description
1. DNVS/thread	29.7	2.2	24.8	Threaded connector/contact
2. DNVS/m_t1	97.6	4.9	32.4	Tubular joint
3. Chen/pkustk13*	94.9	3.4	32.7	Machine element, 21 noded solid
4. DNVS/shipsec8	114.9	3.4	38.9	Ship section
5. GHS_psdef/crankseg_1	52.8	5.3	40.0	Linear static analysis
6. Rothberg/gearbox*	153.7	4.6	40.5	Aircraft flap actuator
7. Rothberg/cfd2	123.4	1.6	41.6	CFD pressure matrix
8. DNVS/shipsec1	140.9	4.0	42.1	Ship section
9. CEMW/tmt_sym	726.7	2.9	43.9	Electromagnetics problem
10. DNVS/fcondp2*	201.8	5.7	55.2	Oil production platform
11. Um/2cubes_sphere	101.5	0.9	56.6	Electromagnetics, 2 cubes in a sphere
12. GHS_psdef/crankseg_2	63.8	7.1	59.8	Linear static analysis
13. DNVS/ship_003	121.7	4.1	63.9	Ship structure—production
14. Boeing/pwtk	217.9	5.9	64.7	Pressurised wind tunnel
15. DNVS/troll*	213.5	6.1	68.6	Structural analysis
16. DNVS/halfb*	224.6	6.3	70.2	Half-breadth barge
17. GHS_psdef/bmwcra_1	148.8	5.4	74.0	Automotive crankshaft model
18. Schmid/thermal2	1228.0	4.9	74.5	Unstructured thermal FEM
19. DNVS/fullb*	199.2	6.0	78.7	Full-breadth barge
20. Schenk_AFE/af_shell3	504.9	17.6	104.8	Sheet metal forming matrix
21. JGD_Trefethen/Trefethen_20000	20.0	0.3	102.5	Integer matrix
22. Chen/pkustk14*	151.9	7.5	111.9	Civil engineering. Tall building
23. ND/nd12k	36.0	14.2	117.8	3D mesh problem
24. GHS_psdef/apache2	715.2	2.8	159.1	3D structural problem
25. GHS_psdef/ldoor	952.2	23.7	164.0	Large door
26. GHS_psdef/inline_1	503.7	18.7	185.4	Inline skater
27. Koutsovasilis/F1	343.8	13.4	210.7	AUDI engine crankshaft
28. AMD/G3_circuit	1585.5	4.6	232.7	Circuit simulation
29. Oberwolfach/boneS10	914.9	28.2	302.1	Bone micro-finite element model
30. ND/nd24k	72.0	28.7	322.9	3D mesh problem

Table 5.3: A comparison of dense Cholesky implementations. Speeds in Gflop/s are reported.

n	HSL_MA87(a)	HSL_MA87(b)	HSL_MP54	dpotrf
100	0.71	0.74	1.63	3.25
500	15.3	15.0	17.7	20.7
1000	31.3	30.3	29.3	35.1
1500	40.8	40.0	35.7	42.6
2000	48.8	47.2	40.8	47.3
2500	51.8	50.9	44.3	51.4
5000	62.2	61.9	55.8	57.3
10000	66.3	65.1	63.7	64.4
20000	69.6	68.9	67.9	67.1

enabled. On our test machine with the Intel compiler we do this by setting the environment variable `KMP_AFFINITY` to `compact`.

`HSL_MA87(b):` `HSL_MA87` **without** processor affinity enabled.

`HSL_MP54:` dense DAG code described in [13]. It is designed to perform both partial and complete Cholesky factorizations.

`dpotrf:` LAPACK Cholesky factorization subroutine supplied by Intel MKL 10.1, which we believe uses a DAG-based algorithm.

The test results were averaged over ten runs. For each run, the factorization times for different problems of the same size were accumulated until the total elapsed time was at least one second; the average speed in Gflop/s was then calculated.

As can be seen from Table 5.3, processor affinity marginally enhances the performance of `HSL_MA87` unless n is small, and so we use this in all further tests reported on in this paper. Comparing `HSL_MA87` with `HSL_MP54` and `dpotrf`, we see that `HSL_MA87` is competitive for sufficiently large n (n greater than about 2000) and its advantage over the other codes increases with n. However, its performance compares less favourably for small problems. This is due to workstealing with the relatively small blocks used for optimal performance on these problems causing an excessive number of blocks to be switched between caches. In the sparse case, we do not anticipate this will cause problems since there are many more tasks and more than one initial task — workstealing should only play a real role near the start and end of the factorization.

5.3 Effect of node amalgamation

The `HSL_MA87` strategy of amalgamating nodes of the tree is taken from `HSL_MA77` and explained in [19]. A child is amalgamated with its parent if both involve less than a given number, `nemin`, of eliminations. We show in Table 5.4 a few results for `HSL_MA87` for a subset of our problems that were selected to illustrate the effects of varying `nemin`.

Table 5.4: Comparison of the number of entries in L (in millions) and the factorize and solve times for values of the node amalgamation parameter `nemin` in the range `1` to `64`. The fastest factorize times (and those within 3 per cent of the fastest) are in bold.

	$nz(L)$					Factorize times					Solve times				
	1	8	16	32	64	1	8	16	32	64	1	8	16	32	64
7.	38.3	40.0	41.6	44.2	49.8	1.27	**1.01**	1.09	1.05	1.09	0.46	0.36	0.35	0.36	0.38
13.	60.2	61.6	63.9	69.1	77.0	2.43	**2.26**	2.37	2.33	**2.31**	0.55	0.52	0.53	0.53	0.57
17.	69.7	71.6	74.0	77.6	83.9	2.18	1.91	1.94	**1.74**	1.88	0.67	0.61	0.60	0.60	0.63
19.	74.5	76.1	78.8	85.5	100.5	**2.92**	**2.90**	**2.86**	**2.89**	3.05	0.70	0.67	0.66	0.67	0.77
25.	144.6	154.7	164.0	182.7	223.6	4.95	3.87	3.39	**3.10**	3.42	1.65	1.59	1.63	1.65	1.84

Looking at the factorize times, we see that for most problems there are worthwhile savings if `nemin` is chosen to be greater than `1`, but no one value of `nemin` consistently gives the best times. For some problems, $nz(L)$ grows rapidly with `nemin` and this leads to slower solve times (for example, for problems 19 and 25 the solve times for `nemin` = `64` are significantly greater than for `nemin` = `32`). Based on our results, in `HSL_MA87` we have chosen the default value to be `32`. However, if the number of entries in L increases slowly with `nemin`, it can be advantageous to use an even larger value of `nemin`. For instance, we ran problem 23 (`ND/nd24k`) with larger `nemin` and found the factorization time reduced from 54 seconds with `nemin` = `32` to 47 seconds with `nemin` = `96`, while the number of entries in L increased only slightly from 326×10^6 to 336×10^6 (so that there was almost no increase in the solve time).

5.4 Block size

The block size nb, discussed in Section 4.1, is a parameter under the user's control. In Table 5.5, we report the factorize time for a range of block sizes on a single core and on 8 cores. The fastest times (and those within 3 per cent of the fastest) are in bold. We see that, on a single core, the best times are obtained using a larger block size than on 8 cores, but in our tests the reductions in time using $nb > 256$ are less than 3 per cent. Based on our experiments, the default block size used by `HSL_MA87` is 256.

Table 5.5: Comparison of the factorize times for different block sizes.The fastest times (and those within 3 per cent of the fastest) are in bold.

	Single core. Block size nb						8 cores. Block size nb				
	128	192	256	320	384	448	128	192	256	320	384
1.	5.75	5.34	**5.24**	**5.22**	**5.17**	**5.12**	1.00	0.97	**0.92**	**0.91**	1.03
5.	7.94	7.48	**7.32**	**7.21**	**7.16**	**7.14**	1.34	**1.28**	**1.28**	1.39	1.53
11.	13.2	12.6	**12.3**	**12.1**	**12.0**	**12.0**	2.42	2.11	**2.10**	**2.00**	2.11
17.	11.5	11.0	**10.7**	**10.6**	**10.6**	**10.5**	1.88	1.86	**1.74**	1.86	2.00
23.	95.3	**87.5**	**86.4**	**84.6**	**85.3**	**85.4**	16.1	15.2	**14.2**	**14.2**	15.2
29.	52.9	50.1	**48.9**	**48.8**	**48.4**	**48.4**	8.64	8.34	**7.69**	**7.66**	**7.72**

5.5 Local task stack size

In Section 4.3, we discussed the use of local task stacks. In Table 5.6, we report the factorization times for a range of local task stack sizes and also for no local task stack. We see that the gains on our 8 core machine achieved using a local task stack are generally modest (for some problems, the time saving is less than 10 per cent but for others, including problem 13, the savings are more significant). Based on our experiments, `HSL_MA87` uses a local task stack of size 100. Note we anticipate that on a machine with a larger number of cores, the gains resulting from the use of local task stacks will be greater.

With the local stacks in place, we found that the priorities used for the global task pool had little effect on our machine for the problems we tried. However, it is our belief that they would be important on a machine with more cores so that task starvation is a serious risk.

Table 5.6: Comparison of the factorize times for different local task stack sizes. The fastest times (and those within 3 per cent of the fastest) are in bold.

	No local	Stack size			
	stack	10	50	100	200
4.	1.28	1.19	1.18	**1.01**	1.09
10.	1.56	1.44	1.44	**1.38**	1.49
13.	2.98	2.33	2.38	**2.17**	**2.17**
19.	2.97	2.97	2.89	**2.79**	**2.73**
25.	3.49	3.45	**3.13**	**3.11**	**3.16**

5.6 Speedups and speed for `HSL_MA87`

One of our concerns is the speedup achieved by `HSL_MA87` as the number of cores increases. In Figure 5.1, we plot the speedups in the factorize times when 2, 4, and 8 cores are used. We see that the speedup on 2 cores is close to 2, on 4 cores it is generally more than 3 and for the largest problems (in terms of $nz(L)$)

it exceeds 3.3. On 8 cores, HSL_MA87 achieves speedups of more than 6 for many of the largest problems and for all our test problems, the speedup exceeds 5.

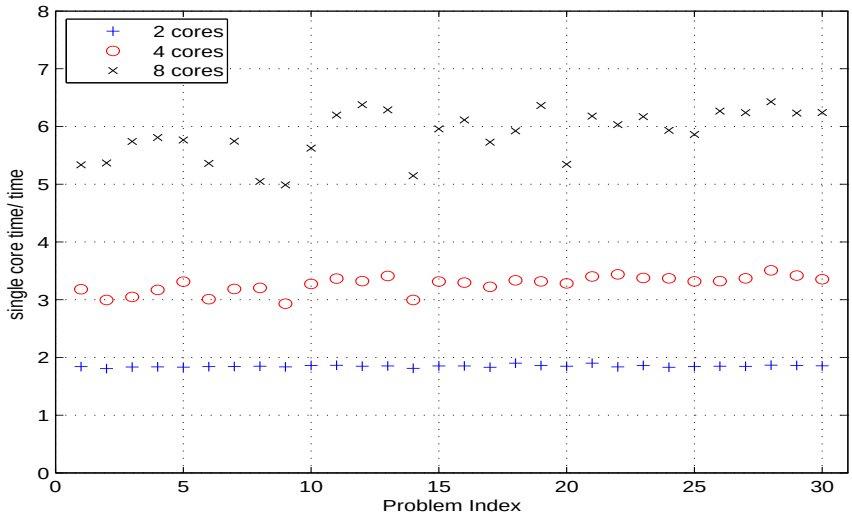


Figure 5.1: The ratios of the factorize times on 2, 4 and 8 cores to the factorize time on a single core.

Of course, our primary concern is the actual speed achieved. We show the speeds in Gflop/s on 8 cores in Figure 5.2. Here we compute the flop count from a run with the node amalgamation parameter nemin having the value 1. We note that for 11 of our 30 test problems, the speed exceeds 36.4 Gflop/s, which is half the maximum dgemm speed, see Table 5.1. Furthermore, on all but two of the problems, it is greater than 24.3 Gflop/s, which is a third of the dgemm maximum.

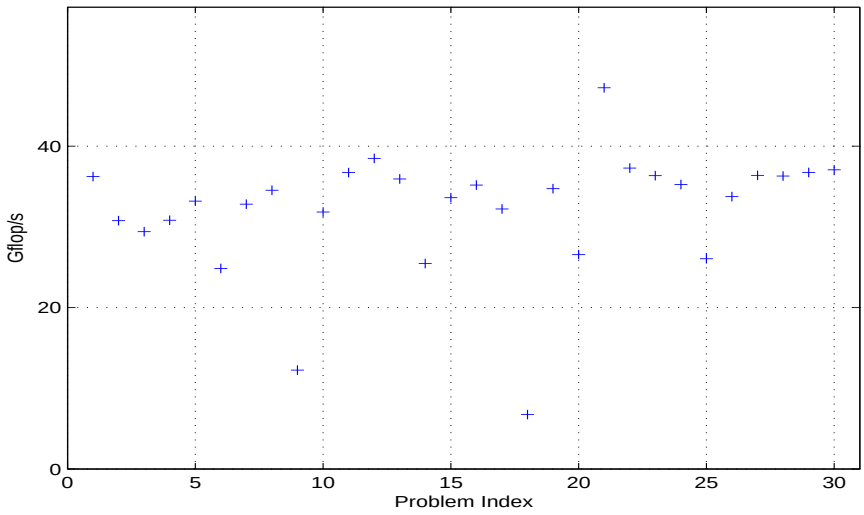


Figure 5.2: The speeds in Gflop/s on 8 cores.

6 Comparisons with other solvers

We wish to compare the performance of HSL_MA87 with some of the other readily available solvers. In this section, we first briefly introduce the state-of-the-art solvers we use in our comparisons.

6.1 MA57

MA57 [8] is a multifrontal sparse direct solver that has been an important part of the HSL mathematical software library since it was first released in 2000. The code is a serial code, written in Fortran 77 (a Fortran 95 interface is also available). MA57 offers a range of options, including several sparsity orderings, solving for multiple right-hand sides, partial solutions, error analysis, scaling, a matrix modification facility, and a stop and restart facility. Although primarily designed for indefinite problems, it can also be used to solve positive-definite linear systems. We use Version 3.2.0 in our tests and, since our problems are all positive definite, the pivoting control is set so that pivoting is switched off; the scaling option is also disabled.

Although MA57 is a serial code, it is included in this study since it is a widely used and well known package so that comparing its performance with that of our new code HSL_MA87 on a single core is of interest.

6.1.1 MUMPS

MUMPS (MUltifrontal Massively Parallel Solver) [2] is a well known package for solving sparse linear systems $Ax = b$, where A can be non-symmetric, symmetric positive definite, or general symmetric. MUMPS uses a multifrontal approach and exploits both parallelism arising from sparsity in A and from dense factorization kernels. Important features of the MUMPS package include the input of the matrix in assembled or elemental format, error analysis, iterative refinement, scaling of the original matrix, detection of null pivots, basic estimate of rank deficiency and null space basis, and computation of a Schur complement matrix. In addition, MUMPS offers limited out-of-core facilities, allowing the factors to be stored on disk. MUMPS incorporates several built-in ordering algorithms, a tight interface to a number of external ordering packages (including MeTiS), and allows the user to input a given ordering. The software is written in Fortran 90 and a C interface is available. The parallel version of MUMPS uses MPI for message passing and makes use of the BLAS, BLACS, and ScaLAPACK libraries. Full details of the MUMPS package are available at `http://graal.ens-lyon.fr/MUMPS/`

We employ Version 4.8.3 in our tests.

6.1.2 PARDISO

PARDISO [20] is a thread-safe software package for solving large sparse symmetric and non-symmetric linear systems of equations on shared memory multiprocessors. During the factorize phase, PARDISO exploits pipelining parallelism and memory hierarchies with a combination of left- and right-looking techniques using Level-3 BLAS. The code employs OpenMP directives. Further details are available at `http://www.pardiso-project.org/`

Originally developed at the University of Basel, PARDISO is now included in the Intel MKL library. We use the Intel MKL Version 10.1.

6.1.3 Comparisons on one and eight cores

We first compare the performance of the above packages with that of HSL_MA87 when run on a single core of our test machine. The ratios of the factorize times for each package to the factorize time for HSL_MA87 are given in Figure 6.1. For all the test problems except 9 and 18, MA57 is significantly slower than HSL_MA87. This is probably because MA57 is not primarily designed for positive-definite systems and the current version does not exploit recently developed dense linear algebra kernels (see, for example, [3, 18, 19]). There are five problems for which MUMPS is more than 10 per cent faster than HSL_MA87 and

twelve for which HSL_MA87 is more than 10 per cent faster than MUMPS; for the remaining problems, the performance of the two codes is comparable. In general, there is little to choose between the HSL_MA87 and PARDISO factorize times, but there are a number of problems for which one significantly outperforms the other. It is not clear why this should be the case and we are not able to predict when one solver will be the faster.

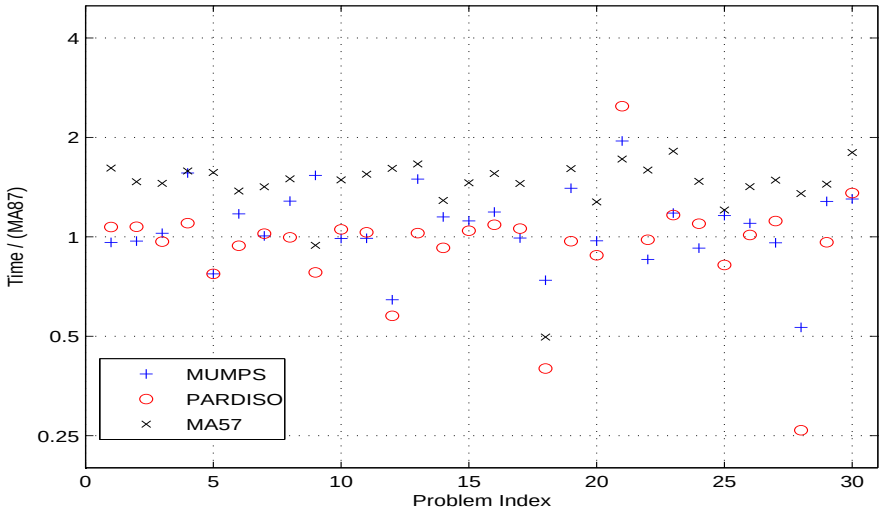


Figure 6.1: The ratios of the MUMPS, PARDISO and MA57 factorize times to the HSL_MA87 factorize time (single core).

In Figure 6.2 the ratios of the factorize times for MUMPS and PARDISO to the factorize time for HSL_MA87 on 8 cores are given. Ratios for problem 21 are omitted as they are outside the plotted range (in this case, the HSL_MA87, MUMPS and PARDISO factorize times were 17, 120 and 96 seconds, respectively). We see that HSL_MA87 generally achieves much better speeds than MUMPS and, with the exception of a small number of problems, the performance of HSL_MA87 is either comparable to or better than that of the Intel version of PARDISO.

7 Future work

Having demonstrated that the sparse DAG approach is promising for use on multicore chips, we must consider how it can be improved. Ideally, we would like to match the capabilities of the serial HSL code HSL_MA77, but with improved performance. This means adding an out-of-core capability and the ability to efficiently and reliably solve indefinite problems. Further, we would like to improve the performance of the sparse DAG approach so that it obtains speedups (and flop rates) in the same range as the dense case.

We believe that our cache-aware scheduler can be improved to more accurately model the caching arrangements of whatever machine we are using, rather than just the shared level-2 caches on our current system. A recursive layer based approach would even allow us to model MPI and out-of-core storage as further levels of cache. Having a queue containing the next few tasks would allow data to be prefetched before it is required, thus enabling the latency to be effectively hidden. A good heuristic for selecting tasks may be able to match current approaches in minimising communication volume.

Buttari et. al. [5] discuss a version of their dense DAG code for *LU* factorization that incorporates a pairwise partial pivoting strategy. It may be possible to build on this approach in the sparse symmetric indefinite case. We plan to follow the work of Reid and Scott [19] and delay columns corresponding to

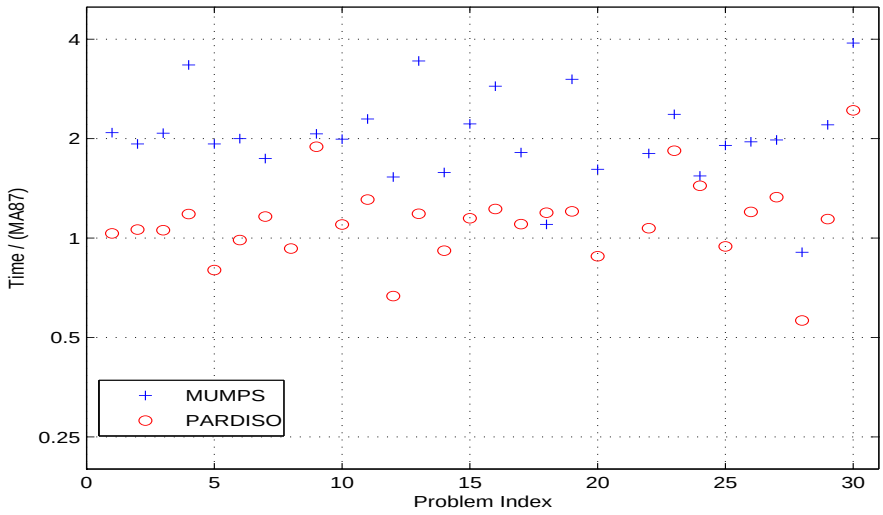


Figure 6.2: The ratios of the MUMPS and PARDISO factorize times to the `HSL_MA87` factorize time (8 cores).

pivots that are rejected at a node because of stability considerations. We will prepend delayed columns to the parent nodal matrix and if necessary increase the task counts for the blocks that are affected.

8 Code availability

Our new sparse parallel Cholesky solver `HSL_MA87` discussed in this paper has been developed for inclusion in the mathematical software library HSL. All use of HSL requires a licence. Individual HSL packages (together with their dependencies and accompanying documentation) are available without charge to individual academic users for their personal (non-commercial) research and for teaching; licences for other uses involve a fee. Details of the packages and how to obtain a licence plus conditions of use are available at `www.cse.scitech.ac.uk/nag/hsl/`.

9 Acknowledgement

We would like to thank our colleague Iain Duff for carefully reading a draft of this paper and making helpful suggestions for improving the presentation.

References

[1] P. Amestoy, T. Davis, and I. Duff, *Algorithm 837: AMD, an approximate minimum degree ordering algorithm*, ACM Transactions on Mathematical Software, 30 (2004), pp. 381–388.

[2] P. Amestoy, I. Duff, J.-Y. L'Excellent, and J. Koster, *A fully asynchronous multifrontal solver using distributed dynamic scheduling*, SIAM J. Matrix Analysis and Applications, 23 (2001), pp. 15–41.

[3] B. Andersen, J. Gunnels, F. Gustavson, J. Reid, and J. Wasniewski, *A fully portable high performance minimal storage hybrid format cholesky algorithm*, ACM Transactions on Mathematical Software, 31 (2005), pp. 201–207.

[4] 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.

[5] 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.

[6] Y. Chen, T. Davis, W. Hager, and S. Rajamanickam, *Algorithm 887: CHOLMOD, supernodal sparse cholesky factorization and update/downdate*, ACM Transactions on Mathematical Software, 35 (2008). Article 22 (14 pages).

[7] T. Davis, *The University of Florida sparse matrix collection*, Technical Report, University of Florida, 2007. http://www.cise.ufl.edu/∼davis/techreports/matrices.pdf.

[8] I. Duff, *MA57– a new code for the solution of sparse symmetric definite and indefinite systems*, ACM Transactions on Mathematical Software, 30 (2004), pp. 118–154.

[9] A. George, *Nested dissection of a regular finite-element mesh*, SIAM J. Numerical Analysis, 10 (1973), pp. 345–363.

[10] N. Gould, J. Scott, and Y. Hu, *A numerical evaluation of sparse direct solvers for the solution of large, sparse, symmetric linear systems of equations*, ACM Transactions on Mathematical Software, 33 (2007). Article 10, 32 pages.

[11] A. Gupta, M. Joshi, and V. Kumar, *WSMP: A high-performance serial and parallel sparse linear solver*, Technical Report RC 22038 (98932), IBM T.J. Watson Reserach Center, 2001. www.cs.umn.edu/˜agupta/doc/wssmp-paper.ps.

[12] P. Hénon, P. Ramet, and J. Roman, *PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems*, Parallel Computing, 28 (2002), pp. 301–321.

[13] J. Hogg, *A DAG-based parallel Cholesky factorization for multicore systems*, Technical Report RAL-TR-2008-029, Rutherford Appleton Laboratory, 2008.

[14] J. Hogg, J. Reid, and J. Scott, *A sparse symmetric out-of-core linear solver for multicore architectures*, Technical Report RAL-TR-2009-005, Rutherford Appleton Laboratory, 2009. To appear.

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

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

[17] ——, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Scientific Computing, 20 (1999), pp. 359–392.

[18] J. Reid and J. Scott, *An efficient out-of-core sparse symmetric indefinite direct solver*, Technical Report RAL-TR-2008-024, Rutherford Appleton Laboratory, 2008. Submitted to ACM Transactions on Mathematical Software.

[19] ——, *An out-of-core sparse Cholesky solver*, ACM Transactions on Mathematical Software, 36 (2009). To appear.

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