Technical Report

Design of a multicore sparse Cholesky factorization using DAGs

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

December 23, 2009

RAL-TR-2009-027

ISSN 1358-6254

Design of a multicore sparse Cholesky factorization using DAGs

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

ABSTRACT

The rapid emergence of multicore machines has led to the need to design new algorithms that are efficient on these architectures. Here, we consider the solution of sparse symmetric positive-definite linear systems by Cholesky factorization. We were motivated by the successful division of the computation in the dense case into tasks on blocks and use of a task manager to exploit all the parallelism that is available between these tasks, whose dependencies may be represented by a directed acyclic graph (DAG). Our algorithm is built on the assembly tree and subdivides the work at each node into tasks on blocks, whose dependencies may again be represented by a DAG. To limit memory requirements, updates of blocks are performed directly.

Our algorithm is implemented within a new solver `HSL_MA87`. It is written in Fortran 95 plus OpenMP and is available as part of the software library HSL. Using problems arising from a range of practical applications, we present experimental results that support our design choices and demonstrate `HSL_MA87` obtains good serial and parallel times on our 8-core test machines. Comparisons are made with existing modern solvers and show that `HSL_MA87` generally outperforms these solvers, particularly in the case of very large problems.

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: jonathan.hogg@stfc.ac.uk john.reid@stfc.ac.uk 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`.

December 23, 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, symmetric positive-definite matrix of order n. A number of direct solvers using the Cholesky factorization $A = LL^T$ have been developed for this problem in recent years, including the serial codes `MA57` [8] and `HSL_MA77` [24] from the HSL software library [17] and CHOLMOD [6] as well as the parallel codes MUMPS [1], PARDISO [25], PaStiX [15], TAUCS [18], and WSMP [14]. A detailed comparison of serial codes is provided by Gould, Hu, and Scott [12]. We summarise the main features of the parallel codes later in this section. They each have three phases: **analyse** the structure, **factorize** the matrix A, and **solve** sets of equations (1.1). Moreover, they each rely on an assembly tree that is constructed by nested dissection and/or other ordering strategies, followed by node amalgamation to make more effective use of Level-3 BLAS at the expense of additional entries in L and operation counts. Some are multifrontal codes (see [9], sections 10.7 and 10.8), relying on temporary storage for the frontal matrices at the active nodes of the assembly tree and the generated-element matrices from their child nodes. Others subdivide the generated-element matrix at each node and add the parts directly into the columns of L associated with the ancestors of the node.

The rapid emergence of multicore machines has led to the need to design new algorithms that are able to effectively exploit these architectures. These architectures differ from traditional SMP by having shared caches and shared links to memory, making efficient reuse of data in (shared) cache more important. Our approach uses the assembly tree and is non-multifrontal. Inspired by recent multicore solvers for dense positive-definite linear systems [4, 5], we aim to achieve good parallelization on a multicore architecture by breaking the computation into tasks that are of modest size while being large enough for good Level-3 BLAS performance on each core and scheduling these with the aid of a task manager. The directed acyclic graph (DAG) that represents the data dependencies between these tasks is an extension of the assembly tree. Both the independence of operations at nodes that do not have an ancestor/descendant relationship and the independence that is available between operations within a single node is exploited within this framework. Further, additional parallelization is available from the independence of updates of blocks of an ancestor node from the node's descendants. Being non-multifrontal avoids the need to hold generated-element matrices that are waiting for assembly at their parent nodes; this reduces the memory requirements.

Our algorithm is implemented within a new sparse Cholesky solver `HSL_MA87` that we have developed for inclusion within HSL. HSL is a Fortran library and so `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.

In recent years, there has been significant interest in the development of parallel sparse symmetric algorithms and solvers. To help put our work into context, we briefly summarise the key algorithmic features of five well-known parallel codes that offer facilities for sparse symmetric positive-definite systems and may be run on multicore machines.

MUMPS (MUltifrontal Massively Parallel Solver) [1] is a multifrontal Fortran/MPI package. While it is designed to solve symmetric and non-symmetric linear systems on distributed-memory machines, it can be run on shared-memory machines. It allocates the nodes of the assembly tree to threads during analyse. Large nodes other than the root of the tree are subdivided into block rows, which may be processed during the factorize phase by separate "slave" threads. The root node is processed by all the threads using ScaLAPACK [3]. Load and memory balancing are achieved by the dynamic scheduling of slave threads, taking account of the current load and memory demands on all the threads. In the solve phase, MUMPS uses the ScaLAPACK parallelism at the root and the parallelism implied by the assembly tree at other nodes.

PARDISO [25] is a non-multifrontal Fortran/C/OpenMP package for solving large sparse symmetric and non-symmetric linear systems of equations on shared-memory multiprocessors. It chooses a set of independent subtrees and begins by processing these in parallel, one for each thread. The subtrees are placed in a queue so that, when a thread has finished processing its subtree, it can request another. The remaining nodes are placed in a queue in bottom-up breadth-first order and each is processed by a thread as it becomes available. For nodes having a large number of eliminations, the corresponding columns are subdivided into sets of columns termed "panels", each of which is updated by a single thread. For each panel, all the processing is performed by a single thread, which means that good use is made of caching. The thread starts by applying all the updates that are then available and applies the others as they become available. Once the updates have been completed, the thread factorizes its diagonal block, then calculates its off-diagonal blocks of L, and finally records that these blocks are available for later updates within ancestor nodes.

PaStiX [15] is non-multifrontal C/Pthreads/MPI code that is primarily designed for positive-definite systems. It uses blocks defined by the variables of the nodes of the assembly tree, with large nodes subdivided. Zero rows within these blocks are held explicitly unless they are leading or trailing rows. Data distribution is as a block column (when the number of variables at the node is small) or as a block, each owned statically by a thread during factorize (when the number of variables at the node is not small). A block-column task involves everything for the block column, including calculating update matrices for later blocks. A block task is to factorize a diagonal block, calculate an off-diagonal block of L and send it to other block owners in the block column, or calculate the update matrices that involve the block. If a thread does several updates to the same block from different block columns, it accumulates them locally. The distribution is found during analyse by simulating all the costs. The sequence of tasks performed by each thread during factorize is fixed then. Recently, in [11], a dynamic scheduling designed for multicore and NUMA (Non-Uniform Memory Access) architectures was added.

TAUCS [18] is a C/Cilk library of sparse linear solvers. In particular, it offers a multifrontal sparse symmetric positive-definite solver. Cilk supports spawning of tasks as special procedure calls that can execute in parallel independently of the caller until synchronization in the caller. This is done recursively at the nodes of the assembly tree for each of the children, with synchronization before processing the frontal matrix at the node. Large nodes are partitioned recursively and the blocks are factorized recursively, which allows Cilk to manage parallelism dynamically both within and between the nodes. A recursive block data structure is used to limit cache movement.

WSMP [14, 13] is a Fortran/C/Pthreads multifrontal package. It is comprised of two parts: one for solving symmetric systems (both LL^T and LDL^T factorizations are offered) and one for general systems. On a shared-memory machine, it assigns all the threads to the root node and recursively assigns the threads of each parent node to its children to balance the load. At the nodes of the assembly tree that are assigned more than one thread, the dense operations on the frontal matrices are parallelized. The threads are managed through a task-parallel engine [19] that achieves fine-grain load balance via workstealing.

The outline of the remainder 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 describes the algorithm implemented within our DAG-based sparse Cholesky solver `HSL_MA87`, highlighting the features that distinguish it from the five codes that we have just described. Results of experiments with `HSL_MA87` and comparisons with a number of the above solvers on selected problems from practical applications are given in Sections 4 and 5. Finally, in Section 6, we make some concluding remarks and comment on the availability of `HSL_MA87`.

Our algorithm was developed independently of PaStiX, but employs similar concepts. Our contribution is to establish the algorithm as an extension to those used in the dense case, to develop a version designed primarily for shared rather than distributed memory and to remove a substantial constraint on the blocking used within the algorithm. We then implement the algorithm using a series of task scheduling techniques from the dense case to exploit cache reuse, all leading to substantial performance improvements on multicore machines.

2 Dense DAG solvers

Recent research by Buttari et al. [4, 5] into efficiently solving dense linear systems of equations on multicore processors has shown that significant parallel speedups may be obtained by subdividing the computation into block operations and performing these in parallel, subject only to their interdependencies.

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_block** factorizes a block on the diagonal, $A_{kk} = L_{kk}L_{kk}^T$, where L_{kk} is lower triangular.
- **solve_block** solves a triangular set of equations $L_{kk}^T L_{ik} = A_{ik}$ to obtain an off-diagonal block L_{ik} of the Cholesky factor.
- **update_block** updates a block of the remaining submatrix $A_{ij} \Leftarrow A_{ij} - L_{ik}L_{jk}^T$, $i \geq j > k$.

Traditionally, these tasks have been ordered in one of these two ways:

right-looking For each block column k in sequence, the factorize_block is performed, then the solve_blocks for the column are performed, and then all the update_blocks involving the blocks $L_{ik}, i > k$.

left-looking For each block column k in sequence, the update_blocks to the blocks A_{ij} are performed using the blocks L_{ij}, $i \leq j < k$, then the factorize_block for the block column is performed, then the solve_blocks for the block column are performed.

For example, ScaLAPACK uses a right-looking algorithm and relies on the parallelization of the updates for most of its speed-up.

The dependencies between the tasks are:

- **factorize_block** must wait for its block A_{kk} to be fully updated.
- **solve_block** must wait for its block A_{ik} to be fully updated and for the factorize_block $A_{kk} = L_{kk}L_{kk}^T$ to be completed.
- **update_block** must wait for solve_block to be complete for both its blocks L_{ik} and L_{jk}.

These dependencies may be represented by a directed acyclic graph (DAG), with a node for each task and an edge for each dependency. A task is ready for execution if and only if all tasks with incoming edges to it are completed. The first node corresponds to the factorization of the first diagonal block and the final node corresponds to the factorization of the final diagonal block. While tasks must be ordered in conformance with the DAG, there remains much freedom for exploitation of parallelism.

Hogg [16] recently implemented a dense Cholesky factorization that takes advantage of all this freedom. He uses 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. The choice of which available task to execute will clearly affect the overall execution time.

To guide this choice, Hogg used the time for a factorize_block (about $nb^3/3$ flops) as his unit of time, which means that solve_block (about nb^3 flops) takes 3 units and update_block (about $2nb^3$ flops for an off-diagonal block) takes 6 units. By solving some recurrence equations, he was able to show that with sufficient parallel threads, the least time needed is $9(n/nb - 1)$ units, and he constructed a corresponding schedule that initiates each task as late as possible. He noted that the assumption of sufficient parallel threads is valid in the important late stages of the factorization when there is insufficient parallelism to keep all threads busy. In Hogg's implementation with a given (probably small) number of threads, whenever a node has to be chosen from a set of candidates, one that is earliest in the schedule is chosen. If there are several such nodes, one that reuses data is preferred, which reduces the transfer of data between caches. Hogg compared his strategy with other strategies for choosing a task from those available and found that it was the most satisfactory, though the performance gains were modest.

Hogg's code is written in Fortran 95 with OpenMP and is available in HSL as `HSL_MP54`. He reports [16] near perfect speedups on an 8-core machine for sufficiently large problems.

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 executing the application or by unbalanced triggering of hardware interrupts.

3 DAG-based sparse direct solver

The sparse factorization work described in this paper was motivated by the aim of applying the ideas of the previous section to the sparse case. In particular, we aimed to work with tasks of a sufficient size for efficient execution on a single thread using Level-3 BLAS while seeking to take advantage of all the potential parallelism available between such tasks.

We chose a non-multifrontal implementation in order to avoid the memory overheads of the multifrontal method and to avoid update operations emanating from a node having to wait for its parent to be active (the TAUCS approach) or until memory for the parent frontal matrix is allocated.

3.1 Nodal matrix data structure

The columns of L associated with a node of the assembly tree consist of a trapezoidal matrix that has zero rows corresponding to variables that are eliminated later in the pivot sequence at nodes that are not ancestors. We have chosen to compress this matrix in the traditional manner (see [9], Section 10.5) 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*.

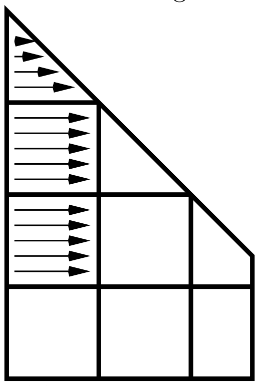
As in the dense case described in the previous section, we subdivide this matrix into blocks under the control of a parameter nb. This is illustrated in Figure 3.1(a). We divide the computation into tasks in which a single block is revised (details in Section 3.2). If the number of columns nc in the nodal matrix is small, this may yield tasks that are too small to justify their associated overheads. Therefore, if nc is less than nb, we base the block size on the value nb^2/nc, rounding up to a multiple of 8 to avoid overlaps between cache lines. PaStiX also treats nodal matrices with few columns differently; it treats such a matrix as a single entity. We discuss the (small) effect of our approach on the factorize time in Section 4.5.

We store the nodal matrix using the row hybrid blocked structure of Anderson et al. [2] with the modification that "full" storage is used for the blocks on the diagonal rather than storing only the actual entries. This is illustrated in Figure 3.1(b). Using the row hybrid scheme rather than the column hybrid scheme facilitates updates between nodes by removing any discontinuities at row block boundaries (we explain the importance of this in Section 3.2). Storing the blocks on the diagonal in full storage allows us to exploit efficient BLAS and LAPACK routines. This structure is illustrated in Figure 3.1(b). Note that the final block on the diagonal is often trapezoidal, to allow the other blocks of its block row to be square.

3.2 Tasks

Following the design of our dense DAG-based code (Section 2), we split the work involved in the sparse factorization of A into the following tasks (illustrated graphically in Figures 3.2 to 3.4):

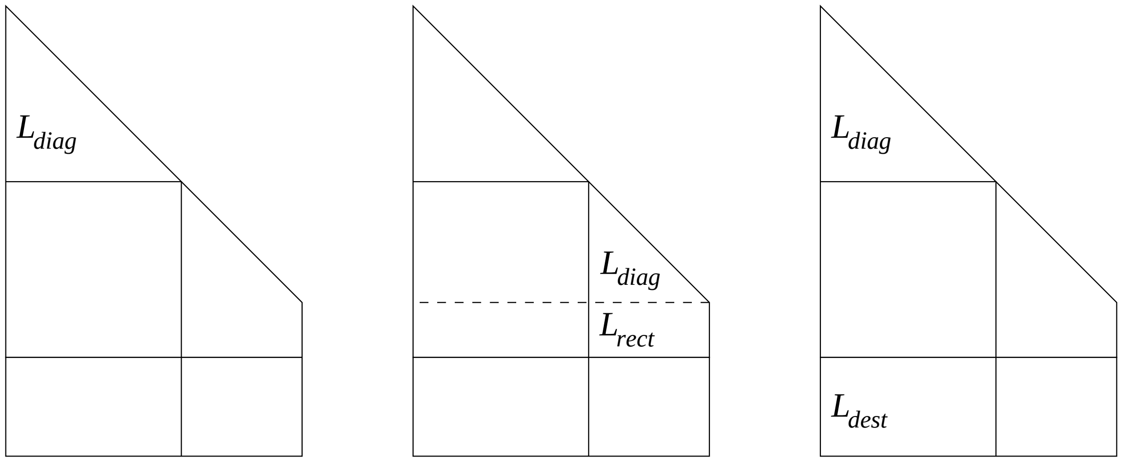
Figure 3.1: Row hybrid block structure for a nodal matrix.


(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

Figure 3.2: factorize_block(`diag`) and solve_block(L_{dest})



(a) factorize_block(`diag`)

(b) factorize_block(`diag`) $L_{rect} \Leftarrow L_{rect}L_{diag}^{-T}$

(c) solve_block(L_{dest}) $L_{dest} \Leftarrow L_{dest}L_{diag}^{-T}$

factorize_block(L_{diag}) computes the Cholesky factor L_{diag} of the triangular part of a block `diag` 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-3 subroutine `_trsm`, see Figures 3.2(a) and 3.2(b).

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

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

using the BLAS-3 subroutine `_trsm`, see Figure 3.2(c).

update_internal(L_{dest}, `scol`) Within a nodal matrix, performs the update of the block L_{dest} from the block column `scol`

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

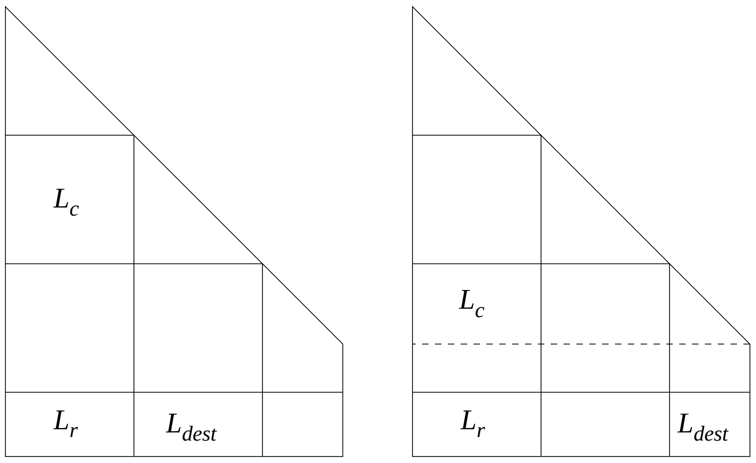
where L_r is a block of the block column `scol` and L_c is a submatrix of this block column. If the L_{dest} is not in the final block column of the node, L_c is block of `scol`, see Figure 3.3(a); otherwise, L_c is the submatrix that corresponds to the columns of L_{dest}, see Figure 3.3(b). If L_{dest} is an off-diagonal block, we use the BLAS-3 subroutine `_gemm` for this. If L_{dest} is on the diagonal, we use the BLAS-3 subroutine `_syrk` for the triangular part and `_gemm` for the rectangular part, if any.

update_between(L_{dest}, `snode`, `scol`) performs the update of the block L_{dest} from the block column `scol` of a descendant node `snode`

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

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. Unless the number of entries updated is very small, we exploit the BLAS-3 subroutine `_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 L_{dest}, see Figure 3.4.

Figure 3.3: update_internal(L_{dest}, `scol`), $L_{dest} \Leftarrow L_{dest} - L_r L_c^T$



(a) L_{dest} not in last block (b) L_{dest} in last block

Note that the submatrices L_r and L_c in Figure 3.4 are determined by the block L_{dest} and are probably not blocks of the block column `scol`. Storing the blocks by rows and continuously (see Figure 3.1(a)) allows us to access the submatrices L_r and L_c needed for update_between as whole arrays.

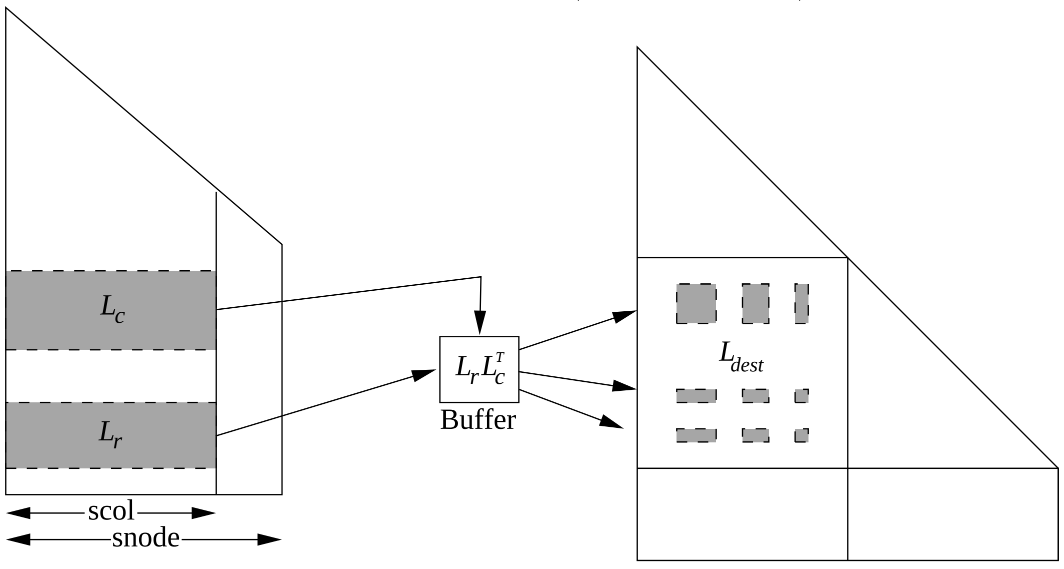
We could have cast update_between as an operation from a pair of blocks, but this would often cause the same destination block to be updated more than once from the same block column. This is undesirable since contested writes cause more cache misses than contested reads (a write may invalidate a cache 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.

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 becoming available. At a moment during factorize, 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. We explain how this is managed in Section 3.3. Initially, the task pool is given a factorize_block task from each leaf node of the assembly tree.

In practice, it seems that the update_between tasks are by far the most demanding of computer resources. In Figure 3.5, we show the percentages of flops needed for the different kinds of tasks for the test problems of Section 4.

We now explain how we determine when a task is ready without needing to represent the whole DAG explicitly. During analyse, we calculate a count for each block of L. If the block is on the diagonal, the

Figure 3.4: update_between(L_{dest}, `snode`, `scol`)



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

count is the number of updates, update_internal or update_between, that will be applied to it. If the block is not on the diagonal, the count is one more than the number of updates that will be applied to it. During factorize, 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 complete, which means that its solve_block task is ready and may be stored.

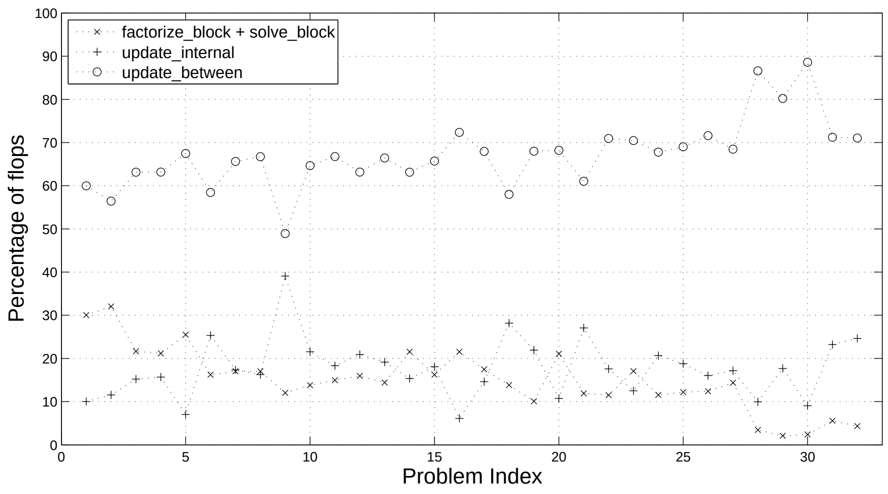
When a factorize_block or solve_block task completes, we decrement its count to flag this event with a negative value. A column lock is set and we store each update task that depends on the completion of this task and does not depend on a task that has not yet completed. Once this has been done, the 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.

3.3 Task dispatch engine

In the dense case (Section 2), we found that complex prioritization schemes that take optimal schedules into account offer very limited benefit. In `HSL_MA87`, we therefore use a simpler prioritization scheme that favours cache awareness by using local task stacks and a single global task pool. For each shared cache, there is a small local stack holding tasks that are intended for use by the threads sharing this cache. During the factorization, each thread adds or draws tasks from the top of its local stack. It is a stack rather than a more complicated data structure because this gives all the properties that are needed. Each stack has a lock to control access by its threads.

When a thread completes the last update_block for a diagonal block, it executes the factorize_block task for this block at once without putting it on its stack. This promotes both cache reuse and the generation of further tasks. When a thread completes a solve_block task, it first places any update_between tasks generated on its stack, then places any update_internal tasks generated. Both will be above any stacked solve_block tasks. Since some of the data needed for an update is likely to be in the local cache, cache reuse is encouraged naturally without the need for explicit management.

Figure 3.5: Percentage of flops in the different kinds of tasks.



If a local stack becomes full, a global lock is acquired and the bottom half (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. We give the tasks in the task pool different priorities, in the descending order factorize_block, solve_block, update_internal, update_between, but our experience has been that this does not significantly improve the execution time over using a single stack for all the tasks in the task pool. To understand why this should be the case, note that, as already observed, a factorize_block task is executed as soon as it is generated and so never reaches the pool. When a factorize_block task completes, several solve_block tasks may be placed on the stack, one of which is probably executed immediately. When an update task completes for an off-diagonal block, a single solve_block task is placed on the stack and probably executed immediately. Few solve_block tasks therefore reach the pool. It follows that prioritization mainly affects the update tasks and update_internal tasks will have already been favoured by the order in which they are added to the local stacks.

When establishing update_between tasks, it is convenient to search the path from the node to the root through links to parents. This leads to the tasks being placed on the stack with those nearest the root uppermost, so that these will be executed first. This is the opposite to the order that will lead to early availability of further tasks. We tried reversing this order and indeed found that the number of available tasks increased. This led to a larger task pool being needed and to an increase in execution time. It seems that as long as there are sufficient tasks available to keep the threads active, this is sufficient. On the basis of numerical experimentation, we have set the default value for the initial size of the task pool to 25,000 (the size of the pool is increased whenever necessary during execution). For many of the test problems of Section 4, the task pool size did not exceed 10,000; the largest was approximately 22,000.

If a 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.

To check the effect of having a stack for each cache, we tried two tests while running the problems of Section 4. First, we ran with a stack for each thread. This led to a small loss of performance (around 1% to 2% for the larger problems). Second, we disabled processor affinity, which means that the threads are not required to share caches in the way the code expects. This led to a similar loss of performance.

We also tried the effect of using the global lock to control the local stacks as well as the global pool. We found that this had no noticeable effect on execution time on our test machine.

3.4 Improving cache locality in update_between

It is desirable for the efficiency of an update_between task that the rows of L_r and L_c correspond to rows and columns of L_{dest} that are in the same order and that many of these rows and columns of L_{dest} are contiguous. This reduces cache misses and assists hardware prefetching. To ensure that they are in the same order, we permute the index lists at each node to pivot order.

The elimination order is usually determined during analyse by a depth-first search of the assembly tree, and we follow this practice. Let $n_1, n_2, \ldots, n_k$ be a path of the assembly tree in which each node other than n_k is the first child of its successor. The index list at n_1 may be written as $v_1, v_2, \ldots, v_k$ where v_i indexes those of its variables that are eliminated at n_i, $i = 1, 2, \ldots, k$ (some of these lists v_i may be empty). The corresponding part of the pivot sequence is $v_1, w_1, s_1, v_2, w_2, s_2, \ldots$ where w_i lists the other variables eliminated at node n_i (w_1 is empty) and s_i lists the variables eliminated at siblings of n_i and their descendants. Therefore, the variables indexed in each of the sets $v_1, v_2, \ldots$ are contiguous in the pivot sequence, which is desirable for updates between node n_1 and each of the nodes $n_2, \ldots, n_k$. It is not all we would like; for example, updating the off-diagonal part of the matrix at n_2 involves the rows indexed by the lists $v_2, v_3, \ldots$ and in general there is a gap in the pivot sequence between each of these lists and the next. However, we can apply the same argument to a subsequence $n_i, n_{i+1}, \ldots, n_k$, and the lists are likely to get progressively larger as we approach the root.

This property does not extend to a node n_1 that is not a first child because some of the indexes v_2 may also index variables of its first-child sibling and be scattered among other indices for the first-child sibling. For example, if the variables indexed 7,8,9 are eliminated at the parent and the index list of the first child is $(7, 8, 9, \ldots)$, the index list of the second child might be $(7, 9, \ldots)$.

To exploit this property, it is desirable for the length of the list of each first child, excluding its eliminated variables, to be long. At every node, we have therefore made the child with the greatest such length be first. The effect of this strategy is reported on in Section 4.2.

4 Numerical results

In this section, we present numerical results for our new sparse Cholesky code `HSL_MA87`. In common with other solvers, `HSL_MA87` offers a number of options through the use of control parameters. These include parameters that control node amalgamation (Section 4.3) and the size of the blocks (Section 4.4). We present numerical experiments that support our design choices for `HSL_MA87` as well as the default settings for the control parameters.

4.1 Test environment

The experiments we report on throughout this section were performed on our multicore test machine `fox`, details of which are given in Table 4.1. Note that the sharing of level-2 caches and memory buses, makes speed-up near 2 on 2 cores much easier to obtain than speed-up near 4 on 4 cores, which in turn is much easier to obtain than speed-up near 8 on 8 cores.

The sparse test matrices used in our experiments are listed in Table 4.2. This set comprises 32 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]. In our tests, we use the nested dissection ordering that is computed by `METIS_NodeND` [20, 21]. In Table 4.2, we include the number of millions of entries in the matrix factor (denoted by $nz(L)$) and the number of billions of floating-point operations (Gflops) when this pivot sequence is used by `HSL_MA87` without node amalgamation (see Section 4.3).

Unless stated otherwise, runs were performed using all 8 cores on our test machine `fox` and all control parameters used by `HSL_MA87` were given their default settings. All times are elapsed times for the factorization phase, in seconds, measured using the system clock. Unfortunately, we found that when the elapsed time on 8 cores was less than a second, it could vary by 20% to 30% between runs. Occasionally,

Table 4.1: Specifications of our 8 core test machine `fox`.

	2-way quad Harpertown (`fox`)
Architecture	Intel(R) Xeon(R) CPU E5420
Operating system	Red Hat 5
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
Level-1 cache	32 K on each core
Level-2 cache	6 M for each pair of cores
Memory	32 GB for all cores
BLAS	Intel MKL 10.1
Compiler	Intel 11.0 with option -fast

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

a time would be greater by much more than this[1]. In each experiment, we therefore averaged over ten complete runs of each of the problems except for the slowest five, each of which requires more that 500 Gflops to factorize A and always took longer than 10 seconds. We will refer to these five problems as the *slow subset.* For these cases, we averaged the times over two runs.

4.2 Effect of reordering the children

In Section 3.4, we explained our strategy for choosing, at each non-leaf node of the assembly tree, which child node to order as the first child. Note that this does not change the flop count or the number of entries in L. The effect of our strategy on the factorize time is illustrated in Table 4.3, which includes results for the slow subset together with three other problems from our test set, chosen to show the range of behaviours found. Though reasonably modest, of particular note is that the performance gains in parallel are often much more than merely an eighth of the serial gains, leading to better speedups. Throughout the remainder of the paper, all results are obtained using the child reordering strategy.

4.3 Effect of node amalgamation

Node amalgamation (see Section 4 of [10]) has become well established as a means of improving factorization speed at the expense of the number of entries in L and the operation counts during factorize and solve. The original strategy relied on a postorder generated by a depth-first search of the tree. A parent and child that were adjacent in this order were merged if both involved fewer than *nemin* eliminations. A more powerful version is used in [24], involving the recursive merge of any child/parent pair if both involve fewer than *nemin* eliminations. We have chosen to use this more powerful version.

In a new environment, a new exploration is needed for the most suitable value of the parameter *nemin*. This is the purpose of this subsection. We expected the best value to be in the range 8 to 64 so ran with the values 8, 16, 32, and 64. To illustrate the value of amalgamation, we also run with the value 1.

In Table 4.4, we show the factorize times for the slow subset and for three others that represent the three kinds of behaviour that we saw: flat (problem 3), $nemin = 1$ much slower (problem 16), and U-shaped (problem 22). The two biggest problems showed flat behaviour, but $nemin = 64$ was best for problems 28 to 30.

[1]We believe that the occasional very slow runs were caused by an executing thread being asked to perform a system task when there were few tasks waiting to be executed.

Table 4.2: Test matrices factorized without node amalgamation. $nz(A)$ is the number of entries in the lower triangular part of A; $nz(L)$ is the number of entries in L.

Identifier	n (10^3)	$nz(A)$ (10^6)	$nz(L)$ (10^6)	Flops (10^9)	Application/description
1. `CEMW/tmt_sym`	726.7	2.9	30.0	9.38	Electromagnetics
2. `Schmid/thermal2`	1228.0	4.9	51.6	14.6	Unstructured thermal FEM
3. `Rothberg/gearbox`*	153.7	4.6	37.1	20.6	Aircraft flap actuator
4. `DNVS/m_t1`	97.6	4.9	34.2	21.9	Tubular joint
5. `Boeing/pwtk`	217.9	5.9	48.6	22.4	Pressurised wind tunnel
6. `Chen/pkustk13`*	94.9	3.4	30.4	25.9	Machine element, 21 noded solid
7. `GHS_psdef/crankseg_1`	52.8	5.3	33.4	32.3	Linear static analysis
8. `Rothberg/cfd2`	123.4	1.6	38.3	32.7	CFD pressure matrix
9. `DNVS/thread`	29.7	2.2	24.1	34.9	Threaded connector/contact
10. `DNVS/shipsec8`	114.9	3.4	35.9	38.1	Ship section
11. `DNVS/shipsec1`	140.9	4.0	39.4	38.1	Ship section
12. `GHS_psdef/crankseg_2`	63.8	7.1	43.8	46.7	Linear static analysis
13. `DNVS/fcondp2`*	201.8	5.7	52.0	48.2	Oil production platform
14. `Schenk_AFE/af_shell3`	504.9	9.0	93.6	52.2	Sheet metal forming matrix
15. `DNVS/troll`*	213.5	6.1	64.2	55.9	Structural analysis
16. `AMD/G3_circuit`	1585.5	4.6	97.8	57.0	Circuit simulation
17. `GHS_psdef/bmwcra_1`	148.8	5.4	69.8	60.8	Automotive crankshaft model
18. `DNVS/halfb`*	224.6	6.3	65.9	70.4	Half-breadth barge
19. `Um/2cubes_sphere`	101.5	0.9	45.0	74.9	Electromagnetics
20. `GHS_psdef/ldoor`	952.2	23.7	144.6	78.3	Large door
21. `DNVS/ship_003`	121.7	4.1	60.2	81.0	Ship structure—production
22. `DNVS/fullb`*	199.2	6.0	74.5	100	Full-breadth barge
23. `GHS_psdef/inline_1`	503.7	18.7	172.9	144	Inline skater
24. `Chen/pkustk14`*	151.9	7.5	106.8	146	Civil engineering. Tall building
25. `GHS_psdef/apache2`	715.2	2.8	134.7	174	3D structural problem
26. `Koutsovasilis/F1`	343.8	13.6	173.7	219	AUDI engine crankshaft
27. `Oberwolfach/boneS10`	914.9	28.2	277.8	282	Bone micro-finite element model
28. `ND/nd12k`	36.0	7.1	116.5	505	3D mesh problem
29. `JGD_Trefethen/Trefethen_20000`	20.0	0.3	90.7	652	Integer matrix
30. `ND/nd24k`	72.0	14.4	320.6	2054	3D mesh problem
31. `bone010`	986.7	36.3	1076.4	3876	Bone micro-finite element model
32. `audikw_1`	943.7	39.3	1242.3	5804	Automotive crankshaft model

* indicates only the sparsity pattern is provided.

Table 4.3: The effect of reordering the children on the `HSL_MA87` factorize times on 1 and 8 cores. Default values are used for all parameters.

	Without reordering			With reordering		
	1	8	speedup	1	8	speedup
9	5.14	1.01	5.08	5.09	0.93	5.50
16	14.5	2.73	5.30	14.1	2.54	5.55
18	11.5	1.93	5.95	11.4	1.89	6.02
28	83.3	13.1	6.36	80.0	12.3	6.49
29	120.	20.1	5.99	120.	19.8	6.03
30	333.	51.8	6.44	317.	48.0	6.62
31	519.	73.1	7.10	507.	70.8	7.17
32	788.	113.	7.06	760.	106.	7.18

In Table 4.4, we also show the number of entries in L and the solve times. The examples illustrate the kinds of behaviour that we saw for the solve times: flat (problems 28, 29), rising slowly (problems 31, 32), rising less slowly (problems 3, 16, 22), and $nemin = 1$ much slower (problem 16).

These considerations led us to choose 32 as our default *nemin* value. However, if the number of entries in L increases slowly with *nemin*, it can be advantageous to use a larger value. This is the case, for instance, with problem 30. However, if a large number of solves is to follow the factorization, we would recommend using a smaller value of *nemin* (for example, $nemin = 8$).

Table 4.4: Comparisons of times on 8 cores for *nemin* in the range 1 to 64. Default values for other parameters. The factorize times within 3% of the fastest are in bold.

	$nz(L)$ (millions)				
	1	8	16	32	64
3	37	39	41	44	49
16	98	119	139	172	228
22	74	76	79	86	101
28	117	117	118	119	121
29	91	92	94	96	99
30	321	322	323	326	331
31	1076	1090	1108	1135	1183
32	1242	1257	1275	1303	1359

	Factorize times					Solve times				
	1	8	16	32	64	1	8	16	32	64
3	0.91	**0.83**	0.86	**0.83**	0.89	0.27	0.27	0.29	0.31	0.34
16	5.00	2.77	**2.59**	**2.61**	3.04	1.52	1.09	1.24	1.49	1.80
22	2.70	**2.57**	**2.59**	**2.64**	2.89	0.50	0.51	0.53	0.58	0.66
28	22.5	15.3	13.7	12.3	**11.1**	0.75	0.74	0.74	0.76	0.74
29	119.9	40.8	27.5	19.9	**15.7**	0.62	0.58	0.59	0.65	0.61
30	80.4	58.4	53.1	48.0	**44.1**	2.05	2.02	2.03	2.08	2.03
31	**72.0**	**71.0**	**70.7**	**70.8**	**71.4**	6.82	6.86	7.00	7.19	7.39
32	**109.**	**107.**	**106.**	**106.**	**106.**	7.82	7.85	8.01	8.23	8.43

4.4 Block size

The block size *nb* was discussed in Section 3.1. In Table 4.5, we report the factorize time for a range of block sizes on a single core and on 8 cores. As in Table 4.4, we show results for the slow subset and three cases representing the behaviour of the others for both factorize and solve times. The fastest times and those within 3% of the fastest are again shown in bold. On a single core, there was never much difference in the times for *nb* in the range 256 to 448, but the times for smaller *nb* were usually greater by more than 3%. On eight cores, 256 almost always gave times within 3% of the best. These considerations led us to use 256 as our default value.

4.5 Rectangular blocks

We explained in Section 3.1 the merits of using rectangular blocks when the number of columns *nc* in the nodal matrix is less than *nb*. We use the block row size nb^2/nc, rounded up to a multiple of 8. To assess the efficacy of this, we ran our tests with a fixed block size of 256. The most significant change was that the factorize time for problem 29 on 8 cores increased from 19.8 (adaptive size) to 25.2 (fixed size). For one problem, the time reduced by about 7% with a fixed size while for another it increased by about 8%. Otherwise, the changes were not greater than 4% and in both directions. Throughout the remainder of this paper (and within HSL_MA87) rectangular blocks are used.

Table 4.5: Comparison of the factorize times for different block sizes *nb*. Default values for other parameters. The factorize times within 3% of the fastest are in bold.

	Single core factorize times						8-core factorize times				
	128	192	256	320	384	448	128	192	256	320	384
3	4.30	**4.13**	**4.09**	**4.08**	**4.06**	**4.06**	**0.77**	0.80	0.82	0.84	0.94
15	10.07	9.58	**9.36**	**9.33**	**9.30**	**9.25**	**1.67**	**1.63**	**1.63**	**1.67**	1.77
22	17.5	16.3	**16.1**	**15.9**	**15.8**	**15.8**	2.79	**2.60**	**2.61**	2.69	2.69
28	90.9	82.6	**80.0**	**79.0**	**79.2**	**80.1**	14.5	**12.6**	**12.3**	**12.4**	13.1
29	148.	125.	120.	**117.**	**117.**	**116.**	23.8	20.5	**19.8**	**19.7**	20.6
30	394.	331.	**318.**	**313.**	**312.**	**316.**	63.8	50.3	**48.0**	**48.1**	50.1
31	580.	528.	508.	**499.**	**492.**	**488.**	82.9	73.4	**70.9**	**69.5**	**69.6**
32	884.	794.	761.	**747.**	**735.**	**729.**	128.	110.	**106.**	**104.**	**103.**

4.6 Local task stack size

In Section 3.3, we discussed the use of local task stacks. We have performed experiments without local task stacks and with local task stacks of size in the range 10 to 300. Except for the slow subset, we found improvements in the factorize time in the approximate range 10% to 20% over running without local stacks. In Table 4.6, we report the results for three representative problems that gave gains at the ends and middle of the range 10% to 20%. For the slow subset, there was a gain but of less than 10%. These experiments led us to use 100 as the default size.

Table 4.6: The factorize times on 8 cores for different local task stack sizes. Default values for other parameters. The times within 3% of the fastest are in bold.

	No local	Stack size				
	stack	10	50	100	200	300
3	0.92	0.86	0.86	**0.82**	**0.84**	0.85
15	1.80	1.74	**1.63**	**1.63**	**1.59**	**1.61**
16	3.14	2.80	**2.60**	**2.58**	**2.58**	**2.61**
28	12.8	12.7	**12.4**	**12.3**	**12.2**	**12.1**
29	20.3	20.3	**20.0**	**19.8**	**19.7**	**19.5**
30	49.9	49.5	48.7	**48.0**	**47.4**	**47.2**
31	74.0	73.6	**71.3**	**70.9**	**70.7**	**70.8**
32	110.	110.	**107.**	**106.**	**106.**	**106.**

For three of the slow problems, we show in Table 4.7 the number of leaf nodes (initially a factorize_block task is put into the global task pool for each leaf), the total number of tasks, the number of tasks taken directly from the local stacks, the number of tasks sent to the global task pool because a local stack became full, and the number of tasks moved to the global task pool by workstealing. We see that the number of tasks moved by workstealing is small. Provided the stack size is at least 100, a good proportion of the tasks are executed directly.

For the smaller problems, the local stacks became full for only eight cases when the stack size was 100. This happened most often for problem 19, shown in Table 4.7. Next was problem 26, where 750 tasks were moved to the pool because of a full stack. For a further 6 problems, fewer than 400 tasks were moved to the pool because of a full stack. For the remaining smaller problems, none of the local stacks became full. With a stack size of 200, a local stack became full only for problem 22 and only 100 tasks were moved to the task pool because of this.

Although the workstealing figures in Table 4.7 are small, workstealing is important for load balance

Table 4.7: Tasks taken directly from local stacks, moved to pool because of a full stack, and moved to pool because of workstealing. 8 cores with different local stack sizes. Default values for other parameters.

	Leaves (10^3)	Tasks (10^3)	Stack size	Direct (10^3)	Full (10^3)	Workstealing (10^3)
19	0.88	48	10	19	28	0.14
			50	40	7	0.30
			100	45	1	0.43
			200	46	0	0.54
			300	46	0	0.47
28	0.06	124	10	19	105	0.16
			50	45	78	0.47
			100	66	56	0.81
			200	92	30	1.61
			300	108	13	1.83
29	0.06	241	10	21	221	0.13
			50	46	195	0.39
			100	63	177	1.25
			200	97	144	0.89
			300	119	121	1.76
32	5.58	772	10	232	534	0.16
			50	573	192	0.92
			100	703	60	2.84
			200	751	11	3.57
			300	759	4	3.95

and its importance increases with the local stack size. To illustrate this, in Table 4.8 we present times on 8 cores with and without workstealing using local stack sizes of 10 and 100.

4.7 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 4.1, 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 at least 3 for all but four of the smallest test problems, and for the largest problems (in terms of flops) it exceeds 3.6. On 8 cores, HSL_MA87 achieves speedups of more than 6 for many of the larger problems, reaching nearly 7.2 for the largest. For all but the three smallest problems, the speedup exceeds 5. It is very encouraging to note that, as the problem size increases, so too does the speedup achieved.

Of course, our primary concern is the actual speed. We show the speeds in Gflop/s on 8 cores in Figure 4.2. Here we compute the flop count from a run with the node amalgamation parameter *nemin* having the value 1 (that is, the flop count reported in Table 4.2). We note that for 14 of our 30 test problems, the speed exceeds 36.4 Gflop/s, which is half the maximum `dgemm` speed (see Table 4.1). Furthermore, for only the two smallest problems is the speed significantly less than 24.3 Gflop/s, which is a third of the `dgemm` maximum.

5 Comparisons with other solvers

We now compare the performance of HSL_MA87 with some of the recent solvers outlined in Section 1. The solvers are listed in Table 5.1. Unless stated otherwise, all control parameters for each of the solvers are set to their default settings and, where offered, the positive definite option is selected (so that none of the

Table 4.8: Factorize times with and without workstealing on 8 cores. Default values for other parameters. The times within 3% of the fastest are in bold.

	Stack size 10		Stack size 100	
	with	without	with	without
2	1.24	1.30	**1.23**	1.28
17	1.86	1.96	**1.71**	2.05
27	7.54	7.67	**7.10**	11.08
28	12.7	12.8	**12.3**	16.3
29	**20.3**	**20.2**	**19.8**	21.9
30	49.5	49.8	**48.0**	49.7
31	73.6	73.7	**70.8**	85.9
32	109.7	110.0	**105.8**	139.3

Figure 4.1: The ratios of HSL_MA87 factorize times on 2, 4 and 8 cores to its factorize time on a single core.

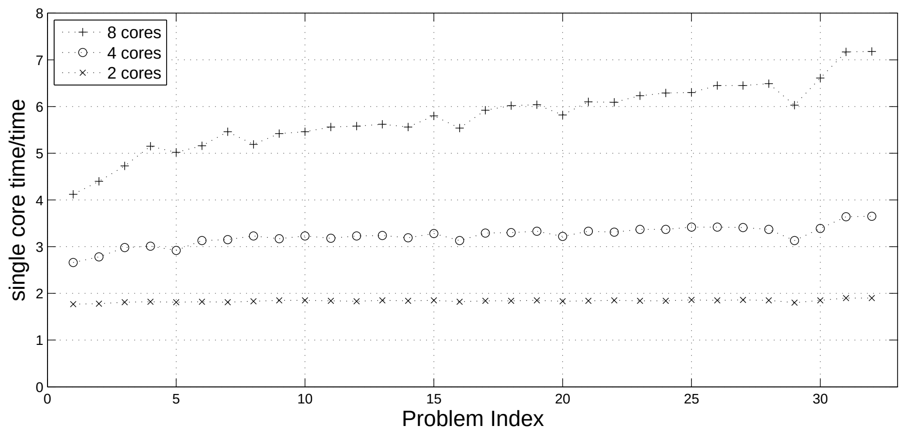
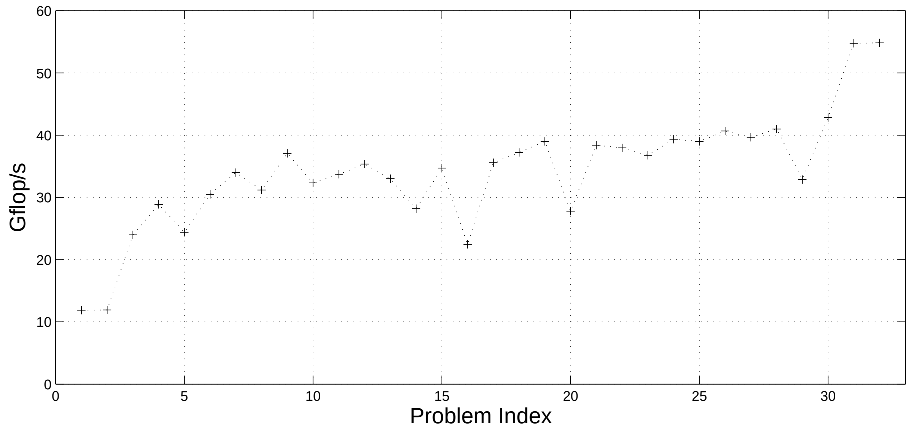


Figure 4.2: The speed of HSL_MA87 factorize in Gflop/s on 8 cores.



codes performs any numerical pivoting). We provide HSL_MA87 and PARDISO with the ordering generated by METIS_NodeND [20]. TAUCS has no option to input an ordering, but it calls METIS internally. This may result in a slightly different ordering because of tie breaking. We tried supplying PaStiX with the same ordering as HSL_MA87 and PARDISO, but found this severely impaired its performance. Instead, for PaStiX, we used its internal call to the SCOTCH nested dissection routine [22, 23]. TAUCS was run using its multifrontal option as this is recommended in the user documentation.

We note that, because of various compilation issues, we were unable to use a single compiler for all our tests. However, our experience of running HSL_MA87 with both the Intel and gcc compilers was that this made very little difference to the factorize times since most of the time was spent in the BLAS or other small kernels that are equally well optimized by the versions of both compilers used.

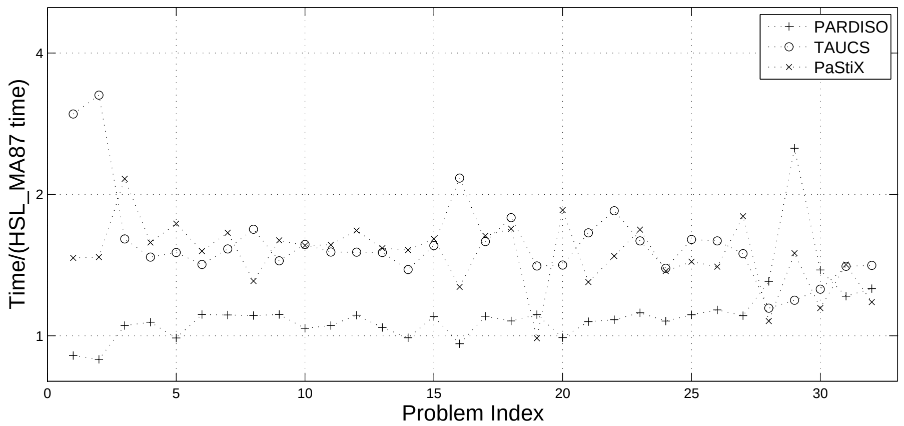
Table 5.1: Sparse direct solvers used in our numerical experiments.

Code	Date/version/compiler	Authors/website
PARDISO	10.2009/ v4.0.0 ifort 10.1 -fast	O. Schenk and K. Gärtner `http://www.pardiso-project.org/`
PaStiX	04.2009/ v5.1.2 icc 10.1/ifort 11.0 -fast	P. Henon, P. Ramet and J. Roman `http://pastix.gforge.inria.fr/files/README-txt.html`
TAUCS	09.2003/ v2.2 cilk 5.4.6/ gcc 4.1.2 -O3	S. Toledo `http://www.cs.tau.ac.il/∼stoledo/taucs`

5.1 Comparisons on one core of fox

Although the solvers are primarily designed to run in parallel, it is of interest to first compare their performances on a single core. In Figure 5.1, the ratios of the factorize times for PARDISO, PaStiX and TAUCS to the factorize time for HSL_MA87 on fox (Table 4.1) are plotted for each of our 32 test problems. We see that HSL_MA87 is generally faster than both PaStiX and TAUCS. On the lowest numbered test problems, PARDISO sometimes outperforms HSL_MA87 but, on the largest problems, HSL_MA87 is consistently faster.

Figure 5.1: The ratios of the PARDISO, TAUCS and PaStiX factorize times to the HSL_MA87 factorize time (single core of fox).



5.2 Comparisons on eight cores of fox

In Figure 5.2, the ratios of the factorize times for PARDISO, PaStiX and TAUCS to the factorize time for HSL_MA87 on 8 cores of fox are given. In Figure 5.3, the speedup ratios are presented. We see that, on 8 cores, HSL_MA87 almost always outperforms the other solvers (the only exceptions being a number of the small problems for which PARDISO is the fastest solver). For the largest problems (the last 5 problems), the performance of TAUCS comes closest to that of HSL_MA87, while the the difference between the performance of PARDISO and that of HSL_MA87 (in terms of both time and speed up) becomes more significant as the problem size increases.

Figure 5.2: The ratios of the PARDISO, TAUCS and PaStiX factorize times to the HSL_MA87 factorize time (8 cores of fox).

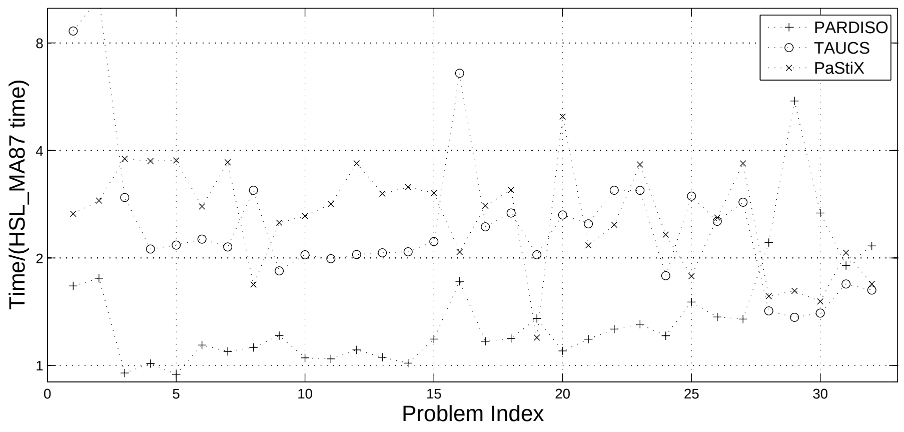
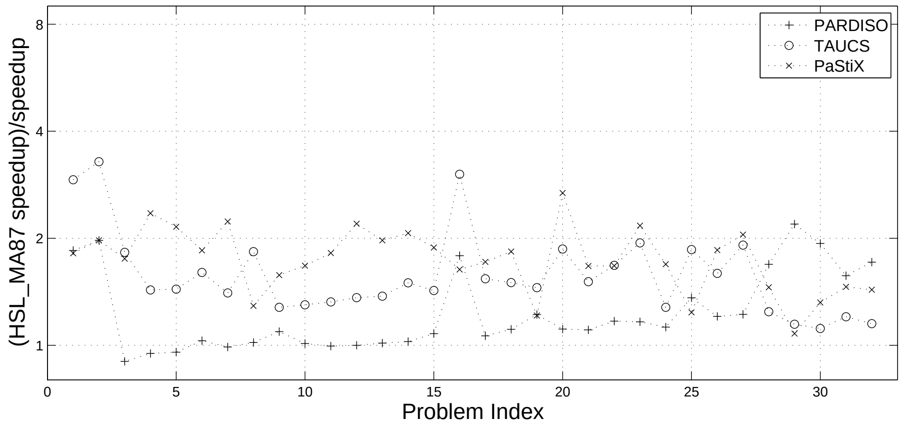


Figure 5.3: The ratios of HSL_MA87 factorize speed-up to those of PARDISO, TAUCS and PaStiX (8 cores of fox)



5.3 Comparisons on other processors

So far, the results have all been for our test machine fox (see Table 4.1). We end this section by presenting runs on two further multicore machines, brief details of which are given in Table 5.2. Since the results we have already reported indicate that, of the solvers tested, PARDISO most closely rivals HSL_MA87, we restrict our runs to these two solvers (MKL 11.0 version of PARDISO is used on both machines). The ratios of the factorize times on a single core and 8 cores are given in Figures 5.4 and 5.5. We observe that on both machines the performance of `HSL_MA87` compares favourably with that of PARDISO and, in particular, on the largest problems running on 8 cores, `HSL_MA87` is significantly faster than PARDISO.

Table 5.2: Specifications of two further test machines .

	Intel Nehalem	AMD Shanghai
Architecture	Intel(R) Xeon(R) CPU E5540	AMD Opteron 2376
Clock	2.53 GHz	2.30 GHz
Cores	2×4	2×4
Level-1 cache	128 K on each core	128 K on each core
Level-2 cache	128 K on each core	512 K on each core
Level-3 cache	8192 K shared by 4 cores	6144 K shared by 4 cores
Memory	24 GB for all cores	16 GB for all cores
BLAS	Intel MKL 11.0	Intel MKL 11.0
Compiler	Intel 11.0 with option -fast	Intel 11.0 with options -ipo -O3 -no-prec-div -static -msse3

Figure 5.4: Comparison of `HSL_MA87` and PARDISO factorize times on the Intel Nehalem architecture.

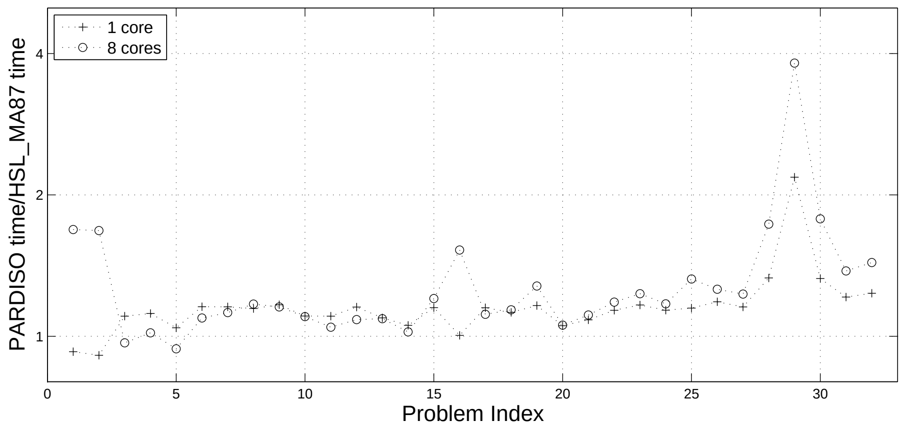
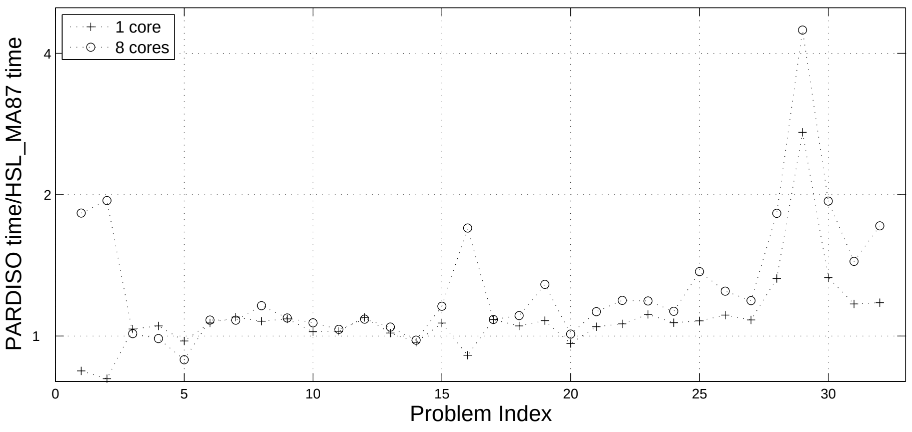


Figure 5.5: Comparison of `HSL_MA87` and PARDISO factorize times on the AMD Shanghai architecture.



6 Concluding remarks

The rapid emergence of multicore architectures demands the design and development of algorithms that are able to effectively exploit the new architectures. In particular, the efficient solution of sparse linear systems on multicore architectures is a challenging and important problem. In this paper, we have reported on the development of a task DAG-based algorithm that we have implemented in a new sparse direct solver, `HSL_MA87`, for solving large-scale symmetric positive-definite linear systems on multicore machines. We have described the main components of the algorithm and have used numerical experiments to support the reasons behind our algorithm choices. In addition, we have presented numerical comparisons with other state-of-the-art sparse direct solvers. These show that our code is performing well and, in particular, it outperforms existing codes on the largest problems to which we had access for testing.

Our DAG-based sparse Cholesky solver `HSL_MA87` has been developed for inclusion in the mathematical software library HSL. Versions exist for A real symmetric and positive definite and A complex Hermitian and positive definite. 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 normally involve a fee. Details of the packages and how to obtain a licence plus conditions of use are available at `http://www.cse.scitech.ac.uk/nag/hsl/`.

7 Acknowledgement

We would like to thank Iain Duff for carefully reading a draft of this paper and making helpful suggestions for improving the presentation. We also thank three anonymous referees who reviewed an earlier version of this paper. Their constructive criticism led to improvements both in `HSL_MA87` and this paper.

References

[1] 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.

[2] 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.

[3] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[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] I. Duff, A. Erisman, and J. Reid, *Direct Methods for Sparse Matrices*, 1986.

[10] I. Duff and J. Reid, *The multifrontal solution of indefinite sparse symmetric linear systems*, ACM Transactions on Mathematical Software, 9 (1983), pp. 302–325.

[11] M. Faverge and P. Ramet, *A NUMA aware scheduler for a parallel sparse direct solver*, Parallel Computing, (2009). Submitted.

[12] 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.

[13] A. Gupta, *WSMP: Watson sparse matrix package (Part-I: Direct solution of symmetric sparse systems)*, Tech. Rep. RC 21886, IBM T. J. Watson Research Center, Yorktown Heights, NY, November 2000. *http://www.cs.umn.edu/˜agupta/wsmp.*

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

[15] 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.

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

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

[18] D. Irony, G. Shklarski, and S. Toledo, *Parallel and fully recursive multifrontal sparse Cholesky*, Future Gener. Comput. Syst., 20 (2004), pp. 425–440.

[19] P. Kambadur, A. Gupta, A. Ghoting, H. Avron, and A. Lumsdaine, *Modern task parallelism for modern high performance computing*, in SC09 (International Conference for High Performance Computing, Networking, Storage and Analysis), 2009. PFunc URL: http://www.coin-or.org/projects/PFunc.xml.

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

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

[22] F. Pellegrini and J. Roman, *Sparse matrix ordering with SCOTCH*, in Proceedings of HPCN'97, Vienna, Austria. Lecture Notes in Computer Science, Vol. 1225, pp. 370-378, 1997.

[23] F. Pellegrini, J. Roman, and P. Amestoy, *Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering*, Concurrency: Practice and Experience, 12 (2000), pp. 69–84.

[24] J. Reid and J. Scott, *An out-of-core sparse Cholesky solver*, ACM Transactions on Mathematical Software, 36 (2009). Article 9, 33 pages.

[25] 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.