SIAM J. SCI. COMPUT.
Vol. 32, No. 6, pp. 3627–3649

DESIGN OF A MULTICORE SPARSE CHOLESKY FACTORIZATION USING DAGs[*]

J. D. HOGG[†], J. K. REID[†], AND J. A. SCOTT[†]

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 sparse algorithm is built on the assembly tree and subdivides the work at each node into tasks on blocks of the Cholesky factor. The dependencies between these tasks may again be represented by a DAG. To limit memory requirements, blocks are updated directly rather than through generated-element matrices. Our algorithm is implemented within a new efficient and portable 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 applications, we present experimental results that support our design choices and demonstrate that 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 performs well, particularly in the case of very large problems.

Key words. Cholesky factorization, sparse symmetric linear systems, DAG-based, parallel, multicore, Fortran 95, OpenMP

AMS subject classifications. 65F05, 65F50, 65Y05

DOI. 10.1137/090757216

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 [19] and HSL_MA77 [40] from the HSL software library [29] and CHOLMOD [10] as well as the parallel codes MUMPS [2], PARDISO [42], PaStiX [26, 27], TAUCS [30], and WSMP [23]. We summarize the main features of the parallel codes in section 2. They each have three phases: *analyze* 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 [15, 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.

[*]Received by the editors April 27, 2009; accepted for publication (in revised form) September 21, 2010; published electronically December 21, 2010. Work supported by EPSRC grants EP/F006535/1 and EP/E053351/1.
http://www.siam.org/journals/sisc/32-6/75721.html

[†]Computational Science and Engineering Department, Rutherford Appleton Laboratory, Chilton, Oxfordshire, OX11 0QX, UK (jonathan.hogg@stfc.ac.uk, john.reid@stfc.ac.uk, jennifer.scott@stfc.ac.uk).

Our focus is on multicore architectures, which differ from traditional symmetric multiprocessing (SMP) by having shared caches and shared links to memory, making efficient reuse of data in (shared) cache more important. Inspired by recent multicore solvers for dense positive-definite linear systems [8, 9], we aim to integrate tree-based and at-a-node parallel schemes into the same task-based parallel framework used in the dense codes. This is done by splitting each computation into tasks of modest size but sufficiently large that good level-3 BLAS performance can be achieved. The dependencies between the tasks are implicitly represented by a directed acyclic graph (DAG) that embeds the assembly tree. This allows the independence of operations both in different subtrees and within a single node to be easily recognized and exploited. Our approach uses the assembly tree and is non-multifrontal. This avoids the need to hold generated-element matrices for later assembly, reducing memory requirements.

Our algorithm is implemented within a new sparse Cholesky solver `HSL_MA87` that we have developed for inclusion within the Fortran library HSL [29]. It 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. We begin by presenting a concise overview of existing parallel sparse direct solvers in section 2. In section 3, we provide a short description of the recent developments for dense linear systems that we will adapt to the sparse case. Section 4 describes the algorithm implemented within our DAG-based sparse Cholesky solver `HSL_MA87`, highlighting the features that distinguish it from the other modern Cholesky codes. Results of experiments with `HSL_MA87` and comparisons with other parallel direct solvers on a range of problems are given in sections 5 and 6. Finally, in section 7, we make some concluding remarks and comment on the availability of `HSL_MA87`.

2. Background and related work.

Since the 1980s, there has been significant interest in the development of parallel sparse symmetric algorithms and solvers. To help put our work into context, we briefly summarize the key algorithmic features of five well-known modern parallel codes that offer facilities for sparse symmetric positive-definite systems and may be run on multicore machines.

MUMPS [2] is a multifrontal Fortran/MPI package. While it is designed to solve symmetric and nonsymmetric 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 the analyze phase. 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 [6]. 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 [42] is a non-multifrontal Fortran/C/OpenMP package for solving large sparse symmetric and nonsymmetric 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 [26, 27] 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 analyze by simulating all the costs. The sequence of tasks performed by each thread during factorize is fixed then. Recently, in [20], a dynamic scheduling designed for multicore and NUMA (Non-Uniform Memory Access) architectures was added.

TAUCS [30] is a C/Cilk library of sparse linear solvers. In particular, it offers a multifrontal sparse symmetric positive-definite solver. Cilk [7] 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 [23, 24] 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 [31] that achieves fine-grain load balance via work-stealing.

Other important shared-memory codes include SuiteSparseQR [13] for sparse QR, while for sparse unsymmetric linear systems there is the well-known and widely used package SuperLU_MT [14] and a parallel version of UMFPACK 4 [11] developed by Avron, Shklarski, and Toledo [5].

SuiteSparseQR [13] is a recent C++ sparse QR factorization package based on the multifrontal algorithm. Within each frontal matrix, LAPACK and multithreaded BLAS are used to achieve good performance on multicore architectures. Parallelism across different frontal matrices is handled with Intel's new threading building blocks (TBB) library for writing parallel applications in C++ on shared-memory multicore architectures [41]. During the analyze phase, the frontal matrices are assigned to TBB tasks; normally, there are fewer tasks than frontal matrices. The relationship between the tasks is held as a tree. During the factorization phase, TBB handles all synchronization and scheduling of the tasks.

SuperLU_MT [14, 34] was developed as a C/Pthreads package for the solution of unsymmetric systems, targeted at SMPs of modest size. More recently, OpenMP support has been added. As explained in the overview given in [35], the factorization uses a left-looking algorithm, with threshold partial pivoting. Sets of adjacent columns are grouped into "panels" for efficient use of BLAS. The kernel is based on a supernode-panel update, which invokes multiple calls to level-2 BLAS, effectively achieving so-called BLAS 2.5 speed. Parallelization uses an asynchronous and barrier-free dynamic allocation algorithm to schedule both coarse-grained and fine-grained parallel tasks and achieve a high level of concurrency. A global task queue is used to store ready panels in the column elimination tree, and whenever a thread becomes free, it takes a panel from this queue. The coarse-grained task is to factorize the independent panels in disjoint subtrees, while the fine-grained task is to update panels by previously computed supernodes. The scheduler facilitates the smooth transition between the two types of tasks and maintains load balance dynamically.

Parallel UMFPACK [5] is a C/Cilk package that implements an unsymmetric-pattern multifrontal approach. It employs column preordering and partial pivoting. Avron, Shklarski, and Toledo started with *UMFPACK* 4, designed and implemented their own serial version, and then parallelized it. Parallelism is exploited at a number of levels, including within the elimination tree, the merging of contribution blocks, and parallel dense operations. Numerical comparisons with SuperLU_MT show that a multifrontal approach can outperform a left-looking algorithm (particularly on a small number of processors) [5].

As is clear from these brief descriptions, the use of task scheduling and dependency management in direct methods is not new. This is primarily because the elimination tree naturally leads to a task-orientated view in which operations and updates within columns or fronts may be viewed as tasks. These ideas are present in a number of early works on the subject, including papers by, amongst others, Duff [18], Duff et al. [16], Geist and Ng [21], and George et al. [22] that use the concept of a pool or queue of tasks that are dynamically scheduled. Static scheduling has often been preferred, however, to minimize expensive communication overheads. Much of the development up to the early 1990s is described in a comprehensive review article by Heath, Ng, and Peyton [25]. Initially, the tasks involved columns, but later involved the nodes of the supernodal approach and the fronts of the multifrontal algorithm.

As methods were developed to use level-3 BLAS to avoid cache overheads (see, for example, [36]), the granularity of the tasks was substantially reduced. To achieve good performance, it became necessary to exploit parallelism in the dense tasks near the root of the tree. The simplest way to do this is to switch to a parallel dense code toward the end of the factorization. More advanced techniques have been developed, such as the master and slave processes of MUMPS and the artificial splitting of supernodes into panels as used by PARDISO and SuperLU. This allows the two levels of parallelism to be used together in a cohesive sense—though they are still fundamentally separate.

A finer-grained level of parallelism can be exposed in the supernodal case if the generation of updates from a given supernode (or panel) is split into tasks depending on the destination supernode, as is done, for example, in PARDISO. A similar grain size can be obtained in the multifrontal method by splitting a front across multiple processors, each of which performs its own update operation; this approach is used by WSMP.

We address a key weakness of many existing shared-memory parallel strategies, which is that the parallelism inherent in the tree and the parallelism inherent in the dense submatrix factorizations are implemented using different mechanisms. Our aim in combining the parallel techniques for sparse and dense tasks is to achieve an algorithm that is simple to schedule.

Although our algorithm was developed independently of PaStiX, it employs similar concepts. By using a supernodal partition of the columns to also partition the rows, PaStiX creates a blocking scheme where the sparsity pattern is stored using blocks that are either zero or nonzero (rather than individual entries). This approximates the sparsity pattern of the factors reasonably well if the separators coming from a nested dissection ordering are preserved as the supernodes. This scheme eliminates sparse operations with blocks, allowing all dense level-3 BLAS operations to be scheduled as tasks.

Our scheme overcomes a number of deficiencies present in the PaStiX algorithm. Firstly, PaStiX may store zero rows in the middle of a block. This potentially increases the size of the stored factors that must be moved through memory and also the number of floating-point operations that must be performed. Secondly, as the PaStiX algorithm avoids sparse expansion operations, the block sizes limit the dimension of matrices passed to the BLAS, reducing the performance. Our algorithm, however, allows the multiplication of a block by an entire column if that is desirable. Finally, the scheduling algorithm and storage schemes of our implementation are better tuned to exploit shared-memory and multicore architectures than the PaStiX implementation, which targets hybrid distributed/shared-memory systems.

Our contribution is to establish a supernodal sparse Cholesky algorithm that extends current state-of-the-art techniques used in the dense case. The task-orientated design allows schedulers developed there to be easily applied to sparse factorizations on multicore architectures. Our approach incorporates and extends many of the features found in existing sparse solvers including a two-dimensional (that is, blocking by rows as well as by columns) supernodal parallel scheme with arbitrary row blocking. We also employ a new variable reordering algorithm at the nodes of the elimination tree to improve cache locality in the sparse operations; this is described in section 4.3.

3. Dense DAG solvers.

In this section, we briefly describe state-of-the-art DAG-based solvers for dense linear systems of equations on multicore processors. In the next section, we discuss how we have extended these ideas to the sparse case.

Recent research by Buttari et al. [8, 9] 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. This must wait until the block A_{kk} has been fully updated.

solve_block solves a triangular set of equations $L_{kk}^T L_{ik} = A_{ik} (i > k)$ to obtain an off-diagonal block L_{ik} of the Cholesky factor. This must wait for the block A_{ik} to be fully updated and for the factorize_block $A_{kk} = L_{kk}L_{kk}^T$ to be completed.

update_block updates a block of the remaining submatrix $A_{ij} \Leftarrow A_{ij} - L_{ik}L_{jk}^T$, $i \geq j > k$. This must wait for solve_block to have completed for blocks L_{ik} and L_{jk}.

The dependencies between tasks may be represented by a 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.

Hogg [28] has implemented a dense Cholesky factorization that takes advantage of 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 formulates and solves a series of recurrence relations describing start times in relation to the critical path, assigning each node a schedule based on the latest start time on an infinitely parallel machine. Given a finite number of threads, whenever a node has to be chosen from a set of candidates, the one 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 advantages were modest.

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

4. DAG-based sparse direct solver.

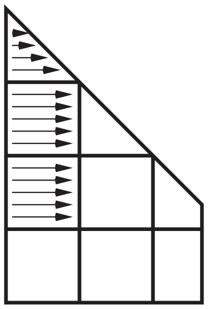
The sparse factorization work of this paper was motivated by the aim of applying the ideas of the previous section to the sparse case. In particular, we aimed to work with tasks of a sufficient size for efficient execution on a single thread using level-3 BLAS while seeking to take advantage of all the potential parallelism available between such tasks. We use a supernodal-based approach. Throughout the rest of the paper, we use the term *node* to refer to a (relaxed) supernode of L.

4.1. 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 compress this matrix in the traditional manner (see [15, 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, we subdivide this matrix into blocks under the control of a parameter nb. This is illustrated in Figure 4.1(a). We divide the computation into tasks in which a single block is revised (details in section 4.2). 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 5.5.

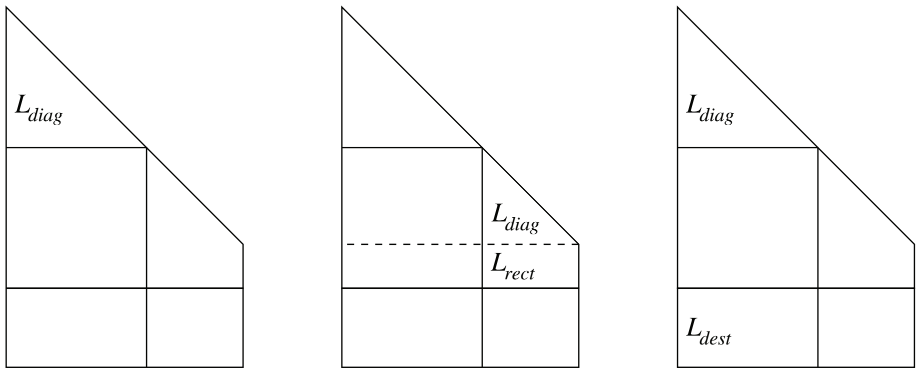
We store the nodal matrix using the row hybrid blocked structure of Andersen et al. [3] with the modification that "full" storage is used for the blocks on the diagonal


(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

FIG. 4.1. *Row hybrid block structure for a nodal matrix.*



(a) factorize_block

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

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

FIG. 4.2. *factorize_block and solve_block.*

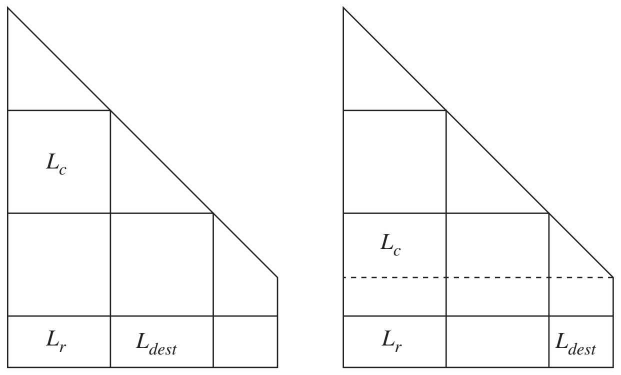
rather than storing only the actual entries ("packed" storage). This is illustrated in Figure 4.1(b). Note that the final block on the diagonal is often trapezoidal, to allow the other blocks of its block row to be square. Using the row hybrid scheme rather than the column hybrid scheme facilitates updates between nodes by removing any discontinuities at row block boundaries (we explain the importance of this in section 4.2). Storing the blocks on the diagonal in full storage allows us to exploit efficient BLAS and LAPACK routines: routines for performing symmetric updates to, or multiple solves with, and Cholesky factorization of triangular matrices stored in packed storage are either nonexistent or suffer from a performance penalty compared with their full storage equivalent.

4.2. Tasks. Following the design of our dense DAG-based code (section 3), we split the work involved in the sparse factorization of A into the following tasks:

factorize_block(L_{dest}) computes the Cholesky factor L_{diag} of the triangular part of a block L_{dest} that is on the diagonal using the LAPACK subroutine `_potrf`. If L_{dest} 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 4.2(a) and 4.2(b).



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

FIG. 4.3. *Update_internal*(L_{dest}, `scol`), $L_{dest} \Leftarrow L_{dest} - L_r L_c^T$.

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 4.2(c).

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

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

where L_r is a block of the block column `scol` and L_c is a submatrix of this block column. If L_{dest} is not in the final block column of the node, L_c is a block of `scol` (see Figure 4.3(a)); otherwise, L_c is the submatrix that corresponds to the columns of L_{dest}; see Figure 4.3(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.

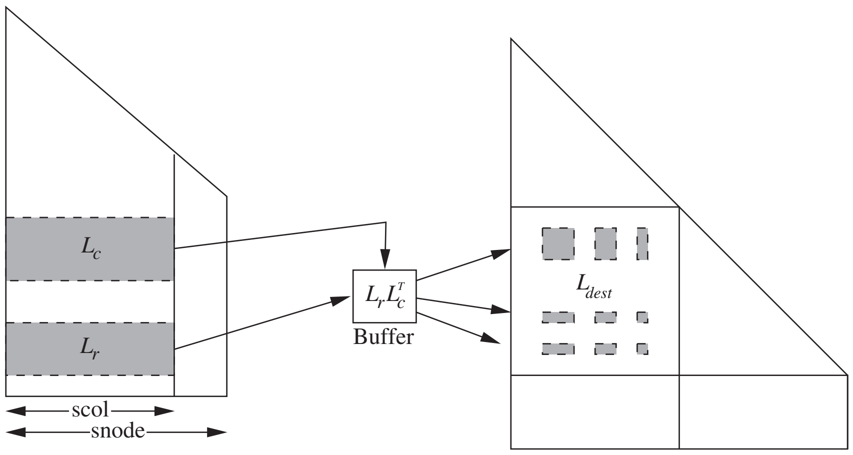
update_between(L_{dest}, `snode`, `scol`) performs the update of the block L_{dest} from the block column `scol` of a descendant node `snode`:

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

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 4.4.

Note that the submatrices L_r and L_c in Figure 4.4 are determined by the block L_{dest} that is being updated, and in general they are not blocks of the block column `scol`. Using row storage for the block column (see Figure 4.1(a)) means that L_r and L_c are stored as contiguous arrays, allowing efficient use of BLAS-3 subroutines.

We could have cast the update_between task as an operation from a pair of blocks, but this would often result in a destination block needing to be updated more than once from the same block column. This is undesirable since contested writes cause



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

FIG. 4.4. *Update_between(L_{dest},* `snode`*,* `scol`*).*

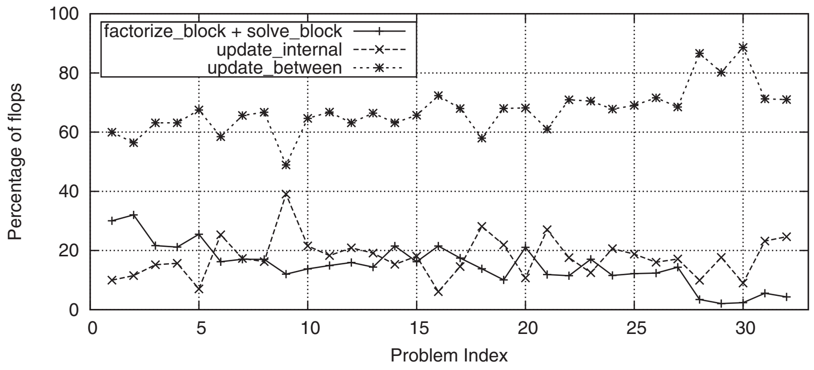


FIG. 4.5. *Percentage of flops in the different kinds of tasks. (Problem index refers to Table* 5.2).

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 destination 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 until all the relevant rows of the block column have become 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 4.3. 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 4.5, we show the percentages of flops needed for the different kinds of tasks for the test problems of section 5.

We now explain how we determine when a task is ready without needing to represent the whole DAG explicitly. During analyze, we calculate a count for each block of L. This is the total number of updates (update_internal and update_between) that will be applied to it, plus one if the block is not on the diagonal. 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. 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, so its solve_block task may be stored.

When a factorize_block or solve_block task completes, we decrement the block's 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. The flagging of blocks with negative values is needed so that when a subsequent solve_block task completes, we can identify which update tasks have now become ready. The column lock is needed to ensure that this identification is made correctly.

4.3. Task dispatch engine. Our task scheduler is fairly basic and has many features and ideas in common with established scheduling engines such as those offered by Cilk [7], Intel's TBB [41], and SMP Superscalar (SMPSs) [39]. As none of these can support our implicit task DAG representation and only SMPSs can be used from Fortran, we decided to improve the portable OpenMP scheduler originally developed for our dense Cholesky code (see section 3). The design is such that this task scheduler can be easily replaced as newer implementations are developed, for example, as part of the PLASMA [1], PFunc [31], or StarPU [4] projects. We note that a similar upgrade to use PFunc has been applied to WSMP while this paper was under review.

Our experience in the dense case was that complex prioritization schemes that take optimal schedules into account offered very limited benefit. In `HSL_MA87`, we have therefore chosen to use a simple prioritization scheme that favors cache awareness by using local task stacks and a single global task pool, in a similar fashion to SMPSs. 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. A stack is used 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 task for a diagonal block, it places the relevant factorize_block task onto the stack and will immediately take it off again for its next task, in effect executing the factorize_block task for this block at once. 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 onto its stack, then it places any update_internal tasks generated. Both will be above any stacked solve_block tasks. Since some of the data needed for an update are likely to be in the local cache, cache reuse is encouraged naturally without the need for explicit management. This corresponds to the depth-first scheduling of Cilk.

If a local stack becomes full, a global lock is acquired and the bottom half is moved to the global task pool. This is similar in concept to the main ready list of SMPSs, but the method of entry is different. SMPSs uses the main ready list for leaf tasks of the DAG, which are unlikely to need data already in a cache. We overflow the tasks that have been in the stack the longest so that their data are unlikely still to be in the local cache. This avoids the need to dynamically resize tasks' lists and significantly reduces the amount of work-stealing. Another difference between our scheduler and that of SMPSs is that rather than having a list of tasks with high priority for immediate execution, we give tasks in the task pool numerical priorities. In descending order of priority, our tasks are factorize_block, solve_block, update_internal, and update_between. However, our experience has been that using these priorities does not significantly improve the execution time. 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 task pool. When a factorize_block task completes, several solve_block tasks may be placed on the local stack, one of which is probably executed immediately. When the final update task for an off-diagonal block completes, a single solve_block task is placed on the local stack and probably executed immediately. Few solve_block tasks therefore reach the task pool. It follows that prioritization mainly affects the update tasks, and update_internal tasks will have already been favored 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 can increase significantly. This led to a larger task pool being needed (for many of our test problems, the maximum size of the pool increased by a factor of between 2 and 4) and to a small increase in execution time (in our experiments, typically less than 2 percent). It is unclear whether this results from additional task handling overheads or differences in cache locality. 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 5, the number of tasks in the pool never exceeded 10,000; the largest number in the pool 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 (work-stealing). This is similar to the breadth-first work-stealing of Cilk, the FIFO work-stealing of SMPSs or the deques of TBB. 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 5. 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 investigated the effect of using the global lock to control the local stacks as well as the global task pool. We found that this had no noticeable effect on execution time on our test machine `fox` (see Table 5.1). We conclude from this that most of the performance advantage we demonstrate in the results section is due to better exploitation of the cache rather than reducing contention for the global lock. Despite this, we have retained the separate locks to avoid problems as core counts increase.

4.4. 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 when the contents of the buffer holding $L_rL_c^T$ are added into L_{dest}; see

Figure 4.4. 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 analyze 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 n_j is the first child of $n_{j+1}, j = 1, \ldots k-1$. The index list at n_1 may be subdivided 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 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 5.2.

5. Numerical results.

In this section, we present numerical results for our new sparse Cholesky code `HSL_MA87`.

5.1. Test environment.

The experiments of this section were performed on our multicore test machine `fox`, details of which are given in Table 5.1. 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 5.2. Each problem is available from the University of Florida Sparse Matrix Collection [12]. The problem indices will be used in tables and figures to identify the problems. In selecting

TABLE 5.1
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

TABLE 5.2

*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. * indicates only the sparsity pattern is provided.*

Problem index	Identifier	n (10^3)	$nz(A)$ (10^6)	$nz(L)$ (10^6)	Flops (10^9)	Application/description
1	`CEMW/tmt_sym`	727	2.9	30.0	9.4	Electromagnetics
2	`Schmid/thermal2`	1228	4.9	51.6	14.6	Unstructured thermal FEM
3	`Rothberg/gearbox`*	154	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`	218	5.9	48.6	22.4	Pressurised wind tunnel
6	`Chen/pkustk13`*	94.9	3.4	30.4	25.9	Machine element
7	`GHS_psdef/crankseg_1`	52.8	5.3	33.4	32.3	Linear static analysis
8	`Rothberg/cfd2`	123	1.6	38.3	32.7	CFD pressure matrix
9	`DNVS/thread`	29.7	2.2	24.1	34.9	Threaded connector
10	`DNVS/shipsec8`	115	3.4	35.9	38.1	Ship section
11	`DNVS/shipsec1`	141	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`*	202	5.7	52.0	48.2	Oil production platform
14	`Schenk_AFE/af_shell3`	505	9.0	93.6	52.2	Sheet metal forming
15	`DNVS/troll`*	214	6.1	64.2	55.9	Structural analysis
16	`AMD/G3_circuit`	1586	4.6	97.8	57.0	Circuit simulation
17	`GHS_psdef/bmwcra_1`	149	5.4	69.8	60.8	Automotive crankshaft
18	`DNVS/halfb`*	225	6.3	65.9	70.4	Half-breadth barge
19	`Um/2cubes_sphere`	102	0.9	45.0	74.9	Electromagnetics
20	`GHS_psdef/ldoor`	952	23.7	145	78.3	Large door
21	`DNVS/ship_003`	122	4.1	60.2	81.0	Ship structure
22	`DNVS/fullb`*	199	6.0	74.5	100	Full-breadth barge
23	`GHS_psdef/inline_1`	504	18.7	173	144	Inline skater
24	`Chen/pkustk14`*	152	7.5	107	146	Tall building
25	`GHS_psdef/apache2`	715	2.8	135	174	3D structural problem
26	`Koutsovasilis/F1`	344	13.6	174	219	AUDI engine crankshaft
27	`Oberwolfach/boneS10`	915	28.2	278	282	Bone micro-FEM
28	`ND/nd12k`	36.0	7.1	117	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	321	2054	3D mesh problem
31	`Oberwolfach/bone010`	987	36.3	1076	3876	Bone micro-FEM
32	`GHS_psdef/audikw_1`	944	39.3	1242	5804	Automotive crankshaft

the test set, our aim was to choose a wide variety of large-scale problems. In our tests, we use the nested dissection ordering that is computed by `METIS_NodeND` [32, 33]. In Table 5.2, we include the number of entries in the matrix factor and the number of flops when this pivot sequence is used by `HSL_MA87` without node amalgamation (see section 5.3).

Unless stated otherwise, runs were performed using all 8 cores on our test machine `fox` with all control parameters of `HSL_MA87` at 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, a time would be greater by much more than this.[1] We therefore averaged over 10 complete runs of each of the problems except for the slowest five, where we averaged over two. We will refer to these five problems as the *slow subset*, each of which requires more that 500 Gflops to factorize A and always took longer than 10 seconds.

[1]We believe that the occasional very slow runs were caused by a core with a critical task being asked to perform work for the operating system.

TABLE 5.3
The effect of reordering the children on the HSL_MA87 *factorize times on* 1 *and* 8 *cores.*

Problem	Without reordering			With reordering		
index	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

5.2. Effect of reordering the children. In section 4.4, we explained our strategy for choosing, at each nonleaf 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 5.3, which includes results for the slow subset together with three other problems, chosen to show the range of behaviors found. In this comparison, we denote the natural ordering of the children (i.e., that derived from pivot order) in the column "without reordering". Though reasonably modest, of particular note is that the performance gains in parallel are often 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.

5.3. Effect of node amalgamation. Node amalgamation (see section 4 of [17]) 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 a chosen number *node_amal* of eliminations. A more powerful version is used in [40], involving the recursive merge of any child/parent pair if both involve fewer than *node_amal* eliminations. We have chosen to use this version.

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

Table 5.4 shows the factorize times for the slow subset and for three others that represent the three kinds of behavior that we saw: flat (problem 3), $node\_amal = 1$ much slower (problem 16), and U-shaped (problem 22). The two biggest problems showed flat behavior, but $node\_amal = 64$ was best for problems 28–30.

Table 5.4 also shows the number of entries in L and the solve times. The examples illustrate the kinds of behavior that we saw for the solve times: flat (problems 28, 29), rising slowly (problems 31, 32), rising less slowly (problems 3, 16, 22), and $node\_amal = 1$ much slower (problem 16).

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

5.4. Block size. The block size *nb* was discussed in section 4.1. In Table 5.5, we report the factorize time for a range of block sizes on a single core and on 8 cores.

Table 5.4

Performance on 8 *cores for values of the node amalgamation parameter node_amal in the range* 1 *to* 64. *The factorize times within* 3% *of the fastest are in bold.*

Problem index	$nz(L)$ (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

Problem	Factorize times					Solve times				
index	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

Table 5.5

Comparison of the factorize times for different block sizes nb. The factorize times within 3% *of the fastest are in bold.*

Problem	Single core factorize times						8-core factorize times				
index	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**

As in Table 5.4, we show results for the slow subset and three cases representing the behavior of the other problems for both factorize and solve times. On a single core, for all but the largest problems, there was little 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 8 cores, 256 almost always gave times within 3% of the best. These considerations led us to use 256 as our default value. While a smaller value is beneficial for the large problems on a single core, we want a single parameter for simplicity and expect large problems to be solved on more than one core.

5.5. Extended rectangular blocks. We explained in section 4.1 the merits of using extended rectangular blocks when the number of columns nc in the nodal matrix is less than nb. 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 (extended) to 25.2 (fixed). 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 in `HSL_MA87`) extended rectangular blocks are used.

Table 5.6
Comparison of the factorize times on 8 cores for different local task stack sizes. The factorize times within 3% of the fastest are in bold.

Problem	No local	Stack size				
index	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**

Table 5.7
Tasks taken directly from local stacks, moved to pool because of a full stack, and moved to pool because of work-stealing (W-S). Results are on 8 cores for a range of local stack sizes.

Problem index	Stack size	Direct (10^3)	Full (10^3)	W-S (10^3)
19	10	19	28	0.14
(880	50	40	7	0.30
leaves)	100	45	1	0.43
(48,000	200	46	0	0.54
tasks)	300	46	0	0.47
28	10	19	105	0.16
(60	50	45	78	0.47
leaves)	100	66	56	0.81
(124,000	200	92	30	1.61
tasks)	300	108	13	1.83
29	10	21	221	0.13
(60	50	46	195	0.39
leaves)	100	63	177	1.25
(241,000	200	97	144	0.89
tasks)	300	119	121	1.76
32	10	232	534	0.16
(5,580	50	573	192	0.92
leaves)	100	703	60	2.84
(772,000	200	751	11	3.57
tasks)	300	759	4	3.95

5.6. Local task stack size. In section 4.3, 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 of 10%–20% over running without local stacks. In Table 5.6, we report the results for three representative problems that gave gains at the ends and middle of the range 10%–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.

For three of the slow problems, we show in Table 5.7 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 work-stealing. We see that the number of tasks moved by work-stealing 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 5.7. Next was problem 26, where 750 tasks were moved to the pool because of a full stack. For a further six problems, fewer than 400 tasks were moved to the pool because of a full stack. 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 work-stealing figures in Table 5.7 are small, work-stealing is important for load balance, and its importance increases with the local stack size. In our

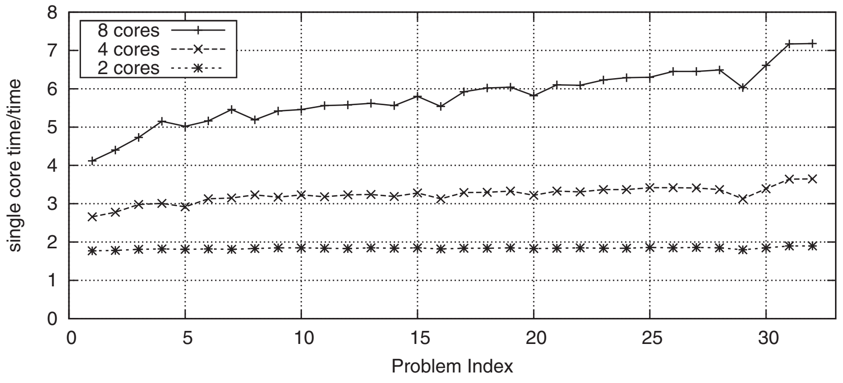


FIG. 5.1. *The ratios of* HSL_MA87 *factorize times on* $2, 4$*, and* 8 *cores to that on a single core.*

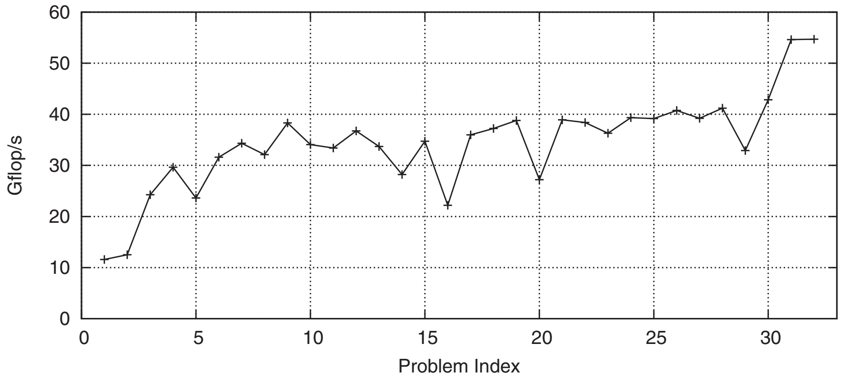


FIG. 5.2. *The speed of* HSL_MA87 *factorize in Gflop/s on* 8 *cores.*

tests at stack size 100, enabling work-stealing reduced the execution time by up to 25% on some problems.

5.7. 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 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 at least 3 for all but 4 of the smallest test problems, and for the 2 largest problems 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 2 largest. For all but the 3 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 5.2. Here we compute the flop count from a run with the node amalgamation parameter *node_amal* having the value 1 (the flop count reported in Table 5.2). 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 5.1). 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.

Table 6.1
Sparse direct solvers used in our numerical experiments on `fox`.

Code	Date/version/compiler	Authors/website
`HSL_MA87`	08.2009/ v1.4.0/ Intel 11.0 -fast	J.D. Hogg, J.K. Reid, and J.A. Scott, HSL `http://www.hsl.rl.ac.uk`
PARDISO	10.2009/ v4.0.0/ Intel 10.1 -fast	O. Schenk and K. Gärtner `http://www.pardiso-project.org`
PaStiX	04.2009/ v5.1.2/ Intel 11.0 -fast	P. Henon, P. Ramet, and J. Roman `http://pastix.gforge.inria.fr`
TAUCS	09.2003/ v2.2/ cilk 5.4.6/ GNU 4.1.2 -O3	D. Irony, G. Shklarski, and S. Toledo `http://www.cs.tau.ac.il/∼stoledo/taucs`
WSMP	06.2010/ v10.5.26/ Intel 11.1	A. Gupta, IBM `http://www-users.cs.umn.edu/∼agupta/wsmp.html`

6. Comparisons with other solvers.

We end by presenting some comparisons of the performance of `HSL_MA87` with that of the recent sparse Cholesky solvers outlined in section 1. Note that the intention here is not to make a detailed study of the advantages and disadvantages of each package or to assist potential users in the selection of a solver. The versions of the solvers used together with other details are given in Table 6.1. MUMPS is not included since it is an MPI-based code and not designed for shared-memory architectures. Where a solver offers more than one version, we use the one designed for shared memory (in particular, we used the Pthreads version of PaStiX, but note that in some shared memory environments the code's authors observe it may be more efficient to mix MPI and Pthreads). 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 codes performs any numerical pivoting). We provide `HSL_MA87`, PARDISO, and WSMP with the ordering generated by `METIS_NodeND` [32]. 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 significantly increased its subsequent factorization time. Instead, for PaStiX, as recommended, we use its internal call to the nested dissection routine of SCOTCH [38, 37]. We found this generally produced slightly sparser factors than `METIS_NodeND`. TAUCS is run using its multifrontal option, as this is the advice given in the user documentation.

We note that because of how the codes were supplied and the mix of languages employed, we were unable to use a single compiler for all our tests. Where possible we have used compilers from the Intel suite. PARDISO was supplied as a library built with a slightly older version of the compiler, and the Cilk extensions to C used by TAUCS supports only the GNU gcc compiler. However, our experience of running `HSL_MA87` with the Intel 10.1, 11.0, and gcc compilers was that it had little effect on the factorize times because 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. On our test machines, each solver used the same Intel BLAS.

6.1. Comparisons on one core of `fox`.

Although the solvers are primarily designed to run in parallel, it is of interest to first run them on a single core. In Figure 6.1, the ratios of the factorize times for PARDISO, PaStiX, TAUCS, and WSMP to the factorize time for `HSL_MA87` on `fox` are plotted for each of our 32

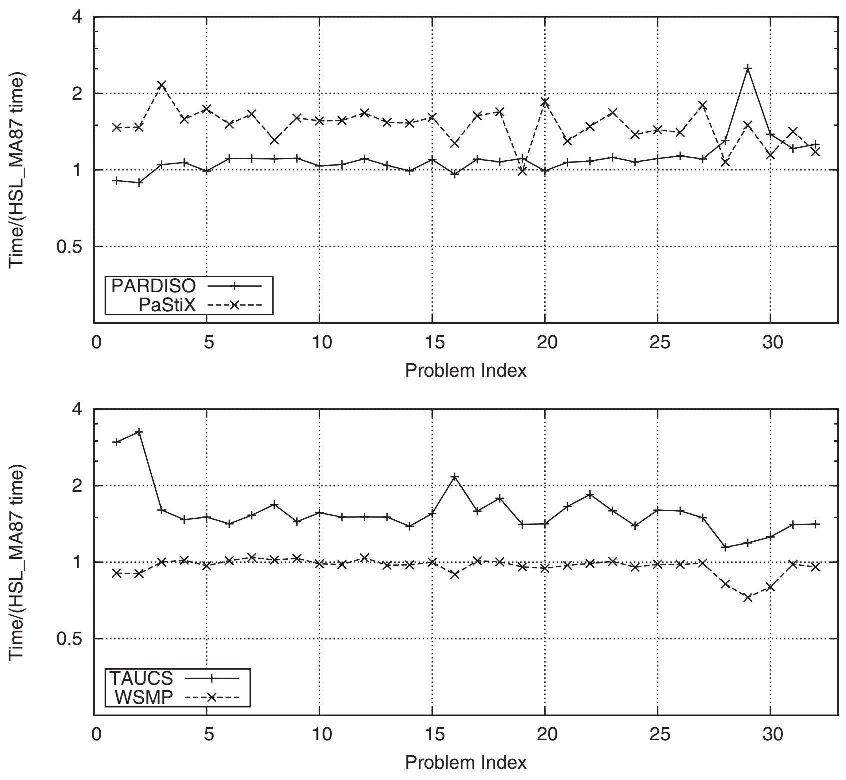


FIG. 6.1. *The ratios of the supernodal solvers PARDISO and PaStiX and multifrontal solvers TAUCS and WSMP factorize times to the* HSL_MA87 *factorize time (single core of* fox*).*

test problems. Compared with the other supernodal solvers, we see that HSL_MA87 is generally faster than PaStiX while it is comparable to PARDISO, although the latter is slightly poorer on the largest problems. Looking at the multifrontal solvers, HSL_MA87 consistently outperforms TAUCS, but for most problems offers similar performance to WSMP. On problems 28, 29, and 30, the multifrontal codes show a better comparative performance, with WSMP having the fastest times. We suspect this is due to the nature of these problems favoring one technique over another (problems 28 and 30 are from the same test set and are related, while problem 29 is an unusual academic problem).

6.2. Comparisons on 8 cores of fox. In Figure 6.2, the ratios of the factorize times for PARDISO, PaStiX, TAUCS, and WSMP to the factorize time for HSL_MA87 on eight cores of fox are given. We see that, on 8 cores, HSL_MA87 performs well (the only exceptions being a small number of problems for which PARDISO or WSMP is faster). Both PaStiX and TAUCS are significantly slower than HSL_MA87, delivering their best performance on the largest problems. Conversely, in our tests, the speedups for PARDISO are poor for the largest problems. We believe that this is because PARDISO does not use a two-dimensional decomposition of the block columns. All the problems where WSMP outperforms HSL_MA87 in parallel are also ones where it outperformed HSL_MA87 in serial. However, with the exception of the smallest problem, the gap is proportionally smaller, as HSL_MA87 achieves better speedups than WSMP.

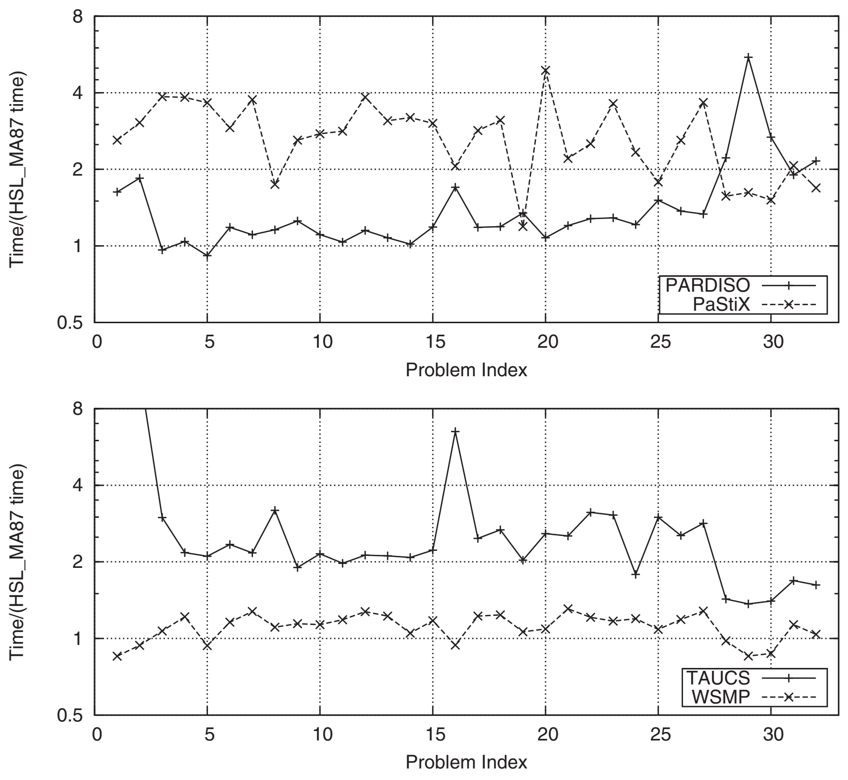


FIG. 6.2. *The ratios of the supernodal solvers PARDISO and PaStiX and multifrontal solvers TAUCS and WSMP factorize times to the* HSL_MA87 *factorize time (8 cores of* fox*).*

TABLE 6.2
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

For a number of problems where the WSMP and HSL_MA87 times are comparable in serial, HSL_MA87 is faster in parallel.

6.3. Comparisons on other processors. So far, the results have all been for our test machine fox (see Table 5.1). We end this section by presenting runs on two further multicore machines, brief details of which are given in Table 6.2. Since the results we have already reported indicate that, of the solvers tested, PARDISO and WSMP most closely rival HSL_MA87, we restrict our runs to these solvers (on both machines the version of PARDISO included in the Intel MKL 11.0 is used). The ratios of the factorize times on 8 cores are given in Figures 6.3 and 6.4. We observe that on both machines the performance of HSL_MA87 compares favorably with that of both PARDISO and WSMP. In particular, on the largest problems running on 8 cores, HSL_MA87 is significantly faster than PARDISO.

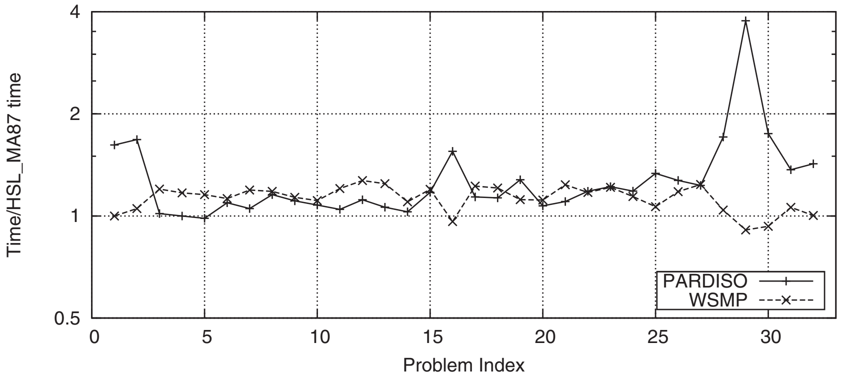


FIG. 6.3. *Comparison of* HSL_MA87*, PARDISO, and WSMP factorize times on the Intel Nehalem architecture (8 cores).*

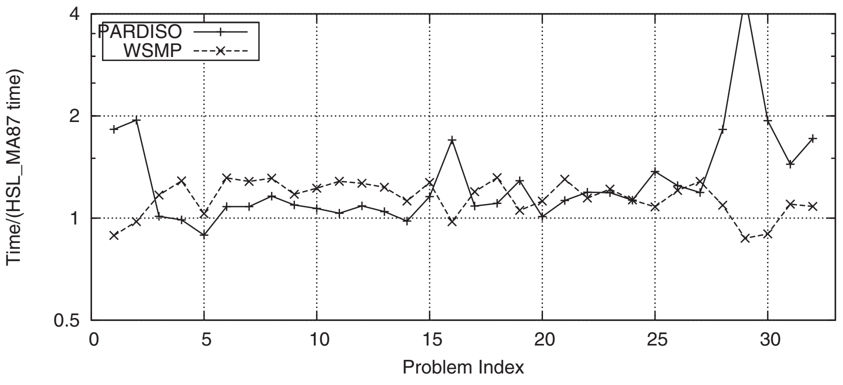


FIG. 6.4. *Comparison of* HSL_MA87*, PARDISO, and WSMP factorize times on the AMD Shanghai architecture (8 cores).*

7. 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 on a range of problems on our 8-core test machines.

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 (noncommercial) 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 [29].

Acknowledgments. Our thanks to Iain Duff for helpful suggestions that improved the presentation of an early draft of this paper. We are grateful to the authors of the solvers tested in section 6 for their help and comments. We also thank three anonymous referees and the associate editor whose constructive criticism led to improvements in both HSL_MA87 and this paper.

REFERENCES

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, *Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects*, J. Phys.: Conf. Ser., IOP Publishing, 180 (2009), article 012037.

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

[3] B. S. Andersen, J. A. Gunnels, F. G. Gustavson, J. K. Reid, and J. Wasniewski, *A fully portable high performance minimal storage hybrid format Cholesky algorithm*, ACM Trans. Math. Software, 31 (2005), pp. 201–207.

[4] C. Augonnet, S. Thibault, and R. Namyst, *StarPU: A Runtime System for Scheduling Tasks Over Accelerator-based Multicore Machines*, Technical report Inria-00467677, Inria, Paris, 2010.

[5] H. Avron, G. Shklarski, and S. Toledo, *Parallel unsymmetric-pattern multifrontal sparse LU with column preordering*, ACM Trans. Math. Software, 34 (2008), article 8.

[6] 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*, SIAM, Philadelphia, 1999.

[7] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, *Cilk: An efficient multithreaded runtime system*, J. Parallel Distrib. Comput., 37 (1996), pp. 55–69.

[8] 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.

[9] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, *A class of parallel tiled linear algebra algorithms for multicore architectures*, Parallel Comput., 35 (2009), pp. 38–53.

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

[11] T. A. Davis, *Algorithm* 832: *UMFPACK, an unsymmetric-pattern multifrontal method*, ACM Trans. Math. Software, 30 (2004), pp. 196–199.

[12] T. A. Davis and Y. Hu, *The University of Florida Sparse Matrix Collection*, ACM TOMS, to appear.

[13] T. A. Davis, *Algorithm 8xx, SuiteSparseQR: A multifrontal multithreaded sparse QR factorization package*, ACM Trans. Math. Software, submitted.

[14] J. W. Demmel, J. R. Gilbert, and X. S. Li, *An asynchronous parallel supernodal algorithm for sparse Gaussian elimination*, SIAM J. Matrix Anal. Appl., 20 (1999), pp. 915–952.

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

[16] I. S. Duff, N. I. M. Gould, M. Lescrenier, and J. K. Reid, *The Multifrontal Method in a Parallel Environment*, Technical report CSS 211, Harwell Laboratory, Oxfordshire, UK, 1987.

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

[18] I. S. Duff, *Parallel implementation of multifrontal schemes*, Parallel Comput., 3 (1986), pp. 193–204.

[19] I. S. Duff, *MA*57*—A new code for the solution of sparse symmetric definite and indefinite systems*, ACM Trans. Math. Software, 30 (2004), pp. 118–154.

[20] M. Faverge and P. Ramet, *Dynamic scheduling for sparse direct solver on NUMA architectures*, in Proceedings of PARA'2008, NTNU, Trondheim, Norway, 2008.

[21] G. A. Geist and E. Ng, *Task scheduling for parallel sparse Cholesky factorization*, Int. J. Parallel Program., 18 (1989), pp. 291–314.

[22] A. George, M. T. Heath, J. Liu, and E. Ng, *Solution of sparse positive definite systems on a shared-memory multiprocessor*, Int. J. Parallel Program., 15 (1986), pp. 309–325.

[23] 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, Yorktown Heights, NY, 2001; also available online from http://www.cs.umn.edu/~agupta/doc/wssmp-paper.ps.

[24] A. Gupta, *WSMP: Watson Sparse Matrix Package (Part* I: *Direct Solution of Symmetric Sparse Systems)*, Technical report RC 21886, IBM T. J. Watson Research Center, Yorktown Heights, NY, 2000; also available online from http://www.cs.umn.edu/~agupta/wsmp.

[25] M. T. Heath, E. Ng, and B. W. Peyton, *Parallel algorithms for sparse linear systems*, SIAM Rev., 33 (1991), pp. 420–460.

[26] P. Hénon, P. Ramet, and J. Roman, *PaStiX: A parallel sparse direct solver based on a static scheduling for mixed* 1*D/*2*D block distributions*, in Workshops IPDPS, Lecture Notes in Comput. Sci. 15, Springer-Verlag, New York, 2000, pp. 519–525.

[27] P. Hénon, P. Ramet, and J. Roman, *PaStiX: A high-performance parallel direct solver for sparse symmetric definite systems*, Parallel Comput., 28 (2002), pp. 301–321.

[28] J. D. Hogg, *A DAG-based Parallel Cholesky Factorization for Multicore Systems*, Technical report RAL-TR-2008-029, Rutherford Appleton Laboratory, Chilton, Oxfordshire, UK, 2008.

[29] HSL, *A Collection of Fortran Codes for Large-scale Scientific Computation*, http://www.hsl.rl.ac.uk/.

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

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

[32] G. Karypis and V. Kumar, *METIS—Family of Multilevel Partitioning Algorithms*, http://glaros.dtc.umn.edu/gkhome/views/metis.

[33] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., 20 (1998), pp. 359–392.

[34] X. S. Li, *An overview of SuperLU: Algorithms, implementation, and user interface*, ACM Trans. Math. Software, 31 (2005), pp. 302–325.

[35] X. S. Li, *Evaluation of SuperLU on multicore architectures*, J. Phys. Conf. Ser., 125 (2008), article 012079.

[36] E. G. Ng and B. W. Peyton, *A supernodal Cholesky factorization algorithm for shared-memory multiprocessors*, SIAM J. Sci. Comput., 14 (1993), pp. 761–769.

[37] 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.

[38] F. Pellegrini and J. Roman, *Sparse matrix ordering with SCOTCH*, in Proceedings of HPCN'97, Vienna, Austria, Lecture Notes in Comput. Sci. 1225, Springer-Verlag, New York, 1997, pp. 370–378.

[39] J. M. Perez, R. M. Badia, and J. Labarta, *A dependency-aware task-based programming environment for multi-core architectures*, in Proceedings of the IEEE International Conference on Cluster Computing, 2008, pp. 142–151.

[40] J. K. Reid and J. A. Scott, *An out-of-core sparse Cholesky solver*, ACM Trans. Math. Software, 36 (2009), article 9.

[41] J. Reinders, *Intel threading building blocks: Outfitting C++ for Multi-core Processor Parallelism*, O'Reilly Media, Sebastopol, CA, 2007.

[42] O. Schenk and K. Gärtner, *Solving unsymmetric sparse systems of linear equations with PARDISO*, J. Future Generation Comput. Syst., 20 (2004), pp. 475–487.