Parallel and Fully Recursive Multifrontal Sparse Cholesky*

Dror Irony Gil Shklarski Sivan Toledo

12th December 2002

Abstract

We describe the design, implementation, and performance of a new parallel sparse Cholesky factorization code. The code uses a multifrontal factorization strategy. Operations on small dense submatrices are performed using new dense-matrix subroutines that are part of the code, although the code can also use the BLAS and LAPACK. The new code is recursive at both the sparse and the dense levels, it uses a novel recursive data layout for dense submatrices, and it is parallelized using Cilk, an extension of C specifically designed to parallelize recursive codes. We demonstrate that the new code performs well and scales well on SMP's. In particular, on up to 16 processors, the code outperforms two state-of-the-art message-passing codes. The scalability and high performance that the code achieves imply that recursive schedules, blocked data layouts, and dynamic scheduling are effective in the implementation of sparse factorization codes.

keywords: sparse Cholesky factorization, parallel Cholesky factorization, multifrontal factorizations, Cilk, recursive factorizations, block layouts, recursive layouts.

1 Introduction

This paper describes the design and implementation of a new parallel direct sparse linear solver. The solver is based on a multifrontal supernodal sparse Cholesky factorization [17] (see also the review [36]). The first analysis parallel multifrontal methods is due to Duff [18], and there are now a number of parallel high-performance implementations [3, 23, 27, 32, 37].

The multifrontal method factors the matrix using recursion on a combinatorial structure called the elimination tree (etree). Each vertex in the tree is associated with a set of columns of the Cholesky factor L. The set of columns is called an amalgamated node [17] or a relaxed supernode [6]. Strictly speaking, in the elimination tree each vertex is associated with a single column. The structure that we call elimination tree in this paper is usually called an assembly tree or a condensed or supernodal elimination tree. Since we never refer to the single-column elimination tree in this paper, no confusion should arise. The set of columns associated with a vertex is called a *front* or a *supernode* and the number of columns in such as set is called the size of the front (or the size of the supernode). The method works by factoring the columns associated with all the proper descendants of a vertex v, then updating the coefficients of the unknowns associated with v, and factoring the columns of v.

The updates and the factorization of the columns of v are performed using calls to the *dense* level-3 BLAS [14, 15]. The ability to exploit the dense BLAS and the low symbolic

*This research was supported in part by an IBM Faculty Partnership Award, by grants 572/00 and 9060/99 from the Israel Science Foundation (founded by the Israel Academy of Sciences and Humanities), and by a VATAT graduate fellowship.

overhead allow the method to effectively utilize modern computer architectures with caches and multiple processors. Our solver includes a newly designed and implemented subset of the BLAS/LAPACK, although it can use existing implementations, such as ATLAS [42] and BLAS produced by computer vendors [1, 2, 12, 28, 29, 33, 38].

While the multifrontal supernodal method itself is certainly not new, the design of our solver is novel. The novelty stems from aggressive use of recursion in all levels of the algorithm, which allows the solver to effectively utilize complex advanced memory systems and multiple processors. We use recursion in three ways, one conventional and two new:

- The solver uses a recursive formulation for both the multifrontal sparse factorization and for the new implementation of the BLAS. This approach is standard in multifrontal sparse factorization, and is now fairly common in new implementations of the dense linear algebra codes [5, 19, 20, 25, 24, 26, 41, 42]. A similar approach was recently proposed by Dongarra and Raghavan for a non-multifrontal sparse Cholesky method [16]. This use of recursive formulations enables us to exploit recursion in two new ways.

- The solver exploits parallelism by declaring, in the code, that certain function calls can run concurrently with the caller. That is, the parallel implementation is based entirely on recursive calls that can be performed in parallel, and not on loop partitioning, explicit multithreading, or message passing. The parallel implementation uses Cilk [21, 39], a programming environment that supports a fairly minimal parallel extension of the C programming language and a specialized run-time system. One of the most important aspects of using Cilk is the fact that it performs dynamic scheduling that leads to both load balancing and locality of reference.

- The solver lays out dense submatrices recursively. More specifically, matrices are laid out in blocks, and the blocks are laid out in memory using recursive partitioning of the matrices. This data layout, originally proposed by Gustavson et al. [24] ensures automatic effective utilization of all the levels of the memory hierarchy and can prevent false sharing and other memory-system problems. The use of a novel indirection matrix enables low-overhead indexing and sophisticated memory management for block-packed formats.

Our performance results indicate that sparse factorization codes implemented in Cilk can achieve high performance and high scalability. The fact that Cilk schedules processors dynamically using only information about the procedure-invocation tree of a running program relieves the programmer from having to implement complex special-purpose schedulers. (Cilk runs on shared-memory machines; we acknowledge that on distributed-memory machines, static and special purpose schedulers might still be necessary.) Cilk supports nested parallelism in a natural way: if there is sufficient parallelism at the top levels of the program, Cilk does not extract any parallelism further down; if there is not, Cilk uses parallelism exposed at lower levels. This ensures that the coarsest possible level of parallelism is exploited, leading to low overhead and good data locality.

Our results also indicate that blocked data layouts for dense matrices enables high-performance sequential and parallel implementations at both the dense and sparse levels of the factorization code. We have found that it is best to keep the data in a blocked layout throughout the computation, even though this results in slightly higher data-access costs at the sparse levels.

The rest of the paper is organized as follows. Section 2 describe the design of the new dense subroutines. Section 3 describes the design of the parallel sparse Cholesky factorization code. Section 4 describes the performance of the new solver, and Section 5 presents our conclusions.

2 Parallel Recursive Dense Subroutines

Our solver uses a novel set of BLAS (basic linear algebra subroutines; routines that perform basic operations on dense blocks, such as matrix multiplication; we informally include in this term dense Cholesky factorizations). The novelty lies in the fusion of three powerful ideas: recursive data structures, automatic kernel generation, and parallel recursive algorithms.

2.1 Indirect Block Layouts

Our code stores matrices by block, not by column. Every block is stored contiguously in memory, either by row or by column. The ordering of blocks in memory is based on a recursive partitioning of the matrix, as proposed in [24]. The algorithms use a recursive schedule, so the schedule and the data layout match each other. The recursive layout allows us to automatically exploit level 2 and 3 caches and the TLB. The recursive data layout also prevents situations in which a single cache line contains data from two blocks, situations that lead to false sharing of cache lines on cache-coherent multiprocessors.

Our data format uses a level of indirection that allows us to efficiently access elements of a matrix by index, to exploit multilevel memory hierarchies, and to transparently pack triangular and symmetric matrices. While direct access by index is not used often in most dense linear algebra algorithms, it is used extensively in the extend-add operation in multifrontal factorizations.

In our matrix representation, shown in Figure 1, a matrix is represented by a two-dimensional array of structures that represent submatrices. The submatrices are of uniform size, except for the submatrices in the last row and column, which may be smaller. This array is stored in a column major order in memory. A structure that represents a submatrix contains a pointer to a block of memory that stores the elements of the submatrix, as well as several meta-data members that describe the size and layout of the submatrix. The elements of a submatrix are stored in either column-major order or row-major order. The elements of all the submatrices are normally stored submatrix-by-submatrix in a large array that is allocated in one memory-allocation call, but the order of submatrices within that array is arbitrary. It is precisely this freedom to arbitrarily order submatrices that allows us to effectively exploit multilevel caches and non-uniform-access-time memory systems.

Note that in Figure 1, the matrix is not only partitioned, but its blocks are *laid out recursively in memory*. That is, the first four blocks in memory are A_{11}, A_{21}, A_{12}, and A_{22}, not the blocks in the first block-row or the first block-column. This arrangement is the one we call recursive partitioning.

Accessing a random element of a matrix requires two div/mod operations and two indexing operations in two-dimensional arrays. In a column- or row-major layout, only one indexing operation is required and no div/mod's. However, even in an extend-add operation, the elements that are accessed are not completely arbitrary, and many accesses are to consecutive elements. In such cases, we can significantly reduce the cost of data accesses by simply advancing a pointer up to the block boundary. Our code uses this optimization.

2.2 Efficient Kernels, Automatically-Generated and Otherwise

Operations on individual blocks are performed by optimized kernels that are usually produced by automatic kernel generators. In essence, this approach bridges the gap between the level of performance that can be achieved by the translation of a naive kernel implementation by an optimizing compiler, and the level of performance that can be achieved by careful hand coding. The utility of this approach has been demonstrated by ATLAS, as well as by earlier projects, such as PHIPAC [7]. We have found that on some machines with advanced compilers, such as SGI Origin's, we can obtain better performance by writing naive kernels and letting the native optimizing compiler produce the kernel. On SGI Origin's, a

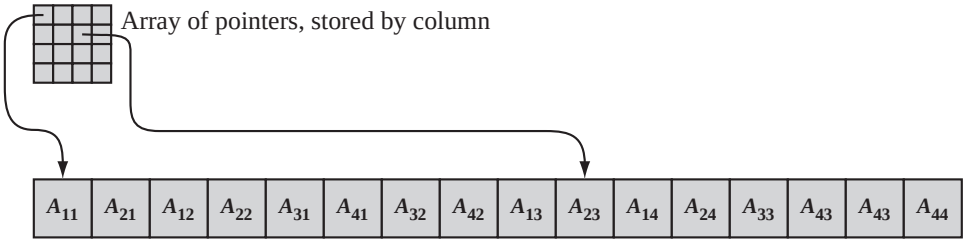

Figure 1: The use of indirection in the layout of matrices by block. The matrix is represented by an array of pointers to blocks (actually by an array of structures that contain pointers to blocks).

compiler feature called the loop-nest optimizer delivers better performance at smaller code size than our automatically-generated kernels. Even on the SGI, our kernels are not completely naive: as in ATLAS, we split the kernel into several cases, one that handles fixed-size blocks and others that handle smaller leftover blocks of arbitrary sizes.

We currently have kernel generators for two BLAS routines: DGEMM and DSYRK. We also use hand-coded kernels for fixed-sized blocks. The hand-coded kernels are written in assembly language and utilize SSE2 instructions (vector instructions on Intel Pentium processors that handle pairs of double-precision floating-point numbers). These kernels are based on the SSE2 DGEMM kernel contributed to ATLAS by Camm Maguires.

The kernel generators accept as input several machine parameter and code-configuration parameters and generate optimized kernels automatically. Our matrix multiplication (DGEMM) kernel generator is essentially the ATLAS generator (by Whaley and Petitet). We have implemented a similar generator for rank-k update (DSYRK). The additional kernels ensure that we obtain high performance even on small matrices; relying only on a fast DGEMM kernel, which is the strategy that ATLAS uses, leads to suboptimal performance on small inputs. Our DSYRK kernel is simpler than ATLAS's DGEMM kernel: it uses unrolling but not optimizations such as software pipelining and prefetching.

The flexibility of our data structures allows us one optimization that is not possible in ATLAS and other existing kernels. Our data structure can store a submatrix either by row or by column; a bit in the submatrix structure signals whether the layout is by row or by column. Each kernel handles one layout, but if an input submatrix is laid out incorrectly, the kernel simply calls a conversion subroutine that transposes the block and flips the layout bit. In the context of the BLAS and LAPACK calls made by the sparse factorization code, it is never necessary to transpose a block more than once. In other BLAS implementations that are not allowed to change the layout of the input, a single block may be transposed many times, in order to utilize the most efficient loop ordering and stride in each kernel invocation (usually in order to perform the innermost loop as a stride-1 inner product).

2.3 Parallel Recursive Dense Subroutines

The fact that the code is recursive allows us to easily parallelize it using Cilk. The syntax of Cilk, illustrated in Figures 2 and 3 (and explained fully in [39]) allows the programmer to specify that a function call may execute the caller concurrently with the callee. A special command specifies that a function may block until all its subcomputations terminate. Parallelizing the recursive BLAS in Cilk essentially meant that we added the `spawn` keyword to function calls that can proceed in parallel and the `sync` keyword to wait for termination of subcomputations. We stress that we use recursion not just in order to expose parallelism, but because recursion improves locality of reference in the sequential case as well.

```
cilk void mat_mult_add(int n,
                       matrix A, matrix B, matrix C) {
  if (n < blocksize) {
    mat_mult_add_kernel(n, A, B, C);
  } else {
    // Partition A into A_11, A_12, A_21, A_22
    // Partition B and C similarly
    spawn mat_mult_add(n/2,A_11,B_11,C_11);
    spawn mat_mult_add(n/2,A_11,B_12,C_12);
    spawn mat_mult_add(n/2,A_21,B_11,C_21);
    spawn mat_mult_add(n/2,A_21,B_12,C_22);
    sync; // wait for the 4 calls to return
    spawn mat_mult_add(n/2,A_12,B_21,C_11);
    spawn mat_mult_add(n/2,A_12,B_22,C_12);
    spawn mat_mult_add(n/2,A_22,B_21,C_21);
    spawn mat_mult_add(n/2,A_22,B_22,C_22);
  }
}
```

Figure 2: Simplified Cilk code for square matrix multiply-add. The code is used as an illustration of the main features of Cilk; this is not the exact code that we use. Our recursive code is about 90 lines long.

3 Multifrontal Sparse Cholesky Factorization

Our multifrontal sparse Cholesky implementation is fairly conventional except for the use of Cilk. The code is explicitly recursive, which allowed us to easily parallelize it using Cilk. In essence, the code factors the matrix using a postorder traversal of the elimination tree. At a vertex v, the code spawns Cilk subroutines that recursively factor the columns associated with the children of v and their descendants. When such a subroutine returns, it triggers the activation of an extend-add operation that updates the frontal matrix of v. These extend-add operations that apply updates from the children of v are performed sequentially using a special Cilk synchronization mechanism called *inlets*.

In this section we first explain the overheads incurred by Cilk programs. We then explain the use of inlets to manage memory efficiently in our code, as well as performance issues arising from the interfaces to the dense BLAS.

3.1 Overheads in a Cilk Program

Cilk programs incur three main kinds of overheads. The first source of overheads is the cost of spawning a Cilk procedure, which is roughly 3 times more expensive than calling a C function [39, Section 5.3]. We address this issue by switching from Cilk to straight C at the bottom levels of the recursion. Since even C function calls are much more expensive than loop iterations, we also switch from recursion to loops on small subproblems, as does almost any high-performance recursive code.

The second source of overheads is Cilk's processor scheduling mechanism. When a processor p_i that participates in the execution of a Cilk program becomes idle, it queries the other processors, and if one of them, say p_j, has work that needs to be performed, p_i "steals" a chunk of work from p_j (the chunk is actually the top-most (oldest) suspended procedure on p_j's execution stack). The other processors are queried by p_i in a random order, which introduces overhead when most of the processors are idle, since this situation leads to many unsuccessful queries and to contention on the work queues of the busy processors. Extensive theoretical and experimental analysis shows that when the program has sufficient parallelism, this overhead is small with very high probability. See [39, Section 2.8] for an

```
cilk matrix* snmf_factor(vertex v) {
  matrix* front = NULL;
  inlet void extend_add_helper(matrix* Fc) {
    if (!front) front = allocate_front(v);
    extend_add(Fc, front);
    free_front(Fc);
  }

  for (c = first_child[v]; c != -1; c = next_child[c]) {
    extend_add_helper( spawn snmf_factor(c) );
  }
  sync; // wait for the children & their extend-adds
  if (!front) front = allocate_front(v); // leaf

  // now add columns of original coefficient matrix to
  // frontal matrix, factor the front, apply updates,
  // copy columns to L, and free columns from front

  return front;
}
```

Figure 3: Simplified Cilk code for the multifrontal Cholesky factorization with inlets to manage memory and synchronize extend-add operations.

overview, and [10, 9] for the technical details.

The third source of overhead is the memory system. Cilk uses a shared-memory model. Most parallel computers, even those with hardware support for shared-memory programming, have memories that are distributed to some extent. In some cases the main memory is shared but each processor uses a private cache, and in other cases the main memory is physically distributed among the processors. Access to a data item by multiple processors (sometimes even to different data items that are stored nearby in memory) often induces communication to communicate values and/or to invalidate or update cached values. This source of overhead can be significant. Our experiments, reported below, show that as long as the memory system is relatively fast, even when physically distributed, as in the case of an Origin 3000 with 16 processors or fewer, this overhead is tolerable. But when the memory system is too slow, as in an Origin 3000 with more than 16 processors, the overheads incurred by our code are intolerable.

3.2 Memory Management and Synchronization using Inlets

Inlets are subroutines that are defined within regular Cilk subroutines (similar to inner functions in Java or to nested procedures in Pascal). An inlet is always called with a first argument that is the return value of a spawned subroutine, as illustrated in Figure 3. The runtime system creates an instance of an inlet only after the spawned subroutine returns. Furthermore, the runtime system ensures that all the inlets of a subroutine instance are performed atomically with respect to one another, and only when the main procedure instance is either at a `spawn` or `sync` operation. This allows us to use inlets as a synchronization mechanism, which ensures that extend-add operations, which all modify a dense matrix associated with the columns of v, are performed sequentially, so the dense matrix is not corrupted. This is all done without using any explicit locks.

The use of inlets also allows our parallel factorization code to exploit a memory-management technique due to Liu [34, 35, 36]. Liu observed that we can actually delay the allocation of the dense frontal matrix associated with vertex v until after the first child of v returns. By cleverly ordering the children of vertices, it is possible to save significant

amounts of memory and to improve the locality of reference. Our sequential code exploits this memory management technique and delays the allocation of a frontal matrix until after the first child returns. In a parallel factorization, we do not know in advance which child will be the first to return. Instead, we check in the inlet that the termination of a child activates whether the frontal matrix of the parent has already been allocated. If not, then this child is the first to return, so the matrix is allocated and initialized. Otherwise, the extend-add simply updates the previously-allocated frontal matrix. Since Cilk's scheduler uses on each processor the normal depth-first C scheduling rule, when only one processor works on v and its descendants, the memory allocation pattern matches the sequential one exactly, and in particular, the frontal matrix of v is allocated after the first-ordered child returns but before any of the other children begin their factorization process. When multiple processors work on the subtree rooted at v, the frontal matrix is allocated after the first child returns, even if it is not the first-ordered child.

3.3 Interfaces to the Dense Subroutines

The sparse Cholesky code can use both traditional BLAS and our new recursive BLAS. Our new BLAS provide two advantages over traditional BLAS: they exploit deep memory hierarchies better and they are parallelized using Cilk. The first advantage affects only large matrices, in particular matrices that do not fit within the level-2 cache of the processor. The second advantage allows a single scheduler, the Cilk scheduler, to manage the parallelism in both the sparse factorization level and the dense BLAS/LAPACK level.

On the other hand, the recursive layout that our BLAS use increases the cost of extend-add operations, since computing the address of the (i, j) element of a frontal matrix becomes more expensive. By carefully implementing data-access operations, we have been able to reduce the total cost of these operations, but they are nonetheless significant.

To explore this issue, we have implemented several variants of the interface code that link the sparse factorization code to the dense kernels. One variant keeps the dense matrices in blocked layout throughout the computation. Two other variants perform extend-add operations on column-major layout but copy the data to and from blocked layout to exploit the blocked-layout dense routines. One of these two variants copies data before and after each call to a dense routine. The other variant copies data once before a sequence of three calls that constitute the factorization of a frontal matrix and copies the data back after the three calls. A final variant keeps the matrices in column-major layout and uses conventional dense routines. Tables 1 and 2 in the next section presents performance results that compare these four variants.

4 Performance

We now present performance results that support our claims regarding the design of the factorization code. We conducted experiments on two different machines. One machine is an SGI Origin 3000 series with 32 R14000 processors running at 500 MHz, and with 8 MB level-2 caches. The Origin 3000 is a ccNUMA machine. Its basic node contains 4 processors and a memory bank in a symmetric configuration (SMP); the combined memory bandwidth of such a node is 3.2 GB/s in full duplex. Four such nodes are connected to a router using 1.6 GB/s full-duplex links. The machine that we used has two routers, which are connected using two 1.6 GB/s full-duplex links. The topology of this machine implies deteriorating bisection bandwidths and increasing latencies when going from four nodes to 16 to 32.

The second computer that we used is a 1.6 GHz Intel Pentium 4 uniprocessor with 256 KB of on-chip cache and 256 MB of 133 MHz SDRAM main memory running Linux. On this machine, both our matrix-multiplication kernel and ATLAS's matrix-multiplication

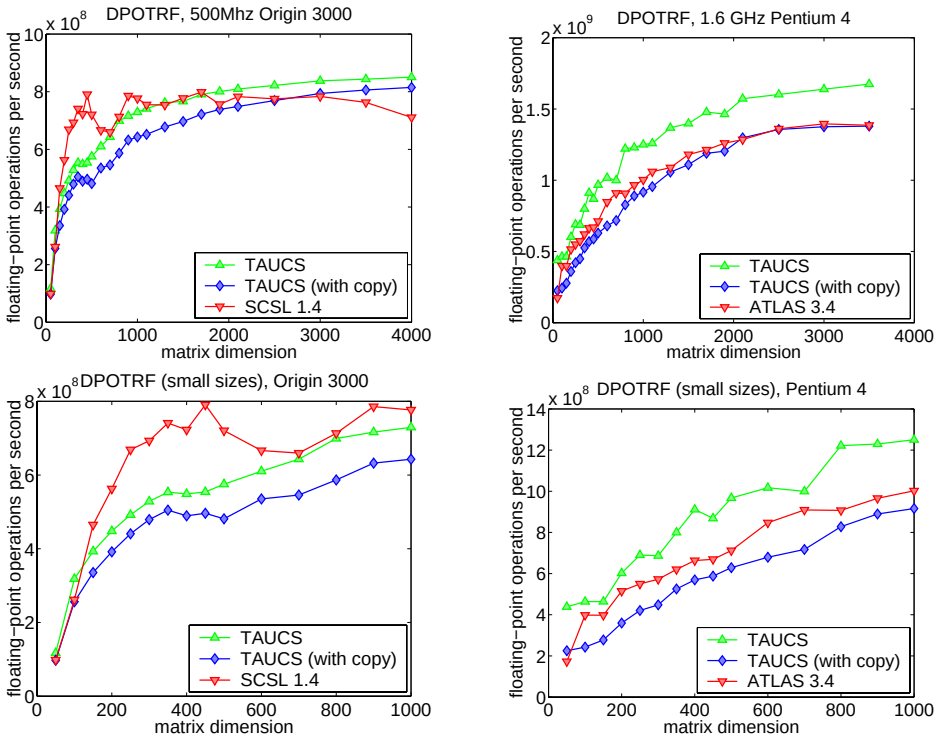

Figure 4: The uniprocessor performance of the new dense Cholesky factorization (denoted TAUCS) compared to the performance of SCSL 1.4, SGI's native BLAS/LAPACK and to ATLAS 3.4. Each plot shows the performance of our new subroutine with recursive layout, the performance of the new subroutine when the input and output are in column-major order (in which case we copy the input and output to and from recursive format), and the performance of SCSL or ATLAS. The graphs on the top row show the performance on matrices whose dimension is up to 4000, and the top row shows the performance on matrices with dimension 1000 or less, to highlight the behavior of the code on smaller matrices.

kernel utilize SSE2 instructions, which allow for two double-precision multiply-add instructions per cycle.

We present results on matrices from the PARASOL test-matrix collection, on one matrix from the Harwell-Boeing collection, and on Laplacians of regular 3-dimensional meshes (i.e., 7-point finite-element discretizations of the model Poisson problem). All the matrices have been reordered using METIS 4.0 with its default settings prior to the factorization.

Figure 4 shows that the uniprocessor performance of our new dense matrix subroutines is competitive and often better than the performance of the vendor library, SGI's SCSL version 1.4, and competitive with the performance of ATLAS (version 3.4). The graphs in the figure also show that even though the cost of copying to and from column-major order is significant, on large matrices our routine outperforms the vendor's routine even when this cost is included. We note that the previous version of SCSL, 1.3, does not perform as well as version 1.4 (see [30]). In particular, version 1.3 slows down significantly when the matrices exceeds the size of the level-2 cache. Neither our code nor version 1.4 suffers from this behavior. The data in the figure shows the performance of dense Cholesky factorization routines, but the performance characteristics of other routines are similar.

In experiments not reported here, we have found that on both the Origin and the Pentium-4 machine, the uniprocessor performance of our new dense codes does not depend on the ordering of blocks. That is, as long as we lay out matrices by block, performance is inde-

Table 1: The performance of the code on a 1.6 GHz Pentium 4 with several variants of the sparse/dense interface, explained in the text. The first matrix is from the Harwell-Boeing collection, the third is from the PARASOL collection, and the two remaining ones are regular 3D meshes. The second column specifies the total number of floating-point operations in the factorization, and columns 3–7 the performance of the different variants in Mflop/s.

	Operation count	no column major	no col-maj. simple SYRK	copy once to blocks	copy for each routine	col-maj. w/ ATLAS
bcsstk32	1.73e9	430	430	343	330	464
30x30x30	2.72e9	686	638	478	466	661
oilpan	3.16e9	483	454	357	349	519
40x40x40	16.00e9	928	878	657	648	845

pendent of the ordering of blocks (recursive vs. block-column-major). It appears that the spatial locality that laying out matrices by block provides is sufficient, and that the additional coarser-grained spatial locality that we achieve by recursive layout of blocks does not contribute significantly to performance.

Tables 1 and 2 demonstrate the performance differences between different sparse/dense interfaces. The tables show the performance of several variants of the sparse/dense interface. The *no-column-major* column describes the performance of a code that keeps the frontal matrices in the indirect blocked layout; the extend-add operations access matrix elements using double indirection. The *no-column-major-simple-SYRK* is a similar code, but which uses a plain SYRK dense kernel rather than an automatically-generated and SSE2-enhanced one (all the other variants use an automatically-generated and SSE2-enhanced SYRK kernel). The *copy-once-to-blocks* code copies from column major to blocked layout before the factorization of each frontal matrix, which consists of 3 dense calls. The *copy-for-each-routine* code copies matrices before and after each of the 3 dense calls. The *column-major-with-ATLAS* code uses a column-major layout for the frontal matrices and calls ATLAS.

The data shows that blocked layouts are beneficial. On the Pentium 4, the blocked layout and column-major layout with ATLAS result in similar performance. On the Origin, blocked layout are clearly more effective than column-major layout with SCSL. On both machines, when we use blocked layout, performing extend-adds directly on the blocked layout is preferable to copying between blocked layouts and column-major layouts.

Table 1 also shows that the code benefits significantly from a highly optimized dense SYRK kernel. The "simple" SYRK kernel uses a fairly simple implementation using three nested loops. The optimized kernel uses SSE2 hand-coded kernel for 80-by-80 blocks and automatically-generated unrolled kernels for smaller block sizes.

Figure 5 shows that our new dense matrix routines scale well unless memory access times vary too widely. The graphs show the performance of the dense Cholesky factorization routines on a 32-processor SGI Origin 3000 machine. The entire 32-processor machine was dedicated to these experiments. On this machine, up to 16 processors can communicate through a single router. When more than 16 processors participate in a computation, some of the memory accesses must go through a link between two routers, which slows down the accesses. The graphs show that when 16 or fewer processors are used, our new code performs similarly or better than SCSL. The performance difference is especially significant on 12–16 processors. But when 32 processors are used, the slower memory accesses slow our code down more than it slows SCSL (but even SCSL slows down relative to its 16-processors performance). We suspect that the slowdown is mostly due to the fact that we allocate the entire matrix in one memory-allocation call (so all the data resides on a single 4-processor node) and do not use any memory placement or migration primitives, which would render the code less portable and more machine specific.

Figures 7 and 8 show that our overall sparse Cholesky code scales well with up to 16

Table 2: The performance of the code on a single processor of the Origin 3000 with several variants of the sparse/dense interface. All the matrices except the last are from the PARASOL collection, last is a regular 3D mesh. The columns are the same as in Table 1. There is no *no-column-major-simple-SYRK* column in this case since on the Origin none of our kernels are automatically generated. The last column shows the performance with column-major layout and the SCSL BLAS.

	Operation count	no column major	no col-maj. simple SYRK	copy once to blocks	copy for each routine	col-maj. w/ SCSL
crankseg1	35e9	620		536	528	616
thread	38e9	681		589	583	612
shipsec1	42e9	629		549	541	606
crankseg2	48e9	631		543	536	614
bmwcra1	66e9	605		522	514	593
50x50x50	63e9	686		592	588	552

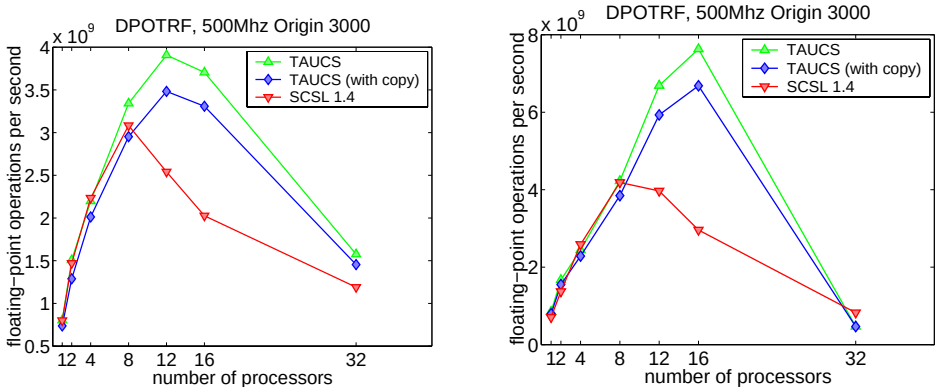

Figure 5: The parallel performance of the new dense Cholesky factorization on matrices of order 2000 (left) and 4000 (right).

processors. Note that our earlier code [30] could not effectively exploit that many processors. The graphs in Figure 7 also show the benefit of parallelizing the sparse and dense layers of the solver using the same parallelization mechanism. The code speeds up best (circles) when both the sparse multifrontal code and the dense routines are parallelized using Cilk. When we limit parallelism to either the sparse layer or to the dense routines (triangles), performance drops significantly. Figure 8, which describes the performance of our best code, shows that our code achieves not only effective speedups but also high absolute performance.

Figure 6 explains the data in Figure 7. The figure shows that the vast majority of fronts in the multifrontal algorithm are small, but most of the floating-point arithmetic is performed on large fronts. On the four test matrices analyzed here, about 80% of the fronts have fewer than 100 columns, but 80% of the arithmetic is performed on fronts with 500 columns or more. This observation is not new [3, 23, 27, 32, 37]; we repeat it here to explain Figure 7. Parallelizing dense operations is important mainly in order to speed up operations on large fronts. Processing multiple fronts in parallel is important since there are numerous small fronts whose processing generates significant overhead, which should be parallelized.

Figure 9 shows that on an Origin 3000 with up to 16 processors, our code is competitive with two other parallel sparse Cholesky factorization codes. In fact, our code is usualy faster. The two other codes are MUMPS [3, 4] and PSPASES [23, 31, 32], both of which

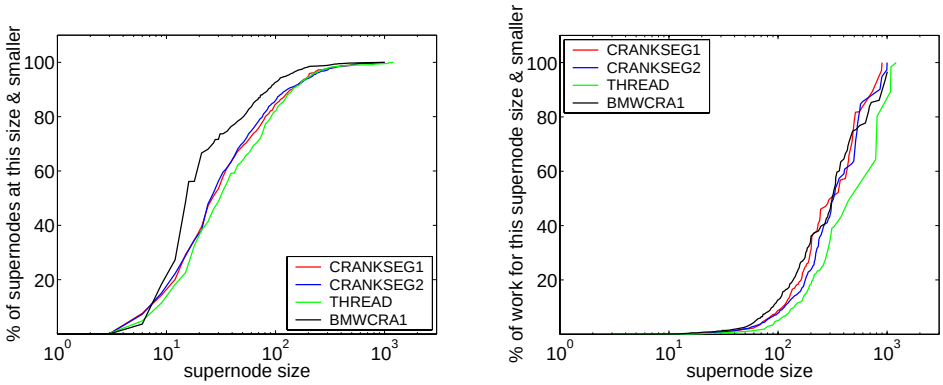

Figure 6: The distribution of front sizes (left) and the distribution of work according to front sizes (right).

are distributed-memory parallel codes. In this experiment we used the latest versions of the codes available at the time of the experiment: MUMPS version 4.1.6, released March 2000, and PSPASES version 1.0.3, dated May 9, 1999. We used SGI's implementation of MPI for both codes, and BLAS from SGI's SCSL version 1.4. We also used ScaLAPACK version 1.7 [8, 11] and MPI-BLACS [13] version 1.1patch03 to run MUMPS. We used exactly the same orderings for all the runs, orderings obtained from METIS. In PSPASES, we timed the routine DPSPACEN, which is the numerical factorization phase, and in MUMPS, we timed the routine MUMPS with parameters that instruct the routine to use our precomputed ordering and that inform the routine that the input matrix is symmetric positive definite. Due to different supernode amalgamation strategies, the actual number of floating-point operations performed by the other codes are slightly different than the number performed by TAUCS, even though the ordering is the same. MUMPS performed up to 0.46% more floating-point operations. PSPASES performed between 5.11% fewer floating-point operations to 16.86% more. In PSPASES the amalgamation strategy depends on the number of processors; in TAUCS and MUMPS it is not.

The data in Figure 9 clearly shows that our code is competitive with other parallel sparse Cholesky codes, and usually faster. This indicates that the overheads hidden in the recursive Cilk implementation are smaller or comparable to the overheads of other parallel environments. We cannot conclude much more than this from the data, since TAUCS is structurally different from the two other codes. MUMPS and PSPASES are distributed-memory codes that use MPI, unlike TAUCS, which relies on shared memory. The memory system of the Origin 3000 is physically distributed, but every processor can access all the memory in a cache coherent and sequentially consistent manner. Using MPI on this machine obviously introduces overhead, which is a disadvantage, but avoids communication induced by the coherency and consistency protocols. Therefore, the poorer performance of PSPASES and MUMPS relative to TAUCS can be explained by the message-passing overheads. However, because PSPASES and MUMPS avoid implicit communication induced by the consistency protocol, we expect them to perform well on more than 16 processors, whereas TAUCS slows down on 32 processors, as indicated in Figure 5. Obviously, the use of MPI also allows PSPASES and MUMPS to run on systems without shared-memory support.

The comparisons are limited to PSPASES and MUMPS since these were the only parallel sparse direct symmetric-positive-definite codes whose sources we could obtain.

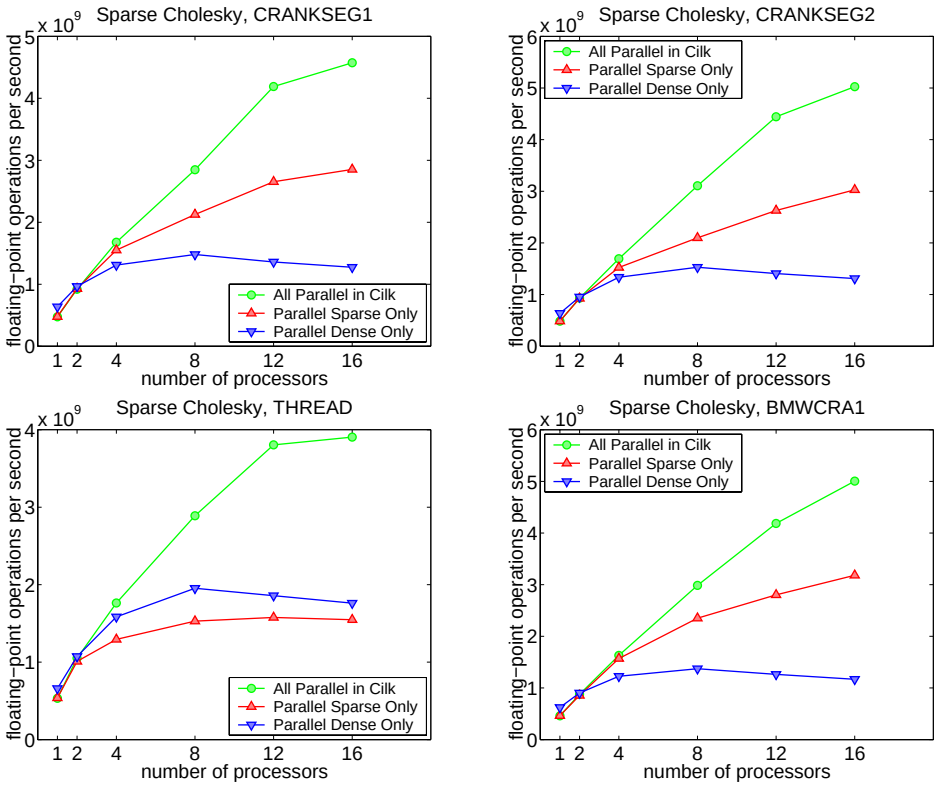

Figure 7: The performance on the Origin 3000 of the parallel multifrontal sparse Cholesky in Cilk on four PARASOL test matrices. The data only shows the performance of the numeric factorization. The three plots in each graph show the performance of our Cilk-parallel sparse solver with our Cilk-parallel dense routines, the performance of our Cilk-parallel sparse solver with sequential recursive dense routines, and of our sequential sparse solver with SCSL's parallel dense routines. The TAUCS dense routines that we used in these experiments copy to and from blocked layout before and after each dense call. Replacing TAUCS's sequential dense routines with SCSL's dense routines does not significantly change the performance of the overall code ("Parallel Sparse Only" plots).

5 Conclusions

Our research addresses several fundamental issues: Can blocked and possibly recursive data layouts be used effectively in a large software project? In other words, is the overhead of indexing the (i, j) element of a matrix acceptable in a code that needs to do that frequently (e.g., a multifrontal code that performs extend-add operations on frontal matrices)? How much can we benefit from writing additional automatic kernel generators and other kernel optimizations? Can Cilk manage parallelism effectively in a multilevel library that exposes parallelism at both the upper sparse layer and the lower dense layer?

Before we address each of these questions, we point out that the performance of our code relative to other parallel sparse Cholesky codes indicates that our experiments, and hence our conclusions, are valid. With up to 16 processors, our shared-memory code outperforms two other parallel distributed-memory codes, MUMPS and PSPASES. Although the distributed memory codes incur higher communication overhead, they avoid unnecessary communication that may be induced by the hardware's shared-memory support mechanisms. Therefore, we believe that the comparison is essentially fair, and indicates that our code performs at state-of-the-art levels. (We acknowledge that our code, like other

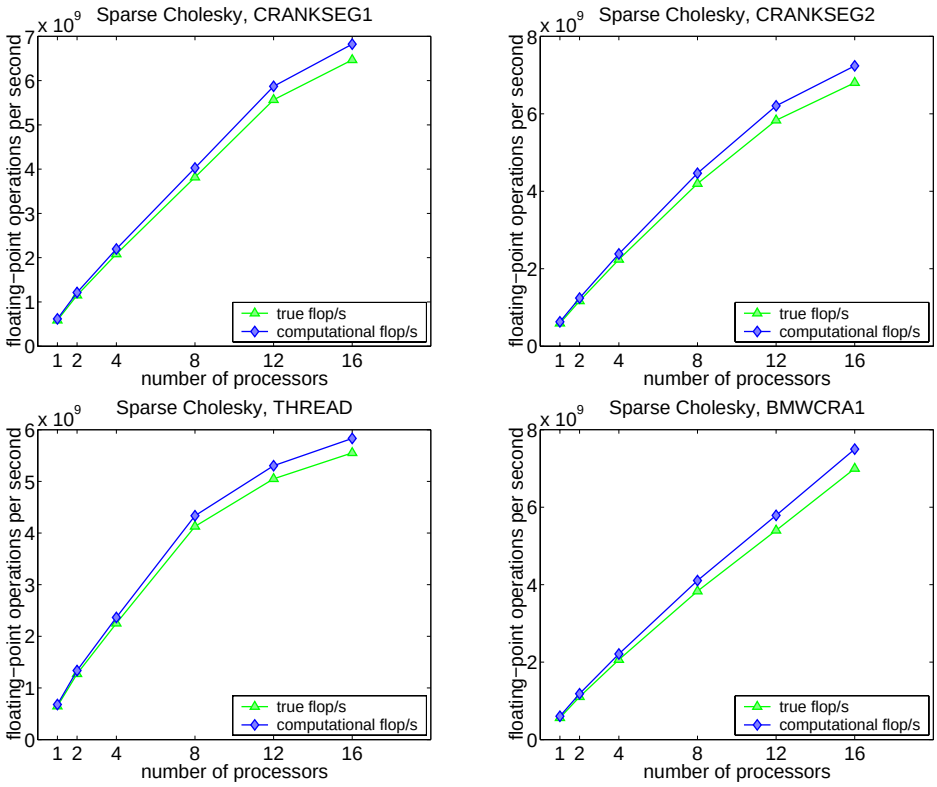

Figure 8: The performance and speedups of the sparse Cholesky code on the Origin 3000. The data only shows the performance of the numeric factorization. The code is the *no-column-major* code (third column in table 2). Each plot shows the performance in computational flop/s, which refers to the number of floating-point operations that the code actually performs, and to the true flop/s, which excludes operations on structural zeros. The code performs some operations on zeros in order to create larger supernodes [6].

shared-memory codes, is sensitive to the performance characteristics of the shared memory system, and that when the memory is too slow our code's performance degrades.) Furthermore, the sequential version of our code has been incorporated by the developers of MATLAB [40] into the next release as the default sparse positive definite solver that the linear-solve operator calls. While this implies nothing concerning the parallel code, it does suggest that the sequential code, which is essentially identical to the parallel one except for the use of C function calls instead of Cilk spawns, is efficient. The code will replace a slower sparse left-looking column oriented code [22].

Our experiments clearly demonstrate that the blocked-recursive data layout does not lower performance and can significantly improve performance (on the Origin). In addition, the blocked layout is clearly scalable up to at least 16 processors. Although we have not performed scalability experiments with column-major layouts, we believe that achieving scalability with such layouts is more difficult. On the other hand, we have not found a significant difference between the performance of column-major layout of the blocks and a recursive layout of the blocks. This finding suggests that the recursive block layouts may not be as important as suggested by [24]. The novelty in our results, compared to earlier work on blocked and recursive layouts [24], is that we have demonstrated the effectiveness of these layouts in a code that (1) often operates on small matrices, and (2) that performs random accesses to matrix elements often.

Our experiments also clearly demonstrate that additional highly optimized kernels do

improve performance significantly. The obvious reason for this is that in a sparse Cholesky code, a significant amount of work is performed in the context of dense operations on small and medium matrices. As a result, a highly optimized matrix-multiplication kernel (GEMM) does not provide near-optimal performance. We expect that additional optimized kernels for POTRF and TRSM would further improve performance. In addition, it is clear that other aspects of the dense kernels, such as data-copying policies and performance on small problems also impact the performance of sparse codes.

So far, it seems clear that the Cilk can help manage parallelism and simplify code in complex parallel codes. In particular, Cilk allows us to easily exploit nested parallelism. The fact that both the elimination-tree parallelism and the dense parallelism are expressed in Cilk allows the Cilk scheduler to effectively manage processors. However, the slowdowns on 32 processors suggest that Cilk codes should manage and place memory carefully on ccNUMA machines. We also note that we have found that the overhead of the Cilk scheduler is indeed small, but higher than we had expected. We found that avoiding spawning Cilk procedures on small problem instances, calling sequential C procedures instead, can improve performance. This annoyance makes Cilk programming more difficult than it should be.

The scalability and absolute performance of our code, as demonstrated by our experiments, demonstrate the effectiveness of dynamic scheduling, at least on share-memory machines. Many parallel sparse factorization codes are statically scheduled, where a specialized scheduler decides in advance which processor or processors should perform which vertices in the supernodal elimination tree. Our experiments demonstrate that a dynamically-scheduled factorization can achieve high performance and scalability. The advantage of dynamic scheduling is that it can adapt better to changes in the computing environments (processors being used or freed by other users) and to parallelism in the application itself.

Our research not only addresses fundamental questions, but it also aims to provide users of mathematical software with state-of-the-art high-performance implementations of widely-used algorithms. A stable version of our sequential code (both real and complex) is freely available at `www.tau.ac.il/~stoledo/taucs`. This version includes the sequential multifrontal supernodal solver. The parallel version with the recursive BLAS and the parallelized Cilk codes is available from us upon request; we plan to incorporate the parallel code into the standard distribution of TAUCS once we clean it up.

References

[1] R. C. Agarwal, F. G. Gustavson, and M. Zubair. Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. *IBM Journal of Research and Development*, 38(5):563–576, 1994.

[2] R. C. Agarwal, F. G. Gustavson, and M. Zubair. Improving performance of linear algebra algorithms for dense matrices using algorithmic prefetch. *IBM Journal of Research and Development*, 38(3):265–275, 1994.

[3] P. R. Amestoy, I. S. Duff, J. Koster, and J. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23:15–41, 2001.

[4] P. R. Amestoy, I. S. Duff, J. L'Excellent, J. Koster, and M. Tuma. MUltifrontal Massively Parallel Solver (MUMPS version 4.1), specification sheets. Available online from `http://www.enseeiht.fr/lima/apo/MUMPS/doc.html`, Mar. 2000.

[5] B. S. Andersen, J. Waśniewski, and F. G. Gustavson. A recursive formulation of cholesky factorization of a matrix in packed storage. *ACM Transactions on Mathematical Software*, 27:214–244, June 2001.

[6] C. Ashcraft and R. Grimes. The influence of relaxed supernode partitions on the multifrontal method. *ACM Transactions on Mathematical Software*, 15(4):291–309, 1989.

[7] J. Bilmes, K. Asanovic, C. W. Chin, and J. Demmel. Optimizing matrix multiply using PHIPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the International Conference on Supercomputing*, Vienna, Austria, 1997.

[8] 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, PA, 1997. Also available online from `http://www.netlib.org`.

[9] R. D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, Sept. 1995.

[10] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 356–368, Santa Fe, New Mexico, Nov. 1994. IEEE Computer Society Press.

[11] J. Choi, J. Dongarra, R. Pozo, and D. Walker. ScaLAPACK: A scalable linear algebra for distributed memory concurrent computers. In *Proceedings of the 4th Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127, 1992. Also available as University of Tennessee Technical Report CS-92-181.

[12] Compaq. Compaq extended math library (CXML). Software and documuntation available online from `http://www.compaq.com/math/`, 2001.

[13] J. Dongarra and R. Whaley. A user's guide to the blacs v1.0. Technical Report UT CS-95-281, LAPACK Working Note 94, University of Tennessee, 1995. Available online from `http://www.netlib.org/blacs/`.

[14] J. J. Dongarra, J. D. Cruz, S. Hammarling, and I. Duff. Algorithm 679: A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):18–28, 1990.

[15] J. J. Dongarra, J. D. Cruz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[16] J. J. Dongarra and P. Raghavan. A new recursive implementation of sparse Cholesky factorization. In *Proceedings of the 16th IMACS World Congress 2000 on Scientific Computing, Applications, Mathematics, and Simulation*, Lausanne, Switzerland, Aug. 2000.

[17] I. Duff and J. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.

[18] I. S. Duff. Parallel implementation of multifrontal schemes. *Parallel Computing*, 3, 1986.

[19] E. Elmroth and F. Gustavson. Applying recursion to serial and parallel QR factorization leads to better performance. *IBM Journal of Research and Development*, 44(4):605–624, 2000.

[20] E. Elmroth and F. G. Gustavson. A faster and simpler recursive algorithm for the LAPACK routine DGELS. *BIT*, 41:936–949, 2001.

[21] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. *ACM SIGPLAN Notices*, 33(5):212–223, 1998.

[22] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.

[23] A. Gupta, G. Karypis, and V. Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):502–520, 1997.

[24] F. Gustavson, A. Henriksson, I. Jonsson, B. Kågström, and P. Ling. Recursive blocked data formats and BLAS's for dense linear algebra algorithms. In B. Kågström, J. Dongarra, E. Elmroth, and J. Waśniewski, editors, *Proceedings of the 4th International Workshop on Applied Parallel Computing and Large Scale Scientific and Industrial Problems (PARA '98)*, number 1541 in Lecture Notes in Computer Science Number, pages 574–578, Ume, Sweden, June 1998. Springer.

[25] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41:737–755, Nov. 1997.

[26] F. G. Gustavson and I. Jonsson. Minimal-storage high-performance Cholesky factorization via blocking and recursion. *IBM Journal of Research and Development*, 44:823–850, Nov. 2000.

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

[28] IBM. Engineering and scientific subroutine library (SCSL). Software and documuntation available online from `http://www-1.ibm.com/servers/eservers/pseries/software/sp/essl.html`, 2001.

[29] Intel. Math kernel library (MKL). Software and documuntation available online from `http://www.intel.com/software/products/mkl/`, 2001.

[30] D. Irony, G. Shklarski, and S. Toledo. Parallel and fully recursive multifrontal supernodal sparse cholesky. In *Proceedings of the International Conference on Computational Science (ICCS 2002)*, pages 335–344 of Part II, Amsterdam, Apr. 2002.

[31] M. Joshi, A. Gupta, F. Gustavson, G. Karypis, and V. Kumar. PSPASES: Scalable parallel direct solver library for sparse symmetric positive definite linear systems: User's manual for version 1.0.3. Technical Report TR 97-059, Department of Computer Science, University of Minnesota, 1997, revised 1999.

[32] M. Joshi, A. Gupta, F. Gustavson, G. Karypis, and V. Kumar. PSPASES: An efficient and scalable parallel sparse direct solver. Unpublished article, presented at the International Workshop on Frontiers of Parallel Numerical Computations and Applications (Frontiers'99), Annapolis, Maryland, February 1999. Available online from `http://www-users.cs.umn.edu/~mjoshi`, 1999.

[33] C. Kamath, R. Ho, and D. P. Manley. DXML: a high-performance scientific subroutine library. *Digital Technical Journal*, 6(3):44–56, 1994.

[34] J. W. H. Liu. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Transactions on Mathematical Software*, 12(3):249–264, 1986.

[35] J. W. H. Liu. The multifrontal method and paging in sparse Cholesky factorization. *ACM Transactions on Mathematical Software*, 15(4):310–325, 1989.

[36] J. W. H. Liu. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Review*, 34(1):82–109, 1992.

[37] O. Schenk and K. Gärtner. Sparse factorization with two-level scheduling in PARADISO. In *Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computing*, page 10 pages on CDROM, Portsmouth, Virginia, Mar. 2001.

[38] SGI. Scientific computing software library (SCSL). Software and documentation available online from from `http://www.sgi.com/software/scsl.html`, 1993–2002.

[39] Supercomputing Technologies Group, MIT Laboratory for Computer Science, Cambridge, MA. *Cilk-5.3.2 Reference Manual*, Nov. 2001. Available online at `http://supertech.lcs.mit.edu/cilk`.

[40] The MathWorks, Inc, Natick, MA. MATLAB *Reference Guide*, Aug. 1992.

[41] S. Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, 1997.

[42] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. Technical report, Computer Science Department, University Of Tennessee, 1998. available online at www.netlib.org/atlas.

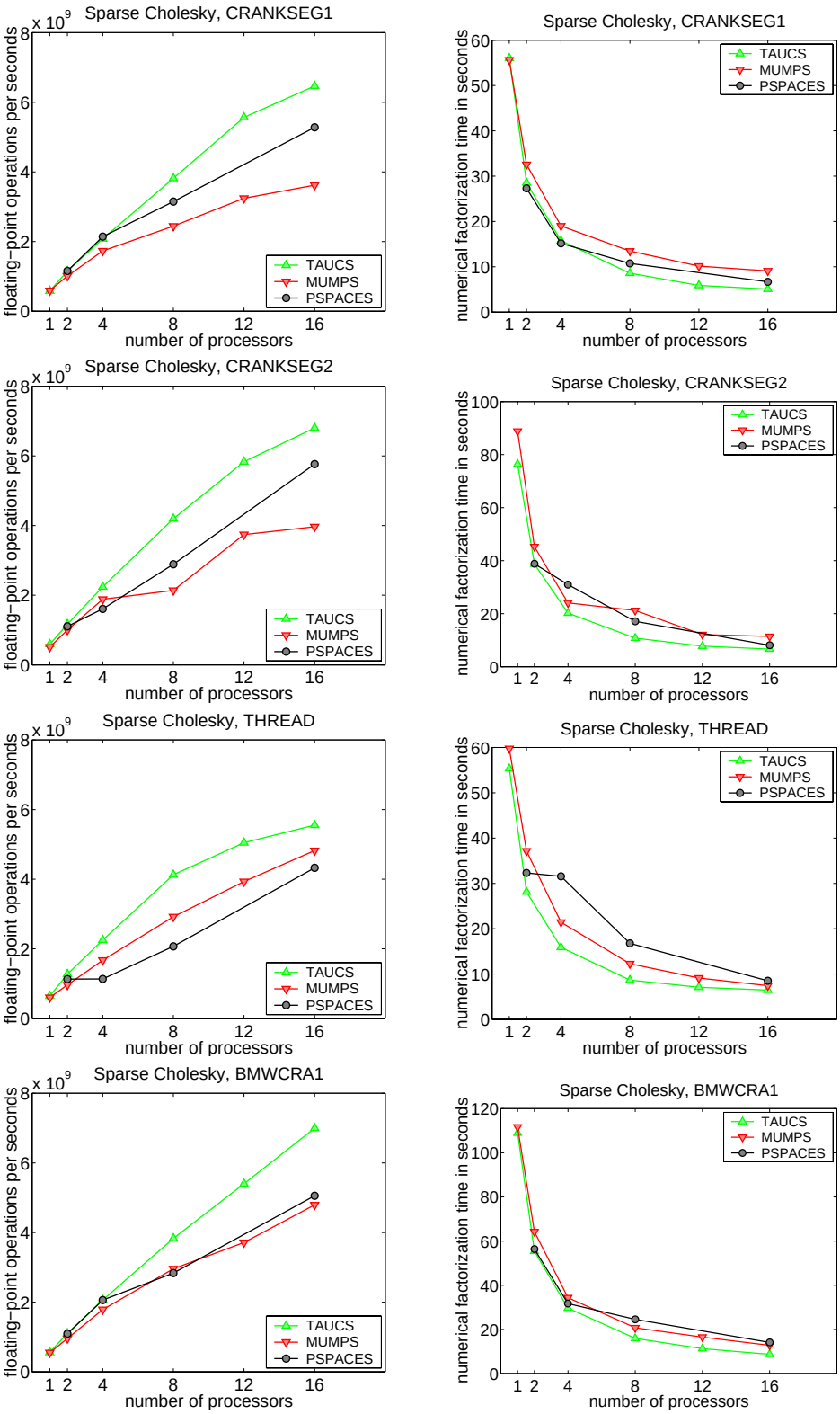

Figure 9: The performance and speedups of three sparse Cholesky codes on the Origin 3000. The data only shows the performance of the numeric factorization. The plots on the left show performance in true flop/s, which excludes operations on structural zeros. The plots on the right show running times in seconds. The number of processors that PSPASES can use must be a power of 2, and at least 2.