Parallel and Fully Recursive Multifrontal Supernodal Sparse Cholesky*

Dror Irony, Gil Shklarski, and Sivan Toledo

School of Computer Science, Tel-Aviv Univsity
http://www.tau.ac.il/~stoledo

Abstract We describe the design, implementation, and performance of a new parallel sparse Cholesky factorization code. The code uses a supernodal 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.

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 (see, e.g., [20]). The multifrontal supernodal 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 (unknowns in the linear system). The method works by factoring the columns associated with all the columns associated with proper decsendants 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 [7, 6]. The ability to exploit the dense BLAS and the low symbolic 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 [26] and BLAS produced by computer vendors [1, 2, 5, 15–17, 23].

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:

* 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.

P.M.A. Sloot et al. (Eds.): ICCS 2002, LNCS 2330, pp. 335−344, 2002.

- 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 [3, 9, 10, 13, 12, 14, 25, 26]. A similar approach was recently proposed by Dongarra and Raghavan for a non-multifrontal sparse Cholesky method [8]. 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 [24, 11], 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. [12] 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.

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 [12]. 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.

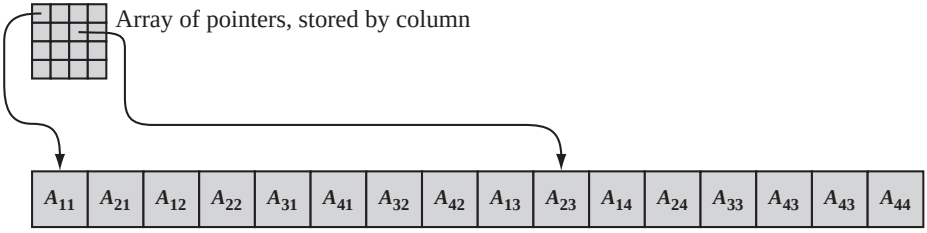

Figure 1. The use of indirection in the layout of matrices by block. The matrix is represented by an array of pointers to blocks (actually by an array of structures that contain pointers to blocks).

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.

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 [4]. 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 compiler feature called the loop-nest optimizer delivers better performance at smaller code size than our automatically-generated kernels.

We currently have kernel generators for two BLAS routines: DGEMM and DSYRK, and we plan to produce two more, for DTRSM, and DPOTRF. 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 on 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 Figure 2 (and explained fully in [24]) 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. The full paper will contain a full example. 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.

3 Multifrontal Supernodal Sparse Cholesky Factorization

Our multifrontal supernodal 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*.

```
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 2. Simplified Cilk code for the supernodal multifrontal Cholesky factorization with inlets to manage memory and synchronize extend-add operations.

3.1 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 2. 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 [18–20]. 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.

3.2 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 is not significant in the sparse factorization code, because it affects only large matrices, in particular matrices that do not fit within the level-2 cache of the processor. Since many of the dense matrices that the sparse factorization code handles are relatively small, this issue is not significant. The second advantage is significant, as we show below, since it 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 macros (and functions that the compiler can inline on the SGI platform), we have been able to reduce the total cost of these operations, but they are nonetheless significant.

4 Performance

Figure 3 shows that the uniprocessor performance of our new dense matrix subroutines is competitive and often better than the performance of state-of-the-art vendor libraries (SGI's SCSL version 1.3 in this case). The graphs show the performance of the vendor's routine declines when the matrix grows above the size of the processor's level-2 cache, but that the performance of our routine does not. 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. Measurements on Pentium-III machines running Linux, now shown here due to lack of space, indicate that our new routines are similar in performance and sometimes faster than ATLAS. They are faster especially on small matrices, where the benefit of using multiple automatically-generated kernels is greatest. 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 the Origin 3000, 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 independent 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.

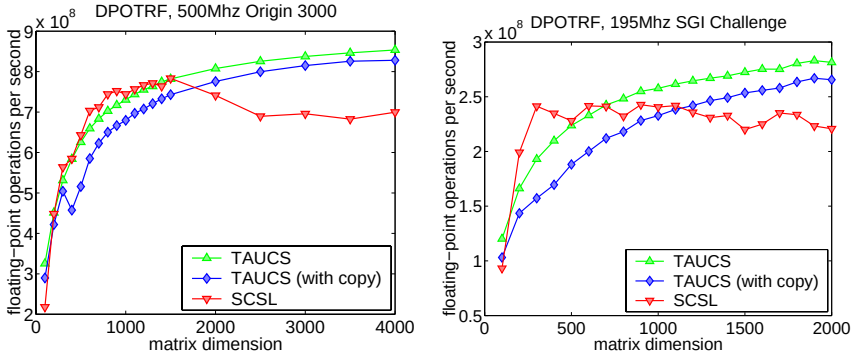

Figure 3. The uniprocessor performance of the new dense Cholesky factorization (denoted TAUCS) compared to the performance of SCSL, SGI's native BLAS/LAPACK. Each plot shows the performance of our new subroutine with recursive layout, the performance of the new subroutine when the input and output is in column-major order (in which case we copy the input and output to/from recursive format), and the performance of SCSL.

Figure 4 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 16 processors. But when 32 processors are used, the slower memory accesses slow our code down much 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.

Figure 5 (left) shows that our overall sparse Cholesky code scales well with up to 8 processors. The code does not speed up further when it uses 16 processors, but it does not slow down either. The graph also shows the benefit of parallelizing the sparse and dense layers of the solver using the same parallelization mechanism. The code speeds up best (green 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 (red/blue triangles), performance drops significantly.

We acknowledge that the absolute performance of the sparse factorization code, as shown in Figure 5 (right), appears to be lower than that of state-of-the-art sparse Cholesky codes. We have not measured the performance of the code relative to the performance of other codes, but from indirect comparisons it appears that the code is slower than PARADISO [22], for example. We have received reports that the code is

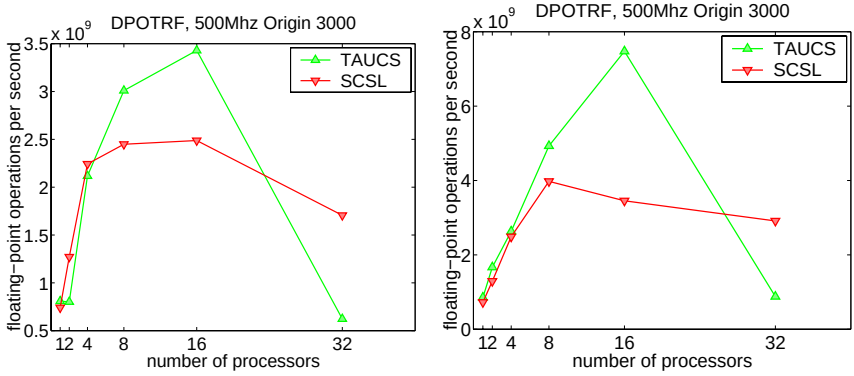

Figure 4. The parallel performance of the new dense Cholesky factorization (without data copying) on matrices of dimension 2000 (left) and 4000 (right). The lack of speedup on 2 processors on the 2000-by-2000 matrix appears to be a Cilk problem, which we have not been able to track down yet.

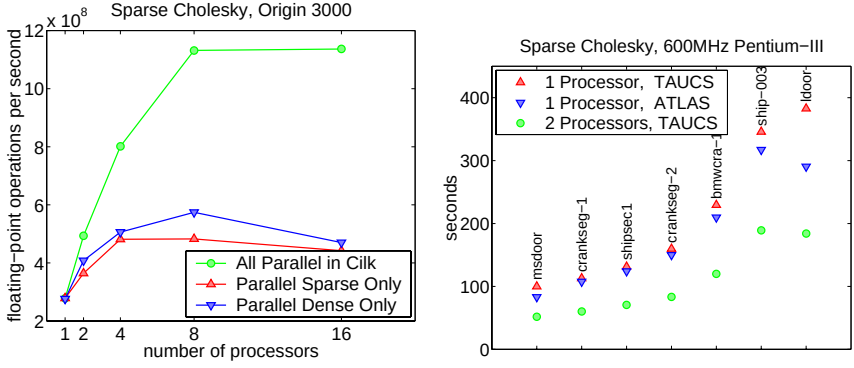

Figure 5. The performance of the parallel supernodal multifrontal sparse Cholesky in Cilk. The graph on the left shows the performance of the numerical factorization phase of the factorization on a 500 MHz Origin 3000 machine, on a matrix that represents a regular 40-by-40-by-40 mesh. The symbolic factorization phase degrades performance by less than 5% relative to the performance shown here. The three plots 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 plot on the right shows the performance on a dual-processor Pentium-III machine for some of the PARASOL test matrices. Performance is given in seconds to allow comparisons with other solvers. The plot shows the performance of our code with recursive dense routines on 1 and 2 processors, and also the performance of our sparse code when it calls ATLAS.

significantly faster on a uniprocessor than the sequential code of Ng and Peyton [21]. The sequential code is 5-6 times faster than Matlab 6's sparse `chol`, which is written in C but is not supernodal. We believe that using relaxed supernodes will bring our code to

a performance level similar to that of the fastest codes (currently the code only exploits so-called fundamental supernodes). We plan to implement this enhancement in the near future.

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? We clearly benefit from the DSYRK generator, but will additional ones help, e.g. for extend-add? Can Cilk manage parallelism effectively in a multilevel library that exposes parallelism at both the upper sparse layer and the lower dense layer?

So far, it seems clear that the Cilk can help manage parallelism and simplify code in complex parallel codes. However, the slowdowns on 32 processors suggest that Cilk codes should manage and place memory carefully on ccNUMA machines. It also appears that additional optimized kernels are beneficial, particularly since many of the dense matrices that sparse solvers handle are quite small. It is not clear yet whether the improved spatial locality of blocked layouts justifies the additional overhead of random accesses to matrix elements.

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 is freely available at `www.tau.ac.il/~stoledo/taucs`. This version includes the sequential multifrontal supernodal solver, but not the recursive BLAS or the parallelized Cilk codes. We plan to freely distribute all the codes once we reach a stable version that includes them.

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. 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.
4. 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.
5. Compaq. Compaq extended math library (CXML). Software and documuntation available online from `http://www.compaq.com/math/`, 2001.
6. 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.

7. 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.
8. 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.
9. 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.
10. E. Elmroth and F. G. Gustavson. A faster and simpler recursive algorithm for the LAPACK routine DGELS. *BIT*, 41:936–949, 2001.
11. 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.
12. 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.
13. 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.
14. 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.
15. 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.
16. Intel. Math kernel library (MKL). Software and documuntation available online from `http://www.intel.com/software/products/mkl/`, 2001.
17. C. Kamath, R. Ho, and D. P. Manley. DXML: a high-performance scientific subroutine library. *Digital Technical Journal*, 6(3):44–56, 1994.
18. 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.
19. J. W. H. Liu. The multifrontal method and paging in sparse Cholesky factorization. *ACM Transactions on Mathematical Software*, 15(4):310–325, 1989.
20. J. W. H. Liu. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Review*, 34(1):82–109, 1992.
21. E. G. Ng and B. W. Peyton. Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM Journal on Scientific Computing*, 14(5):1034–1056, 1993.
22. 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.
23. SGI. Scientific computing software library (SCSL). Software and documuntation available online from from `http://www.sgi.com/software/scsl.html`, 1993–2001.
24. Supercomputing Technologies Group, MIT Laboratory for Computer Science, Cambridge, MA. *Cilk-5.3 Reference Manual*, June 2000. Available online at `http://supertech.lcs.mit.edu/cilk`.
25. S. Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, 1997.
26. 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.