CONF-9206185—2

DE92 016829

PARALLEL SPARSE CHOLESKY FACTORIZATION ALGORITHMS FOR SHARED-MEMORY MULTIPROCESSOR SYSTEMS*

Edward Rothberg and Anoop Gupta
Dept. of Computer Science
Stanford University
Stanford, CA 94305
and
Esmond Ng and Barry Peyton
Mathematical Sciences Section
P.O. Box 2008, Building 6012
Oak Ridge National Laboratory
Oak Ridge, TN 37831-6367

*Research sponsored by the Dept. of Computer Science, Stanford University, and the Applied Mathematical Sciences Research Program, Office of Energy Research, U.S. Department of Energy under contract DE-AC05-840R21400 with the Martin Marietta Energy Systems, Inc.

PARALLEL SPARSE CHOLESKY FACTORIZATION ALGORITHMS FOR SHARED-MEMORY MULTIPROCESSOR SYSTEMS

Edward Rothberg and Anoop Gupta
Department of Computer Science
Stanford University
Stanford, CA 94305

Esmond Ng and Barry Peyton
Mathematical Sciences Section
Oak Ridge National Laboratory
P.O. Box 2008, Oak Ridge, TN 37831-6367

Abstract

In this paper, we consider sparse Cholesky factorization on a multiprocessor system that possesses a globally shared memory. Our algorithm is a parallel version of a serial blocked left-looking factorization algorithm described in [12,17]. Unlike previous parallel left-looking algorithms, the new algorithm uses a matrix-matrix multiplication operation to implement its computationally intensive primitives. Consequently, careful implementation of these primitives enables extensive reuse of data in cache for most realistic problems, and thus reduces the volume of traffic to and from main memory. Reducing memory traffic is crucial on many shared-memory multiprocessors because the interconnect to main memory is often a serious bottleneck. We shall compare the performance of the parallel blocked algorithm with an earlier parallel left-looking algorithm studied in [13].

1 Introduction

In this paper we introduce a simple parallel blocked sparse Cholesky algorithm designed for shared-memory multiprocessors with a modest number of processors. The algorithm is particularly appropriate for shared-memory multiprocessors that:

- are based on one of the fast new microprocessors which rely heavily on the reuse of data in cache to achieve high computational rates, and/or
- do *not* have a data bus that is capable of meeting near-peak demand from all processors at once.

The rapid increase in computational speeds seen in recent years has not been matched by commensurate increases in data access speeds. Consequently, in many instances data bandwidth is the critical resource that limits performance. In particular, many current and forthcoming shared-memory multiprocessors have at least one of the two characteristics listed above. It appears moreover that this state of affairs will continue for some time to come.

The parallel factorization algorithm described in this paper is based on a sequential left-looking blocked sparse Cholesky factorization algorithm studied by Ng and Peyton [12] and Rothberg and Gupta [17]. The basic task around which the algorithm is built is a *supernode block column task*, which computes all the columns belonging to a particular *supernode*. (A supernode is a set of contiguous factor columns that share the same external sparsity structure.) The supernode block column task is organized around computational primitives that update every member of one set of columns with a multiple of each member of another set of columns. The matrix-matrix multiplication operation used to implement these primitives enables the reuse of "local data", thereby reducing data traffic to and from more distant parts of the memory hierarchy.

Straightforward parallelization of this sequential method unfortunately does not immediately lead to an efficient parallel method. The supernode block column task results in a parallel task granularity that is far too large during the middle and later portions of the computation to keep the processors busy and the load evenly balanced, even when the number of processors is quite small. We thus split the columns of each supernode into subsets of contiguous columns, which we shall call *panels*. The block column task upon which our parallel algorithm is based is the computation of all columns that belong to a particular panel. In selecting the panels, our goal is to include enough columns in each to maintain good data reuse and low bus traffic, while avoiding the inclusion of too many columns, which leads to deteriorating performance due to loss of concurrency in the computation.

Section 2 briefly discusses background material and previous work in this area. Section 3 contains an informal discussion of the parallel panel-based sparse Cholesky factorization algorithm. Timing results on an SGI 4D/380 are presented in Section 4. Finally, Section 5 contains a few concluding remarks.

2 Background and previous work

Let A be a symmetric positive definite matrix. The Cholesky factor of A, denoted by L, is a lower triangular matrix whose main diagonal is positive and for which $A = LL^T$. When A is sparse, fill occurs during the factorization; that is, some of the zero elements in A will become nonzero elements in L. In order to reduce time and storage requirements, only the nonzero positions of L are stored and operated on during *sparse* Cholesky factorization. Techniques for accomplishing this task and for reducing fill have been studied extensively (see [9] for details). In this paper we restrict our attention to the numerical factorization phase. We assume that the preprocessing steps, such as reordering to reduce fill and symbolic factorization to set up the compact data structure for L, have been performed. Details on the preprocessing can also be found in [9].

2.1 Sequential left-looking sparse Cholesky algorithms

Column-oriented, left-looking, sparse Cholesky: The standard column-oriented, left-looking, sparse Cholesky algorithm is widely used in many sparse matrix packages, such as SPARSPAK [5]. When column $L_{*,j}$ is to be computed, the algorithm first modifies column $A_{*,j}$ with multiples of previous columns of L, namely those columns $L_{*,k}$ for which $1 \leq k \leq j-1$ and $L_{j,k} \neq 0$; it then scales the modified column to obtain $L_{*,j}$. We let $cmod(j,k)$ denote the operation of updating column $A_{*,j}$ with a multiple of the appropriate entries of $L_{*,k}$, and we let $cdiv(j)$ denote the final scaling operation. Thus we can write the column task $Tcol(j)$ as

$$Tcol(j) := \{cmod(j,k) \mid k < j \text{ and } L_{j,k} \neq 0\} \cup \{cdiv(j)\}.$$

We shall refer to this algorithm as the `col-col` algorithm.

Supernodal sparse Cholesky: For many realistic problems, especially those from structural analysis applications, the fill generated during the factorization creates groups of contiguous columns that share the same sparsity structure. A group of such columns is referred to as a *supernode*. The reader should consult [11,14] for a detailed description of various *supernode partitions*. The first reference also presents an efficient algorithm for generating a supernode partition.

Supernodes have been used to organize left- and right-looking sparse Cholesky factorization algorithms around matrix-vector or matrix-matrix operations that reduce memory traffic, thereby making more efficient use of vector registers [3,4] or cache [1,16]. The role of supernodes in improving sparse Cholesky factorization algorithms is well documented [1,3,4,7,16,19].

Let $\boldsymbol{K} = \{p, p+1, \ldots, p+q\}$ denote a supernode in L. These columns have a *dense* diagonal block and have *identical* column structure below row $p+q$. Because the external sparsity pattern is shared by all columns in $\boldsymbol{K}$, any column $A_{*,j}$ $(j > p+q)$ will be modified by either *all* columns of $\boldsymbol{K}$ or *no* columns of $\boldsymbol{K}$. (Throughout, supernodes will be denoted by bold-faced capital letters.) In the supernodal Cholesky factorization algorithm, the basic task around which the algorithm is organized is again $Tcol(j)$. However, the primitive at the heart of the column task $Tcol(j)$ is no longer $cmod(j,k)$. Instead, the new computationally-intensive primitives are $cmod(j,\boldsymbol{K})$, $j \notin \boldsymbol{K}$, which modifies column j with a multiple of each column in the supernode $\boldsymbol{K}$, and $cmod(j,\boldsymbol{J})$, $j \in \boldsymbol{J}$, which modifies column j with a multiple of each column $k \in \boldsymbol{J}$ for which $k < j$. We shall refer to this formulation of left-looking sparse Cholesky factorization as the `sup-col` algorithm. Careful implementation of these primitives can significantly reduce memory traffic and indirect addressing. The $cmod(j,\boldsymbol{J})$ operation is implemented as a *dense matrix-vector product* and is accumulated directly into factor storage for the target column $L_{*,j}$; the $cmod(j,\boldsymbol{K})$ update first accumulates a dense matrix-vector product into a small work vector, after which the accumulated column update is applied to the target column $L_{*,j}$ using a single column operation that requires indirect addressing. In both cases the outer loop of the matrix-vector multiplication is unrolled to reduce memory traffic [6].

Block sparse Cholesky: Rothberg and Gupta [17] and Ng and Peyton [12] have investigated a left-looking block Cholesky algorithm first suggested in [4]. The basic task around which the computation is organized is a *block column task* that consists of computing every column of L in a particular supernode. More precisely, the basic task is given by

$$Tsup(\boldsymbol{J}) := \{Tcol(j) \mid j \in \boldsymbol{J}\},$$

where J is a supernode of L. This algorithm preserves all of the performance enhancements incorporated into the `sup-col` algorithm and enables new improvements in performance as well. The block column task $Tsup(J)$ is organized around the following computational primitives. The $smod(J,K)$ primitive, $J \neq K$, modifies with a multiple of each column of the supernode K every column of the supernode J that receives updates from K. After all such *external* updates to J have been computed by the $smod(J,K)$ primitive, the $Chol(J)$ primitive performs all the *internal* updates and the column scaling operations needed to complete the computation of the columns of J. At the heart of these primitives is a *dense matrix-matrix product* that, when carefully implemented, permits extensive reuse of data in cache memory (see [12,17] for details). Many of the algorithms incorporated into the LAPACK linear algebra software package [2] are also organized in this fashion for similar reasons. We shall refer to the block sparse Cholesky algorithm as the `sup-sup` algorithm.

2.2 Parallel implementation techniques

Parallel versions of two of these sequential algorithms have been explored previously. In [8], George *et al.* introduced a parallel `col-col` algorithm for shared-memory multiprocessors. Their algorithm is a straightforward parallel implementation of the sequential `col-col` algorithm found in [9]. Ng and Peyton [13] used the same straightforward approach to produce a parallel version of the `sup-col` algorithm. We shall use the same techniques to implement a parallel block algorithm.

These parallel algorithms maintain a pool of column tasks, and the assignment of column tasks to processors is dynamic: when a processor completes one column task, it immediately seeks another column task from the pool. This approach is particularly appropriate for a shared-memory multiprocessor system with a fast bus, and as might be expected it typically achieves good load balance. The instructions that manipulate the lists of updating columns (supernodes) needed by the parallel `col-col` (`sup-col`) algorithm form a critical section in the code, access to which is limited by means of standard "lock" and "unlock" synchronization primitives. For a more complete discussion of the implementation details, the reader should consult [8,13].

In a series of studies, Rothberg and Gupta [15,16,17,18] have looked into many of the issues and techniques needed to implement sparse Cholesky factorization algorithms on modern workstations. Their work establishes the importance of using blocked algorithms to fully exploit the memory hierarchy of these machines, including those with multiple processors. In the next section, we discuss a parallel left-looking sparse Cholesky algorithm that is appropriate for this class of machines.

3 A parallel blocked sparse Cholesky algorithm

For most realistic problems, the work required to compute $Tsup(J)$ for each supernode J increases dramatically as the computation proceeds. As a result, straightforward parallelization of the `sup-sup` algorithm leads to an extremely inefficient parallel algorithm due to its coarse granularity, particularly in the middle and later stages of the computation. A simple and natural way to solve this problem is to split each supernode J into sets of contiguous columns, which we shall call *panels*. The task on which our parallel algorithm is based is the computation of every column of L that belongs to a particular panel. This task will be denoted as $Tpan(J)$, where J is a panel of contiguous columns all belonging to the same supernode. (We shall use non-bold capital letters to denote a panel.) We shall refer to the panel-based algorithm as the `sup-pan` algorithm.

The block column task $Tpan(J)$ is organized around the following three computational primitives. The $xpmod(J,K)$ primitive ($J \not\subseteq K$) modifies with a multiple of each column of the supernode K every column of the panel J that receives updates from K. After all such *external* updates to J have been computed, the $ipmod(J,J)$ primitive updates every column in panel J with the columns of every preceding panel in the supernode J to which J belongs. That is, the $ipmod(J,J)$ primitive performs all updates to the panel J that are internal to the supernode J yet external to the panel J itself. The $Chol(J)$ primitive performs all the updates internal to the panel and also the column-scaling operations needed to complete the computation of the columns of J. As with the primitives for the `sup-sup` algorithm, these primitives are implemented around a *dense matrix-matrix product* that, when carefully implemented, permits extensive reuse of data in cache memory.

The performance of our algorithm to a large extent depends on how the panels are selected. The trade-off between available concurrency and data latency can perhaps be best illustrated by looking at the two extreme panel selection strategies. Consider first the case where each panel J is as *small* as possible, that is, each panel J contains a single

column of L. In this case the parallel sup-pan algorithm is equivalent to the parallel sup-col algorithm studied in [13], and thus shares its strengths and weaknesses. This "paneling" exposes more parallelism than any other; it results, however, in the least efficient use of the cache and the global data bus. Consider now the case where each panel J is as *large* as possible, that is, each panel J is a supernode of L. In this case the parallel sup-pan algorithm is equivalent to the parallel sup-sup algorithm. This paneling exposes less parallelism than any other; it results, however, in the most efficient use of the cache and the global data bus. Clearly then we seek a paneling strategy that lies between these two extremes; we seek panels that are

- large enough to retain a large fraction of the data latency advantages of the parallel sup-sup algorithm, and, at the same time
- small enough to expose a large fraction of the concurrency available in the parallel sup-col algorithm.

Preliminary results indicate that, as long as extremely fine and extremely coarse panelings are avoided, the parallel sup-pan algorithm performs quite well, and moreover its performance is remarkably robust with respect to a wide range of paneling strategies. One reason for this state of affairs is that the marginal increases in data reuse fall off rapidly as the panel size increases beyond a modest number of columns. Consequently, partitioning a large supernode into panels, each with several columns, sacrifices a small fraction of the data-latency advantages of the sup-sup algorithm. Our preliminary results also indicate that for modest numbers of processors the paneling must become fairly coarse before we observe serious deterioration in performance due to insufficient concurrency within the computation.

In this preliminary study we use the following simple paneling strategy, which we have observed to be quite effective in our preliminary tests. Choose a constant panel size w, where w is a positive integer. Divide each supernode J into panels J of w contiguous columns, with the last panel containing anywhere from 1 to w "leftover" columns. In our tests we have essentially completed a parameter study on the paneling parameter w, testing different values of w between 2 and 32. How best to panel the supernodes merits further study and is still under investigation.

4 Timing results

We had hoped to test the parallel sup-pan algorithm on two machines: the SGI 4D/380 and the new Cray C90. While we have obtained results on the SGI 4D/380, we have unfortunately been unable to gain access to a Cray C90. The Cray C90 is of interest to us because it can experience significant performance degradation when there is heavy contention for the shared memory. Our parallel sup-pan algorithm should reduce this contention, and we hope to study the performance of the algorithm on this machine in the near future.

The SGI 4D/380 is a high-performance multiprocessor workstation containing eight MIPS R3000/R3010 processors. New and forthcoming multiprocessor workstations, such as the SGI 4D/380, make up an important new class of machines, with perhaps the best overall cost/performance ratio available. The fast, relatively inexpensive, off-the-shelf processors used by these machines depend heavily on the reuse of data in cache for good performance. The relatively inexpensive data bus technology used in these machines provides far less bandwidth to main-memory than is typically required by an unblocked factorization algorithm such as the parallel sup-col algorithm. In consequence, this class of machines is an ideal target for the parallel sup-pan algorithm.

We tested the performance of our algorithm on the matrices from the Harwell-Boeing sparse matrix collection listed in Table 1. Some statistics for each matrix are also shown in the table. Each matrix was reordered by the multiple minimum degree ordering heuristic [10] before the factorization was performed.

In our tests we compared the performance of the parallel sup-col algorithm with that of the parallel sup-pan algorithm. We tested a set of panel widths ranging from $w = 2$ to $w = 32$. The algorithms were coded in C and compiled with optimization turned on. The timing results are presented in Table 2. The table records the performance of the parallel sup-col (sup-pan) algorithm in terms of speed-ups relative to a fast serial implementation of the sup-col (sup-sup) algorithm. For the sup-pan method, we report only the best performance obtained over all panel sizes tested. (The performance of the parallel sup-pan algorithm was remarkably robust with respect to all but the smallest of the panel sizes tested.) The panel size for which the best timing was obtained is recorded in parentheses.

As observed in [12,17], the performance of the serial sup-sup algorithm is far better than that of the serial sup-col algorithm because of reuse of the data in cache memory. In addition, the efficiencies of the parallel sup-col algorithm range from 41–47%, while those of the parallel sup-pan algorithm range from 73–80%. It appears then that the following two statements hold true for these algorithms on this machine:

problem	n	$\|A\|$	$\|L\|$
BCSSTK14	1,806	63,454	112,267
BCSSTK15	3,948	117,816	651,222
BCSSTK16	4,884	290,378	741,178
BCSSTK17	10,974	428,650	1,005,859
BCSSTK18	11,948	149,090	662,725
BCSSTK23	3,134	45,178	420,311
BCSSTK25	15,439	252,241	1,416,568
BCSSTK29	13,992	619,488	1,694,796

Table 1: Test problems.

problem	serial sup-col time (in secs)	parallel sup-col speed-up $p=4$	parallel sup-col speed-up $p=8$	serial sup-sup time (in secs)	parallel sup-pan speed-up $p=4$	parallel sup-pan speed-up $p=8$
BCSSTK14	2.44	2.84	3.70	1.58	3.29 (16)	5.27 (12)
BCSSTK15	38.24	2.88	3.25	21.50	3.54 (16)	6.13 (12)
BCSSTK16	33.96	2.96	3.72	20.46	3.49 (12)	6.00 (12)
BCSSTK17	32.63	2.84	3.73	20.55	3.37 (16)	6.12 (12)
BCSSTK18	36.28	2.76	3.42	21.71	3.47 (16)	5.84 (16)
BCSSTK23	28.58	2.84	3.36	17.53	3.73 (16)	6.37 (16)
BCSSTK25	72.49	2.90	3.53	45.05	3.54 (16)	5.87 (12)
BCSSTK29	98.23	2.83	3.29	59.35	3.59 (12)	6.35 (12)

Table 2: Performance statistics on an SGI 4D/380. The numbers in parentheses record the panel size w for which the best timing was obtained.

- The parallel sup-pan algorithm retains a large fraction of the serial sup-sup algorithm's ability to reuse data in cache.
- The data bus of the SGI 4D/380 is overtaxed by the parallel sup-col algorithm.

The combination of superior sequential performance and superior parallel performance makes the parallel sup-pan algorithm an attractive alternative on this particular multiprocessor workstation and others similar to it. Over all the problems, the parallel sup-pan algorithm on eight processors performed from a low of 9.7 to a high of 10.9 times faster than the sequential sup-col algorithm.

5 Concluding remarks

In this brief note we have introduced a simple parallel left-looking blocked sparse Cholesky algorithm, which we call the parallel sup-pan algorithm. The target machine for such an algorithm is any shared-memory multiprocessor for which the data bus is a potential bottleneck and/or the individual processors rely on extensive reuse of data in cache for good performance. Our timing results on an SGI 4D/380 multprocessor workstation indicate that the method performs quite well on such machines.

In future work, we hope to study the performance of this method on one of the latest vector supercomputers with multiple processors, the Cray C90. We hope to include other machines of interest, as well. The primary goal of our future efforts however will be to devise a simple and efficient algorithm to compute good panelings for the algorithm. This problem is currently under study.

Finally we wish to note two other issues connected with the parallel sup-pan algorithm. First, any ordering of the columns within a supernode leads to identical fill in the factorization. Some supernode column orderings however are better at placing into panels those columns that are updated by many of the same supernodes. Panels with this property better perform their role of reusing data in cache and limiting bus traffic. Second, the notion of *domains* has played a significant role in parallel sparse Cholesky algorithms for distributed-memory machines. A domain is

a set of columns whose computation is an atomic task assigned to a single processor. The choice of the columns in a domain depends on the sparsity structure of L (corresponding to a subtree of the elimination tree). The key is that no synchronization is required when the columns of a domain are computed. To implement this technique successfully, one must balance the reduction in synchronization overhead against the increase in grain size incurred by the technique. Preliminary experimental results have indicated that reordering within supernodes and the use of domains have little impact on the performance of the parallel sup-pan algorithm when the number of processors is small.

Acknowledgements

This research is supported in part by DARPA contract N00039-91-C-0138, and by the Applied Mathematical Sciences Research Program, Office of Energy Research, U.S. Department of Energy under contract DE-AC05-85OR21400 with Martin Marietta Energy Systems Inc. Anoop Gupta is also supported by an NSF Presidential Young Investigator award.

References

[1] P.R. Amestoy and I.S. Duff. Vectorization of a multiprocessor multifrontal code. *Internat. J. Supercomp. Appl.*, 3:41–59, 1989.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings of Supercomputing '90*, pages 1–10. IEEE Press, 1990.

[3] C. Ashcraft. A vector implementation of the multifrontal method for large sparse, symmetric positive definite linear systems. Technical Report ETA-TR-51, Engineering Technology Applications Division, Boeing Computer Services, Seattle, Washington, 1987.

[4] C.C. Ashcraft, R.G. Grimes, J.G. Lewis, B.W. Peyton, and H.D. Simon. Progress in sparse matrix methods for large linear systems on vector supercomputers. *Internat. J. Supercomp. Appl*, 1:10–30, 1987.

[5] E.C.H. Chu, A. George, J. W-H. Liu, and E. G-Y. Ng. User's guide for SPARSPAK-A: Waterloo sparse linear equations package. Technical Report CS-84-36, University of Waterloo, Waterloo, Ontario, 1984.

[6] J.J. Dongarra and S.C. Eisenstat. Squeezing the most out of an algorithm in Cray Fortran. *ACM Trans. Math. Software*, 10:219–230, 1984.

[7] I.S. Duff and J.K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Trans. Math. Software*, 9:302–325, 1983.

[8] A. George, M.T. Heath, and J. W-H. Liu. Parallel Cholesky factorization on a shared-memory multiprocessor. *Linear Alg. Appl.*, 77:165–187, 1986.

[9] A. George and J. W-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1981.

[10] J. W-H. Liu. Modification of the minimum degree algorithm by multiple elimination. *ACM Trans. Math. Software*, 11:141–153, 1985.

[11] J.W.H. Liu, E. Ng, and B.W. Peyton. On finding supernodes for sparse matrix computations. Technical Report ORNL/TM-11563, Oak Ridge National Laboratory, Oak Ridge, TN, 1990.

[12] E. Ng and B. Peyton. Block sparse Cholesky algorithms on advanced uniprocessor computers. Technical Report ORNL/TM-11960, Oak Ridge National Laboratory, Oak Ridge, TN, 1991. (To appear in SIAM J. Sci. Stat. Comput.).

[13] E. Ng and B. Peyton. A supernodal Cholesky factorization algorithm for shared-memory multiprocessors. Technical Report ORNL/TM-11814, Oak Ridge National Laboratory, Oak Ridge, TN, 1991. (Submitted to SIAM J. Sci. Stat. Comput.).

[14] A. Pothen. Simplicial cliques, shortest elimination trees, and supernodes in sparse Cholesky factorization. Technical Report CS-88-13, Department of Computer Science, The Pennsylvania State University, University Park, Pennsylvania, 1988.

[15] E. Rothberg and A. Gupta. A comparative evaluation of nodal and supernodal parallel sparse matrix factorization: detailed simulation results. Technical Report STAN-CS-90-1305, Stanford University, Stanford, California, 1990.

[16] E. Rothberg and A. Gupta. Efficient sparse matrix factorization on high-performance workstations—exploiting the memory hierarchy. *ACM Trans. Math. Software*, 17:313–334, 1991.

[17] E. Rothberg and A. Gupta. An evaluation of left-looking, right-looking, and multifrontal approaches to sparse Cholesky factorization on hierarchical-memory machines. Technical Report STAN-CS-91-1377, Stanford University, Stanford, California, 1991. submitted to the International Journal of High Speed Computing.

[18] E. Rothberg and A. Gupta. Techniques for improving the performance of sparse matrix factorization on multiprocessor workstations. In *Proceedings of Supercomputing '90*, Los Alimitos, California, 1991. IEEE Computer Society Press.

[19] H.D. Simon, P.A. Vu, and C.W. Yang. Sparse matrix at 1.68 Gflops. Technical report, Boeing Computer Services, Seattle, Washington, 1989.