Towards a Machine-Independent Solution of Sparse Cholesky Factorization

Calvin Lin and W. Derrick Weathersby [a] *

[a] Dept. of Computer Science and Engineering, FR-35,
University of Washington
Seattle, WA 98195

This paper reports on our efforts to develop a machine-independent solution to sparse Cholesky factorization that performs well across all general-purpose MIMD computers. We present results for a diverse set of machines and a diverse set of algorithmic techniques. In particular we show results for the Kendall Square KSR-1, Sequent Symmetry and Intel iPSC/2, and we compare supernodal and nodal techniques, fan-in and fan-out algorithms, and shared and non-shared memory programming models.

1. INTRODUCTION

Software portability is an important means of reducing program development costs. Here, a "portable" program is one that achieves good performance across some desired set of parallel computers. In particular, a portable – or machine-independent – program achieves performance that is competitive with the best machine-dependent programs.[2] Previous studies have suggested that the non-shared memory programming model leads to more portable programs than the shared memory model because of advantages in improved locality of reference [1–3]. These studies, however, focus on applications that are fairly regular in their data access patterns. Here, we report on efforts to find a machine-independent solution to a very irregular problem: sparse Cholesky factorization.

In some sense direct sparse Cholesky methods represent the ultimate challenge in parallel portability. The sparse, data-dependent nature of the problem can lead to poor locality of reference and, for static partitionings, significant load imbalance. These characteristics would seem to favor shared memory solutions, but because shared memory machine architectures may not scale as well as non-shared memory machines, efficient non-shared memory implementations are desirable. Furthermore, Cholesky factorization plays an important role in many scientific applications. Examples include finite element methods, structural analysis, and various VLSI circuit simulators. Finally, unlike dense matrix computations, sparse matrix computations can scale to sizes that would invite massively parallel implementations.

Previous work in the parallel solution of sparse matrix factorization have generally been machine-specific [4–10]. Thus, for example, because distributed memory approaches were only studied on distributed memory machines, it was not clear whether the resulting

*This research was supported in part by ARPA Grant N00014-92-J-1824, NSF Grant CDA-9211095, and an NSF Graduate Engineering Education Fellowship

[2] By this definition, portable solutions may not exist for all problems and all machines.

performance numbers were due to the programming model or to characteristics of the actual machines. By comparing both shared and non-shared memory implementations on shared memory machines, we can decouple the influence of the programming model from the influence of the underlying hardware. We can thus determine the extent to which the various programming models affect performance.

This paper experimentally explores four dimensions of the space of possible solutions to sparse Cholesky factorization. This study is not exhaustive– for example, multi-frontal techniques [11] are not considered– but will move us closer to a machine-independent solution. We show results for both shared and non-shared memory machines, namely, the Kendall Square KSR-1, the Sequent Symmetry, and the Intel iPSC/2. We consider fan-in and fan-out algorithms; shared and non-shared memory programming models; nodal and supernodal techniques; as well as the effects of various mapping and reordering functions.

This paper is organized as follows. Section 2 provides background by describing the different aspects and variations of sparse Cholesky factorization. Section 3 describes our experimental results, and we conclude in Section 4.

2. BACKGROUND

2.1. Sparse Cholesky Factorization

Sparse Cholesky factorization can be used to solve a system of linear equations, $Ax = b$, whenever A is a symmetric, positive definite matrix. We can solve for x by factoring the A matrix into a lower triangular matrix, L, such that $A = LL^T$. To zero out all entries above the diagonal, multiples of one column are added to the values of another column. Such *column modifications* are performed once for each column to compute the L matrix in a process known as *numeric factorization.*

For sparse matrices the overall solution involves four steps. First, the A matrix is ordered to reduce *fill.* Fill refers to the non-zero elements that are introduced in the process of numeric factorization and represents unnecessary work. The second step, *symbolic factorization*, computes the structure of the eventual factor matrix. The third step is numeric factorization, and the final step is the triangular solve where the factor matrix is used to compute the value of x. Because numeric factorization is the most time consuming step we will not consider steps 2 or 4 in this paper. The first step, though computationally inexpensive, is significant in that it can affect the performance of numeric factorization.

2.2. Experimental Parameters

Our experiments compare fan-in and fan-out approaches, shared memory and non-shared memory approaches, and nodal and supernodal approaches. This section briefly introduces these terms.

Two general approaches to computing the Cholesky factorization are the left-looking and right-looking approaches. These sequential algorithms differ in their memory access patterns and the order in which column modifications are performed. The left-looking algorithm leads to the fan-in parallel algorithm, while the right-looking algorithm leads to the fan-out algorithm. The fan-in approach is synchronous and demand driven, while the fan-out approach is asynchronous and data driven. We show results comparing the fan-in and fan-out approaches. We do not implement any fan-in algorithms in the shared memory programming model because we found no such algorithm in the literature.

Our shared memory implementation of sparse Cholesky factorization comes from the SPLASH suite of benchmarks. This program uses a work queue execution model that yields good load balance. By contrast, our non-shared memory implementations statically partition the data into disjoint sets and assign each set to a different processor. Our implementations of the shared and non-shared memory models result in almost identical amounts of communication. We do not produce a shared memory implementation for the iPSC/2 because we have no way to emulate shared memory on this machine.

Supernodal techniques exploit the structure of the non-zero elements to increase granularity and improve locality of reference [10]. Consecutive columns with non-zero elements in the same rows below the diagonal are combined and treated as a unit, allowing multiple column modifications to be performed at once. When supernodes are restricted to contain a single column, the result is a nodal technique. Supernodes reduce execution time for the best sequential implementations, but nodal techniques have the potential for greater parallelism. Our implementation of supernodes comes from the Rothberg and Gupta implementation in which supernode size is controlled based on cache size [10].

Finally, non-shared memory implementations must consider two additional factors: reordering and *mapping*. The load balance of the numeric factorization step depends on the partitioning of the nodes or supernodes, which in turn is dependent on the order in which the columns are modified. In sequential implementations, the goal of the ordering step is simply to reduce the amount of fill. For parallel solutions, this ordering step can also be used to balance the load. If the factorization process is viewed as a logical tree with arcs representing data dependencies, orderings that produced short balanced trees will lead to balanced loads. *Mapping* refers to the mapping of nodes to processors. A good mapping will minimize communication costs and give good load balance.

In the reordering step we employ the multiple minimum degree (MMD) [7] ordering heuristic followed by Jess and Kees tree height minimization technique. The MMD reordering arranges the columns to minimize fill in the factor matrix.

We implement three different mapping algorithms. The simplest load balancing strategy is a cyclic, round-robin assignment of nodes to processors, but this leads to excessive communication volume [7]. At the other extreme, the subtree-to-subcube mapping reduces communication but leads to poor load balance for unbalanced work trees [7]. We introduce a new algorithm that attempts to balance the load as well as reduce communication. In this algorithm the elimination tree is traversed in depth-first order until a subtree is found with less than or equal to $\frac{1}{P}$ percent of the total work (P is the number of processors), plus some tolerance. This user-specified tolerance determines how far the mapping may deviate from the optimal mapping. In general, we found the new algorithm to yield the best performance.

2.3. Hardware

We describe experiments on a 20 processor Sequent Symmetry A, a 56 processor Kendall Square KSR-1, and a 32 processor Intel iPSC/2. The 20 processors of the Sequent are connected to main memory by a shared bus. Each processor has a 64K coherent cache for both data and instructions. Our KSR-1 has 56 processors organized as a ring of rings. The KSR's shared main memory is viewed as a large coherent cache that allows data to migrate on demand. Finally, our Intel iPSC/2 is a hypercube connected distributed

memory machine. Each node consists of a processor and 4MB of local memory. Processors communicate via message passing.

3. RESULTS

3.1. Methodology

We wish to compare the aspects of sparse Cholesky factorization outlined above. Unfortunately, these different aspects interact. That is, the various experimental parameters are not independent. Thus, for example, we cannot make a general statement comparing *all* fan-in and fan-out algorithms without first limiting our discussion to either nodal or supernodal approaches. Our methodology is to search for the best solutions along each of the three dimensions mentioned above. Thus, we will find the best fan-in implementation independently of the best fan-out implementation, and then compare the two results.

3.2. Interpretation

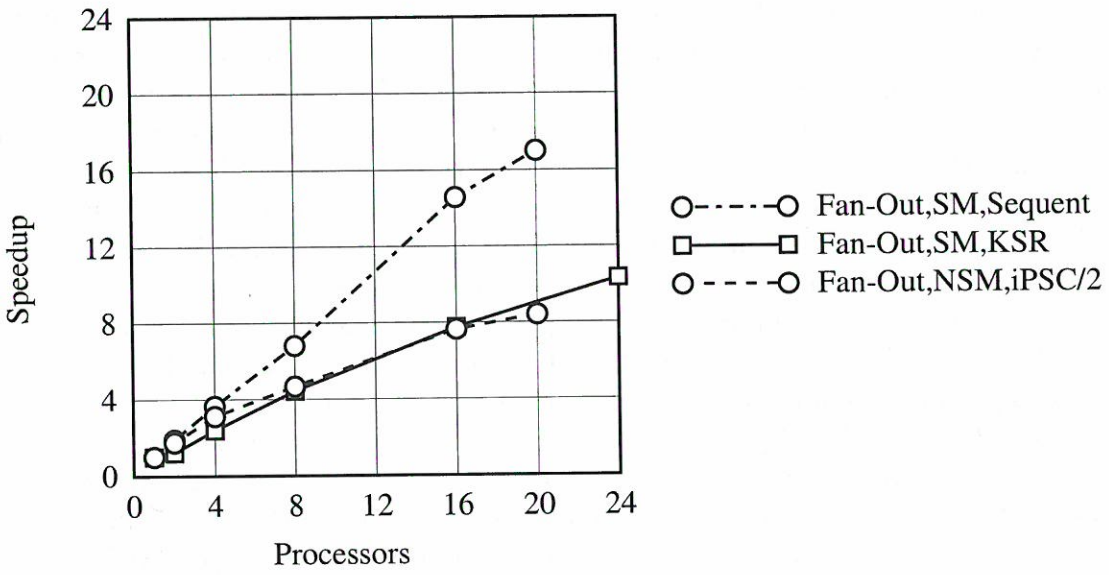

Figure 1. Best performance on each machine.

We first show the best algorithm for machine. (See Figure 1.) On the two shared memory machines where we could directly compare the shared and non-shared memory models, the supernodal fan-out shared memory program outperforms our statically partitioned programs. The success of the shared memory programs is due to superior load balance and lower overhead.

On our machines, both implementations of the fan-out algorithm consistently outperform the fan-in algorithm. Figure 2 shows results for the Sequent Symmetry, KSR-1 and the Intel iPSC/2. The poor performance of the fan-in algorithm can be attributed to the implementation's synchronous nature, its lack of locality and its buffered updates.

To understand these effects, we must describe our algorithm in more detail. We use the fan-in algorithm described by Ashcraft et al. [4]. which is synchronous because a process performing a column modification must wait for all data to arrive before computing a modification. This synchronization leads to idle time. The locality and buffering effects can also be explained. Abstractly, each column modification is computed in a distributed fashion and requires the merging of multiple partial modifications, where each partial modification is computed by a different process. When the granularity is large and each process owns several columns, these partial modifications are advantageous because they trade reduced communication cost for increased local computation. When the granularity decreases, however, the savings in communication goes to zero while the extra computation becomes strictly overhead. Note that modifications to the algorithm might avoid some of these problems, but at the expense of added complexity. For example, idle time can be reduced by allowing a process to asynchronously work on one column modification while it awaits the arrival of other partial modifications.

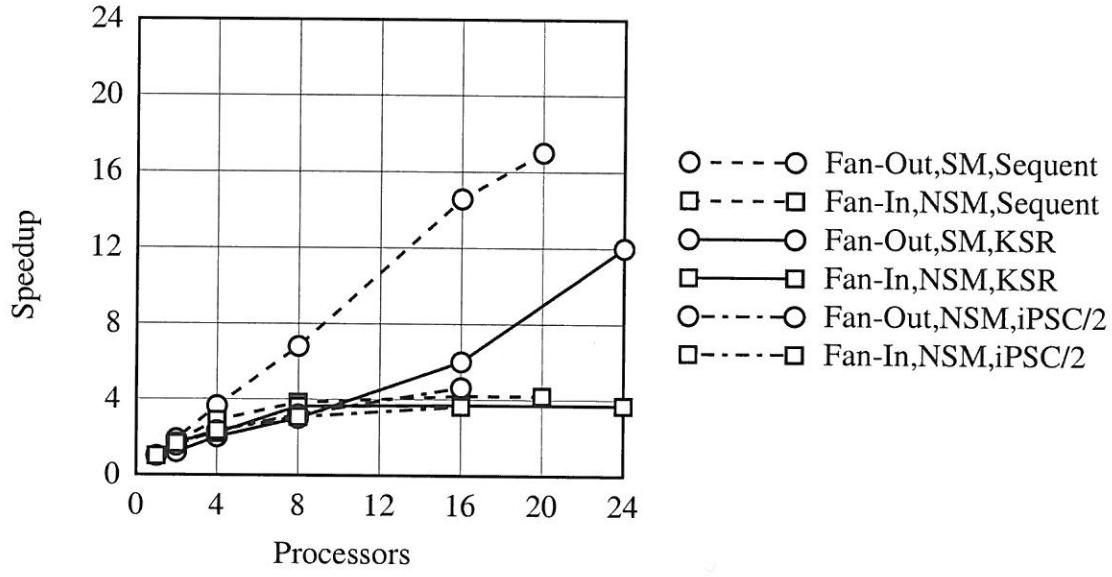

Figure 2. Best fan-in vs. best fan-out.

In comparing nodes versus supernodes, we observe a scaling effect where supernodes are superior for small numbers of processors but inferior for large numbers of processors. This cross-over occurs because although supernodes improve locality and decrease overhead, they also increase granularity and limit parallelism. Figure 3 shows results for the Symmetry where the cross-over point is at eight processors. A solution to this scaling problem is to limit the size of the supernodes. Current schemes place upper bounds on the size of supernodes based on the target machine's cache size [10], but do not consider the amount of physical parallelism. We believe that supernodal techniques can scale better if the size of supernodes is based on the number of processors, and the latency and communication startup costs of the underlying machine.

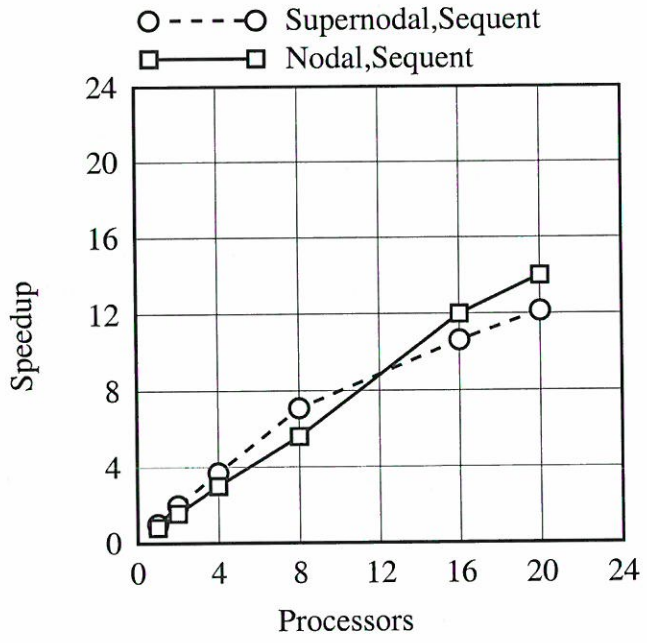

Figure 3. Best nodal vs. best supernodal.

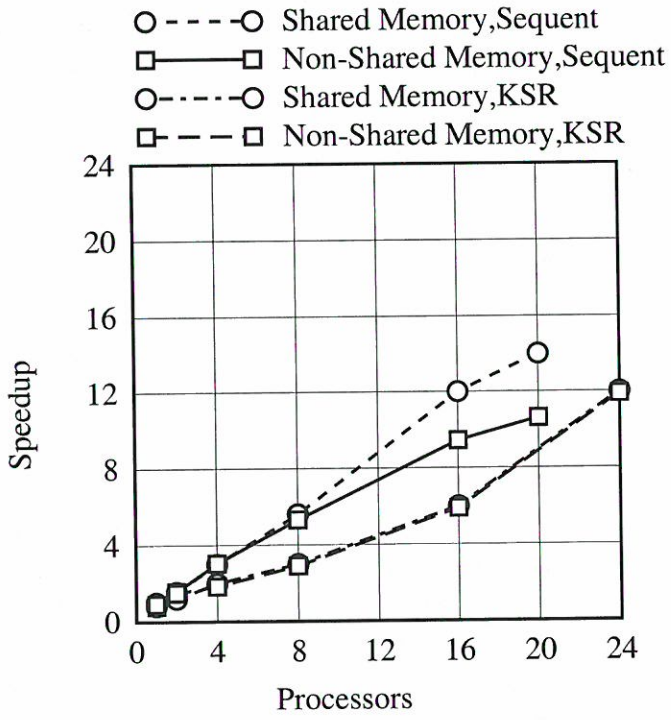

Figure 4. Best SM program vs. best NSM program.

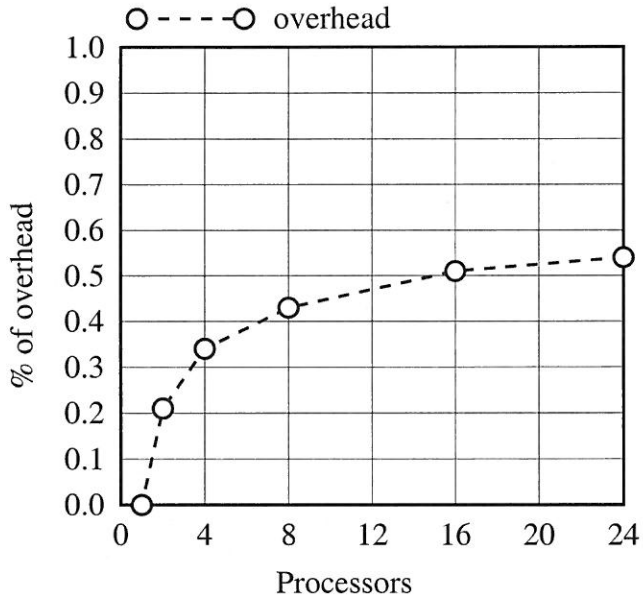

Figure 5. Percent of SM-NSM performance gap due to overhead.

In comparing the influence of programming models, we observe a performance advantage for the shared memory model. (See Figure 4.) The difference in performance can be explained by the load imbalance and the overhead suffered by the non-shared memory program. The static partitioning necessarily forces the non-shared memory program to do extra work. Through several pointer de-references and arithmetic calculations, each processor determines the next column to be updated. With the static partitioning, this work is useless if the column belongs to another processor. The shared memory implementation has no such overhead. Figure 5 shows the significance of this overhead on the KSR-1. The overhead is *conservatively* calculated as the percentage of time spent executing the code to determine a non-local column with respect to the difference in execution time of the two memory models. We instrumented the code to count the number of times this overhead code was executed and then measured the amount of time to execute one instance of this code. The performance difference becomes very small for machines like the KSR where locality is extremely important.

4. CONCLUSION

We have compared various aspects of sparse Cholesky factorization on a variety of parallel computers. Three observations are noteworthy. First, for this application the shared memory programming model performs better than the non-shared memory programming model. A significant portion of this performance gap stems from additional overhead that exists in the non-shared memory approach. Thus, we have identified one circumstance where the ability to share a single address space fundamentally affects performance. We contrast this with many other applications where the single address space is primarily a programming convenience. The second observation is that the supernodal technique can

inhibit performance as the number of processors grows. As mentioned earlier, sophisticated techniques for limiting supernode sizes should lead to better scalability. Finally, the best fan-out implementations perform better than the best fan-in implementations. From these observations we conclude that a machine-independent supernodal fan-out implementation can eventually be found. The issues of load-imbalance and supernode size must first be addressed before this ultimate goal is reached.

Acknowledgments

We thank Larry Snyder for many helpful discussions regarding this work. We are also thank John Lewis for his valuable suggestions and his help in providing access to a reordering tool.

REFERENCES

1. Calvin Lin and Lawrence Snyder. A comparison of programming models for shared memory multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages II 163–180, 1990.
2. Ton A. Ngo and Lawrence Snyder. On the influence of programming models on shared memory computer performance. In *The Scalable High Performance Computing Conference*, to appear, 1992.
3. Calvin Lin. *The Portability of Parallel Programs Across MIMD Computers.* PhD thesis, University of Washington, Department of Computer Science and Engineering, 1992.
4. C. Ashcraft, S.C. Eisenstat, and J.W.H. Liu. A fan-in algorithm for distributed sparse numerical factorization. *SIAM Journal on Scientific and Statistical Computing*, 11, 1990.
5. C. Ashcraft, S. Eisenstat, J. Liu, and A. Sherman. A comparison of three distributed sparse factorization schemes. In *SIAM Symposium on Sparse Matrices*, May 1989.
6. A. Geist and C.H. Romine. Lu factorization on distributed memory multiprocessor architectures. *SIAM Journal on Scientific and Statistical Computing*, 9(4).
7. A. George, J. Liu, and E. Ng. Communication reduction in parallel sparse cholesky factorization on a hypercube. In *Hypercube Multiprocessors*, pages 576–586, 1987.
8. A. George, M. Heath, J.W.H. Liu, and E. Ng. Sparse cholesky factorization on a local memory multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 9, 1988.
9. M.T. Heath, E. Ng, and B.W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33, 1991.
10. E. Rothberg and A. Gupta. Techniques for improving the performance of sparse factorization on multiprocessor workstations. In *Proceedings of Supercomputing '90*, November 1990.
11. J. Liu. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Review*, 34(1):82–109, March 1992.