

A Comparison of 1-D and 2-D Data Mapping for Sparse LU Factorization with Partial Pivoting*

Cong Fu[†] Xiangmin Jiao[†] Tao Yang[†]

Abstract

This paper presents a comparative study of two data mapping schemes for parallel sparse LU factorization with partial pivoting on distributed memory machines. Our previous work has developed an approach that incorporates static symbolic factorization, nonsymmetric L/U supernode partitioning and graph scheduling for this problem with 1-D column-block mapping. The 2-D mapping is commonly considered more scalable for general matrix computation but is difficult to be efficiently incorporated with sparse LU because partial pivoting and row interchanges require frequent synchronized inter-processor communication. We have developed an asynchronous sparse LU algorithm using 2-D block mapping, and obtained competitive performance on Cray-T3D. We report our preliminary studies on speedups, scalability, communication cost and memory requirement of this algorithm and compare it with the 1-D approach.

1 Introduction

Efficient parallelization for sparse LU factorization with pivoting is important to many scientific applications. Different from sparse Cholesky factorization, for which the parallelization problem has been relatively well solved [3, 8, 12, 13], the sparse LU factorization is much harder to be parallelized due to its dynamic nature caused by pivoting operations. The previous work has addressed parallelization issues using shared memory platforms or restricted pivoting [6, 7, 9]. In [4] we proposed a novel approach that integrates three key strategies together in parallelizing this algorithm on distributed memory machines: 1) adopt a static symbolic factorization scheme [7] to eliminate the data structure variation caused by dynamic pivoting; 2) identify data regularity from the sparse structure obtained by the symbolic factorization so that efficient dense operations can be used to perform most of the computation; 3) make use of graph scheduling techniques and efficient run-time support called RAPID [3] to exploit irregular parallelism. The preliminary experiments are encouraging and good performance results are obtained with 1-D data mapping [4].

In the literature 2-D mapping has been shown more scalable than 1-D for dense LU factorization and sparse Cholesky [13, 14]. In [10] a sparse solver with element-wise 2-D mapping is presented. For better cache performance, block partitioning is preferred [1]. However there are several difficulties to apply the 2-D block-oriented mapping to the case of sparse LU factorization even the static structure is predicted in advance. First of all, pivoting operations and row interchanges require frequent and well-synchronized inter-processor communication when submatrices in the same column block are assigned to

*Supported by NSF RIA CCR-9409695 and CDA-9529418, and by a startup fund from University of California at Santa Barbara.

[†]Department of Computer Science, University of California, Santa Barbara, CA 93106

different processors. In order to exploit irregular parallelism, we need to deal with irregular and asynchronous communication, which requires delicate message buffer management. Secondly, it is difficult to model irregular parallelism from sparse LU. Using the elimination tree of $A \times A^T$ is possible but not accurate. Lastly, the space complexity is another issue. Exploiting irregular parallelism to a maximum degree may need more buffer space.

This paper presents our preliminary work on the design of an asynchronous algorithm for sparse LU with 2-D mapping. Section 2 briefly reviews the static symbolic factorization, sparse matrix partitioning and 1-D algorithms. Section 3 presents the 2-D algorithm. Section 4 discusses the advantages and disadvantages of the 1-D and 2-D codes. Section 5 presents the experimental results. Section 6 concludes the paper.

2 Background and 1-D approaches

The purpose of sparse LU factorization is to find two matrices L and U for a given nonsymmetric sparse matrix A such that $PA = LU$, where L is a unit lower triangular matrix, U is an upper triangular matrix, and P is a permutation matrix containing pivoting information. It is very hard to parallelize sparse LU because the pivot selection and row interchange dynamically increase fill-ins and change L/U data structures. Using the precise pivoting information at each elimination step can certainly optimize data space usage and improve load balance, but lead to high run-time overhead. In this section, we briefly discuss some techniques used in our S^* parallel sparse LU algorithm [4].

Static symbolic factorization. Static symbolic factorization is proposed in [7] to identify the worst case nonzero patterns. The basic idea is to statically consider all the possible pivoting choices at each step. And then the space is allocated for all the possible nonzeros that would be introduced by *any* pivoting sequence that could occur during the numerical factorization. The static approach avoids data structure expansions during the numerical factorization. The dynamic factorization, which is used in an efficient sequential code SuperLU [11], provides more accurate data structure prediction on the fly, but it is challenging to parallelize SuperLU on distributed memory machines. Currently the SuperLU group has been working on shared memory parallelizations [11].

L/U supernode partitioning. After the nonzero fill-in patterns of a matrix is predicted, the matrix is further partitioned using a supernode approach to improve the cache performance. In [11], a nonsymmetric supernode is defined as a group of consecutive columns in which the corresponding L factor has a dense lower triangular block on the diagonal and the same nonzero pattern below the diagonal. Based on this definition, in each column block the L part only contains dense subrows. We call this partitioning method L supernode partitioning. Here by “subrow” we mean the contiguous part of a row within a supernode. For SuperLU, it is difficult to explore structure regularity in a U factor after L supernode partitioning. However in our approach, nonzeros (including overestimated fill-ins) in the U factor can be clustered as dense columns or subcolumns. We discuss the U partitioning strategy as follows. After a L supernode partition has been obtained on a sparse matrix A , i.e., a set of column blocks with possible different block sizes, the same partitioning is applied to the rows of the matrix to further break each supernode into submatrices. Now each off-diagonal submatrix in the L part is either a dense block or contains dense blocks. Furthermore, in [4] we have shown that each nonzero submatrix in the U factor of A contains only dense subcolumns. This is the key to maximizing the use of BLAS-3 subroutines in our algorithm. Figure 1(a) illustrates the dense patterns in a partitioned sparse matrix.

Data Mapping. For block-oriented matrix computation, 1-D column-block cyclic mapping and 2-D block cyclic mapping are commonly used. In 1-D column-block cyclic mapping, as illustrated in Figure 1(b), a column-block $A_{:,j}$ is assigned to the same processor $P_{j \bmod p}$, where p is the number of the processors. Each column-block is called a panel in [12]. A 2-D block cyclic mapping views the processors as a 2-D $r \times s$ grid, and a block is the minimum unit for data mapping. A nonzero submatrix block $A_{i,j}$ is assigned to the processor $P_{i \bmod r, j \bmod s}$ as illustrated in Figure 1(c). The 2-D mapping is usually considered more scalable than 1-D mapping for Cholesky because it tends to have better computation load balance and lower complexity of communication volume. However the 2-D mapping introduces more overhead for pivoting and row swapping.

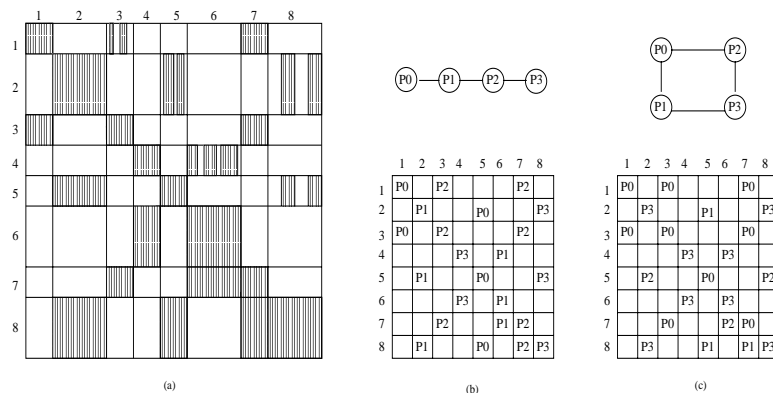


FIG. 1. (a) A partitioned sparse matrix; (b) A 1-D cyclic mapping of column blocks(panels); (c) 2-D cyclic mapping of blocks.

The 1-D program partitioning. Given an $n \times n$ matrix A , assume after the matrix partitioning it has N panels or $N \times N$ blocks. The maximum number of columns in a panel is b . Each panel k is associated with two types of tasks: $Factor(k)$ and $Update(k, j)$ for $1 \leq k < j \leq N$. 1) A task $Factor(k)$ is to factorize all the columns in the k -th panel, including finding the pivoting sequence associated with those columns and updating the lower triangular portion of panel k . The pivoting sequence is held until the factorization of the k -th panel is completed. Then the pivoting sequence is applied to the rest of the matrix. This is called “delayed pivoting” [2]. The $Factor()$ tasks mainly use BLAS-1 and BLAS-2 subroutines. 2) Task $Update(k, j)$ is to use panel k to modify panel j . That includes “row swapping” using the result of pivoting derived by $Factor(k)$, “scaling” which uses the factorized $A_{k,k}$ to scale $A_{k,j}$, and “updating” which uses $A_{i,k}$ and $A_{k,j}$ to modify $A_{i,j}$. The $Update()$ tasks mainly use BLAS-2 and BLAS-3 subroutines.

1-D approaches. We call the set of updating tasks $Update(k, j), k < j \leq N$ as the stage k matrix updating. In [4], we have implemented two different parallel methods with 1-D data mapping as follows.

- **The compute-ahead algorithm (CA).** This algorithm uses the column-block cyclic mapping. The pivoting process proceeds in the order of k , i.e., $Factor(k+1)$ is started after $Factor(k)$. But the $Update()$ tasks can be overlapped. The CA schedule tries to execute the pivoting tasks as early as possible since they are normally in the critical path of the computation.

- The 1-D RAPID code. This code uses a DAG to model irregular parallelism and the RAPID system [3] to schedule the tasks. Then RAPID will execute the DAG on a distributed memory machines using a low-overhead communication scheme. In [4], we show that the 1-D RAPID code outperforms the CA code because graph scheduling exploits the parallelism more aggressively.

While the above approaches are reasonable as the first step towards efficient parallelization of sparse LU on distributed memory machines, they have a disadvantage that 1-D partitioning can expose limited parallelism. Also in order to support the concurrency among multiple updating stages in CA, multiple buffers are needed to keep pivoting panels on each processor. Thus the space complexity is high. The current RAPID implementation in [4] also uses extra memory space for supporting general irregular computations.

3 The 2-D algorithm

In this section we present an asynchronous sparse LU factorization using 2-D mapping. We still employ the optimization strategies such as static symbolic factorization, nonsymmetric L/U supernode partitioning and amalgamation strategy. The computation of a task $Factor(k)$ or $Update(k, j)$ is distributed to a column of nodes in the processor grid. The main challenge is the coordination of pivoting and data swapping across a subset of processors to minimize the overhead, exploit maximum amount of parallelism and maintain lower space usage. The ideas of our approach is summarized below. The goal is to eliminate global barriers in the code design, which maximizes concurrency. For the simplicity of description, we assume that the processor grid is a square of size $\sqrt{p} \times \sqrt{p}$.

- The **computation flow** is still controlled by the pivoting tasks $Factor(k)$. Namely, $Factor(k+1)$ will start after $Factor(k)$. However, the $Update()$ tasks can be overlapped. The maximum degree of overlapping is \sqrt{p} . Naturally it is ideal if pivoting operations can be conducted in parallel as much as possible. But we have not found a good way to do this unless we use the graph scheduling techniques. Thus a certain amount of parallelism is lost while the code is relatively easy to be implemented.
- The **pivoting process** within $Factor(k)$ is parallelized as follows. Let the processor that owns the block $A_{k,k}$ be called as $P_{k,k}$. For each column m in the k -th panel, each processor in this column finds the local maximum value and the corresponding subrow index independently. Then it sends the subrow with the local maximum to $P_{k,k}$. After receiving all the subrows with the local maxima, processor $P_{k,k}$ finds the global maximum and the corresponding subrow index t from the collected subrows, and multicasts the subrow t to processors in the same column for further updating of the lower triangular portion of panel k . It also sends the original m -th subrow in $A_{k,k}$ to the processor owning the global maximum for swapping. Notice that the whole subrow is communicated when each processor reports its local maximum to $P_{k,k}$. In this way $P_{k,k}$ can swap with the subrow t locally without further synchronization. This shortens the waiting time to conduct further updating with a little more communication volume. In the algorithm, each processor maintains \sqrt{p} buffers to collect the possible selected subrows for pivoting and each buffer is of size b . The synchronization point is maintained at $P_{k,k}$ which performs a global maximum selection when each processor reports its local maximum.

The **Updating** of the lower triangular portion of panel k in $Factor(k)$ can be conducted in parallel as soon as the selected subrow t is available locally. After all the columns in panel k are factorized, the **pivoting sequence** derived by $Factor(k)$ is broadcasted to all processors. Since the $Factor(k)$ is performed in a column of the processor grid, each

processor in this column has the pivoting sequence. Thus the pivoting sequence information can be broadcasted along the rows of the processor grid simultaneously.

While other processors in the grid are conducting swapping operations, processor P_{kk} needs to broadcast factorized $A_{k,k}$ along its own processor row. Similarly each processor in the same processor column as $P_{k,k}$ broadcasts the updated panel k along the row direction of the processor grid.

Considering there could be at most \sqrt{p} stages of matrix updating overlapped, each processor should have \sqrt{p} buffers to receive the result of $Factor()$. Each buffer has size Nb^2/\sqrt{p} .

- The first thing to be done in $Update(k, j)$ is **row swapping**. Swapping in different updating stages can be overlapped on different columns of processors. Each processor needs to swap some subrows out while receiving some subrows. At most there are \sqrt{p} buffers, each buffer needs to hold swapping subrows from other processors with size Nb/\sqrt{p} .

Scaling in $Update(k, j)$ uses $A_{k,k}$ to modify $A_{k,j}$. They can be done in parallel immediately after row swapping. After scaling, the nonzero blocks of $A_{k,j}$, $k+1 \leq j \leq N$, are broadcasted along the column direction of the processor grid. Considering the maximum number of overlapping of updating, each processor has \sqrt{p} buffers and each buffer is of size Nb^2/\sqrt{p} . The final **updating** of the submatrix $A_{k+1:N, k+1:N}$ in $Update(k, j)$, $k+1 \leq j \leq N$, can be done in parallel. If nonzero submatrices $A_{i,k}$ and $A_{k,j}$ are available, they can be used to update the nonzero submatrix $A_{i,j}$.

4 A comparison with 1-D data mapping

We study the performance issues of the 2-D algorithm and compare it with the two 1-D algorithms. Let the number of nonzero lower triangular submatrices be X and the number of nonzero upper triangular submatrices be Y . The total number of nonzero blocks in matrix A is $Z = X + Y + N$. Let n be the dimension of the matrix. Again for the simplicity of description, we assume that the processor grid is $\sqrt{p} \times \sqrt{p}$, and that each submatrix is a $b \times b$ dense block. In the actual implementation this is not always true. We also assume that $Y \approx X \approx Z/2$ and $X, Y \gg N$.

- **Parallelism and load balancing.** Since the operating units are submatrices instead of panels, the 2-D algorithm exploits parallelism in the submatrix level. The degree of possible parallelism in the 2-D algorithm is $O(Z)$ while that for the 1-D algorithms is $O(N)$. The finer granularity in the 2-D algorithm gives more flexibility for load balancing. The 1-D approach has limited parallelism thus the load balancing is more difficult.

- **Memory requirement.** Blocks are distributed evenly onto processors. But each processor needs extra buffer space to hold messages for asynchronous execution. In the CA algorithm, the extra memory requirement for communication is about $Zb^2(1 - 1/p)$. The 2-D algorithm needs about $2Nb^2$ buffer space. Thus the 2D algorithm is capable of solving large problem instances. The implementation of RAPID in [3] also requires that each processor have about $Zb^2(1 - 1/p)$ space for holding messages.

- **Communication volume.** Assume at each stage, the probability for swapping rows (due to pivoting) across processors is q . In the 1-D approach, the total communication volume is about $p(X + N)b^2 \approx 0.5pZb^2$ while in the 2-D approach, the total communication volume is about $(\sqrt{p}(Z + 2N) + 2qY)b^2 \approx \sqrt{p}Zb^2$. Thus the 2-D algorithm has less communication volume comparing to the 1-D algorithm.

- **The total number of messages.** The total number of messages for 1-D is np , all of which are for broadcasting panels. In the 2-D approach it is about $\sqrt{p}(2n + Z) + 2qn$ and

the messages include those for finding pivoting, row swapping and submatrix broadcasting.

The above analysis shows that 2-D mapping has better scalability than 1-D mapping, because 2-D mapping has more flexibility for load balancing, less memory requirement and smaller communication volume. However 2-D mapping leads to more frequent communications.

5 Comparative experimental studies

Our experiments are conducted on Cray-T3D. Each processing element of T3D has 64 megabytes of memory with an 8K cache. The BLAS-3 routine DGEMM can achieve 103 MFLOPS. Testing matrices are from the Harwell-Boeing collection except “goodwin” matrix.

Overall performance of the 2-D code. The overall speedups of the 2-D code are shown in Figure 2(a). The results show that the 2-D code scales up to 64 processors for those matrices. The speedups are reasonable. All speedups are obtained by comparing with the S^* sequential code which is in average 1.4 times slower than the SuperLU sequential code for those cases [4].

In Figure 2(b), we compare the average parallel time differences between the 2-D approach and the two 1-D approaches by computing the ratio: $\frac{PT_{1D}}{PT_{2D}} - 1$. The 1-D RAPID code achieves the best performance because it uses sophisticated graph scheduling technique to guide the mapping of panels which results in better overlapping of communication with computation. The 1-D CA code uses a little bit scheduling and performs slightly better than the 2-D code on a small number of processors. But on 32 and 64 processors the 2-D code catches up. We will explain this issue by analyzing the load balance factor below.

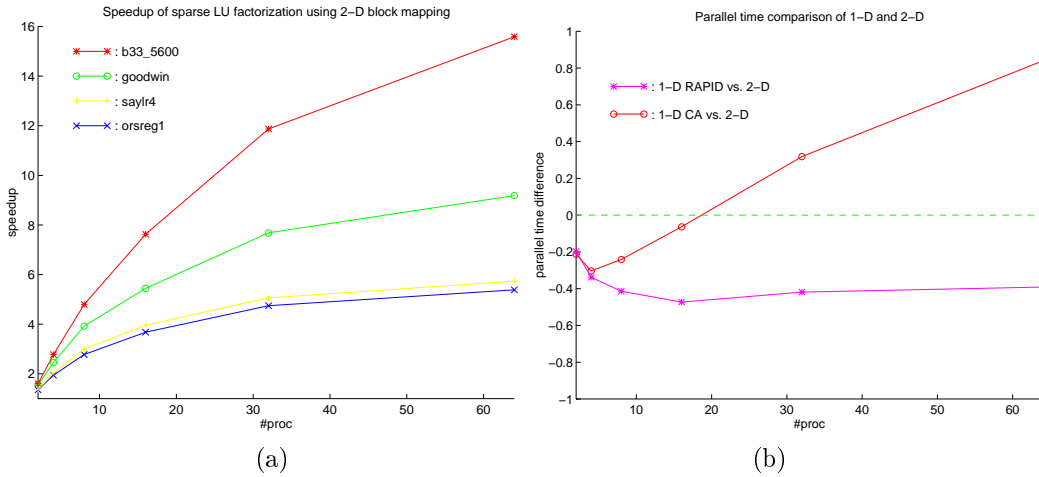


FIG. 2. (a) Parallel speedups on Cray-T3D; (b) Average parallel time difference comparisons.

Scalability and load balancing. We measure the scalability of an algorithm by $S(2^i) = PT(2^{i-1})/PT(2^i)$, where $PT(2^i)$ is the parallel time on 2^i processors. Figure 3(a) shows that 2-D approach has the best scalability on a large number of processors. For instance, the 2-D algorithm achieves 201.35 MFLOPS on 16 processors for B33.5800, while RAPID code achieves 302.5 MFLOPS. But on 64 processors the 2-D algorithm achieves 533.68 MFLOPS, better than the RAPID code which achieves 519.3 MFLOPS.

We analyze the load balancing factor in Figure 3(b). The load balance factor is defined as $work_{total}/(P \cdot work_{max})$ [13]. Here we only count the work from the updating part

because it is the major part of the computation. The 2-D code has the best load balance, which can make up for the impact of lacking of efficient task scheduling. That is the main reason to make the 2-D the most scalable approach.

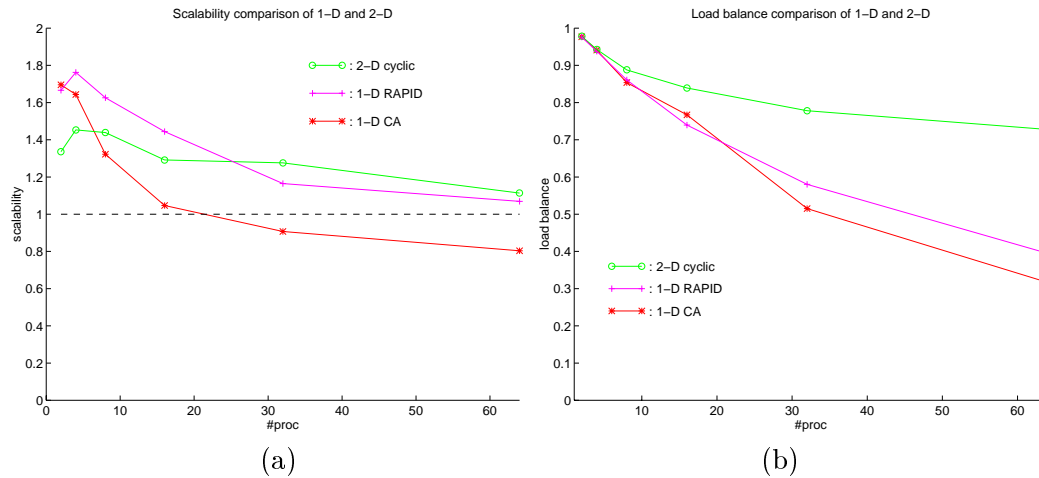


FIG. 3. Scalability and load balancing comparisons.

Impact of inter-processor pivoting and swapping. We further examine the impact of the overhead caused by inter-processor pivoting and swapping. In Table 1, it shows the distribution of the execution times for pivoting and swapping in the 1-D CA code and the 2-D code. The first item is for “Factorizing and waiting”. This is the period when a column of processors (only one processor in the 1-D case) are working on $Factor(k)$, other processors have to wait for the factorizing results. The “Swapping” is for the time of row-interchanges.

TABLE 1
Execution time distributions for 1-D CA code and 2-D code

| Part | P=4 | | P=16 | | P=64 | |
|-----------------------|--------|--------|--------|--------|--------|--------|
| | 1-D CA | 2-D | 1-D CA | 2-D | 1-D CA | 2-D |
| Factorizing + waiting | 18.44% | 23.71% | 56.25% | 39.66% | 81.17% | 52.80% |
| Swapping | 4.05% | 6.77% | 2.03% | 8.77% | 0.77% | 7.88% |

The results show that the increase of “Swapping” time is insignificant. However the “Factorizing + waiting” dominates the execution time and it affects the parallel time significantly because most likely it will be in the critical path of the computation. For a small number of processors, the 2-D code has more overhead percentage for this part. But as the number of processors increases, this part becomes the bottleneck of 1-D CA code. The reason is as follows. When a small number of processors are used, the 1-D code has good load balancing. However on a large number of processors, the load balancing for the 1-D code deteriorates. On the other hand, the 2-D approach starts to perform better because the parallelization of $Factor()$ tasks begins to show advantages and the loads remain well balanced.

Synchronous versus asynchronous 2-D code. Using global barrier in the 2-D code simplifies the implementation, but it cannot overlap computations among different updating stages. We list average parallel time reduction using our asynchronous design in

Table 2. We obtain an average 5-26% improvement, which demonstrates the importance of the barrier free design.

TABLE 2
Performance improvement of 2-D asynchronous code over 2-D synchronous code

| | P=2 | P=4 | P=8 | P=16 | P=32 | P=64 |
|---------|-------|-------|--------|--------|--------|--------|
| Average | 5.27% | 7.13% | 15.57% | 16.95% | 25.93% | 26.07% |

6 Conclusions

In this paper, we present an asynchronous sparse LU factorization algorithm using 2-D block mapping, compare its performance with two algorithms using 1-D column-block mapping. The preliminary experimental results show that 2-D mapping results in better scalability and load balance than 1-D column-block mapping, although the 2-D mapping requires frequent communication. Another important point of the 2-D code is that it requires less memory and therefore has more potential to solve larger problems. We are currently working on the memory-efficient design of RAPID [5] and testing larger problem instances for the RAPID and 2-D code.

References

- [1] C. Ashcraft, R. Grimes, J. Lewis, B. Peyton, and H. Simon, *Progress in Sparse Matrix Methods for Large Sparse Linear Systems on Vector Supercomputers*, International Journal of Supercomputer Applications, 1 (1987), pp. 10–30.
- [2] J. Demmel, *Numerical Linear Algebra on Parallel Processors*. Lecture Notes for NSF-CBMS Regional Conference in the Mathematical Sciences, June 1995.
- [3] C. Fu and T. Yang, *Run-time Compilation for Parallel Sparse Matrix Computations*, in Proceedings of ACM Inter. Conf. on Supercomputing, Philadelphia, May 1996, pp. 237–244.
- [4] ———, *Sparse LU Factorization with Partial Pivoting on Distributed Memory Machines*, in Proceedings of Supercomputing’96, Pittsburgh, Nov. 1996.
- [5] ———, *Space and Time Efficient Execution of Parallel Irregular Computation*, 1997. In preparation.
- [6] K. Gallivan, B. Marsolf, and H. Wijshoff, *The Parallel Solution of Nonsymmetric Sparse Linear Systems using H^* Reordering and an Associated Factorization*, in Proc. of ACM International Conference on Supercomputing, Manchester, July 1994, pp. 419–430.
- [7] A. George and E. Ng, *Parallel Sparse Gaussian Elimination with Partial Pivoting*, Annals of Operations Research, 22 (1990), pp. 219–240.
- [8] A. Gupta and V. Kumar, *Optimally Scalable Parallel Sparse Cholesky Factorization*, in Proc. of 7th SIAM Conf. on Parallel Processing for Scientific Computing, Feb. 1995, pp. 442–447.
- [9] S. Hadfield and T. Davis, *A Parallel Unsymmetric-pattern Multifrontal Method*, Tech. Rep. TR-94-028, CIS Department, University of Florida, Aug. 1994.
- [10] S. C. Kothari and S. Mitra, *A Scalable 2-D Parallel Sparse Solver*, in Proc. of Seventh SIAM Conference on Parallel Processing for Scientific Computing, Feb. 1995, pp. 424–429.
- [11] X. Li, *Sparse Gaussian Elimination on High Performance Computers*, PhD thesis, CS, UC Berkeley, 1996.
- [12] E. Rothberg, *Exploiting the Memory Hierarchy in Sequential and Parallel Sparse Cholesky Factorization*, PhD thesis, Dept. of Computer Science, Stanford, Dec. 1992.
- [13] E. Rothberg and R. Schreiber, *Improved Load Distribution in Parallel Sparse Cholesky Factorization*, in Proc. of Supercomputing’94, Nov. 1994, pp. 783–792.
- [14] R. Schreiber, *Scalability of Sparse Direct Solvers*, vol. 56 of Graph Theory and Sparse Matrix Computation (Edited by Alan George and John R. Gilbert and Joseph W.H. Liu), Springer-Verlag, New York, 1993, pp. 191–209.