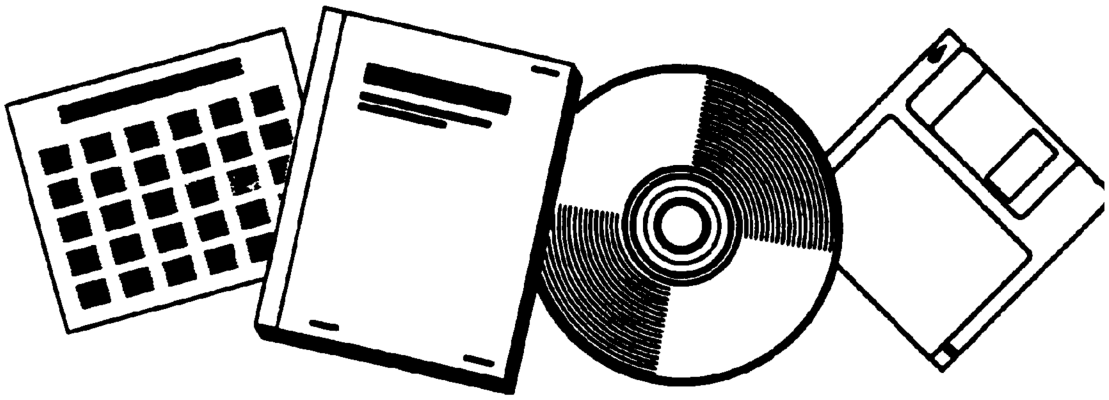
EFFICIENT BLOCK-ORIENTED APPROACH TO PARALLEL SPARSE CHOLESKY FACTORIZATION

STANFORD UNIV., CA

13 JUL 92



BIBLIOGRAPHIC INFORMATION

PB96-151543

Report Nos: STAN-CS-92-1438, CSL-TR-92-533

Title: Efficient Block-Oriented Approach to Parallel Sparse Cholesky Factorization.

Date: 13 Jul 92

Authors: E. Rothberg and A. Gupta.

Performing Organization: Stanford Univ., CA. Computer Systems Lab.

Sponsoring Organization: *Defense Advanced Research Projects Agency, Arlington, VA.

Contract Nos: DARPA-N00039-91-C0138

NTIS Field/Group Codes: 62B (Computer Software)

Price: PC A03/MF A01

Availability: Available from the National Technical Information Service, Springfield, VA. 22161

Number of Pages: 28p

Keywords: *Cholesky factorization, *Computation, *Parallel processing, Computer systems performance, Run time(Computers), Matrices(Mathematics), Decomposition, Massively parallel processors.

Abstract: This paper explores the use of a sub-block decomposition strategy for parallel sparse Cholesky factorization, in which the sparse matrix is decomposed into rectangular blocks. Such a strategy has enormous theoretical scalability advantages over a more traditional column-oriented decomposition for large parallel machines. However, little progress has been made in producing a practical sub-block method. This paper describes and evaluates an approach that is both simple and efficient.

July 1992

Report No. STAN-CS-92-1438
Also Numbered CSL-TR-92-533

PB96-151543

An Efficient Block-Oriented Approach to Parallel Sparse Cholesky Factorization

by

Edward Rothberg and Anoop Gupta

Department of Computer Science

Stanford University

Stanford, California 94305

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1.

2. REPORT DATE

3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE

An Efficient Block-Oriented Approach to Parallel Sparse Cholesky Factorization

5. FUNDING NUMBERS

N00039-91-C0138

6. AUTHOR(S)

Edward Rothberg and Anoop Gupta

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Computer Science Dept./Computer Systems Laboratory
Stanford University
Stanford, CA 94305

8. PERFORMING ORGANIZATION REPORT NUMBER

STAN-CS-92-1438
CSL-TR-92-533

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

DARPA/CSTO
3707 N. Fairfax
Arlington, VA 22203-1714

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Abstract

This paper explores the use of a sub-block decomposition strategy for parallel sparse Cholesky factorization, in which the sparse matrix is decomposed into rectangular blocks. Such a strategy has enormous theoretical scalability advantages over a more traditional column-oriented decomposition for large parallel machines. However, little progress has been made in producing a practical sub-block method. This paper describes and evaluates an approach that is both simple and efficient.

14. SUBJECT TERMS

Sparse Cholesky factorization, parallel processing

15. NUMBER OF PAGES

25

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	

An Efficient Block-Oriented Approach To Parallel Sparse Cholesky Factorization

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

July 13, 1992

Abstract

This paper explores the use of a sub-block decomposition strategy for parallel sparse Cholesky factorization, in which the sparse matrix is decomposed into rectangular blocks. Such a strategy has enormous theoretical scalability advantages over a more traditional column-oriented decomposition for large parallel machines. However, little progress has been made in producing a practical sub-block method. This paper describes and evaluates an approach that is both simple and efficient.

1 Introduction

The Cholesky factorization of sparse symmetric positive definite systems is an extremely important computation, arising in a variety of scientific and engineering applications. Sparse Cholesky factorization is unfortunately also extremely time-consuming, and is frequently the computational bottleneck in these applications. Consequently, there is significant interest in performing the computation on large parallel machines. Several different approaches to parallel sparse Cholesky factorization have been proposed. While great success has been achieved for small parallel machines, success has unfortunately been quite limited for larger machines. The reasons vary, depending on the particular approach.

One important group of parallel sparse factorization approaches includes those that assume that all processors access a single, shared memory. While a shared-memory programming model greatly simplifies the parallelization task, and has led to extremely high-performance implementations [1, 6, 18], it also inherently limits the parallel scalability of the approach. A large number of processors simply can not share a single memory.

A second important group of parallel methods avoids this scalability problem by working within a distributed memory model instead. In nearly all such approach, columns of the sparse matrix are distributed among the memories of the processors [4, 12, 16]. Unfortunately, such approaches have achieved only limited success. Primary among the limitations of these approaches are the enormous amounts of interprocessor communication they require, leading to saturation of the processor interconnection networks, and the limited amounts of concurrency they expose in the sparse problem, limiting the number of processors that can be effectively used for a particular sparse matrix.

Fortunately, these limitations can be overcome (in theory) by moving to a different matrix distribution strategy. Rather than representing the sparse matrix as a set of columns, it can instead be represented as a set of rectangular blocks, with these blocks distributed among the processors. Such a 2-D decomposition has been shown to be extremely effective for parallel dense factorization [21]. It is not clear, however, whether a similar decomposition would be practical for sparse problems. A few investigations [2, 20, 22] have been performed, but these contained little or no exploration of practical algorithms.

This paper focuses on two practical and important questions related to a 2-D decomposition approach. First, we consider the complexity of a parallel sparse factorization program that manipulates sub-blocks. We find that a block approach need not be much more complicated than a column approach. We describe a simple strategy

for performing a block decomposition and a simple parallel algorithm for performing the sparse Cholesky computation in terms of these blocks. The approach retains the theoretical scalability advantages of block methods. We term this block algorithm the *block fan-out method*, since it bears a great deal of similarity to the parallel column fan-out method [12].

Another important issue in a block approach is the issue of efficiency. While parallel scalability arguments can be used to show that a block approach would give better performance than a column approach for extremely large parallel machines, these arguments have little to say about how well a block approach performs on smaller machines. Our goal is to develop a method that is efficient across a wide range of machine sizes. We explore the efficiency of the block approach in two parts. We first consider a sequential block factorization code and compare its performance to that of a true sequential program to determine how much efficiency is lost in moving to a block representation. The losses turn out be quite minor. We then consider parallel block factorization, looking at the issues that potentially limit its performance. The parallel block method is found to give extremely high performance even on small parallel machines. For larger machines, performance is good but not excellent, primarily due to load balance problems. We quantify the load imbalances and investigate the causes.

This paper is organized as follows. We begin in Section 2 with some background on sparse Cholesky factorization. Section 3 then discusses our experimental environment, including a description of the sparse matrices we use as benchmarks and the machines we use to study the parallel block factorization approach. Section 4 describes our strategy for decomposing a sparse matrix into rectangular blocks. Section 5 describes a parallel method that performs the factorization in terms of these blocks. Section 6 then evaluates the parallel method, both in terms of communication volume and achieved parallel performance. Section 7 gives a brief discussion, and finally conclusions are presented in Section 8.

2 Sparse Cholesky Factorization

The goal of the sparse Cholesky computation is to factor a sparse symmetric positive definite matrix A into the form $A = LL^T$, where L is lower triangular. The computation is typically performed as a series of three steps. The first step, *heuristic reordering*, reorders the rows and columns of A to minimize *fill* in the factor matrix L. The second step, *symbolic factorization*, performs the factorization symbolically to determine the non-zero structure of L given a particular reordering. Storage is allocated for L in this step. The third step is the *numerical factorization*, where the actual non-zero values in L are computed. This step is by far the most time-consuming, and it is the focus of this paper. We refer the reader to [13] for more information on these steps.

The following pseudo-code performs the numerical factorization step:

```
1.  for k = 0 to n do
2.      for i = k to n do
3.          L_ik ← L_ik / sqrt(L_kk)
4.      for j = k+1 to n do
5.          for i = j to n do
6.              L_ij ← L_ij − L_ik L_jk
```

The computation is typically expressed in terms of columns of the sparse matrix. Within a column-oriented framework, steps 2 and 3 are typically thought of as a single operation, called a column division or $cdiv()$ operation. Similarly, steps 5 and 6 form a column modification, or $cmod(j, k)$, operation. The computation then looks like:

```
1.  for k = 0 to n do
2.      cdiv(k)
3.      for j = k+1 to n do
4.          cmod(j, k)
```

Only the non-zero entries in the sparse matrix are stored, and the computation only performs operations on non-zeroes.

This column-oriented formulation of the sparse factorization has formed the basis of several parallel sparse factorization algorithms, including the fan-out method [12], the fan-in method [4], and the distributed multifrontal method [16]. The details of these various methods are not relevant to our discussion, so we refer the reader to the relevant papers for more information. We simply note that for each of these methods, communication volume can be shown to grow linearly in the number of processors [14]. Since available communication bandwidth in a multiprocessor typically grows much more slowly, this communication growth represents a severe scalability limitation.

Recent research in parallel sparse Cholesky factorization [3] has shown that the communication needs of column-oriented sparse factorization can be greatly reduced. Through limited replication of data and careful assignment of tasks to processors, communication can be made to grow as the square root of the number of processors, thus improving scalability. Communication volume is not the only thing that limits scalability in column-oriented approaches, however. A column formulation also leads to very long critical paths, thus placing a large lower bound on parallel runtime. For a dense $n \times n$ matrix, the sequential computation requires $O(n^3)$ runtime while the best case parallel runtime is $O(n^2)$. Similar bounds apply for sparse problems.

An alternative formulation of the factorization problem divides the matrix into rectangular subblocks. The dense Cholesky computation, expressed in terms of subblocks, would look like:

```
1. for K = 0 to N do
2.     L_KK — Factor(L_KK)
3.     for I = K + 1 to N do
4.         L_IK — L_IK L_KK^-1
5.     for J = K + 1 to N do
6.         for I = J to N do
7.             L_IJ — L_IJ − L_IK L_JK^T
```

In this pseudo-code, I, J, and K iterate over rows and columns of sub-blocks. It can be shown that this formulation leads to greatly reduced communication volumes and exposes significantly more concurrency. Specifically, communication volume can be shown to grow as the square root of the number of processors, and the critical path can be shown to grow as $O(n)$. It is an open question whether this formulation can be efficiently applied to parallel sparse factorization, and this is the question that we address here.

Before we begin our discussion of a block decomposition of the sparse matrix, we first discuss two important concepts in sparse factorization that will be relevant to our presentation. The first is the concept of a *supernode* [6]. A supernode is a set of adjacent columns in the factor matrix L with identical non-zero structures. Supernodes arise in any sparse factor, and they are typically quite large. By formulating the sparse factorization computation as a series of supernode-supernode modifications, rather than column-column modifications as described before, the computation can make substantial use of dense matrix operations. The result is substantially higher performance on vector supercomputers and on machines with hierarchical memory systems. For more details on supernodal factorization, see [1, 6, 17]. The regularity in the sparse matrix captured by this supernodal structure will prove useful in this paper for producing an effective decomposition of the sparse matrix into rectangular blocks. We will return to this issue shortly.

Another important notion in sparse factorization is that of the *elimination tree* of the sparse matrix [19]. This structure concisely captures important dependency information. If each column of the sparse matrix is thought of as a node in a graph, then the elimination tree is defined by the following parent relationship:

$$parent(j) = \min\{i | l_{ij} \neq 0, i > j\}.$$

It can be shown that a column is only modified by descendent columns in the elimination tree, and equivalently that a column only modifies ancestors. The most important property captured in this tree for parallel factorization is the property that subtrees are independent, and consequently can be processed in parallel. This fact will be relevant later in this paper.

Table 1: Benchmarks.

	Name	Description	Equations	Non-zeroes
1.	GRID100	5-point discretization of 2D region	10,000	39,600
2.	GRID200	Larger instance of 1	40,000	159,200
3.	BCSSTK15	Module of an offshore platform	3,948	113,868
4.	BCSSTK16	Corps of Engineers dam	4,884	285,494
5.	BCSSTK17	Elevated pressure vessel	10,974	417,676
6.	BCSSTK18	R.E. Ginna nuclear power station	11,948	137,142
7.	BCSSTK29	Boeing 767 rear bulkhead	13,992	605,496

Table 2: Factor matrix statistics.

	Name	FP ops	Non-zeroes in L	Supernodes
1.	GRID100	15,707,205	250,835	6,672
2.	GRID200	137,480,183	1,280,743	26,669
3.	BCSSTK15	165,039,042	647,274	1,295
4.	BCSSTK16	149,105,832	736,294	691
5.	BCSSTK17	144,280,005	994,885	2,595
6.	BCSSTK18	140,919,771	650,777	7,438
7.	BCSSTK29	393,059,150	1,680,804	3,231

3 Experimental Environment

Since one of our interests in this paper is to consider practical performance issues for block methods, we will present several performance numbers for realistic sparse matrices factored on real machines. This section briefly describes both the matrices that we use as benchmarks and the machine on which we perform the factorizations.

3.1 Benchmark Matrices

The benchmark matrices we consider in this paper are drawn from the Boeing/Harwell sparse matrix test set [7]. Since our interest is on factorization on large machines, we have chosen some of the largest sparse matrices in the collection. Table 1 gives brief descriptions of the matrices. Table 2 gives information about the factors of these matrices. The first column of numbers shows a count of the number of floating-point operations required for the factorization. The second column gives the number of non-zeroes in L. The third gives the number of distinct supernodes in the factor matrices. Note that all matrices except the two grid problems are preordered using the multiple minimum degree ordering heuristic [15] before being factored. A simple nested dissection ordering is used for the grid problems.

3.2 Target Machine

Although many of the results presented in this paper will be machine-independent, the paper will also include some performance numbers from real parallel machines. We now briefly describe the parallel machines that are considered.

3.2.1 Moderately-Parallel Machine

Performance numbers for sequential and moderately-parallel machines are obtained from a Silicon Graphics 4D/380 multiprocessor. The 4D/380 contains eight high-performance RISC processors, each consisting of a MIPS R3000 integer unit and an R3010 floating-point co-processor. The processors execute at 33 MHz, and are rated at 27 MIPS and 4.9 double-precision LINPACK MFLOPS. They are connected with a bus having a peak throughput of approximately 67 MBytes per second.

Each processor in the 4D/380 has a 64 KByte instruction cache, a 64 KByte first-level data cache, and a 256 KByte second-level data cache. The caches play a crucial role in determining performance. Memory references that hit in the first-level cache are serviced in a single cycle. References that miss in both levels of the cache require roughly 50 cycles to service, and they bring 4 16-byte cache lines into the cache. In contrast, a floating-point multiply requires 5 cycles and a floating-point add requires 2 cycles.

3.2.2 Larger Parallel Machine

In order to provide performance results for machines with more than 8 processors, this paper also makes use of multiprocessor simulation. To keep simulation costs manageable, we perform this simulation in terms of high-level factorization tasks. A single task might represent a block update operation or the transmission of a large message from one processor to another. The costs of the individual high-level tasks are obtained through a simple performance model. The parallel simulation is performed as a discrete-event simulation of these tasks.

The three most important costs that are modeled in this simulation are the costs of performing floating-point operations, fetching data from the local memory of a processor, and communicating data between processors. We now describe our model for each in more detail.

An important cost in a parallel factorization will clearly be the cost of executing the machine instructions that perform the required floating-point operations. For the sake of normalization, we assume that one floating-point operation requires one time unit. Note that this machine instruction term is meant to capture not only the cost of the single instruction that actually performs the floating-point operation, but also the costs of all instructions that are required to support a floating-point operation (such as index computations and memory load instructions).

Another, potentially even more important component of performance is the cost of moving data between a processor's memory and its cache. Our assumption is that a cache miss on one double-precision value takes roughly 5 times as long as a floating-point operation. This number is quite accurate for the SGI 4D/380 and is a reasonable estimate for a wide range of current generation hierarchical-memory machines as well. The cache is assumed to be large enough to hold three 32 by 32 blocks of data. To simplify our simulation, we further assume that all reuse of cached data occurs within block operations. That is, we assume that each block operation begins with an empty cache.

While this computational cost model may appear too simplistic to capture the intricacies of current and future microprocessors, which might include such complex features as heavily pipelined floating-point units and non-blocking caches, we believe this model will in fact provide relatively accurate estimates. No matter what the internal structure of a processor, we believe that its performance for this computation can be understood using the answers to two questions. First, what performance is achieved when virtually no memory references miss in cache? And second, what performance is achieved when all references miss in cache? While heavier pipelining will allow the processor to hide some of the latencies of memory accesses, at the same time memory latencies will continue to increase so there will be more latencies to hide. As a result, we believe that the processor will have to pay some cost for each cache miss.

The third cost that is modeled is the cost of interprocessor communication. Our model assumes that all communication takes place in the form of interprocessor messages. It also assumes that messages are handled by a message co-processor, and therefore cost the sending and receiving processors nothing to process. The true cost of a message is the time it spends in the interconnection network. Our model assumes that this time is determined by the length of the message and the available communication bandwidth between the sending and receiving processors. Communication bandwidth is assumed to be one-tenth of computation bandwidth. That is, a processor can perform ten floating-point operations in the time required to send one floating-point value. This number is roughly average for today's parallel machines. The Intel Touchstone machine, for example, has a roughly 8 to 1 computation to communication bandwidth ratio. The Intel Paragon machine will have a roughly 2 to 1 ratio. On the other hand, the Thinking Machines CM-5 has a roughly 50 to 1 ratio.

For an example of how this performance model would be applied, consider the following three example operations: (1) send a 32 by 32 block of data from one processor to another; (2) multiply the received block by another 32 by 32 block; (3) add the result into another 32 by 32 block. Operation 1 would send 1024 double-precision values and thus would require 10240 time units. Operation 2 would load 3 blocks into the processor cache, requiring 15360 time units, and would perform 65536 floating-point operations, requiring 65536 time units. Finally, operation 3 would load two blocks into the cache, requiring 10240 time units, and would perform

1024 floating-point operations, requiring 1024 time units.

We believe our model captures the most important aspects of parallel machine performance for modern multiprocessors, with one exception. Our model does not capture the effect of message contention in the processor interconnection network. Since the effect of contention is extremely difficult to model accurately, we instead discuss the issue in a qualitative way in sections where the performance model is used.

4 Block Formulation

Having described our evaluation environment, we now move on to the question of how to structure the sparse Cholesky computation in terms of blocks. Our first step in describing a block-oriented approach is to propose a strategy for decomposing the sparse matrix into blocks. Our goal in this decomposition is to retain as much of the efficiency of a sequential factorization computation as possible. Thus, we will keep a careful eye on the amount of computational overhead that is introduced.

4.1 Block Decomposition

We begin our discussion by considering some of the general issues that are important for a block approach. We also discuss how our approach addresses these issues. We believe the main issues that must be addressed are the following. First, blocks should be relatively dense. Since the blocks will be distributed among several processors, there will certainly be some overheads associated with manipulating and storing them. These overheads should be amortized over as many non-zeroes as possible. The block decomposition must therefore be tailored to match the non-zero structure of the sparse matrix. Another important issue is the ways in which blocks in the matrix interact with each other. If the interactions are complex, then the parallel computation can easily spend more time figuring out how blocks interact than it would spend actually performing the block operations. Finally, the individual block operations should be efficient.

The primary motivation behind our decomposition approach is to keep the block computation as simple and regular as possible. Our hope is that a regular computation will be an efficient computation. We keep the computation simple by avoiding two distinct types of irregularity: irregular interactions between blocks and irregular structure within blocks.

4.1.1 Irregular Interactions

Since a sparse matrix in general contains non-zeroes interspersed with zeroes throughout the matrix, it would appear desirable for a block decomposition to possess a large amount of flexibility in choosing blocks. This flexibility could be used to locally tailor the block structure to match the actual structure of the sparse matrix. One seemingly reasonable approach to a block decomposition of a sparse matrix, for example, would locate clumps of contiguous non-zeroes in the matrix and group these clumps together into blocks. This approach has serious problems, however, and we now discuss the advantage of giving up some flexibility and instead imposing a significant amount of rigidity on the decomposition.

The primary problem with a flexible approach to block decomposition concerns the way in which the resulting blocks would interact with each other. Recall that in sparse Cholesky factorization a single non-zero L_{ik} is multiplied with non-zeroes above it in the same column L_{jk} to produce updates to non-zeroes L_{ij} in row i and column j. When the matrix is divided into a set of rectangular blocks, the blocks interact in a similar manner. Consider the simple example in Figure 1. This figure shows a small set of dense blocks from a potentially much larger matrix. During the factorization, the block in the lower left will interact with a portion of the block above it to produce the indicated update, which must be subtracted from portions of the blocks to its right. Keep in mind that each of these blocks is potentially assigned to a different processor. Thus, for each update operation the processor performing that update must keep track of the set of blocks that are involved, the portions of these blocks that are affected, the processors on which these blocks can be found, and it must dole out the computed update to the relevant processors. Keeping track of all such block interactions would be enormously complicated and expensive. With a large number of blocks scattered throughout the matrix, the costs of this irregularity would quickly become prohibitive.

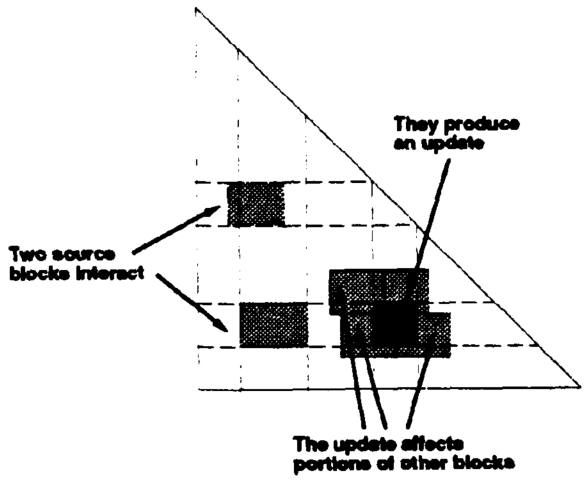


Figure 1: Example of irregular block interaction. Dotted lines indicate boundaries of affected areas.

In order to remove this irregularity and greatly simplify the structure of the computation, we decompose the matrix into blocks using global partitions of the rows and columns. In other words, the columns of the matrix $(1 \ldots n)$ are divided into contiguous sets $(\{1 \ldots p_2 - 1\}, \{p_2 \ldots p_3 - 1\}, \ldots, \{p_N \ldots n\}$, where N is the number or partitions and p_i is the first column in partition i). An identical partitioning is performed on the rows. A simple example is shown in Figure 2. A block L_{IJ} (we refer to partitions using capital letters) is then the set of non-zeroes that fall simultaneously in rows $\{p_I \ldots p_{I+1} - 1\}$ and columns $\{p_J \ldots p_{J+1} - 1\}$. The main advantage of this rigid distribution comes from the fact that blocks share common boundaries. A block L_{IK} now interacts with block L_{JK} in the same block column partition to produce an update to block L_{IJ}.

One possible weakness of a global partitioning strategy is that its global nature may not allow for locally good blocks. We will soon show that this is only a minor problem.

4.1.2 Irregular Block Structure

Another issue that can have a significant impact on the efficiency of the overall computation is the internal non-zero structure of a block. Just as we restricted the choice of block boundaries earlier to increase regularity across block operations, we now consider restrictions on the internal structures of blocks to increase regularity within a block operation.

Note first that allowing arbitrary partitionings of the rows and columns of the matrix would lead to blocks with arbitrary internal non-zero structures. Recall that a block update operation is performed by multiplying a block by the transpose of a block above it (as a matrix-matrix multiplication). With arbitrary non-zero structure within the blocks, the corresponding computation would be a sparse matrix multiplication, which is an inefficient operation in general.

In order to simplify the internal structure of the blocks and keep the computation as efficient as possible, we take advantage of the supernodal structure of the sparse matrix. Specifically, we choose partitions so that all member columns belong to the same supernode. Since the columns in a supernode all have the same non-zero structures, all resulting blocks will share this property. Thus, a block L_{IJ} will consist of some set of dense rows. A block may not be completely dense, since not all rows are necessarily present. A single structure vector keeps track of the set of rows present in a block. This sparsity within a block has little effect on the efficiency of the computation, as we shall soon show.

Before proceeding, we note that Ashcraft [2] proposed a similar decomposition strategy independently.

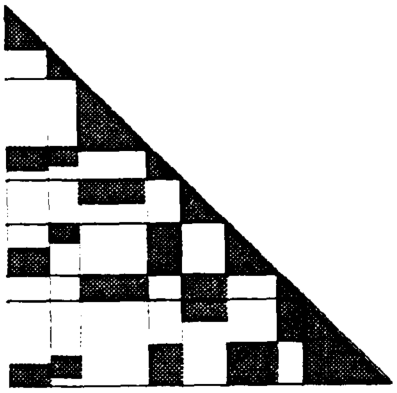


Figure 2: Example of globally partitioned matrix.

4.2 Structure of the Block Factorization Computation

Our goal in placing the above restrictions on blocks in the sparse matrix is to retain as much efficiency as possible in the block factorization computation. We now describe a sequential algorithm for performing the factorization in terms of these blocks and evaluate that algorithm's efficiency. The parallelization of the sequential approach that we derive here will be described later.

At one level, the factorization algorithm expressed in terms of blocks is quite obvious. The following pseudo-code, a simple analogue of dense block Cholesky factorization, performs the factorization. Note that I, J, and K iterate over the partitions in the sparse matrix.

```
1. for K = 0 to N do
2.     L_KK ← Factor(L_KK)
3.     for I = K + 1 to N with L_IK ≠ 0 do
4.         L_IK ← L_IK L_KK^{-1}
5.     for J = K + 1 to N with L_JK ≠ 0 do
6.         for I = J to N with L_IK ≠ 0 do
7.             L_IJ ← L_IJ − L_IK L_JK^T
```

The first thing to note about the above pseudo-code is that it works with a column of blocks at a time. Steps 2 through 4 divide block column K by the Cholesky factor of the diagonal block. Steps 5 through 7 compute block updates from all pairs of blocks in column K. We therefore store the blocks so that all blocks in a column can be easily located. The easiest way to do this is to store one column of blocks after another. One potential problem here is that step 7 updates some destination block L_{IJ} whose location can not easily be determined from the locations of the source blocks. To make this step efficient, a hash table of all blocks is kept.

Now consider the implementation of the individual operations in the pseudo-code. The block factorization in step 2 is quite straightforward to implement. Diagonal blocks are guaranteed to be dense, so this step is simply a dense Cholesky factorization. The multiplication by the inverse of the diagonal block in step 4 is also quite straightforward. This step does not actually compute the inverse of L_{KK}. Instead, it solves a series of triangular systems. While the block L_{IK} is not necessarily dense, the computation can be performed without consulting the non-zero structure of the block.

The remaining step in the above pseudo-code, step 7, is both the most important and the most difficult to implement. It is the most important because it sits within a doubly-nested loop and thus performs the vast majority of the actual computation. It is the most difficult because it works with blocks with potentially different

non-zero structures and must somehow reconcile these structures. More precisely, recall that a single block in L consists of some set of dense rows from among the rows that the block spans (see the example in Figure 2). When an update is performed in step 7 above, the structure of L_{IK} determines the set of rows in L_{IJ} that are affected. Similarly, the structure of L_{JK} determines the set of *columns* in L_{IJ} that are affected.

The block update computation is most conveniently viewed as a two stage process. A set of updates is computed in the first stage, and these updates are subtracted from the appropriate entries in the destination block in the second, or scatter stage. The first stage, the computation of the update, can be performed as a dense matrix-matrix multiplication. The non-zero structures of the source blocks L_{IK} and L_{JK} are ignored temporarily; the two blocks are simply multiplied to produce an update.

During the second stage, the resulting update must be subtracted from the destination. The most simple case occurs when the update has the same non-zero structure as the destination block. We have coded our dense matrix-matrix multiplication routine as a multiply-subtract (i.e., $C = C - AB^T$), rather than a multiply-add, so the destination block can be used as the destination directly, without the need for a second scatter stage.

Consider the more difficult case where the non-zero structures differ. The first step in this case is to compute a set of *relative indices* [19]. These indices indicate the corresponding position in the destination for each row in the source. Two sets of relatives indices are necessary in order to scatter a single block update; rel_i, the affected set of rows and rel_j, the affected set of columns.

The computation of relative indices is quite expensive in general, since it requires a search through the destination to find the row corresponding to a given source row. Fortunately, such a search is only rarely necessary due to an important special case. When the destination block has dense structure, the relative indices bear a trivial relationship to the source indices. Note that the rel_j indices always fall into this category, since the destination block always has dense column structure. We will be more precise about exactly how often relative index computations are necessary shortly.

Once relative indices have been computed, the actual scatter is performed as follows:

```
1.  for i = 0 to length_IK do
2.      for j = 0 to length_JK do
3.          L_IJ[rel_i[i]][rel_j[j]] -= L_IJ[rel_i[i]][rel_j[j]] - update[i][j]
```

Scattering is also somewhat expensive, and it is much more prevalent than relative index computation. The frequency with which relative index computations and scatters must be performed will be considered shortly.

In summary, the efficiency of a block update operation depends heavily on the non-zero structures of the involved blocks.

- The best case occurs when the update has the same structure as the destination. In this case, the $C = C - AB^T$ operation can use the destination block as its destination.
- The next best case occurs when the destination block is dense. The update must be scattered, but the relative indices can be computed inexpensively.
- The worst case occurs when the update has different structure from the destination and the destination block is sparse. The update must be scattered, and relative indices are relatively expensive to compute.

4.3 Performance of Block Factorization

We now look at the performance obtained with a sequential program that uses a block decomposition and block implementation. Since our end goal is to create an efficient *parallel* approach, performance is studied for the case where the matrix is divided into relatively small blocks. The blocks should not be too small, however, because of the overheads that will be associated with block operations. We consider 16 by 16, 24 by 24, and 32 by 32 block sizes. To produce blocks of the desired size B, we form partitions that contain as close to B rows/columns as possible. Since partitions represent subsets of supernodes, some partitions will naturally be smaller than B.

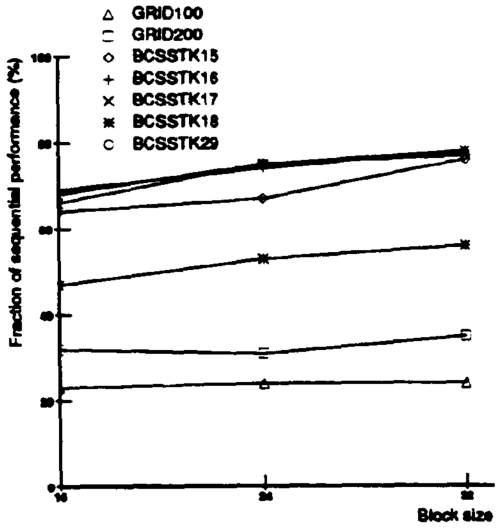


Figure 3: Performance of a sequential block approach, relative to a sequential left-looking supernode-supernode approach.

Table 3: Frequency of relative index computations and scatters for block method, compared with floating-point operations ($B = 16$).

Problem	Relative indices (relative to FP ops)	Scatters (relative to FP ops)
GRID100	0.37%	4.0%
GRID200	0.18%	2.4%
BCSSTK15	0.04%	1.6%
BCSSTK16	0.02%	1.4%
BCSSTK17	0.04%	1.8%
BCSSTK18	0.11%	2.6%
BCSSTK29	0.01%	1.0%

The performance obtained with the sequential block approach on a single SGI 4D/380 processor is shown Figure 3. This performance is expressed as a fraction of the performance obtained with an efficient sequential code (a supernode-supernode left-looking method; one of the most efficient sequential approaches [17]). From the figure, it is clear that the block approach is quite efficient. Typical efficiencies are roughly 65% for a block size of 16 and roughly 75% for a block size of 32. We will discuss the reasons why three of the matrices, GRID100, GRID200, and BCSSTK18, achieve significantly lower performance shortly.

Our earlier discussion indicated that the performance of the block method might suffer because of the need for relative index calculations and update scattering. In order to gauge the effect of these two issues on overall performance, Table 3 relates the amounts of scattering and relative index computation (for $B = 16$) to the number of floating-point operations performed in the factorization. The numbers are quite similar for the other block size choices. The first column compares the number of distinct relative indices computed against the number of floating-point operations. The second column compares distinct element scatters against floating-point operations. The table shows that even if relative index computations and scatters are much more expensive than floating-point operations, the related costs will be small. Clearly, the vast majority of block update operations produce an update with the same structure as the destination block.

It is also interesting to compare relative indices and scatters to those performed by a true sequential method.

Table 4: Frequency of relative index computations and scatters for block method, compared with sequential multifrontal method ($B = 16$).

Problem	Relative indices (relative to seq MF)	Scatters (relative to seq MF)
GRID100	78%	72%
GRID200	80%	69%
BCSSTK15	109%	105%
BCSSTK16	50%	88%
BCSSTK17	61%	90%
BCSSTK18	163%	91%
BCSSTK29	32%	40%

Table 4 gives the relevant numbers. In this case, the comparison is with a sequential multifrontal method, where notions of relative indices and scatters are easily quantified. The comparison is relevant for the left-looking supernode-supernode as well, however, since the two methods perform similar computations. Note that the block method performs a comparable number of relative index computations and scatters.

Ashcraft [2] has described methods for improving block structure and thus decreasing the need for scattering. It is our belief that a very simple block decomposition is more than adequate for keeping such costs in check.

4.4 Improving Performance

It is clear from the previous section that the block method is generally quite efficient. Recall, however, that the method was much less efficient than a true sequential method for several problems. Data on relative index and scatter frequency showed that these were not the source of the losses. The losses are actually due to overheads in the block operations.

Consider a single block update operation. It must find the appropriate destination block through a hash table, determine whether the source and destination blocks have the same structure, and then pay the loop startup costs for the dense matrix multiplication to compute the update. While these costs are trivial when all involved matrices are 32 by 32, in fact many blocks in the sparse matrix are quite small. In the case of matrix GRID100, for example, the average block operation when $B = 32$ performs only 96 floating-point operations, as compared to the 65536 operations that would be performed with 32 by 32 blocks. The average number of floating-point operations per block operation across the whole benchmark set is shown in Figure 4. Note that this figure quite accurately predicts the performance numbers seen in the previous figure.

The primary cause of small blocks in the block decomposition is the presence of small supernodes, and thus small partitions. To increase the size of these partitions, we now briefly consider the use of *supernode amalgamation* [5, 8] techniques. The basic goal of supernode amalgamation is to find pairs of supernodes that are nearly identical in non-zero structure. By relaxing the restriction that the sparse matrix only store non-zeroes, some zeroes can be introduced into the sparse matrix in order to make the sparsity structures of two supernodes the same. These supernodes can then be merged into one larger supernode. We refer the reader to [5] for more details on supernode amalgamation, and simply note that our amalgamation strategy merges supernodes quite aggressively.

In Figure 5 we show the average block operation sizes both before and after amalgamation. It is clear that amalgamation significantly increases the block operation grain size.

Before presenting performance comparisons, we first note that amalgamation does have a cost. By introducing zeroes into the sparse matrix, the amount of floating-point work is increased. To be fair, the performance of the block computation after amalgamation should therefore be compared with the performance of the sequential computation before this extra work is introduced. However, amalgamation also provides some benefit for sequential factorization, primarily related to improved use of the processor cache. We found that the benefit in fact outweighed the cost for the amalgamation strategy we employed on all benchmark matrices, with performance improvements ranging from 1% to 14% (see Table 5) for the true sequential method. Block method performance is therefore compared to the performance of the true sequential method *after* amalgamation.

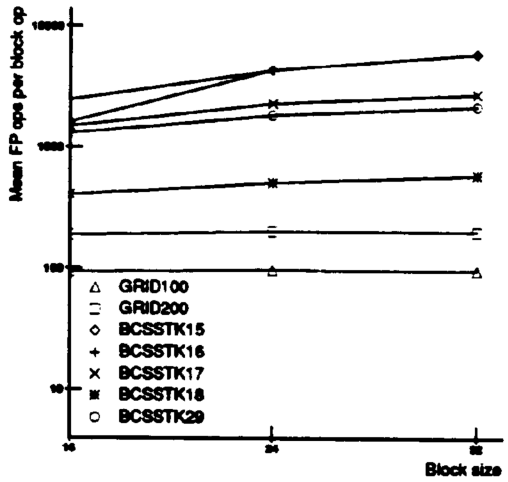


Figure 4: Average floating-point operations per block operation.

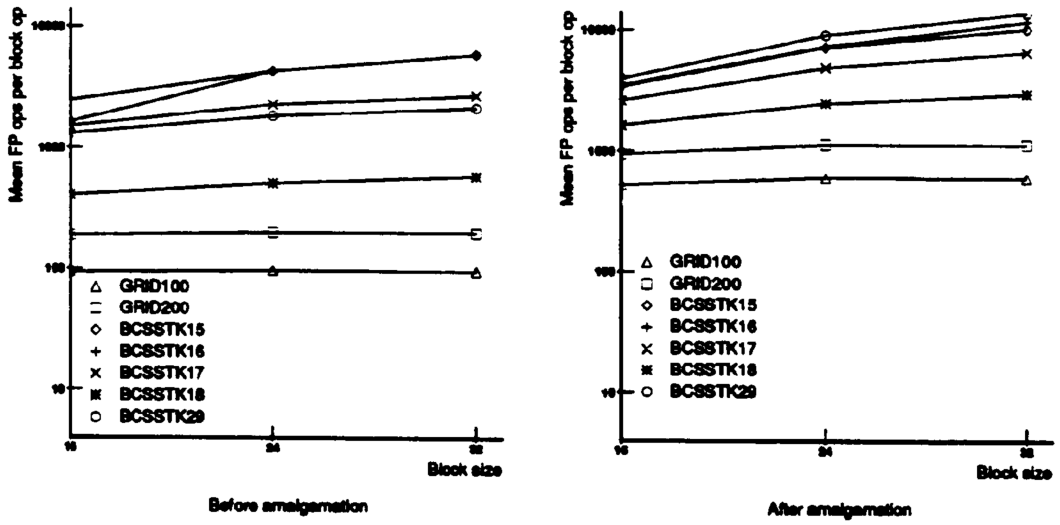


Figure 5: Average floating-point operations per block operation, before and after supernode amalgamation.

Table 5: Supernode amalgamation results.

	Name	Supernodes		Performance improvement
		before amalgamation	after amalgamation	for true seq. method
1.	GRID100	6,672	2,786	5%
2.	GRID200	26,669	11,243	6%
3.	BCSSTK15	1,295	525	1%
4.	BCSSTK16	691	434	3%
5.	BCSSTK17	2,595	1,622	2%
6.	BCSSTK18	7,438	3,727	7%
7.	BCSSTK29	3,231	1,193	14%

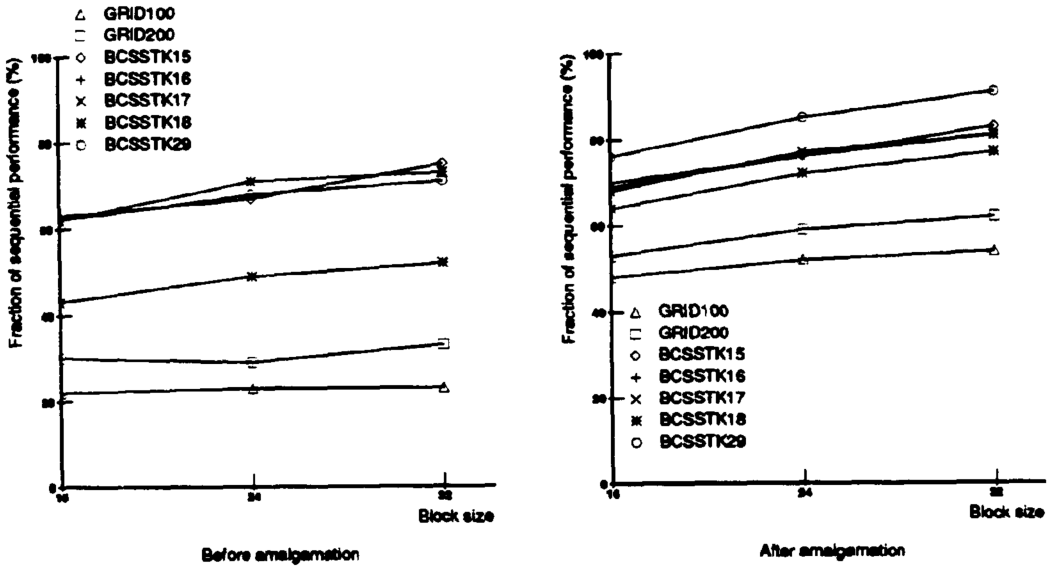


Figure 6: Performance of a sequential block approach, before and after supernode amalgamation, relative to a sequential left-looking supernode-supernode approach.

Figure 6 shows relative performance levels after amalgamation. The results indicate that amalgamation is quite effective at reducing overheads. Performance roughly doubles for GRID100, where the average task grain size increases for $B = 32$ increases from 96 floating-point operations to 597. Performance increases for the other matrices as well. With only two exceptions, block method performance is roughly 85% of that of a true sequential method for $B = 32$. Performance falls off somewhat when $B = 24$, and it decreases further when $B = 16$, but the resulting efficiencies are still more than 70%.

Note that our chosen range of blocks sizes, 16 to 32, is meant to span the range of reasonable choices. Blocks that are smaller than 16 by 16 would be expected to lead to large overheads. Indeed, performance was observed to fall off quite quickly for block sizes of less than 16. The marginal benefit of increasing the block size beyond 32 by 32 would be expected to be small. This expectation was also confirmed by the empirical results.

4.5 Block Decomposition Summary

This section has described a simple means of decomposing a sparse matrix into a set of rectangular blocks. The performance of a method based on such blocks on a sequential machine is nearly equal the performance of a true sequential method. Of course, our goal here is not an efficient sequential method, but instead an efficient parallel method. The next section will consider several issues related to the parallelization of the above approach.

5 Parallel Block Method

The question of how to parallelize the sequential block approach described so far can be divided into two different questions. First, how will processors cooperate to perform the work assigned to them? And second, what method will be used to assign this work to processors? This section will address these two questions in turn.

```
1.  while some L_IJ with map[L_IJ] = MyID is not complete do
2.      receive some L_IK
3.      if I = K /* diagonal block */
4.          Diag_{K,MyID} ← L_IK
5.          foreach L_JK ∈ Wait_{K,MyID} do
6.              L_JK ← L_JK L_IK^{-1}
7.              send L_JK to all P that could own blocks in
                        row J or column J
8.      else
9.          Rec_{K,MyID} ← Rec_{K,MyID} ∪ L_IK
10.         foreach L_JK ∈ Rec_{K,MyID} do
11.             if map[L_IJ] = MyID then
12.                 Find L_IJ
13.                 L_IJ ← L_IJ − L_IK L_JK^T
14.                 nmod[L_IJ] ← nmod[L_IJ] − 1
15.                 if (nmod[L_IJ] = 0) then
16.                     if I = J then /* diagonal block */
17.                         L_IJ ← Factor(L_IJ)
18.                         send L_IJ to all P that could own blocks in
                                    column J
19.                     else if (Diag_{J,MyID} ≠ ∅) then
20.                         L_IJ ← L_IJ L_KK^{-1}
21.                         send L_IJ to all P that could own blocks in
                                    row I or column I
22.                     else
23.                         Wait_{J,MyID} ← Wait_{J,MyID} ∪ L_IJ
```

Figure 7: Parallel block fan-out algorithm.

5.1 Parallel Factorization Organization

We begin our description of the parallel computation by assuming that each block will have some specific owner processor. In our approach, the owner of a block L_{IK} performs all block update operations with L_{IK} as their destination. With this choice in mind, we present the parallel block fan-out algorithm in Figure 7. The rest of this discussion will be devoted to an explanation of the algorithm

The most important notion for the block fan-out method is that once a block L_{IK} is *complete*, meaning that it has received all block updates and has been multiplied by the inverse of the diagonal block, then L_{IK} is sent to all processors that could own blocks updated by it. Blocks that could be updated by L_{IK} fall in block-row I or block-column I of L. When a block L_{IK} is received by a processor p (step 2 in Figure 7), processor p performs all related updates to blocks it owns. The block L_{IK} only produces blocks updates when it is paired with blocks in the same column K. Thus, processor p considers all pairings of the received block L_{IK} with completed blocks it has already received in column K (these blocks are held in set $Rec_{K,p}$) to determine whether the corresponding destination block is owned by p (steps 10 and 11). If the destination L_{IJ} is owned by p ($map[L_{IJ}] = p$), then the corresponding update operation is performed (steps 12 and 13). Each processor maintains a hash table of all blocks assigned to it, and the destination block is located through this hash table.

A count is kept with each block ($nmod[L_{IK}]$), indicating the number of block updates that still must be done to that block. When the count reaches zero, then block L_{IK} is ready to be multiplied by the inverse of L_{KK} (step 20 if L_{KK} has already arrived at p; step 6 otherwise). A diagonal block L_{KK} is kept in $Diag_{K,p}$, and any blocks waiting to be modified by the diagonal block are kept in $Wait_{K,p}$. The sets $Diag$, $Wait$, and Rec can be kept as simple linked lists of blocks.

One issue that is not addressed in the above pseudo-code is that of block disposal. As described above,

the parallel algorithm would retain a received block for the duration of the factorization. To determine when a block can be thrown out, we keep a count $ToRec_{K,p}$ of the number of blocks in a column K that will be received be a processor p. Once $|Rec_{K,p}| = ToRec_{K,p}$, then all blocks in column K are discarded.

We note that a small simplification has been made in steps 11 through 14 above. For all blocks L_{IJ}, I must be greater than J, a condition that is not necessarily true in the pseudo-code. The reader should assume that I is actually the larger of I and J, and similarly that J is the smaller of the two.

5.2 Block Mapping for Reduced Communication

We now consider the issue of mapping blocks to processors. Our general approach is to restrict the set of processors that can own blocks modified by a particular block L_{IK} and thus decrease the number of processors the block must be sent to. The actual restriction is done by performing a *scatter decomposition* [9] of the blocks in the sparse matrix.

More precisely, assume that P processors are used for the factorization, and assume for the sake of simplicity that P is a perfect square ($P = p \times p$). Furthermore, assume that the processors are arranged in a 2-D grid configuration, with the bottom left processor labeled $P_{0,0}$, and the upper right processor labeled $P_{p-1,p-1}$. To limit communication, a row of blocks is mapped to a row of processors. Similarly, a column of blocks is mapped to a column of processors. We choose round-robin distributions for both the rows and columns, where

$$map[L_{IJ}] = P_{I \bmod p, J \bmod p}.$$

Other distributions could be used. By performing the block mapping in this way, a block L_{IK} in the sparse factorization need only be sent to the row of processors that could own blocks in row I and the column of processors that could own blocks in column I. Every block in the matrix would thus be sent to a total of $2p = 2\sqrt{P}$ processors. Note that communication volume is independent of the block size with this mapping; every block in the matrix is simply sent to $2\sqrt{P}$ processors.

The scatter decomposition is appealing not only because it reduces communication volume, but also because it produces an extremely simple and regular communication pattern. All communication is done through multicasts along rows and columns of processors. This pattern is simple enough that one might reasonably expect parallel machines with 2-D grid interconnection networks to provide hardware multicast support for it eventually. In the absence of hardware support, an efficient software multicast scheme can be used. We will return to this issue later in this paper.

5.3 Enhancement: Domains

Before presenting performance results for the block fan-out approach, we first note that the method as described above produces more interprocessor communication than competing column-based approaches for small parallel machines. This is despite the fact that it has much better asymptotic communication behavior. To understand the reason, consider a simple 2-D $k \times k$ grid problem. The corresponding factor matrix contains $O(k^2 \log k)$ non-zeroes, and the parallel factorization of this matrix using a column approach can be shown to generate $O(k^2 P)$ communication volume [14]. In the block approach, every non-zero in the matrix is sent $O(\sqrt{P})$ processors, so the total communication volume grows as $O(k^2 \log k \sqrt{P})$. While the communication in the block approach grows less quickly in P, for any given 'k' it also has a larger 'constant' in front.

An important technique for reducing communication in column methods involves the use of *domains* [2, 4]. Domains are large sets of columns in the sparse matrix that are assigned en masse to a single processor. They are perhaps most easily understood in terms of the elimination tree of L. Recall that disjoint subtrees in the elimination tree are computationally independent, and consequently can be processed concurrently. By assigning the columns of an entire subtree (a domain) to a single processor, the communication that would have resulted had these columns been distributed among processors is avoided.

More precisely, by localizing all columns in a domain to a single processor, all updates to these columns can be performed without the need for interprocessor communication. More importantly, the updates from all columns within a domain to all other entries in the matrix can be computed and aggregated within the owner processor, again with no communication. That processor can then send the aggregate updates to the appropriate

destinations. In a column approach, the aggregate update is sent out on a column-wise basis. We refer the reader to [4] for more details.

Ashcraft suggested [2] that domains can be incorporated into a block approach as well. The basic approach is as follows. The non-zeroes within a domain are stored as they would be in a column-oriented method. The domain factorization is then performed using a column method. The aggregate domain update is computed column-wise as well. We use an extremely efficient left-looking supernode-supernode method for both. Once the aggregate update has been computed, it is sent out in a block-wise fashion to the appropriate destination blocks.

Of course, the domains must be carefully assigned to processors so that processors do not sit idle, waiting for other processors to complete local domain computations. Geist and Ng [10] described an algorithm for assigning a small set of domains to each processor so that the amount of domain work assigned to each processor is evenly balanced. All results from this point on use the algorithm of Geist and Ng to produce domains.

With the introduction of domains, the parallel computation thus becomes a three phase process. In the first phase, the processors factor the domains assigned to them and compute the updates from these domains to blocks outside the domains. In the second phase, the updates are sent to the processors that own the corresponding destination blocks and are added into their destinations. Finally, the third phase performs the block factorization, where blocks are exchanged between processors. Note that these are only logical phases; no global synchronizations is necessary between the phases.

Consider the effect of domains on communication volume in a block method for a 2-D grid problem. We first note that the number of non-zeroes not belonging to domains in the sparse matrix can be shown to grow as $O(k^2 \log P)$, versus $O(k^2 \log k)$ without domains. Total communication volume for these non-zeroes using a block approach is thus $O(k^2 \log P \sqrt{P})$. The other component of communication volume when using domains is the cost of sending domain updates to their destinations. The total size of all such updates can be shown to be $O(k^2)$, independent of P, so domain update communication represents a lower-order term. Total communication for a 2-D grid problem is thus $O(k^2 \log P \sqrt{P})$.

Note that domains produce the added benefit of reducing the number of small blocks in the matrix, and thus reducing related overheads. Recall that small supernodes are the main source of small blocks. In a sparse problem, most small supernodes lie towards the leafs of the elimination tree, where they are likely to be contained within domains.

6 Evaluation

This section evaluates the parallel block fan-out approach proposed in the previous section. The approach is evaluated in three different contexts. First, we look at performance on a small-scale multiprocessor. Then, we consider performance on moderately-parallel machines (up to 64 processors), using our multiprocessor simulation model. Finally, we consider issues for more massively parallel machines.

6.1 Small Parallel Machines

The first performance numbers we present come from the Silicon Graphics SGI 4D/380 multiprocessor. Parallel speedups are shown in Figure 8 for 1 through 8 processors. All speedups are computed relative to a left-looking supernode-supernode sequential code. The figure shows that the block fan-out method is indeed quite efficient for small machines. In fact, performance is comparable to that of a highly efficient column-based code that distributes supernodes among processors [18]. Speedups on 8 processors are roughly 5.5-fold, corresponding to absolute performance levels of between 45 and 50 double-precision MFLOPS. Speedups are less than linear in the number of processors for two simple reasons. First, the block method is slightly less efficient than a column method. We believe this accounts for a roughly 15% performance reduction. Second, the load is unevenly distributed among the processors. A simple calculation reveals that processors spend roughly 15% of the computation on average sitting idle. These two factors combine to give a relatively accurate performance prediction.

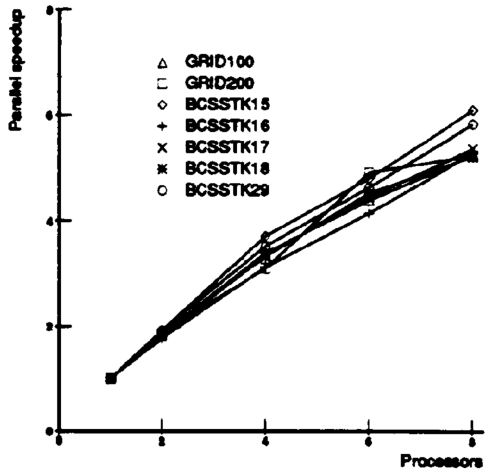


Figure 8: Parallel speedups for block fan-out method on SGI 4D-280, $B = 24$.

6.2 Moderately Parallel Machines

We now perform an evaluation of parallel performance of the block fan-out approach on machines with up to 64 processors, using the multiprocessor simulation model described earlier. We also discuss issues of communication volume.

6.2.1 Simulated Performance

To get a feel for how the block fan-out method would scale to a larger number of processors, Figure 9 shows simulated processor utilization levels for between 4 and 64 simulated processors, using a block size of 24. It is clear from the figure that the block approach exhibits less than ideal behavior as the machine size is increased. On 64 processors, for example, utilization levels drop to roughly 40%. Further investigation reveals that the primary cause of the drop in performance is a progressive decline in the quality of the load balance. Figure 10 compares simulated performance for matrices BCSSTK15 and BCSSTK29 with the best performance that could be obtained with the same block distribution. The load balance performance bound is obtained by computing the time that would be required if there were no dependencies between blocks and if interprocessor communication were free.

The quality of the load distribution clearly depends on the method used to map blocks to processors. Recall that we use a very rigid mapping strategy, where block L_{IJ} is assigned to processor $P_{I \bmod p, J \bmod p}$. One possible explanation for the poor behavior of this strategy is that it does not adapt to the structure of the sparse matrix; it tries to impose a very regular structure on a matrix that is potentially comprised of a very irregular arrangement of non-zero blocks.

While the mismatch between the regular mapping and the irregular matrix structure certainly contributes to the poor load balance, it is our belief that a more important factor is the wide variability in task sizes. In particular, since a block is modified by some set of blocks to its left, blocks to the far right in the matrix generally require much more work than blocks to the left (more accurately, blocks near the top of the elimination tree require more work than blocks near the leafs). Furthermore, since the matrix is lower-triangular, the number of blocks in a column decreases towards the right. The result is a small number of very important blocks in the bottom-right corner of the matrix.

To support our contention that the sparse structure of the matrix is less important than the more general task distribution problem, Figure 11 compares the quality of the load balance obtained for two sparse matrices, BCSSTK15 and BCSSTK29, to the load balance obtained using the same mapping strategy for a dense matrix. The curves show the maximum obtainable processor utilization levels given the block mapping. The dense

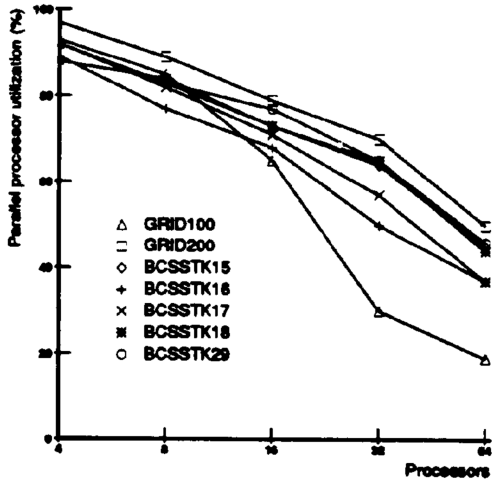


Figure 9: Simulated parallel efficiencies for block fan-out method, $B = 24$.

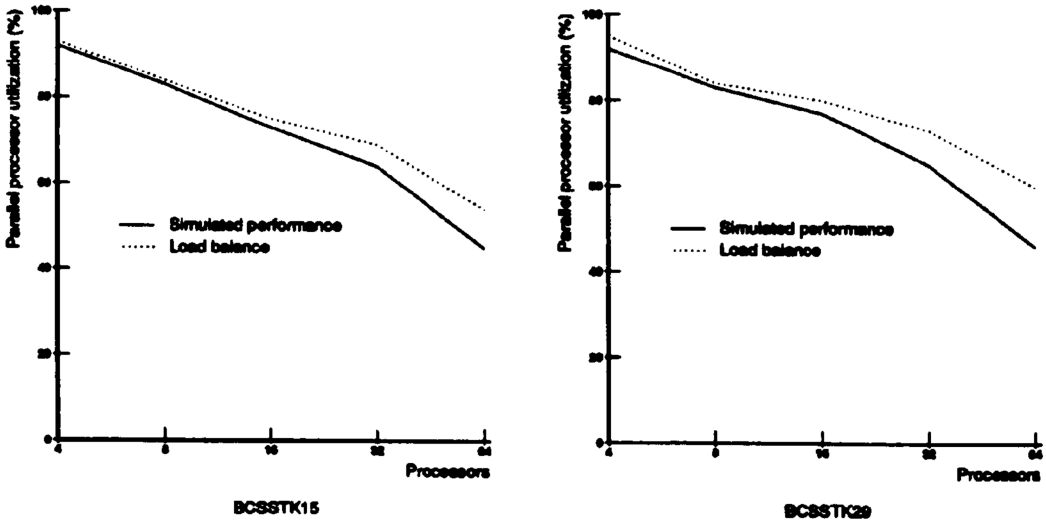


Figure 10: Simulated parallel performance, compared with load balance upper bound ($B = 24$).

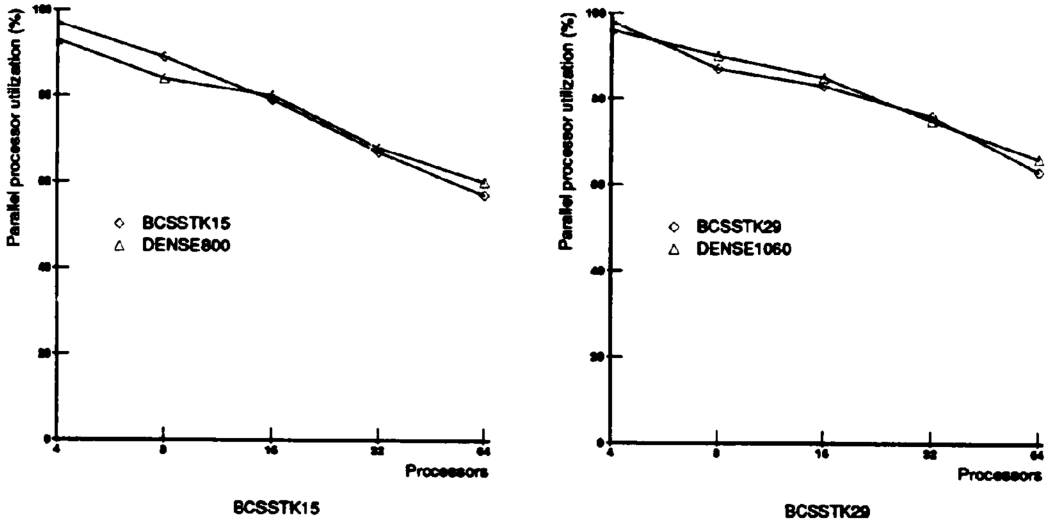


Figure 11: Parallel utilization upper bounds due to load balance for BCSSTK15 and BCSSTK29, compared with load balance upper bounds for dense problems ($B = 24$). In both plots, sparse and dense problems perform the same number of floating-point operations.

problems are chosen so as to perform roughly the same number of floating-point operations as the two sparse problems.

Note that the load balance can be improved by moving to a smaller block size, thus creating more distributable blocks and making the block distribution problem easier. However, as discussed earlier, smaller blocks also increase block overheads. For the larger benchmark sparse matrices, decreasing the block size from $B = 24$ to $B = 16$ increases simulated parallel efficiencies for $P = 64$ from 40%-45% for $B = 24$ to 50%-55% for $B = 16$. A block size of less than 16 further improves the load balance, but achieves lower performance due to overhead issues.

The general conclusion to be drawn from these simulation results is simply that large machines require relatively large problems to achieve high processor utilization levels. In particular, the sparse matrices that we study here are too small to make good use of a 64 processor machine. Of course, it may be possible to significantly improve parallel load balance with a better mapping strategy. A more general function could be used to map columns of blocks to columns of processors, and to map rows of blocks to rows of processors. This matter will require further investigation.

6.2.2 Communication Volume

So far, our analysis has assumed that parallel performance is governed by two costs: the costs of executing block operations on individual processors and the latencies of communicating blocks between processors. Another important, although less easily modeled component of parallel performance is the total interprocessor communication volume. Communication volume will determine the amount of contention that is seen on the interconnection network. Such contention can have severe performance consequences, and can in many cases govern the performance of the entire computation (see [20], for example).

Rather than try to integrate these costs into our simple performance model, we instead look at interprocessor communication in a more qualitative way. To obtain a general idea of how much communication is performed, Figure 12 compares total interprocessor communication volume with total floating-point operation counts for a variety of sparse matrices and machine sizes. This figure shows the average number of floating-point operations performed by a processor per floating-point value sent by that processor. Sustainable values will depend on the relative computation and communication bandwidths of the processor and the processor interconnect in the parallel machine. Current machines would most likely not have trouble supporting the roughly 40 to 1 ratio seen for 16 processors on these matrices. The roughly 20 to 1 ratio on 64 processors, however, could prove

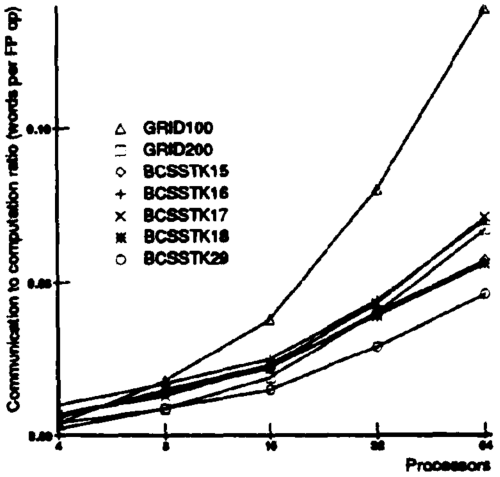


Figure 12: Communication versus computation for the block fan-out method.

troublesome.

To put these communication figures into better perspective, we now compare them to the communication volumes that would be seen with a column-oriented factorization method. Figure 13 shows relative communication volume, compared with a parallel column multifrontal method. Interestingly, the advantages of the block approach on 64 or fewer processors are quite modest. While the growth rates, $O(P)$ for columns and $O(\log P\sqrt{P})$ for blocks, favor the block approach, constants make these rates less relevant for small P.

An interesting thing to note here is that relative communication is quite a bit higher for the two grid problems than for the other matrices. The reason is that the column multifrontal approach does very well communication-wise for sparse matrices whose elimination trees have few nodes towards the root and instead quickly branch out into several independent subtrees. The two grid problems have this property. The block approach derives no benefit from this property.

6.2.3 Summary

To summarize this subsection, we note that our block fan-out approach provides good performance for moderately-parallel machines, although parallel speedups are well below linear in the number of processors for the matrices we have considered. An important limiting factor is the load balance that results from our very rigid block distribution scheme. Regarding communication volumes, we find that the block approach produces comparable amounts of traffic to a column approach on 64 or fewer processors. As to how the performance of a block approach would compare with that of a column approach for such machines, we believe performance would be higher on machines where matrix-matrix, or BLAS3 operations execute significantly more quickly than vector-vector, or BLAS1 operations. On machines that achieve high performance on vector-vector operations, the load balance problems of the block approach would most likely tilt the comparison somewhat in favor of a column approach.

6.3 Massively-Parallel Machines

Having concentrated on issues of efficiency on smaller machines in the first part of this section, we now turn our attention to three issues that will be important for very large parallel machines. First, we look at available concurrency in the problem. In other words, we look at how many processors can be productively used for a particular problem. Next we turn to the issue of per-processor storage requirements, and we consider how they grow as the number of processors and the problem size is increased. A common assumption for large parallel

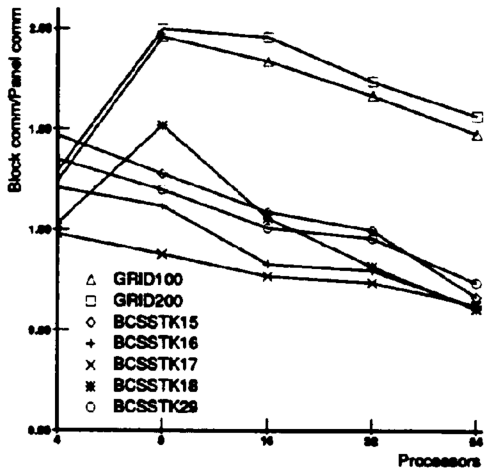


Figure 13: Communication volume of block approach, relative to a column-oriented parallel multifrontal approach.

machines is that each processor will contain some constant amount of memory. Thus, it would be desirable for the amount of storage required per processor to remain constant. Finally, we consider interprocessor communication issues. Our discussions will use 2-D grid problems as examples.

Before further discussing these issues, we should first explain our goals. The primary advantage of a block approach over a column approach for a massively parallel machine is that it allows more processors to cooperate for the same sparse problem. For a $k \times k$ 2-D grid problem, for example, the column approach can be shown to allow $O(k)$ processors to participate. By some measures, a block approach can use $O(k^2)$. Our goal is to determine whether the use of $O(k^2)$ processors is a realistic goal, and to understand the difficulties that might be encountered in trying to reach this goal.

6.3.1 Concurrency

One important bound on the parallel performance of a computation is the length of the critical path. Determining the critical path in a computation requires an analysis of the dependencies between the various tasks in that computation. Such an analysis for block-oriented sparse Cholesky factorization reveals that the length of the critical path is proportional to the height of the elimination tree, assuming some constant block size. For a 2-D grid problem, the elimination tree can be shown to have height $3k$. Thus, in the best case the $O(k^3)$ work of the entire factorization can be performed in $O(k)$ time. Consequently, at most $O(k^2)$ processors can be productively applied to this problem. This figure is consistent with our goals for the block approach.

6.3.2 Storage

We now look at the issue of how per-processor storage requirements grow as the size of the machine and the size of the problem is increased. We first note the obvious fact that the processor must store the portion of the matrix assigned to it. If the factorization is performed on P processors, and the problem being factored is a $k \times k$ grid problem, then each processor must store $O(\frac{k^2 \log k}{P})$ non-zeroes. Keeping per-processor storage requirements constant would thus require that the number of processors grow slightly faster than k^2. Since the critical path analysis showed that only $O(k^2)$ processors can be used productively for this problem, we must resign ourselves to a slow growth rate in per-processor storage.

Now consider the storage requirements of the auxiliary data structures that a processor must maintain. One important set of auxiliary data is the per-block information. An example is the count of how many times a

block is modified. Another is the particular row and column of processors to which a particular block is sent when complete. This data adds a small constant to the size of each block, and consequently it represents a small constant factor increase in overall storage.

Another important set of auxiliary data is the column-wise data. One example is the arrival count information, which keeps track of how many blocks will arrive in a particular column. Since the number of columns in the matrix is k^2, this data structure would occupy $O(k^2)$ space per processor if every entry were kept. Fortunately, only $O(k^3/P)$ of these entries must be stored. The reason is as follows. If the factorization work is distributed evenly among the processors, then the work performed per processor is $O(k^3/P)$. Since a received block is only retained in a processor if it participates in some useful work, clearly the number of such retained blocks and thus the number of arrival counts that must be stored is also $O(k^3/P)$. We can keep a hash table, indexed by column number, of all non-zero arrival counts. When a block arrives, the corresponding arrival count is located and decremented. Note that not all blocks that arrive at a processor participate in an update on that processor. If no arrival count is found for the block column of an arriving block, then the block is immediately discarded. Similar hash structures can be used for the other column-wise data structures.

Regarding per-processor storage growth rates, note that if P grows as k^2, then the per-processor matrix storage costs grow as $O(\log k)$ while the arrival count storage costs grows as $O(k^3/P) = O(k)$. Fortunately, the $O(k)$ term has a very small constant in front of it, so this term will not be particularly constraining for practical P. However, asymptotic per-processor storage requirements will grow with P.

6.3.3 Communication

A crucial determinant of performance on massively parallel machines is the bandwidth of the processor interconnection network. In order to obtain a rough feel for whether the bandwidth demands of the block fan-out method are sustainable as the machine size increases, we look at these demands in relation to two common upper bounds on available communication bandwidth, in a manner similar to that used by Schreiber in [20]. The two upper bounds are based on bisection bandwidth and total available point-to-point bandwidth in the multiprocessor. We consider a 2-D mesh machine organization, which is in some sense a worst case since it offers lower connectivity than most alternative organizations.

A bisection bandwidth bound is obtained by breaking some set of point-to-point interconnection links in the parallel machine to divide it into two halves. Clearly, all communication between processors in different halves must be travel on one of the links that is split. The bisection bandwidth bound simply states that the parallel runtime is at least as large as the time that would be required for these bisection links to transmit all messages that cross the bisector.

In the case of the block fan-out method applied to a 2-D grid problem, recall that $O(k^2 \log P)$ messages are sent, and each is multicast to $O(\sqrt{P})$ processors (a row and column of processors). Figure 14 shows an example mesh of processors, an example bisector, and the communication pattern that can be used to multicast a message. For any simple bisector, a multicast to a row and column of processors crosses that bisector twice. Thus, total traffic across the bisector is $O(k^2 \log P)$. This traffic must travel on one of $O(\sqrt{P})$ communication links in the bisector, and this communication occurs in the $O(k^3/P)$ time required for the factorization. If we assume that communication is evenly distributed among the bisector links, then communication per bisector link per unit time is $O(\frac{k^2 \log P}{\sqrt{P}(k^3/P)}) = O(\frac{\log P \sqrt{P}}{k})$. If P grows as k^2, communication per link per unit time is thus $O(\log P)$. Since the amount of data that can travel on a single link per unit time is constant, this growth rate represents a small problem. The number of processors P must grow slightly slower than k^2 in order to keep message volume per link constant.

Another common communication-based bound on parallel performance is the total amount of traffic that appears on any link in the machine, expressed as a fraction of the total number of links in the machine. For our example, there are $O(k^2 \log P)$ multicasts, each of which traverses $O(\sqrt{P})$ links. The number of links in the machine is $O(P)$, and again this communication occurs in $O(k^3/P)$ time. Thus, global traffic per link per time unit is $O(\frac{k^2 \log P \sqrt{P}}{(k^3/P)(P)})$, or $O(\frac{\log P \sqrt{P}}{k})$. If P is $O(k^2)$, we obtain $O(\log P)$ traffic per link, which is identical to the bisector traffic.

We should note that the preceding arguments have said nothing about *achieved* performance. Demonstrating that certain performance levels can actually be achieved would require a detailed analysis of the structure of

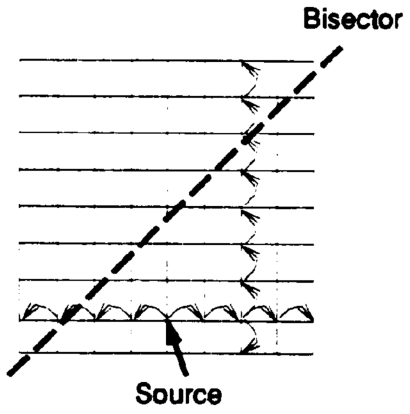


Figure 14: Communication pattern for row/column multicast.

the sparse matrix, the way in which the factorization tasks are mapped to processors, and the order in which these tasks are handled by their owners. This would certainly be a daunting task. This discussion has simply shown that the approach is not constrained away from achieving high performance by any of the most common performance bounds.

6.4 Summary

To summarize our evaluation, we have found that the block fan-out method is quite appealing across a range of parallel machines. Overheads are low enough that the method is quite effective for small parallel machines. It is also effective for moderately parallel machines, although performance is somewhat limited by the quality of the computational load balance. For massively parallel machines, we found that the approach is not perfect. Per-processor storage requirements grow with the number of processors. Bisection bandwidth considerations also limit the number of processors to below ideal. However, these constraints are mild enough that the block fan-out approach appears to be quite practical even for very large P.

7 Future Work

While this paper has explored several practical issues related to parallel block-oriented factorization, it also has brought up a number of questions that will require further investigation. Foremost among these is the question of whether the load balance could be significantly improved. We are currently investigating more flexible block mapping strategies.

Another interesting question concerns the choice of partitions for the 2-D decomposition. Recall that our partitions are chosen to contain sets of contiguous columns from within the same supernode. Ashcraft has shown [2] that by choosing columns that are not necessarily contiguous, it is often possible to divide the sparse matrix into fewer, denser blocks. While our results indicate that the simpler approach is quite adequate, we are currently looking into the question of how large the benefit of a more sophisticated approach may be.

We also hope to compare the block fan-out approach we have proposed here with the block multifrontal approach proposed by Ashcraft [2]. One thing we are certain of is that the block fan-out method is much less complex. So far, we have not discovered any significant advantages to a multifrontal approach, but the issue requires further study. We also hope to investigate a block analogue of the fan-in method.

Once a matrix A has been factored into the form $A = LL^T$, the next step is typically the solution of a

number of triangular systems $Ly = b$, where b is given. An issue that we have left unaddressed in this paper is the efficiency of this backsolve computation when L is represented as a set of blocks. Our belief is that this backsolve will be more efficient than the backsolve for a column representation, but further investigation will be required to fully answer this question.

8 Conclusions

It is becoming increasingly clear that column approaches are inappropriate for sparse Cholesky factorization on large parallel machines. One thing that has been much less clear is whether the alternative, a 2-D matrix decomposition, is truly practical. This paper has described a parallel block algorithm that is both practical and appealing. The primary virtues of our approach are: (1) it uses an extremely simple decomposition strategy, in which the matrix is divided using global horizontal and vertical partitions; (2) it is straightforward to implement; (3) it is extremely efficient, performing the vast majority of its work within dense matrix-matrix multiplication operations; (4) it is efficient across a wide range of machine sizes, providing performance that is comparable to that of efficient column methods even on small parallel machines.

Acknowledgments

We would like to thank Rob Schreiber and Sid Chatterjee for their discussions on block-oriented factorization. This research is supported under DARPA contract N00039-91-C-0138. Anoop Gupta is also supported by an NSF Presidential Young Investigator Award.

References

[1] Amestoy, P.R., and Duff, I.S., "Vectorization of a multiprocessor multifrontal code", *International Journal of Supercomputer Applications*, 3:41-59, 1989.

[2] Ashcraft, C.C., *The domain/segment partition for the factorization of sparse symmetric positive definite matrices*, Boeing Computer Services Technical Report ECA-TR-148, November, 1990.

[3] Ashcraft, C.C., *The fan-both family of column-based distributed Cholesky factorization algorithms*, in Workshop on Sparse Matrix Computations: Graph Theory Issues and Algorithms, 1992.

[4] Ashcraft, C.C., Eisenstat, S.C., Liu, J.L., and Sherman, A.H. "A comparison of three column-based distributed sparse factorization schemes. Research Report YALEU/DCS/RR-810, Computer Science Department, Yale University, 1990.

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

[6] Ashcraft, C.C., Grimes, R.G., Lewis, J.G., Peyton, B.W., and Simon, H.D., "Recent progress in sparse matrix methods for large linear systems", *International Journal of Supercomputer Applications*, 1(4): 10-30, 1987.

[7] Duff, I.S., Grimes, R.G., and Lewis, J.G., "Sparse Matrix Test Problems", *ACM Transactions on Mathematical Software*, 15(1): 1-14, 1989.

[8] Duff, I.S., and Reid, J.K., "The multifrontal solution of indefinite sparse symmetric linear equations", *ACM Transactions on Mathematical Software*, 9(3): 302-325, 1983.

[9] Fox, G., et al, Solving Problems on Concurrent Processors: Volume 1 - General Techniques and Regular Problems, Prentice Hall, 1988.

[10] Geist, G.A., and Ng, E., "A partitioning strategy for parallel sparse Cholesky factorization", Technical Report TM-10937, Oak Ridge National Laboratory, 1988.

[11] George, A., Heath, M., Liu, J. and Ng, E., "Sparse Cholesky factorization on a local-memory multiprocessor", *SIAM Journal on Scientific and Statistical Computing'*, 9:327-340, 1988.

[12] George, A., Heath, M., Liu, J., and Ng, E., *Solution of sparse positive definite systems on a hypercube*, Technical Report TM-10865, Oak Ridge National Laboratory, 1988.

[13] George, A., and Liu, J., *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, 1981.

[14] George, A., Liu, J. and Ng, E., "Communication results for parallel sparse Cholesky factorization on a hypercube", *Parallel Computing*, 10: 287-298, 1989.

[15] Liu, J., "Modification of the minimum degree algorithm by multiple elimination", *ACM Transactions on Mathematical Software*, 12(2): 127-148, 1986.

[16] Lucas, R. Solving Planar Systems of Equations on Distributed-Memory Multiprocessors, PhD thesis, Stanford University, 1988.

[17] Rothberg, E., and Gupta, A., "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, 1991.

[18] Rothberg, E., and Gupta, A., "Techniques for improving the performance of sparse matrix factorization on multiprocessor workstations", *Supercomputing '90*, p. 232-243 , November, 1990.

[19] Schreiber, R., "A new implementation of sparse Gaussian elimination", *ACM Transactions on Mathematical Software*, 8:256-276, 1982.

[20] Schreiber, R., "Are sparse matrices poisonous to highly parallel machines?", in Workshop on Sparse Matrix Computations: Graph Theory Issues and Algorithms, 1992.

[21] Van De Geijn, R., *Massively parallel LINPACK benchmark on the Intel Touchstone Delta and iPSC/860 systems*, Technical Report CS-91-28, University of Texas at Austin, August, 1991.

[22] Venugopal, S., and Naik, V.K., "Effects of partitioning and scheduling sparse matrix factorization on communication and load balance", *Supercomputing '91*, November, 1991.