# An Efficient Block-Oriented Approach To Parallel Sparse Cholesky Factorization

Edward Rothberg

Intel Supercomputer Systems Division
14924 N.W. Greenbrier Parkway
Beaverton, OR 97006

Anoop Gupta

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

## 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 more traditional column-oriented and panel-oriented decompositions. However, little progress has been made in producing a practical sub-block method. This paper proposes and evaluates an approach that is simple to implement, provides slightly higher performance than column (and panel) methods on small parallel machines, and has the potential to provide much higher performance on large parallel machines.*

## 1 Introduction

The Cholesky factorization of sparse symmetric positive definite matrices 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.

Virtually all parallel approaches to sparse Cholesky factorization [3, 9, 16] perform a 1-dimensional decomposition of the sparse matrix. That is, they distribute either rows or columns of the matrix among processors. Such a decomposition has two major limitations. The first is that it produces enormous volumes of interprocessor communication. Communication grows linearly in the number of processors [11], resulting in communication volumes that are difficult to sustain on all but the smallest parallel machines. The second limitation is that a 1-D decomposition produces extremely long critical paths. Since the critical path represents a lower bound on parallel runtime, parallel speedups are severely limited.

Both of these limitations can be overcome (in theory) by moving to a sub-block, or 2-D decomposition. Such a decomposition has been shown to be extremely effective for parallel dense factorization [22]. It is not clear, however, whether a similar decomposition would be practical for sparse problems. A few investigations [1, 21, 23] have been performed, but these contained little or no exploration of practical algorithms. This paper provides a detailed analysis of a new block-oriented algorithm, including performance results from an efficient implementation.

This paper focuses on two practical and important issues related to a 2-D decomposition approach. The first is implementation complexity. The fact that most sparse factorization methods use a 1-D decomposition indicates that this decomposition is more natural. A block approach might significantly complicate the implementation. The second issue is the efficiency of a parallel block-oriented method for practical machine sizes. 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.

Regarding complexity, 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 [9].

Regarding efficiency, we explore this issue 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 to be quite minor, with the block approach producing roughly 80% of the performance of an efficient sequential method. We then consider parallel block factorization, looking at the issues that potentially limit its performance. The parallel block method is found to give high performance on a range of parallel machine sizes. 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. Conclusions are presented in Section 7.

## 2 Sparse Cholesky Factorization

The goal of the sparse Cholesky computation is to factor a sparse symmetric positive definite $n \times n$ 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 reduce *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 [10] for more information on these steps.

The following pseudo-code performs the numerical factorization step:

```
1.  for k = 1 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
```

Only the non-zero entries in the sparse matrix are stored, and the computation performs operations only on non-zeroes. The factorization is most often 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, often called a column division or $cdiv(k)$ operation. Similarly, steps 5 and 6 form a column modification, or $cmod(j,k)$, operation.

This column-oriented formulation of the sparse factorization has formed the basis of several parallel sparse factorization algorithms, including the fan-out method [9], the fan-in method [3], 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 volumes grow linearly in the number of processors [3, 11]. 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 [2] 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 $O(n^3)$ operations while the length of the critical path and thus the best case parallel runtime is $O(n^2)$ operations. Similar bounds apply for sparse problems.

An alternative formulation of the factorization problem divides the matrix into rectangular sub-blocks. This formulation leads to greatly reduced communication volumes and exposes significantly more concurrency. Specifically, communication volumes grow as the square root of the number of processors, and the critical path grows as $O(n)$ [22]. It is an open question whether this formulation can be efficiently applied to parallel sparse factorization, and this is the question 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* [5]. A supernode is a set of adjacent columns in the factor matrix $L$ whose non-zero structure consists of a dense lower-triangular block on the diagonal, and an identical set of non-zeroes for each column below the diagonal. 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 [5, 17, 18]. 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.

One thing we should note is that it is possible to improve the performance of parallel sparse column-oriented methods by grouping sets of adjacent columns from within the same supernode into *panels*, and distributing these panels among the processors [17, 18]. We use the term *column-oriented* in this paper to refer to methods that treat columns as indivisible entities. Thus, panel methods fit this description. When we compare the performance of our parallel block-oriented method to that of a parallel column-oriented method, we will actually compare against the higher-performance parallel panel method.

Another important notion in sparse factorization is that of the *elimination tree* of the sparse matrix [15, 20]. 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 modified only by descendent columns in the elimination tree, and equivalently that a column modifies only ancestors [15]. The most important property captured in this tree for parallel factorization is the property that disjoint subtrees are independent, and consequently can be processed in parallel. This fact will be relevant later in this paper.

Table 1: Benchmark matrices.

| Name | Equations | NZ in A | NZ in L | FP ops to factor |
|---|---|---|---|---|
| GRID100 | 10,000 | 39,600 | 250,835 | 15.7M |
| GRID200 | 40,000 | 159,200 | 1,280,743 | 137M |
| BCSSTK15 | 3,948 | 113,868 | 647,274 | 165M |
| BCSSTK16 | 4,884 | 285,494 | 736,294 | 149M |
| BCSSTK17 | 10,974 | 417,676 | 994.885 | 144M |
| BCSSTK18 | 11,948 | 137,142 | 650,777 | 141M |
| BCSSTK29 | 13,992 | 605,496 | 1,680,804 | 393M |

# 3 Experimental Environment

Since our interest in this paper is to consider practical performance issues for block methods, we will present performance numbers for realistic sparse matrices factored on real machines. This section briefly describes both the matrices we use as benchmarks and the machines 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 [6]. Since our interest is in factorization on large machines, we have chosen some of the largest sparse matrices in the collection. We also include two regular 2-D, 5-pt grid problems. Table 1 gives brief descriptions of the matrices. For each matrix, the table shows the number of rows and columns in the matrix, the number of non-zeroes in the matrix, the number of non-zeroes in the factor, and the number of floating-point operations (in millions) required for the factorization. All matrices except the two grid problems are preordered using the multiple minimum degree ordering heuristic [14] before being factored. A simple nested dissection ordering is used for the grid problems.

## 3.2 Target Machines

This paper will present performance numbers from several parallel machines. We now briefly describe the parallel machines that are considered.

Performance numbers for sequential and small-scale 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. The machine has a hierarchical memory organization; memory references serviced from the processor cache are significantly less expensive than references that must be serviced from main memory.

We also provide performance numbers from the Stanford DASH machine, a 48-processor distributed-shared-memory machine [13]. The DASH machine is built out of a network of 12 4-processor SGI 4D/340 nodes. Each 4D/340 node contains some portion of the global shared memory. A processor can cache any location in the global memory. A processor memory reference that is serviced from its cache requires a single cycle. A reference to a location held in the memory local to a processor requires roughly 30 cycles. A reference to a location held in a non-local memory requires roughly 100 cycles. Our factorization implementation for the Stanford DASH machine explicitly places matrix data in the memory local to the processor that owns that part of the matrix.

In order to provide a more detailed understanding of the performance of parallel machines on this computation, 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 matrix block modification operation or the transmission of a large message from one processor to another. We model the costs of these high-level operations in terms of what we believe are the three most important determinants of performance on a parallel machine: the number of floating-point operations performed, the number of data items fetched from memory, and the amount of data moved between processor memories. The parallel simulation is performed as a discrete-event simulation of these tasks. We do not have space in this paper to describe the exact details of our simulation; details can be found in [17]. We simply note that the costs we use for floating-point operations, memory fetches, and interprocessor communication roughly match those of the DASH machine, and they are quite comparable to those of several other distributed-memory parallel machines.

# 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 the sequential factorization computation as possible.

## 4.1 Block Decomposition

When dividing a matrix into blocks, we believe the three most important issues are: (1) producing blocks with simple internal non-zero structures, so that block operations can be performed efficiently; (2) producing blocks that interact with other blocks in simple ways, so that bookkeeping overheads are minimized; and (3) producing blocks that are as dense as possible, so that per-block computation and storage overheads are minimized. With these goals in mind, the approach we take to decomposing the sparse matrix into blocks is to perform a global partitioning on the matrix, guided by the supernodal structure. More precisely, we divide the columns of the matrix $(1 \ldots n)$ into contiguous sets $(\{1 \ldots p_2 - 1\}, \{p_2 \ldots p_3 - 1\}, \ldots, \{p_N \ldots n\}$, where $N$ is the number of partitions and $p_i$ is the first column in partition $i$). All columns within a particular partition must be members of the same supernode (although a partition will frequently be a subset of a supernode). An identical partitioning is performed on the rows. A simple example is shown in Figure 1. 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 $\{p_J \ldots p_{J+1} - 1\}$.

This global partitioning approach addresses the above-mentioned issues quite well. Each block has a very simple

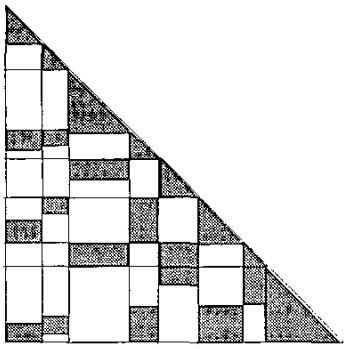

Figure 1: Example of globally partitioned matrix.

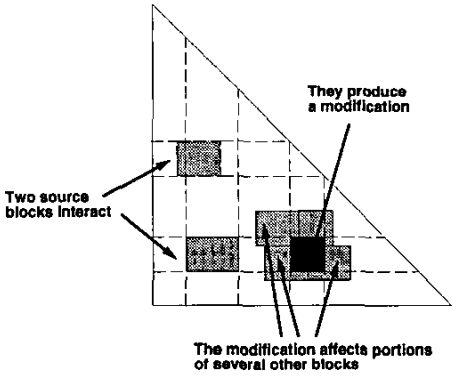


Figure 2: Example of irregular block interaction. Dashed lines indicate boundaries of affected areas.

non-zero structure. Since the block is a portion of a supernode, all rows in the block are dense. The blocks also share common boundaries. As a result, block interactions are extremely regular. As we will soon demonstrate, this decomposition leads to a computation structure where a block interacts with a block above it to produce a modification to a block to its right. Without these common boundaries, block modifications would be quite complicated, with portions of blocks modifying portions of other blocks (see Figure 2).

One issue that this distribution scheme does not address is the block density issue. The global nature of the partitions does not allow the blocks to be tailored to match the local sparsity structure of the matrix. We will see in the next section that this is not actually a significant problem. While blocks will often not be completely dense, this sparsity has little effect on the efficiency of the overall computation.

Before proceeding, we note that Ashcraft [1] proposed a similar decomposition strategy independently.

## 4.2 Structure of the Block Factorization Computation

One important goal we had in choosing this block decomposition was 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 = 1 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 above pseudo-code 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 modifications from all pairs of blocks in column $K$. We store the blocks by columns, so that all blocks in a column can be easily located. We also keep a hash table of all blocks (hashing on the row and column index), so that destination block $L_{IJ}$ in step 7 can be located quickly.

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 it 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 1). When a modification 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.

Block modification is most conveniently viewed as a two stage process. A set of modification values is computed in the first stage, and these values are subtracted from the appropriate entries in the destination block in the second, or scatter stage. The first stage 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 a modification.

During the second stage, the resulting modification must be subtracted from the destination. If the modification has the same non-zero structure as the destination block, then the subtraction is trivial. Otherwise, we must first determine the relationship between the non-zero structures of the modification

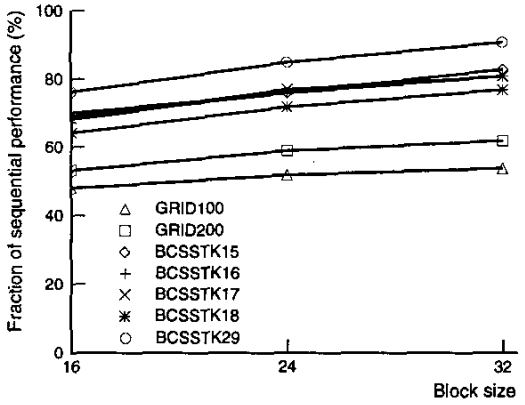


Figure 3: Performance of a sequential block approach, relative to a sequential left-looking supernode-supernode approach.

and of the destination. This information can then be used to scatter the modification into the destination. While this scatter is much more expensive than direct addition of the modification into the destination, it is also much less common.

### 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 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. For example, with a block size of $B = 16$, a supernode of width 51 would be split into three partitions of size 17. Since partitions represent subsets of supernodes, some partitions will naturally be much smaller than $B$.

We found that the block approach as described above actually produces quite low performance for several of the matrices. The reason was the presence of many small supernodes, which led to many small blocks and significant overheads. We were able to improve performance dramatically by performing *supernode amalgamation* [4] before executing the block factorization. Amalgamation is a process of selectively adding non-zeroes to the factor matrix in order to combine small supernodes with nearly-identical non-zero structures into larger supernodes.

The performance obtained with the sequential block approach on a single processor of the SGI 4D/380 after amalgamation is shown in Figure 3. This performance is expressed as a fraction of the performance obtained with an efficient sequential code (a supernode-supernode left-looking method; among the most efficient sequential approaches [17]). Performance numbers for the sequential method are given in Table 2. The results indicate that the block approach is quite efficient. With only two exceptions, block method performance for $B = 32$ is roughly 80% of that of a true sequential method. Performance falls off somewhat when $B = 24$, and it decreases further when

Table 2: Sequential performance on SGI 4D/380 (supernode-supernode left-looking method).

| Name | Performance (MFLOPS) |
|---|---|
| GRID100 | 5.6 |
| GRID200 | 6.7 |
| BCSSTK15 | 7.9 |
| BCSSTK16 | 7.8 |
| BCSSTK17 | 7.5 |
| BCSSTK18 | 7.2 |
| BCSSTK29 | 7.8 |

$B = 16$, but the resulting efficiencies are still roughly 70%.

## 5 Parallel Block Method

We now turn to the question of how to parallelize the sequential block computation. This question 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.

### 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 modification operations with $L_{IK}$ as their destination. With this choice in mind, we present the parallel block fan-out algorithm in Figure 4. The rest of this section 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 modifications and has been multiplied by the inverse of the diagonal block, then $L_{IK}$ is sent to all processors that could own blocks modified by it. Blocks that could be modified 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 4), processor $p$ performs all related modifications to blocks it owns. The block $L_{IK}$ produces block modifications only 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 modification 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 ($nmod[L_{IJ}]$), indicating the number of block modifications that still must be done to that block. When the count reaches zero, then block $L_{IJ}$ is ready to be multiplied by the inverse of $L_{JJ}$ (step 20 if $L_{JJ}$ has already arrived at $p$; step 6 otherwise). A diagonal block $L_{JJ}$ is kept in $Diag_{J,p}$, and any blocks waiting to be modified by

```
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_KK
5.          foreach L_JK ∈ Wait_{K,MyID} do
6.              L_JK := L_JK L_KK^{-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_JJ := Factor(L_JJ)
18.                         send L_JJ to all P that could own blocks in column J
19.                     else if (Diag_{J,MyID} ≠ ∅) then
20.                         L_IJ := L_IJ L_JJ^{-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 4: Parallel block fan-out algorithm.

the diagonal block are kept in $Wait_{J,p}$. 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 by a processor $p$. Once $|Rec_{K,p}| = ToRec_{K,p}$, then the storage associated with blocks in column $K$ is reclaimed.

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 to which the block must be sent. The actual restriction is done by performing a *scatter decomposition* [7] (sometimes referred to as a *torus-mapping*) 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 ($P = s \times s$). 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 $p_{s-1,s-1}$. 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 $map[L_{IJ}] = p_{I \bmod s, J \bmod s}$. 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 $2s = 2\sqrt{P}$ 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.

## 5.3 Enhancement: Domains

Before presenting performance results for the block fan-out approach, we first note that the method as described above actually produces more interprocessor communication than competing column approaches for small parallel machines. 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 generates $O(k^2 P)$ communication volume [11]. In the block approach, every non-zero in the matrix is sent to

$O(\sqrt{P})$ processors, so the total communication volume grows as $O(\sqrt{P}k^2 \log k)$. The communication in the block approach grows less quickly in $P$, but it grows more quickly in $k$. The $k$ term is more important for small $P$.

An important technique for reducing communication in column methods involves the use of *domains* [1, 3]. 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 modifications to these columns can be performed without the need for interprocessor communication. Furthermore, the modifications 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 modifications to the appropriate destinations. In a column approach, the aggregate modification is sent out on a column-wise basis. We refer the reader to [3] for more details.

Ashcraft suggested [1] 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 modification is computed column-wise as well. We use an extremely efficient left-looking supernode-supernode method for both. Once the aggregate modification 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 [8] described an algorithm for assigning a small set of domains to each processor so that the amount of domain work assigned to the processors 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 becomes a three phase process. In the first phase, the processors factor the domains assigned to them and compute the modifications from these domains to blocks outside the domains. In the second phase, the modifications 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 [12]. Total communication volume for these non-zeroes using a block approach is thus $O(\sqrt{P}k^2 \log P)$. The other component of communication volume when using domains is the cost of sending domain modifications to their destinations. The total size of all such modifications is $O(k^2)$, independent of $P$, so domain modification communication represents a lower-order term. Total communication for a 2-D grid problem is thus $O(\sqrt{P}k^2 \log P)$.

# 6 Evaluation

This section evaluates the parallel block fan-out approach proposed in the previous section. We first 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 and using the DASH machine.

## 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 5 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

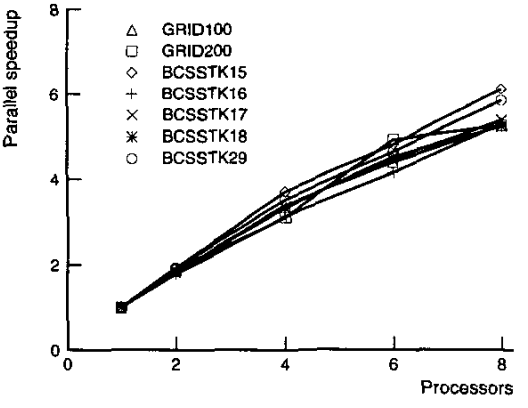


Figure 5: Parallel speedups for block fan-out method on SGI 4D-380, $B = 24$.

is indeed quite efficient for small machines. In fact, performance is slightly higher than that of our highly efficient panel-based parallel code [19]. Speedups on 8 processors are roughly 5.5-fold, corresponding to absolute performance levels of 40 to 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.

## 6.2 Moderately Parallel Machines

We now consider performance on larger machines.

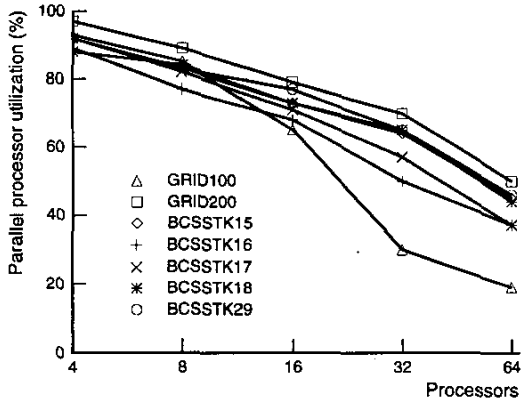


Figure 6: Simulated parallel efficiencies for block fan-out method, $B = 24$.

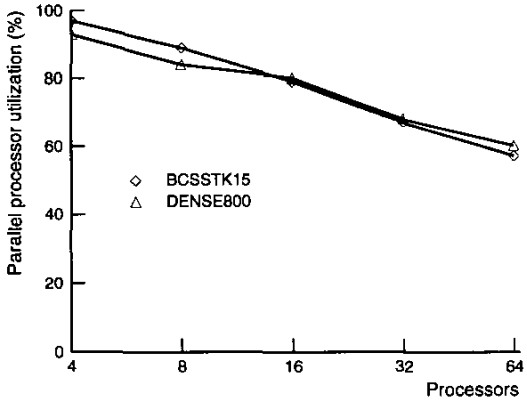


Figure 7: Parallel utilization upper bounds due to load balance for BCSSTK15, compared with load balance upper bounds for a dense problem ($B = 24$). The two problems require roughly the same number of floating-point operations.

### 6.2.1 Simulated Performance

Figure 6 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.

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 $p_{I \bmod s, J \bmod s}$. 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 7 compares the quality of the load balance obtained for matrix BCSSTK15 to the load balance obtained using the same mapping strategy for a dense matrix. The curves show the maximum obtainable processor utilization levels with the block mappings. The dense problem is chosen so as to perform roughly the same number of floating-point operations as the sparse problem.

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, 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

An important determinant of parallel performance that we have not considered so far is interprocessor communication volume. Figure 8 shows the volume of communication that a block fan-out method generates. The figure shows relative communication, as compared with a parallel column multifrontal method. Interestingly, the block approach does not always produce less communication than the column approach on 64 or fewer processors. While the growth rates, $O(P)$ for columns and $O(\sqrt{P} \log P)$ for blocks, favor the block approach, constants make these rates less relevant for small $P$. However, the trends clearly favor the block approach.

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.

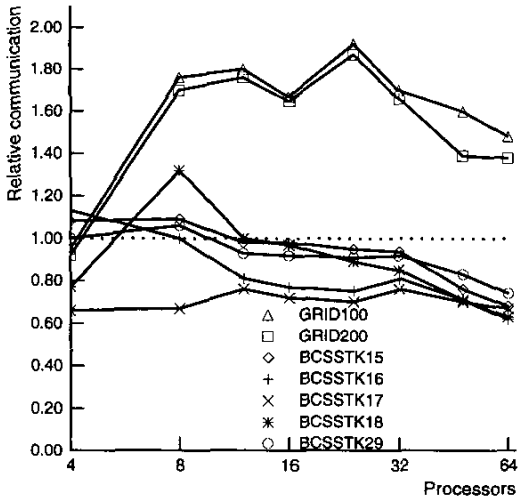


Figure 8: Communication volume of block approach, relative to a column-oriented parallel multifrontal approach.

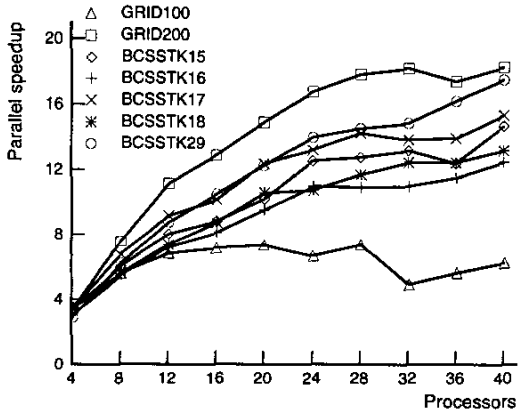


Figure 9: Parallel speedups for block fan-out method on the Stanford DASH machine, $B = 24$.

#### 6.2.3 DASH Performance

We now provide performance numbers from a block fan-out implementation on the Stanford DASH machine. Figure 9 shows achieved parallel speedups on 1 to 40 processors, again compared with a sequential left-looking supernode-supernode method. Recall that the sequential method obtains between 7 and 8 MFLOPS on these problems. The figure shows that speedups are relatively low, ranging from 12 to 18 on 40 processors. These speedups are somewhat lower than those predicted by the simulation. We believe the main cause of this difference is an assumption we made in the simulation. We assumed that the processor could perform computation simultaneous with communication. The DASH machine has limited ability to hide communication latencies.

While these speedups are relatively low, we should note two important items about the results. First, the absolute parallel performance levels of the DASH machine are still quite respectable. The 40 processor machine achieves roughly 100 double-precision MFLOPS. Second, we note that these performance numbers are roughly 10% to 40% higher than corresponding numbers from our panel-oriented parallel multifrontal implementation [17].

#### 6.2.4 Summary

To summarize this section, 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 relatively small size of the matrices and the relatively poor load balance that results from our 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. Even so, we found that the block approach produces higher performance that a competing panel-oriented approach on the 8 processor SGI 4D/380 and the 40 processor Stanford DASH machine.

At this point, we wish to reiterate that communication and concurrency growth rates greatly favor the block method on large parallel machines. The fact that a block approach produces slightly better performance than column approaches for relatively small machines leads us to conclude that the block approach will provide significant benefits for practical parallel machine sizes.

## 7 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 proposed a parallel block algorithm that is quite practical. 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 provides good per-processor performance, since it performs the vast majority of its work within dense matrix-matrix multiplication operations; (4) it is efficient on moderately parallel machines, providing performance that is comparable to that of efficient column (and panel) methods; and (5) it shows good promise for large 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] 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.

[2] Ashcraft, C.C., "The fan-both family of column-based distributed Cholesky factorization algorithms", in *Graph Theory and Sparse Matrix Computation*, IMA Volumes in Mathematics and its Applications, Volume 56, Springer-Verlag, New York, 1993.

[3] 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.

[4] 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.

[5] 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.

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

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

[8] Geist, G.A., and Ng, E., "Task scheduling for parallel sparse Cholesky factorization", *Internation Journal of Parallel Programming*, 18(4): 291-314, 1989.

[9] George, A., Heath, M., Liu, J., and Ng, E., "Solution of sparse positive definite systems on a hypercube", *Journal of Computational and Applied Mathematics*, 27(1): 129-156, 1989.

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

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

[12] Hurlbert, L, and Zmijewski, E., "Limiting communication in parallel sparse Cholesky factorization", *SIAM Journal on Scientific and Statistical Computing*, 12: 1184-1197, 1991.

[13] Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W.D., Gupta, A., Hennessy, J., Horowitz, M., and Lam, M., "The Stanford DASH multiprocessor", *IEEE Computer*, 23(3):63-79, March, 1992.

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

[15] Liu, J., "The role of elimination trees in sparse factorization", *SIAM Journal on Matrix Analysis and Applications*, 11:134-172, 1990.

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

[17] Rothberg, E., *Exploiting the memory hierarchy in sequential and parallel sparse Cholesky factorization*, Ph.D. thesis, Stanford University, January, 1993.

[18] 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.

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

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

[21] Schreiber, R., "Scalability of sparse direct solvers", in *Graph Theory and Sparse Matrix Computation*, IMA Volumes in Mathematics and its Applications, Volume 56, Springer-Verlag, New York, 1993.

[22] 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.

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