An Environment for Run-time Partioning and Scheduling of Parallel Block Sparse Cholesky Factorization

Sesh Venugopal
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903.

Vijay K. Naik
IBM Research Division
T. J. Watson Research Center
Yorktown Heights, NY 10598.

Abstract: In this paper, we describe a model for 2-D block-based sparse Cholesky factorization algorithm for scalable architectures. To use this model effectively and efficiently, we have built a scalable software system called Sparse Hybrid Automatic Parallelization Environment (SHAPE). This environment allows automatic, yet parameterized, partitioning and scheduling of computations and communication at run-time. In this paper, we describe the design of various components of SHAPE and present some of the implementation details with particular emphasis on the partitioner. We present performance data from experiments on a 32-node iPSC/860, using various structured and unstructured sparse matrices. In addition to explaining the preprocessing costs associated with SHAPE, we also compare the performance of the resulting parallel block sparse factorization with a parallel column-based factorization method.

1 Introduction

Unstructured computations are quite common in the solution of sparse linear systems of equations. As a result, efficient partitioning and scheduling of these computations for distributed memory message-passing machines is a hard problem. Distributed sparse Cholesky factorization, as a method of solving such sparse linear systems, has been studied extensively in recent years. In a recent survey of this research [16], the authors point out that the performance of existing methods on message passing systems has been mediocre. Most of these methods use column-based partitioning and scheduling techniques – partitions assigned to processors consist of collections of entire columns and the individual computations scheduled on each processor are in the form of operations on an entire column. Analysis and experimental observations indicate that although there is sufficient parallelism in sparse Cholesky factorization, poor exploitation of locality in the currently used methods leads to high communication costs, low cache and floating-point unit utilization, and hence poor performance.

To overcome these difficulties in sparse direct solvers, and particularly in sparse Cholesky factorization, in recent years block-based partitioning and scheduling schemes have been proposed and analyzed [20, 29, 31, 36]. Using simulations and analysis, in [34, 36] it is shown that distributed sparse Cholesky factorization based on block partitioning promises better scalability with respect to communication and load-imbalance. In [39], experimental results from iPSC/860 are presented to compare the performance of a column-based method with that of a 2-D block-based method for distributed sparse Cholesky factorization. These results indicate that the block-based methods reduce the total number of messages communicated as well as the total volume of data transferred among processors. In addition, these methods improve cpu performance by better utilization of the cache and the floating-point unit.

Although distributed sparse solvers based on 2-D partitioning have shown promise for better performance and scalability, a major practical difficulty lies in partitioning the sparse matrix and allocating and scheduling the partitions among processors so that the resulting communication overhead is low and the load is well balanced. Manually extracting suitable block partitions of a general sparse matrix is tedious and error-prone. Large structured and unstructured matrices encountered in real-life applications may contain several million non-zero entries and, in such cases, any type of systematic block partitioning by hand is simply impractical. To address these issues, we started our investigation of general 2-D block partitioning and scheduling in [36], where we proposed an automatic, general purpose partitioning scheme that takes into account the sparsity structure of the matrix. Using the principles expounded in that paper, we have developed a parallelization tool called Sparse Hybrid Automatic Parallelization Environment (SHAPE) for sparse Cholesky factorization on message passing multiprocessor systems [37]. SHAPE partitions a sparse matrix into a hybrid mix of dense blocks and sparse columns so as to facilitate efficient allocation and scheduling of the computations among processors. In addition to the partitioner, the environment includes an automatic allocator and scheduler. The environment also supports a set of primitives that allow the user to write a 2-D block based distributed sparse Cholesky factorization code. The distributed code written using SHAPE primitives is marginally more complex than a sequential sparse Cholesky factorization code and delivers performance that is comparable to hand-tuned codes.

In this paper, we describe the design of various components of SHAPE, with emphasis on the partitioner. To motivate the design issues, we construct 2-D block-based parallel sparse Cholesky factorization algorithms. We discuss some of the salient points of our implementations and present detailed performance results obtained from experiments on a 32-node iPSC/860. To demonstrate the performance and versatility of SHAPE, for these experiments we have used a variety of large structured and unstructured test matrices.

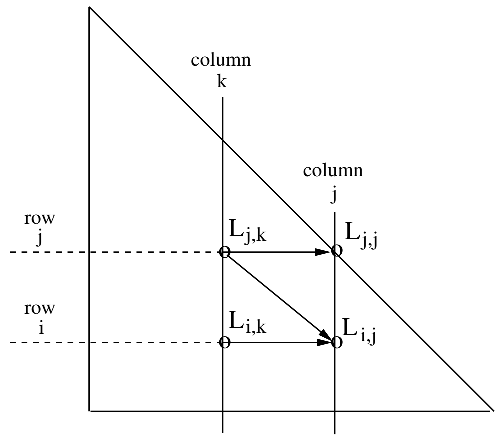
The rest of the paper is organized as follows. In Section 2, we present background information on sparse Cholesky factorization. In Section 3, we describe four models for parallel sparse Cholesky factorization, introduce 2-D block-based methods, and highlight the differences between the commonly used column-based methods and the 2-D block-based methods. In Section 4, we give an overview of the design and the functionality of SHAPE, and describe the primitives supported by this environment. The details of the pre-partitioning and the partitioning phases of SHAPE are discussed in Sections 5 and 6. The allocator and scheduler in SHAPE are briefly discussed in Section 7. Design and implementation details on the efficient 2-D block operations in the numerical factorization are described in Section 8. Presented in Section 9 are the results on the performance of SHAPE both in terms of the time required to partition and schedule various sparse matrices and in terms of the quality of the actual numerical factorization step. The paper concludes with Section 10.

2 Background, definitions, and notation

To introduce the problem and to define terms that we will be referring to in the rest of the paper, in this section we briefly describe the sequential Cholesky factorization algorithm and the element-level dependencies therein. We also describe the special steps necessary to efficiently factor a sparse matrix. For a detailed introduction on this topic, please refer to any elementary book on sparse matrix computations (see, for example, [14]).

Consider the direct solution of $A\mathbf{x} = \mathbf{b}$, where A is an $N \times N$ sparse symmetric positive definite matrix. The Cholesky factorization algorithm factors A into L and L^T, where L is a lower triangular matrix [15]. Shown in Figure 1(a) are the element-level data dependencies of the Cholesky factorization algorithm. In that figure, the direction of the arrows indicates the direction of data flow. The steps in computing L are shown in Figure 1(b). The algorithm represents a particular method of factorization that is commonly referred to as the column-oriented sparse Cholesky factorization. In that algorithm, initially L is set to the lower triangular part of A and $L_{i,j}$ denotes the element in row i and column j of matrix L. Elements $L_{j,k}$ and $L_{i,k}$ from column k of the factor L are required in computing element $L_{i,j}$ in the statement $L_{i,j} = L_{i,j} - L_{i,k} * L_{j,k}$ of the Cholesky factorization algorithm. We refer to this operation as a single *update* operation. In this operation, we refer to $L_{j,k}$ as the *base* element, $L_{i,k}$ as the *multiplier* element and $L_{i,j}$ as the *updated* element. For the diagonal element $L_{j,j}$, in each update, the base and the multiplier are one and the same. We refer to this as *base update*. For the off-diagonal elements, where an update is performed by a base and a multiplier pair, the update is referred to as *base-multiplier update*. Note that in computing the final value of $L_{i,j}$, it must be updated by all pairs of nonzero elements $L_{j,\bar{k}}$ and $L_{i,\bar{k}}$, $1 \leq \bar{k} < j$. After all the updates are performed, the element is *scaled*. In this step, the updated element is divided by the square root of the updated diagonal element in that column.

The notion of *elimination tree* structure has been used extensively to describe and implement many aspects of sparse factorizations (see e.g., [19, 25, 33]). Each node in the elimination tree corresponds to a column of A. If the nodes in this tree are labeled by the indices of the corresponding columns of A, then the parent of any node i, $i < N$, is the node defined by $\min\{k | L_{k,i} \neq 0\}$; i.e., there is an edge (i, k) in the tree if and only if k is the row number of first nonzero element in column i



(a)

for $k = 1$ **to** n **do**
  $L_{k,k} = \sqrt{L_{k,k}}$
  foreach $i \in [k+1, n]$ s.t. $L_{i,k} \neq 0$ **do**
    $L_{i,k} = L_{i,k} / L_{k,k}$
  end foreach
  foreach $j \in [k+1, n]$ s.t. $L_{j,k} \neq 0$ **do**
    foreach $i \in [j, n]$ s.t. $L_{i,k} \neq 0$ **do**
      $L_{i,j} \leftarrow L_{i,j} - L_{i,k} * L_{j,k}$
    end foreach
  end foreach
end for

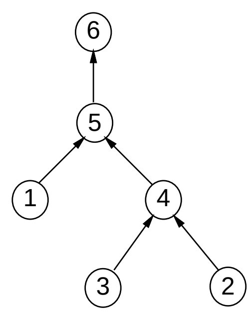
(b)

FIGURE 1
(a) Inter-element dependencies in Sparse Cholesky factorization (schematic) and (b) sparse Cholesky factorization algorithm.

below the diagonal of A. The elimination tree represents a computation graph for the factorization process and can be defined using the structure of L and hence from the structure of A. Figure 2(a)

$$\begin{bmatrix} \times & & & & & \\ & \times & & & & \\ & & \times & & & \\ & \times & \times & \times & & \\ \times & & & \times & \times & \\ & & \times & \circ & \times & \times \end{bmatrix}$$

(a)



(b)

FIGURE 2
An example of sparse L and its elimination tree

shows the sparsity structure of a lower triangular matrix, where the symbol $\times$ indicates position of a nonzero in the lower triangular part of A and o indicates a *fill*. The corresponding elimination tree is shown in Figure 2(b). Note that column 5 of L cannot be computed until columns 1 and 4 are computed. In the elimination tree, this is indicated by the fact that nodes 1 and 4 are children of node 5. Note also that, by definition, the elimination tree does not have transitive edges: column 6's computation depends on columns 3, 4 and 5 but the edges $(3, 6)$ and $(4, 6)$ are not included.

Another useful concept in sparse Cholesky factorization is that of a *supernode* [4, 12]. We define supernode as a set of contiguous columns $i, i+1, \cdots, j$ in L such that each column in this set, except for column i, has only one child in the corresponding elimination tree and the zero-nonzero structure of columns $i+k$ and $i+k+1$ in L, $i \leq i+k < j$, differ only row $i+k$ where the former has a nonzero element. Examples of supernodes are shown in Figure 3 where the columns of a 49×49 sparse matrix are partitioned into supernodes by the vertical lines running through the matrix.

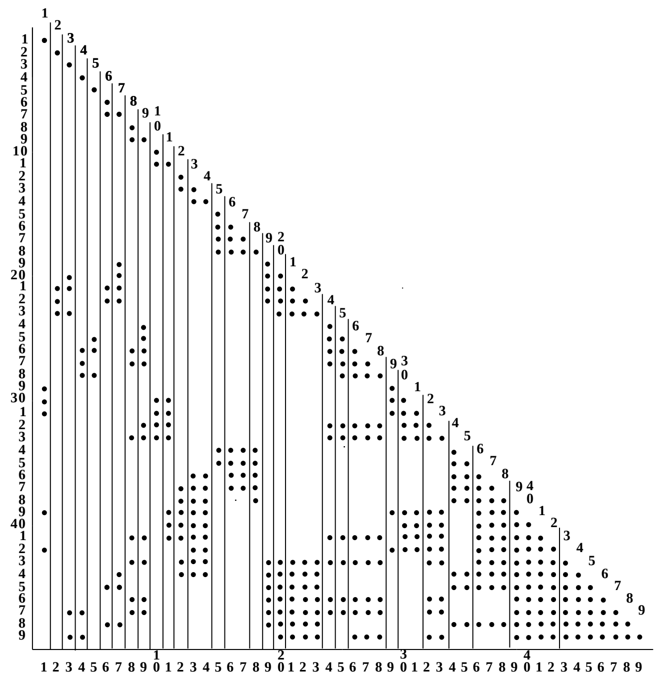


FIGURE 3

The zero-nonzero structure in the lower triangular factor of a 49×49 matrix. A $\bullet$ represents a nonzero and supernodes are represented by vertical partitions.

In practice, to improve the efficiency of sparse Cholesky factorization additional steps are applied prior to the actual numerical factorization. These are *ordering* and *symbolic factorization*. Further details on these steps can be found in any reference book on sparse matrix computations (see, for example, [14]). Most often, a "good" ordering implies one which leads to a sparse factor and fewer arithmetic operations in the numerical factorization step. Parallelism exposed by ordering is an additional consideration in distributed Cholesky factorization. Similarly, the symbolic representation of the factor L is also necessary in the partitioning and scheduling phases of distributed sparse Cholesky factorization that we describe in this paper.

3 Distributed sparse Cholesky factorization

3.1 Models for parallel sparse Cholesky factorization

In sparse Cholesky factorization, parallelism is available at various levels. Depending on the granularity at which parallelism is extracted, parallel factorization methods can be classified into four categories, each with its own model of parallel computation. In [23], the author has described three computational models based on task parallelism. We prefer to classify the models based on data parallelism. This classification is better suited for the methods proposed in this paper.

The finest grain parallelism in Cholesky factorization is extractable at the level of individual add-multiply and division operations associated with the update and scale computations. This is the *operation-level model* of parallel factorization. Under this model, a very high degree of parallelism is achievable, but because of its fine-grain nature, methods based on this model do not perform well on the existing and proposed MIMD architectures where memory latency costs are relatively high. Also, the associated bookkeeping costs tend to be high on general purpose systems. The parallel factorization method presented in [41] is based on this model.

The granularity of parallel operations is increased if parallelism is extracted at the level of nonzero elements in L. We refer to this as the *element-level model* of parallel factorization. In this model, all the computations corresponding to any element are grouped into an atomic unit, and the parallelism is viewed at this level of atomicity. The parallelism in this model is still too fine-grained to be efficiently exploited on the current MIMD architectures.

The level of granularity can be further increased by grouping together elements of the matrix into larger units. A single column (or row) in a matrix is the most common example of such grouping of elements. In the update and scale operations, a column is considered as an atomic entity. Under such a model, the data structures and the operations are all column oriented and parallelism is extracted at the level of column-update-column or column-scale operations. Hence this is the *column-level model* of parallel factorization. In this model, the data distribution schemes are also column based; i.e., processors are assigned one or more columns of L. We shall refer to such data distribution schemes as *column-based* or *column partitioning* methods. Most commonly used and studied parallel sparse factorization methods are based on this model [5, 9, 24, 42, 10, 2]. This is so because of its relative simplicity in analysis and the close resemblance of the data structures with the algebraic representation of Cholesky factorization. In Section 3.2, we discuss the column-based methods in some detail.

Instead of grouping the elements of L only along its columns (or rows), one can group these into two dimensional blocks; i.e., along both rows and columns and not necessarily grouping together all the elements of a row or a column. The update and scale operations in Cholesky factorization can then be performed using these blocks as single entities. We refer to this as the *2-D block-level model* of parallel factorization. Under this model, the matrix data is partitioned in the form of 2-D blocks. The blocks may contain only the nonzeros or may contain both the zeros and nonzeros. Once the matrix is partitioned, the resulting 2-D blocks are then distributed among the processors. This type of partitioning is the *2-D block partitioning* method. Because of the greater flexibility offered in choosing the extents of each block, methods based on the 2-D block model have greater potentials for exploiting parallelism as well as locality than it is possible with the column-based methods. However, using the flexibility effectively is a difficult proposition, especially for large unstructured sparse matrices.

We view the 2-D block model of parallel factorization as a generalization of the element-level model and in Section 3.3, we present a parallel sparse factorization algorithm based on this model. This algorithm uses predefined block partitions of any sparse matrix. In the subsequent sections, we describe SHAPE, an environment we have developed for automatic partitioning of a sparse matrix into blocks and for efficiently using the blocks in 2-D block-based factorization methods. One feature of our partitioner is that only the nonzero regions of L are taken into account in forming the blocks.

3.2 Column-based methods

Reconsider the sequential sparse Cholesky algorithm presented in Figure 1(b). We repeat this algorithm in Figure 4(a). The outermost **for** loop in this algorithm can be treated as steps or stages of factorization. At step k of the computation, column k of L is formed by dividing the nonzero entries in that column by the square root of its diagonal element. The columns of L that are to the right of column k are then updated by column k in the doubly nested j, i loop. In [13] and [23], all the scaling operations on column k are grouped into a single *cdiv*(k) task and all the update operations of column j by column k are grouped into a *cmod*(j, k) task. We use the same terminology here. The collection of update operations involving two sparse columns will be referred to as a *cmod* operation and the task of updating column j by column k will be referred to as *cmod*(j, k) task. Similarly, we define *cdiv* operation and *cdiv* task. Using these terms, in Figure 4(b)

(a)

```
for k = 1 to n do
    L_{k,k} = sqrt(L_{k,k})
    foreach i ∈ [k+1, n] s.t. L_{i,k} ≠ 0 do
        L_{i,k} = L_{i,k} / L_{k,k}
    end foreach
    foreach j ∈ [k+1, n] s.t. L_{j,k} ≠ 0 do
        foreach i ∈ [j, n] s.t. L_{i,k} ≠ 0 do
            L_{i,j} ← L_{i,j} − L_{i,k} * L_{j,k}
        end foreach
    end foreach
end for
```

(b)

```
for k = 1 to n do
    cdiv(k)
    foreach j ∈ [k+1, n] s.t. L_{j,k} ≠ 0 do
        cmod(j, k)
    end foreach
end for
```

FIGURE 4

(a) Generic version of sparse Cholesky factorization algorithm and (b) a column-based sparse Cholesky factorization algorithm.

we rewrite the algorithm of Figure 4(a). The two algorithms are shown side-by-side to show the construction of *cmod* and *cdiv* from the individual operations. All operations in the algorithm of Figure 4(b) are at the column level and so this is a column-based Cholesky factorization algorithm. Clearly, in the column-based algorithm, after a *cdiv*(k) task, all the *cmod*(j, k) tasks, $j > k$, can be performed in parallel. Similarly, all the *cmod*(k, i) tasks, $i < k$, can be performed in parallel provided the stores to the memory locations of column k are performed either sequentially or in some consistent manner such as by using a logarithmic tree. Thus, sparse Cholesky factorization allows for several *cdiv* and *cmod* operations to take place concurrently.

The column-based parallel sparse factorization methods used in practice are variations of the algorithm of Figure 4(b). The three main variations are: fan-in, fan-out, and multifrontal methods. These differ primarily in the data-flow of the intermediate results among the processors. In the following, we present the salient points of these three basic implementations. This will allow us to discuss and compare the 2-D block based methods with these commonly used methods. For the sake of bravity, however, we have omitted all the details including important optimization steps that are applied to reduce various overheads. The references indicated in following provide some of these details. In all parallel implementations of the column-based methods, the matrix is partitioned along columns (individually or in groups) and these partitions are mapped onto the processors according some scheme such as wrap mapping or subtree-to-subcube mapping. In the following discussion, we say that a processor *owns* column i, if that processor performs the $cdiv(i)$ task and we denote that processor by $P_{(i)}$.

In the distributed fan-out method (see, for example [11]), for any column j, all $cmod(j,k)$ tasks and the $cdiv(j)$ task take place in the processor $P_{(j)}$. After column j is factored, it is sent to all the processors that perform a $cmod(i,j)$ task, for some column i, $i > j$; i.e., the factored column j is *fanned-out.* In the distributed fan-in method (see, for example [3]), for any column j, the $cmod(j,k)$ tasks for different k's are distributed among several processors. Typically, the product formation part of a $cmod(j,k)$ task is performed by processor $P_{(k)}$. Each such processors transmits the partial updates of column j to processor $P_{(j)}$, i.e., the undates are *fanned-in.* As in the fan-out method, the $cdiv(j)$ task is performed in processor $P_{(j)}$.

In both fan-out and fan-in methods, the main source of computational inefficiency is in the use of indirect addressing. This difficulty is overcome in the multifrontal method where dense matrix partial factorizations are performed [7, 26]. The multifrontal method makes use of the structure associated with the columns of a supernode. All *cmod* tasks associated with the columns of a supernode can be computed using the same structure. In computing these partial updates, the affected columns are represented by a dense lower triangular matrix called *frontal matrix*. An obvious advantage is that these updates can be done efficiently without resorting to indirect addressing. The *cdiv* tasks are performed in the corresponding part of the dense frontal matrix prior to performing the partial updates. A supernodal elimination tree is used in determining the order of in which the supernodes are evaluated and the order in which the partial updates are merged to get the final updates. The key aspects to note here are that the columns are grouped together so that the corresponding *cdiv* and *cmod* operations can be performed on dense matrices.

Among the three methods, the distributed multifrontal methods have delivered the best overall performance. This is primarily because of lower communication costs and superior single processor performance resulting from the use of dense data structures. However, even with the multifrontal methods there are limitations on the expected parallel performance. Note that grouping columns into larger units increases the data locality, but decreases the available parallelism. This gives rise to difficulties in load balancing and scheduling. As a result, these methods do not scale well when the number of processors is increased.

3.3 Two dimensional block-based methods

The 2-D block methods may be considered to be a generalization of the element-level methods. Consider again the sequential sparse Cholesky algorithm presented in Figure 1(b). This algorithm can be cast into an element-level factorization method as shown in Figure 5(a). In the presentation of this and of the subsequent algorithms, we make use of data-flow concepts where instructions are executed as the data becomes available and not necessarily in the order in which the statements appear. This leads naturally to parallel methods that we describe shortly. Note from Figure 5(a) that the nonzero elements in L can be updated in any order so long as, in each update operation, the final values of the base-multiplier pair are available.

```
E ← list of nonzero elements in L
foreach e ∈ E do
    U_e ← list of element-pairs
        that update e
    s ← element
        that scales e
    if U_e = ∅ then
        Scale e by s


    else
        foreach (e', e'') ∈ U_e do
            update e by (e', e'')
            delete (e', e'') from U_e
        end foreach
    endif
end foreach
```

(a)

```
B ← list of blocks in L
foreach b ∈ B do
    U_b ← list of block-pairs
        that update b
    S_b ← list of blocks
        that scale b
    if U_b = ∅ then
        foreach b' ∈ S_b do
            bdiv(b, b')
        end foreach
    else
        foreach (b', b'') ∈ U_b do
            bmod(b, b', b'')
            delete (b', b'') from U_b
        end foreach
    endif
end foreach
```

(b)

FIGURE 5

(a) Element-level sparse Cholesky factorization algorithm and (b) block-level sparse Cholesky factorization algorithm.

A generalization of the element-level method to the 2-D blocks can be made by considering a block as a collection of one or more (neighboring) nonzero elements of L. The elements in a block are updated by elements belonging to one or more other blocks; some of the elements in a block may be updated by other elements in the same block. Furthermore, the elements in a block may be scaled by the diagonal elements that may belong to the same block or to one or more other blocks. In describing the block methods, it is convenient to describe the operations in terms of blocks instead of elements in a block. For this we define *bmod* and *bdiv* tasks that operate on the 2-D blocks. These tasks are analogous to the *cmod* and *cdiv* tasks that operate on the columns; *bmod* is a collection of update operations and *bdiv* is a collection of scaling operations. Unlike *cmod*, but as in an element update, in performing a *bmod* task, three blocks are needed: base, multiplier, and the block to be updated. Similarly, in a *bdiv* task two blocks are involved: the scaling block and the block to be scaled. Note that one or more of the blocks in a *bmod* or *bdiv* task may refer to the same block and that to complete the scaling operations on a block, more than one *bdiv* operation may be necessary. These characteristics distinguish the block operations from the element-level operations. Shown in Figure 5(b), is such a 2-D block factorization algorithm. It is assumed that the symbolic structure of L is somehow partitioned into blocks and these blocks are made available to the block factorization algorithm. No assumption is made on the partitioning strategy used nor is there any assumption on the shape of the blocks. The element-level and the block-level algorithms in Figure 5 are shown side-by-side to bring out the analogies and the differences.

3.3.1 Distributed block sparse Cholesky factorization

The block algorithm described in Figure 5(b) brings out the essence of the 2-D block model of Cholesky factorization. In that algorithm, on each block, *bmod* and *bdiv* operations are performed as the necessary data from other blocks become available. Clearly, several *bmod* and *bdiv* operations could be carried out concurrently. To express the parallelism at the level of these operations, in

```
T ← list of bmod and bdiv tasks
foreach τ ∈ T do

    if τ is bdiv(b, b') then
        get block b
        get block b'
        bdiv(b, b')

    endif
    if τ is bmod(b, b', b'') then
        get block b
        get block b'
        get block b''
        bmod(b, b', b'')
    endif
end foreach
```

(a)

```
task_list ← SET_UP()
while task_list not empty do
    τ ← delete_front(task_list)
    if  τ is bdiv(b, b') then
        GET_BLOCK(b)
        GET_BLOCK(b')
        bdiv(b, b')
        READY_BLOCK(b)
    endif
    if τ is bmod(b, b', b'') then
        GET_BLOCK(b)
        GET_BLOCK(b')
        GET_BLOCK(b'')
        bmod(b, b', b'')
    endif
end while
```

(b)

FIGURE 6

(a) A parallel block sparse Cholesky factorization algorithm and (b) an SPMD code for parallel block sparse Cholesky factorization.

Figure 6(a), we rewrite the block factorization algorithm. As before, the algorithm is written in the data-flow fashion. The list T consists of all *bmod* and *bdiv* tasks in the block Cholesky factorization. All these tasks are set for execution in the **foreach** loop and are executed as the data becomes available.

On the existing MIMD architectures, one cannot efficiently spawn off all the tasks as implied in the algorithm of Figure 6(a). Also under the single program multiple data (SPMD) paradigm of programming, better performance is obtained if the tasks on individual processors are scheduled ahead of the actual execution phase. We have shown this for the column-based algorithms in [40]. To allow for such optimizations, in Figure 6(b), we show an SPMD code that implements the parallel block sparse Cholesky factorization of Figure 6(a). It is assumed that prior to the factorization, the sparse matrix L is (symbolically) partitioned into blocks and that an ordered task list is available for the execution of *bmod* and *bdiv* tasks. In the distributed sparse factorization code shown in Figure 6(b), we use three special primitives that hide many architecture and implementation specific details. These primitives are SET_UP, GET_BLOCK and READY_BLOCK. A call to SET_UP prepares the data structures for the block partitions and the *task_list* with the desired task schedule. It also interfaces with the partitioner as we describe in the next section. When a block is available to update other blocks, the primitive READY_BLOCK is called. In a distributed environment, the newly computed block is made available to all the processors that execute tasks in which this block is an operand. Finally, GET_BLOCK returns the data structure of a block required in a *bmod* or *bdiv* task. The block fetched by GET_BLOCK could be a local block or may reside on a remote processor.

We believe that for parallel block sparse factorization schemes to become practical, these methods should be no more complex than the code shown in Figure 6(b).

3.3.2 Partitioning for general 2-D block methods

Thus far we have not addressed the issue of partitioning the sparse matrix L into blocks and neither the issue of preparing, for each processor, a schedule of tasks that operate on these blocks. It is relatively straightforward to specify block partitions for sparse matrices that represent regular grids [29, 28]. Based on the grid dimensions and the ordering used, the partitions can be constructed. However, in the case of unstructured sparse matrices, which are most often encountered in real-life applications, any systematic manual partitioning into blocks is next to impossible. For the large unstructured matrices, load balancing and scheduling of computations are even more complex. If the partitioning, mapping, and scheduling of the computations are not done properly, then the advantages of block partitioning disappear rapidly.

To illustrate these difficulties, we refer back to the sparse lower triangular factor shown in Figure 3. Note that the nonzero regions in that matrix may be partitioned into blocks in many different ways. However, a general strategy for partitioning any arbitrary sparse matrix into blocks is not obvious. To address the issue of partitioning large, unstructured sparse matrices into blocks, in [36], we first proposed a block-based automatic partitioning scheme. Our study made it clear that partitioning and scheduling general sparse matrices in a portable fashion is too complex to be done manually, and therefore requires efficient automatic software tools. Based on the partitioning scheme proposed in [36], we have developed a system called Sparse Hybrid Automatic Parallelization Environment, or SHAPE for short, which partitions a sparse matrix into blocks with minimum burden on the user [37].

In Figure 7, we show a block partitioning of the sparse matrix shown in Figure 3, obtained using SHAPE. Each shaded area covering the nonzeros represents a 2-D block. In block sparse Cholesky factorization algorithms presented earlier, no restrictions are imposed on the shape of the 2-D block partitions, whereas the dense blocks in Figure 7 are either triangular or rectangular in shape. Notice also that the sparse matrix is partitioned into dense triangular and rectangular blocks as well as into single sparse columns, which appear mostly in the left region of the matrix. We treat a single sparse column as a degenerate block partition and consider our partitioning to be hybrid in nature. To accommodate the partitions containing sparse columns, in our implementations we have modified the *bmod* and *bdiv* operations. As a result, our implementations can handle traditional column-based methods as well. In addition to partitioning the matrix, SHAPE prepares block dependencies, the task list, and a schedule of execution of these tasks in a distributed environment. The three primitives, SET_UP, GET_BLOCK, and READY_BLOCK, used in the distributed factorization algorithm shown in Figure 6 are also implemented in SHAPE. This facilitates an efficient interface with the user applications without increasing the programming complexity. In the following sections we elaborate more on SHAPE.

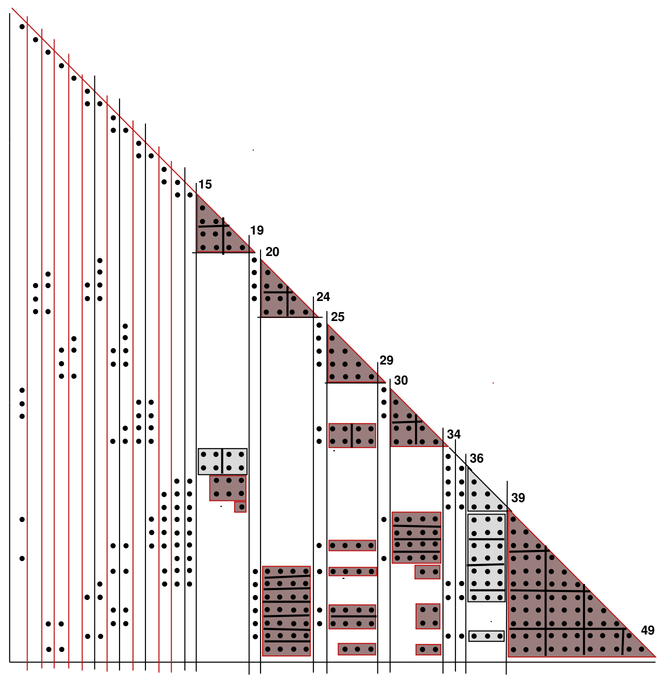


FIGURE 7
Block partitions of the lower triangular symbolic factor shown in Figure 3, obtained using SHAPE.

4 Overview of SHAPE

SHAPE is an environment that can be used either in a stand-alone mode or can be interfaced with user programs through primitives. The stand-alone mode is an interactive system, with SHAPE handling the interactions with the user and the system. In this mode, SHAPE also gathers various performance statistics and allows the user to experiment with the partitioner. This mode is most useful for studying and characterizing sparse matrices. In the noninteractive mode, where SHAPE is interfaced with user programs via primitives, SHAPE performs the role of a run-time preprocessor and of a communication manager. In either mode, SHAPE is used to solve a sparse linear system of equations on a distributed multiprocessor system.

Shown in Figure 8 is a functional diagram illustrating the steps involved in solving the system of equations $A\mathbf{x} = \mathbf{b}$, using SHAPE. After the sparse matrix A is permuted and symbolically factored, the symbolic lower triangular matrix L is input to SHAPE. The output is a schedule of tasks, in the form of a total ordering for each processor, and a profile of communication that can be used to automatically schedule inter-processor communication on each processor. In Figure 8, a dotted box encloses both SHAPE and the numerical factorization step. This is to indicate that during the numerical factorization step, there is an interaction between the two – at run-time SHAPE handles the communication and task coordination on each processor.

The important tasks performed by SHAPE are partitioning the sparse matrix, allocating and scheduling the resulting tasks on a given number of processors, and scheduling and coordinating the information sharing among processors. As a partitioner and scheduler, the output of SHAPE is a total ordering of the *bmod* and *bdiv* tasks on all processors. In this process, SHAPE also produces a communication profile that can be used to schedule and optimize the communication

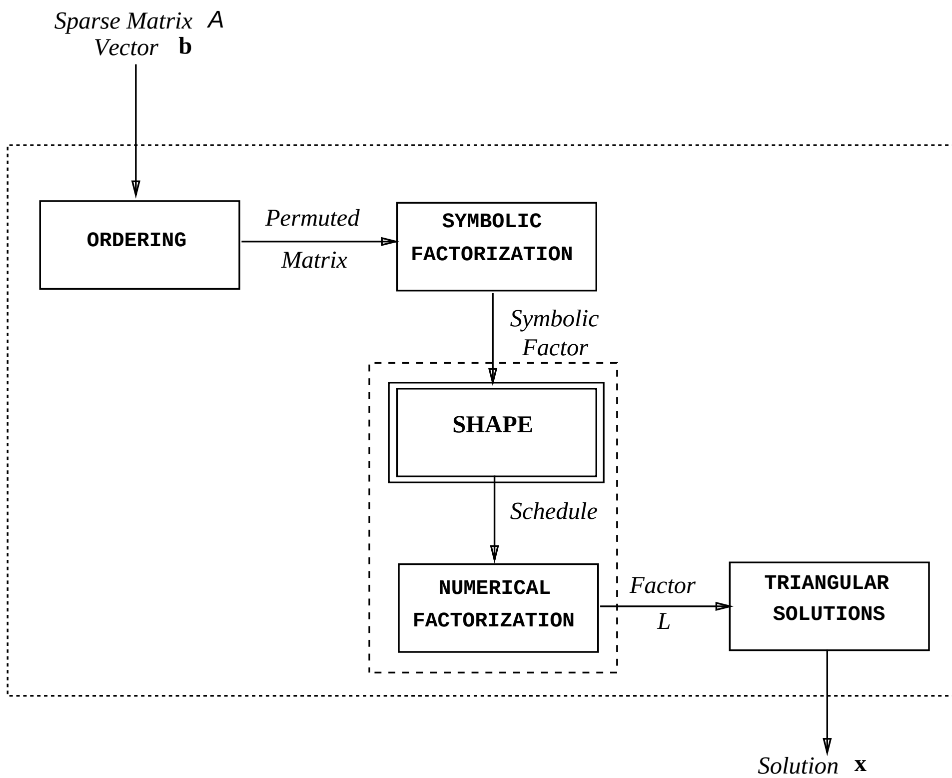


FIGURE 8
Solution of a sparse linear system using SHAPE.

on each processor during numerical factorization. To assist SHAPE in partitioning and scheduling, parameters that characterize the matrix and the target multiprocessor systems can be supplied to SHAPE. The parametrization also allows for efficient portability and scalability across a diverse range of sparse matrix applications and architectures. In the following sections, we will describe, in detail, the design and functionality of SHAPE and the role played by these parameters.

4.1 Functional description of SHAPE

Functionally, SHAPE operates in four stages: *pre-partitioning*, *partitioning*, *allocation* and *scheduling*. The interactions among these four functions are shown schematically in Figure 9. The input to the pre-partitioner is the symbolically factored lower triangular matrix, L, along with two parameters, W and G. These parameters dictate the size and shape of the resulting partitions. Using the parameter W, the pre-partitioner decomposes L into *clusters* of columns and identifies triangular and rectangular shaped blocks within each multi-column cluster. The partitioner, using the parameter G, divides each cluster into *unit blocks* of non-zeros, producing partitions that are either single sparse columns or are triangular or rectangular in shape. In addition to partitioning the sparse matrix, the partitioner also produces a *block dependency DAG* (*BDAG*), a directed acyclic graph of inter-block dependencies, and a *cluster DAG* (*CDAG*), which is a directed acyclic graph of inter-cluster dependencies. The allocator uses the CDAG and produces an assignment of

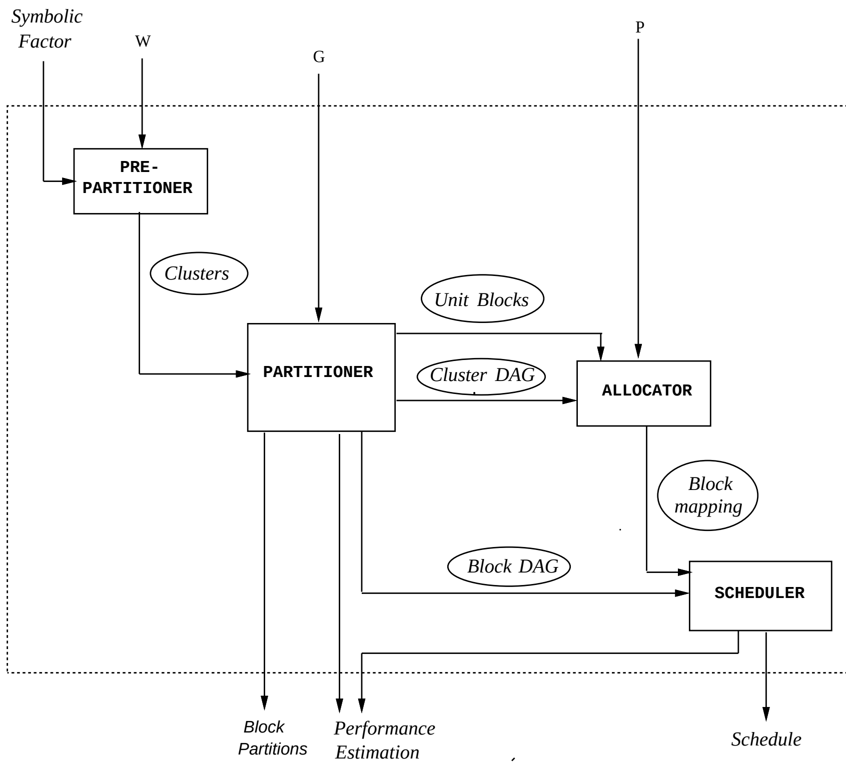


FIGURE 9
Flow diagram illustrating the functionality of SHAPE.

blocks onto the P processors. The scheduler then executes the BDAG using the block-to-processor assignment and assuming the block-owner compute rule – that all computations on a block are performed on the processor onto which that block is mapped. This determines a schedule in the form of a total ordering of tasks to processors, and a communication profile. If the computation and communication cost functions for the target architecture can be modeled, then SHAPE can use these cost functions to give an estimate of the total execution cost as well as the communication and load imbalance overheads for the partitioning and scheduling of the tasks. In addition to the pre-factorization scheduling (pre-scheduling, for short), the function of the scheduler includes the coordination of tasks and of the messages on each processor. This is brought about by means of the execution of the primitives which we describe next.

4.2 Support for primitives

SHAPE supports the three primitives: SET_UP, GET_BLOCK, and READY_BLOCK. These primitives are used in the distributed general block sparse Cholesky factorization code as shown in Figure 6(b). SHAPE provides an efficient implementation of these so that the complexities of programming for a distributed memory system are minimized and, at the same time, optimizations in computation and communication can be used. An additional advantage is that the architectural details remain transparent to the user.

The primitive SET_UP invokes the scheduler in SHAPE, which generates a complete static ordering of tasks on each processor and a communication profile. Both of these are used during the numerical factorization. If the sparse matrix has not been already partitioned into blocks and allocated to processors, then those steps are completed prior to the pre-scheduling. Thus, a call to SET_UP accomplishes all functions of SHAPE up to and including the pre-scheduling function described above. Using the task and the communication schedule to compute the factor in an efficient manner requires manipulation of distributed data structures and a detailed understanding of the capabilities and limitations of the underlying architecture. This is taken care of by the two primitives GET_BLOCK and READY_BLOCK. These primitives use the task and the communication schedules from SET_UP to conduct optimized computations and message-passing at run-time. The functions of GET_BLOCK and READY_BLOCK are as described in Section 3.3.1. In addition to these three primitives, SHAPE also supports primitives that efficiently execute *bmod* and *bdiv* tasks that are used in the 2-D block factorization algorithm shown in Figure 6(b). Thus, in the numerical factorization code, the user simply walks through the complete schedule of the *bmod* and *bdiv* tasks on each processor, without having to manipulate any data structures for computation or communication. As a result, the SPMD code written by the user is simple and resembles closely with a sequential program.

In the interest of space, here we briefly describe some of highlights of the implementations of these primitives. Additional details will be presented elsewhere. One key aspect of the implementations is that all blocks are globally enumerated. For each block, a set of all dependent blocks is maintained in a list called *send_set*. When READY_BLOCK is invoked, the computed block is sent to each dependent processor in its *send_set*. The *entire* block is sent to each of the receiving processors. When GET_BLOCK is invoked to access a block data structure required in a *bmod* or a *bdiv* task, local blocks are returned immediately. If the request is for a remote block and if the block has already been received, then that block is returned from the received buffers. If the block has not been received, a message is posted for receiving the block and the control is returned only after the arrival of the requested block. Finally, the primitives have been designed to work for any block-based scheduling. Currently, they have been implemented for the block sparse Cholesky factorization on the iPSC/860 [37].

5 The Pre-partitioning Stage in SHAPE

In most practical cases, the sparsity structure of L is such that the nonzero elements conglomerate towards the right edge, forming dense blocks. Towards the left of the matrix, there is very little block structure, but the sparsity decreases gradually from left to right, i.e., the matrix becomes more dense. Moreover, one observes that there are triangular blocks of nonzero elements at the diagonal and rectangular or near-rectangular nonzero blocks below these triangles. Accurate identification of such naturally occurring triangular and rectangular *dense* blocks in L is the first step in partitioning the sparse matrix, and is performed by the pre-partitioner.

The pre-partitioner first divides the matrix L into disjoint bands of columns called *clusters*, in the *cluster identification step*. Then, in each cluster, it identifies a set of disjoint rectangles, in the *rectangle identification* step. Informally, a cluster is characterized by a dense triangle at the diagonal and several dense rectangles below the triangle. In the degenerate case, a cluster is a column with the diagonal element representing the triangle. For instance, in Figure 10, columns 6 through 7 form a cluster, columns 15 through 18 form a cluster and so do columns 39 through 49. Columns 1 through 5 are not part of any cluster. Rectangle identification is done only for multi-column clusters.

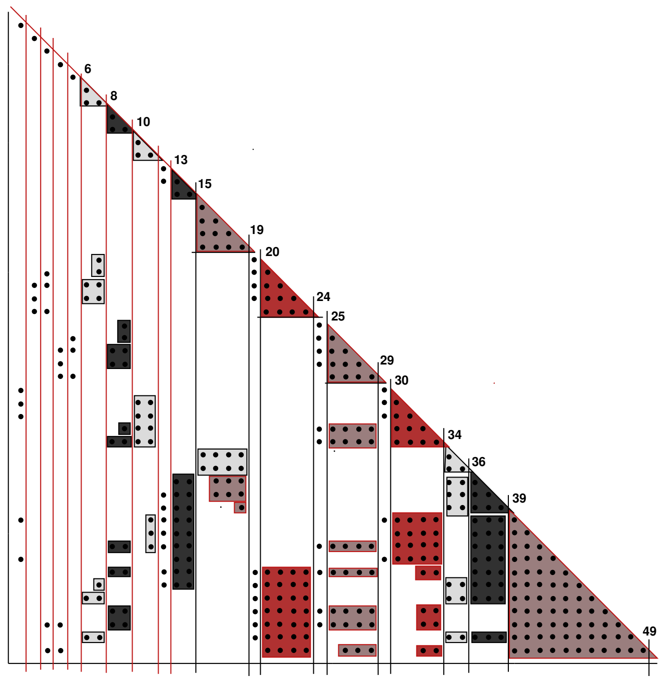


FIGURE 10

Clusters in the sparse matrix of Figure 3. Shaded areas identify the block rectangles and triangles.

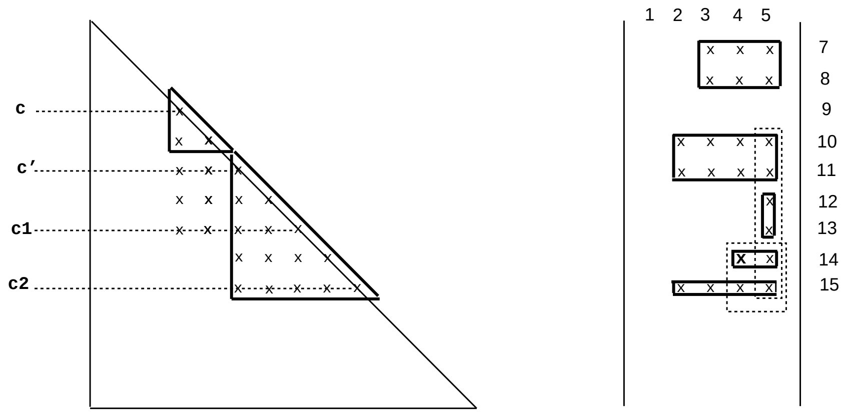
To be more precise, we define the following terms with reference to the structure of the symbolically factored lower triangular matrix L.

Definition 1: A triangle $T_{m,n}$ is the block of elements $\{L_{i,j} \mid m \le i, j \le n; i \ge j\}$. It is the triangle of elements at and below the diagonal formed by the intersection of rows m through n and columns m through n. $T_{m,n}$ is a *dense* triangle if none of its elements is zero. $T_{m,n}$ is *maximally* dense if neither $T_{m,n+1}$ nor $T_{m-1,n}$ is dense, i.e., it is not properly contained in another dense triangle.

For example, in Figure 11(a), $T_{c,c2}$ is a triangle which contains some zeros. $T_{c',c1}$ is a dense triangle, but it is not maximally dense. $T_{c,c1}$ and $T_{c',c2}$ are maximally dense triangles. In Figure 10, $T_{19,23}$ is a triangle with one zero in $L_{23,19}$. $T_{19,22}$ and $T_{20,23}$ are maximally dense triangles.

Definition 2: A cluster $C_{m,n}$ is a band of columns m through n such that $T_{m,n}$ is a dense triangle. The *cluster width* of $C_{m,n}$ is $n - m + 1$.

One objective of the pre-partitioner is to identify non-overlapping clusters, that are as large as possible. Thus, the clusters associated with maximally dense triangles are of most interest. If all maximally dense triangles in L are non-overlapping, then identifying all the maximally dense triangles would complete the cluster identification step. However, as shown in Figure 11(a), maximally dense triangles may overlap and intersect. Thus, identifying only the maximally dense triangles is not sufficient. We use the following strategy to obtain non-overlapping clusters. If a maximally dense triangle $T_{m,n}$ does not intersect with another, then $C_{m,n}$ defines a cluster. If maximally dense triangles $T_{c,c1}$ and $T_{c',c2}$, $c < c' < c1$, intersect each other (see Figure 11(a)), then the columns c through $c' - 1$ form one cluster and columns c' through $c2$ form the second cluster. In other words,



(a) (b)

FIGURE 11
Dense triangles and rectangles.

of the two maximally dense triangles, the cluster defined by the triangle on the right is retained and the one formed by the left triangle is truncated. To give another example, in Figure 10, the maximally dense triangles $T_{34,38}$ and $T_{36,45}$ intersect each other. $T_{34,38}$ contributes the cluster $C_{34,35}$ and $T_{36,45}$ contributes $C_{36,45}$. Note that the clusters as identified above are not necessarily supernodes.

The following is a definition of a rectangle which is always defined in the context of a cluster.

Definition 3: A rectangle $R_{r_1,r_2,c,n}$ in a cluster $C_{m,n}$ is the set of elements $\{L_{i,j} \mid n < r_1 \leq i \leq r_2; m \leq c \leq j \leq n\}$. R is a *dense* rectangle if none of its elements is zero. R is *maximally* dense if none of the rectangles $(r_1 - 1, r_2, c, n)$ or $(r_1, r_2 + 1, c, n)$ or $(r_1, r_2, c - 1, n)$ is dense.

In Figure 11(b), we show a portion of some matrix below the triangle in cluster $C_{1,5}$. There are five maximally dense rectangles: $R_{7,8,3,5}$, $R_{10,11,2,5}$, $R_{10,15,5,5}$, $R_{14,15,4,5}$ and $R_{15,15,2,5}$. Rectangles $R_{12,13,5,5}$ and $R_{14,14,4,5}$ are dense, but not maximally dense rectangles.

Given a cluster $C_{m,n}$, disjoint dense rectangles are obtained from maximally dense rectangles as follows. If a maximally dense rectangle does not intersect another, it is reported as a disjoint rectangle. For instance, this is true of the maximally dense rectangle $R_{7,8,3,5}$ in Figure 11(b). If a maximally dense rectangle R intersects one or more maximally dense rectangles, a disjoint rectangle is obtained by extracting the largest sub-rectangle from R in such a way that none of the intersecting rectangles is split vertically. For instance, the maximally dense rectangle $R_{10,15,5,5}$ in Figure 11(b) gives the disjoint rectangle $R_{12,13,5,5}$. Adding $L_{11,5}$ to the rectangle in an attempt to enlarge it would split $R_{10,11,2,5}$ vertically. The other disjoint rectangles reported in Figure 11(b) are $R_{10,11,2,5}$, $R_{12,13,5,5}$, $R_{14,14,4,5}$ and $R_{15,15,2,5}$. In Figure 10, the disjoint dense rectangles reported in $C_{15,18}$ are $R_{34,35,15,18}$, $R_{36,37,15,18}$ and $R_{38,38,19,19}$.

In practice, sparse matrices contain several columns that do not form multi-column clusters. The pre-partitioner reports these as single column clusters without any further partitioning. For example, in Figure 10, columns 1 through 5, and columns $12, 19, 24$ and 29 remain as single columns at the end of the pre-partitioning process. An observation is that multi-column clusters with small width perform well if decomposed into single columns, i.e., it does not pay to partition small-width multi-column clusters into triangles and rectangles. The cutoff point depends on the overall structure of the sparse matrix as well as on the underlying architecture. To maintain this flexibility in deciding the cutoff point, the pre-partitioner makes use of the input parameter W in the cluster identification and rectangle identification steps. The parameter W specifies the *minimum acceptable cluster width* and acts as a switch between column-based and block-based partitioning for different regions of the matrix. Only those clusters with width at least W are accepted as multiple column clusters. The rest are decomposed into single column clusters. For instance, in Figure 10, if W is set to 4, the clusters $C_{6,7}$, $C_{8,9}$, $C_{10,11}$, $C_{13,14}$ and $C_{34,35}$ are all discarded, and their constituent columns are treated as single columns.

For the sake of brevity, here we do not present the details of the cluster and rectangle identification steps. The complete algorithms describing these steps appear in [37].

6 The Partitioning Stage of SHAPE

In this section, we describe the strategies used in block partitioning and the factors that determine the size and shape of blocks in the partitioning process. In the column-based partitioning methods, the *cmod* tasks are implicitly defined by the structure of the nonzeros within the columns. For block partitioning, however, the blocks themselves are non-uniform in nature and the partitioning varies with the sparsity structure of the matrix. The partitioner, therefore computes an explicit block dependency directed acyclic graph (BDAG) that encapsulates the inter-block dependency information. The block partitioning and the formation of the BDAG are done simultaneously. In the following, we present an overview of the partitioner and refer the reader to [37] for details of the algorithms and data structures used in the partitioner.

6.1 Block partitions and their structure

In the partitioning step, the dense triangle and rectangles in each cluster, as identified by the pre-partitioner, are partitioned into triangular and rectangular shaped sub-blocks. All the elements in each sub-block are assigned to a processor as a single unit and all the computations on that sub-block are scheduled together. For that reason we refer to a sub-block as an atomic block. Single sparse columns produced by the pre-partitioner are not partitioned by the partitioner. For uniformity, these single sparse columns are also considered to be atomic blocks. When there is no ambiguity, we refer to atomic blocks simply as blocks. The *bmod* and *bdiv* tasks operate on atomic blocks. Figure 7 shows block partitions in the lower triangular part of a sparse matrix, as produced by the partitioner in SHAPE. The triangle $T_{30,33}$ is partitioned into two atomic triangles and one atomic rectangle. The rectangle $R_{39,45,36,38}$ in cluster $C_{36,38}$ is partitioned into four atomic rectangles.

The shape and size of the dense blocks extracted by the partitioner are influenced by several factors, including the sparsity structure of the matrix, the parallelism in the problem, and the characteristics of the target architecture. In the following, we quantify these factors and discuss how these are taken into account in the design of the partitioner.

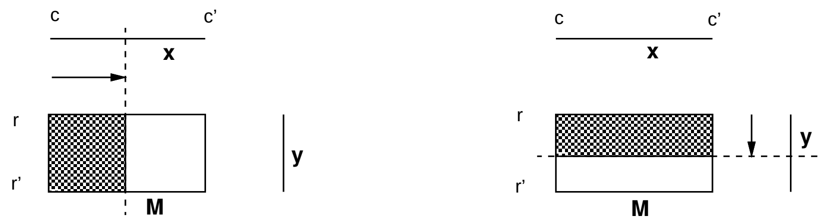
The structure of the nonzeros in the symbolic factor of most sparse matrices shows naturally occurring triangular blocks at the diagonal and polygonal blocks off the diagonal. Thus, one can partition the sparse matrix into triangular and off diagonal polygonal blocks. However, manipulating polygonal blocks is complex and makes the coding process extremely difficult. Moreover, experience shows that computations can be performed efficiently if the polygons are further partitioned into rectangles. For these reasons, we restrict the shapes of atomic blocks to equilateral triangles and rectangles. The triangular block at the diagonal is partitioned into one or more dense triangles and dense rectangles. The off-diagonal polygonal blocks are partitioned into dense rectangles. The rectangle formation of these polygons is partially accomplished in the pre-partitioning stage, as described in Section 5.

The sizes of the atomic blocks are determined by the *grain size* or the parameter G. The value of G specifies the minimum block size in partitioning the triangle and rectangles in a cluster. In general, the value of G is specified based on two factors: the computation to communication ratio of the target architecture and the need to load balance the computations across processors. G determines the size of a block on the basis of either the number of nonzeros in that block and/or the amount of computational work required to factor that block. Typically, the computations associated with nonzeros in the factor increase towards the right of the matrix, whereas the elements towards the left tend to be required in many more computations than those towards the right. Thus, in partitioning certain regions of the matrix, communication overhead must be given higher priority and in some other regions load imbalance overhead must be given higher priority. In the communication intensive regions, G is used to partition the matrix so that the locality is improved and in the load imbalance prone regions, G is used to gain more flexibility in load balancing. In the former case, the parameter G is used in determining the minimum number of nonzero elements per block and in the latter case, it is used in determining the amount of computational work per block. In its two forms, the value of G is scaled appropriately to take into account the architectural characteristics.

A third factor, the *data access* within blocks, is used by the partitioner to further refine the partitioning process. Data access is the order in which the individual elements of data in a block are accessed during computation. On a single processor, the computational performance depends on two major factors – the locality of computations and the vector lengths used. Increased locality leads to improved cache reuse and better performance. The locality of data is dictated by the size and shape of the blocks. Large vector lengths generally result in improved utilization of vector processors and/or pipelined floating point units. The vector lengths are also dictated by the size and the shape of the blocks, and also the data access used during the computations. Given a certain amount of computation to be performed, few iterations with long vectors result in better performance than several iterations with short vectors, if the vectors can be fitted into the cache. As far as possible, data in the blocks should be organized in such a manner that the computational tasks satisfy the former condition.

We consider two forms of data access in computations: *column-major* and *row-major*. In the former case, the data values within a triangular or rectangular block are stored in column-major order, while in the latter, they are stored in row-major order. To highlight the tradeoffs that must be considered in choosing one of two forms, we compare these methods using the computations involved in Cholesky factorization.

Consider the task *bmod*(b, b', b'') where block b is updated using blocks b' and b''. Without loss of generality, assume that b and b' are 1-D blocks and b'' is a 2-D rectangle. The block update using column-major and row-major data access are shown in Figure 12(a) and 12(b), respectively. In that figure, vectors $\mathbf{x}$ and $\mathbf{y}$ represent blocks b and b', respectively (the horizontal and vertical lines are used to indicate the extents of these vectors); the block matrix $\mathbf{M}$ represents the block b''. $\mathbf{M}$ and $\mathbf{x}$ have the same column range, and $\mathbf{M}$ and $\mathbf{y}$ have the same row range. $\mathbf{M}$, $\mathbf{x}$, and $\mathbf{y}$



(a) Column-major (b) Row-major

FIGURE 12
Block computation using column-major and row-major data access.

are all dense. In a sparse matrix, $\mathbf{y}$ is part of column k and $\mathbf{x}$ is part of row k. To understand the differences in the two forms of data access, first note that the basic element-level update in Cholesky factorization is of the form $y \leftarrow y - \alpha \cdot x$. The *bmod*$(b, b', b'')$ computation is a sequence of the element-level updates, given by:

$$y_i \leftarrow y_i - M_{i,j} * x_j, \quad r \leq i \leq r', \text{and} \quad c \leq j \leq c'.$$

In Figure 12, the shaded portion of $\mathbf{M}$ shows the part that has been already used in updating $\mathbf{y}$. In the case of column-major access, the progression is column-wise from left to right, and for every column of $\mathbf{M}$ accessed, one element of $\mathbf{x}$ is accessed and the entire vector $\mathbf{y}$ is updated once. In the case of row-major access, the progression of computation is row-wise from top to bottom, and for every row of $\mathbf{M}$ accessed, the entire vector $\mathbf{x}$ is used once, and one element of $\mathbf{y}$ is computed. The code for column-major access is given in the left half of Figure 13(a). After loop-interchange, this results in the code for row-major access as shown in the left half of Figure 13(b). On the right side

for $j = c$ **to** c' **do**
  for $i = r$ **to** r' **do**
    $y_i \leftarrow y_i - M_{i,j} * x_j$

for $j = c$ **to** c' **do**
  $\mathbf{y} \leftarrow \mathbf{y} - M_{*,j} * x_j$

(a)

for $i = r$ **to** r' **do**
  for $j = c$ **to** c' **do**
    $y_i \leftarrow y_i - M_{i,j} * x_j$

for $i = r$ **to** r' **do**
  $y_i \leftarrow y_i - M_{i,*} \bullet \mathbf{x}$

(b)

FIGURE 13
Element and vector updates using (a) column-major and (b) row-major data access.

of Figure 13(a) is an *axpy* enclosed in the j loop, while on the right side of Figure 13(b) is a *dot* product enclosed in the i loop.

The decision of whether to use axpy or dot product depends on the relative efficiencies of the axpy and dot product operations on the machine and on the structure of the blocks under consideration. If we assume that the axpy and dot operations execute with the same efficiency for a fixed vector length, then the decision depends purely on the block structure. It would be better to use row-major access and storage if there are relatively large number of horizontally long and flat blocks in the partitioning. On the other hand, if there are relatively large number of vertically tall and thin blocks, then it would be better to use column-major access. Moreover, towards the left, it may be advantageous to use the column major access and to use the row major access in the rest of the matrix partitions. Clearly, the selection of the access pattern to choose would depend on the sparsity structure of the matrix. Based on this, the actual shape of a rectangular atomic block is decided. In all the test matrices that we have studied, the sparsity structure is more suited for row-major access.

6.2 Triangle and rectangle partitioning mechanism

Taking into account the factors described above, we partition the blocks in each cluster into atomic blocks as follows. First consider the triangular block in a cluster. Let the width of the triangle in a cluster be w. Note that w is both the height and the base of the triangle. The partitioner divides the triangle into, say, k horizontal and vertical strips. An example is shown in Figure 14(a), where k is four. All the strips are of the same width except, possibly, the bottom-most horizontal strip and

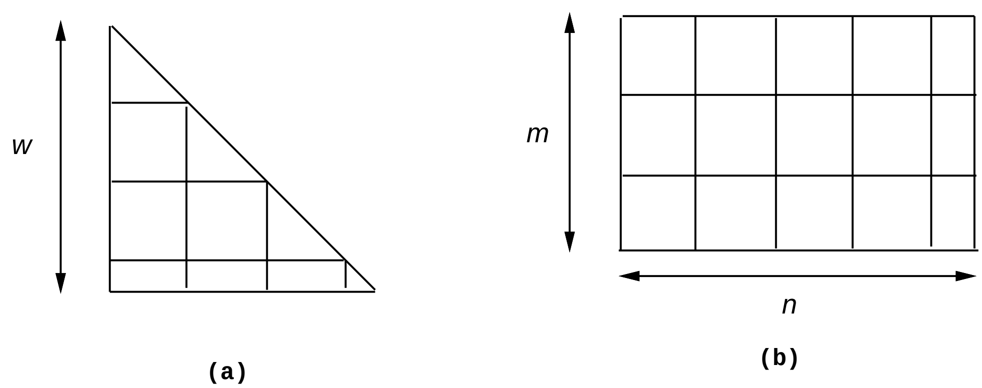


FIGURE 14
Partitioning of a dense triangle and rectangle into atomic blocks.

the rightmost vertical strip which may be smaller in width. After the partitioning, all rectangular blocks are of the same size and all triangular blocks are of the same size, except, possibly, those at the "edges". The triangle is thus partitioned into $k(k+1)/2$ atomic blocks: k triangular blocks and $k(k-1)/2$ rectangular blocks. The size of each partition (and hence k) is controlled by the grain size parameter T. The parameter T imposes a lower bound on the block size and hence an upper bound on the number of partitions. Let T allow at most t partitions in a triangle. Then the partitioner will choose the largest k for which $k(k+1)/2$ is at most t.

Next, consider a rectangle of size $m \times n$, where m is the height and n is the width. The parameter R is used to specify the minimum grain size of the partitions of a rectangle. Given m, n, and R the maximum number of partitions of a rectangle are fixed. As discussed earlier, depending on the desired amount of locality or the desired form of the vectors, the preferred shape of the partitions might be square or flat horizontal strips, or tall vertical strips. When square blocks are desired, the partitioner divides the rectangle into p horizontal and q vertical strips so that the resulting blocks are square in shape, except at two edges, and each block has at least R elements. The "edge" strips may be narrower than the rest. For example, in Figure 14(b), p is equal to three and q is five. When partitions are desired to be "flat" or "tall", additional parameters are specified. These are L_h or L_v and the orientation. For flat horizontal strip partitions, L_h specifies the minimum width of the partitions and thus the size of the row vectors. For tall vertical strip partitions, L_v specifies the minimum height of the partitions and thus the size of the column vectors. The rectangle is partitioned in such a manner that except at the "edges", all partitioned blocks yield such vectors.

Note that the grain size parameter, G, discussed in Section 6.1 is specified in terms of the parameters T, R, and (possibly) L_h and L_v.

6.3 Construction of inter-block dependencies

For computing the inter-block dependencies, we use the idea of *projection* and the intersection of projections. We illustrate this procedure using the example shown in Figure 15. In that figure,

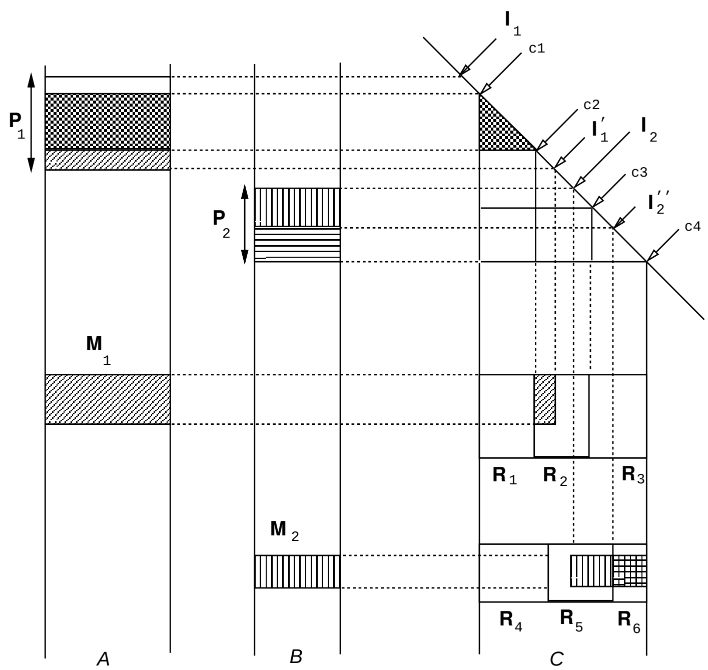


FIGURE 15
Inter-block dependencies in Cholesky factorization.

A, B, and C are clusters. Cluster A has two rectangular blocks, P_1 and M_1, cluster B has two rectangular blocks, P_2 and M_2. The triangle in cluster C is partitioned into six atomic blocks: three triangular and three rectangular in shape. The two off-diagonal rectangular blocks in C are each partitioned into three atomic blocks; the upper rectangle is partitioned into R_1, R_2 and R_3 and the lower rectangle is partitioned into R_4, R_5 and R_6.

Definition 4: The *horizontal projection* of a rectangular block R_{r_1,r_2,c_1,c_2}, formally denoted as $HP(R)$, is the *interval* $[r_1,r_2]$, obtained by extending the top and bottom edges of R_{r_1,r_2,c_1,c_2} to the diagonal. The horizontal projection of a triangular block $T_{m,n}$, is the interval $[m,n]$. This is denoted as $HP(T)$.

Definition 5: The *vertical projection* of a rectangular block R_{r_1,r_2,c_1,c_2}, formally denoted as $VP(R)$, is the interval $[c_1,c_2]$, obtained by extending the left and right edges of R_{r_1,r_2,c_1,c_2} to the diagonal. The vertical projection of a triangular block $T_{m,n}$ is the interval $[m,n]$ and is denoted as $VP(T)$.

A single point P at (i,j) is defined to be the degenerate rectangle $P_{i,i,j,j}$ if $i \neq j$; otherwise, it is defined to be the degenerate triangle $T_{i,i}$. In both cases the projections are degenerate zero-length intervals. For the example in Figure 15, $HP(P_1)$ is the interval $[I_1, I_1']$, $HP(P_2)$ is the interval $[I_2, c_4]$, and $VP(R_2)$ is the interval $[c_2, c_3]$.

Projections are used by the partitioner in computing the inter-block dependencies. When a cluster is partitioned, all the rectangles in that cluster are horizontally projected to the diagonal, just before they are partitioned into atomic blocks. The horizontal projection of each rectangle finds a *match* with one or more clusters. A cluster $C_{m,n}$ *matches* a horizontal projection if the projection interval overlaps the interval $[m,n]$. For example, in Figure 15, the cluster C matches the projection $HP(P_1)$, and the matching interval is $[c_1, I_1']$. When a rectangle is projected, the matching interval, along with the projecting rectangle's row and column extents, is stored in the matching cluster(s).

If the cluster being processed is a single column, each of its nonzeros is horizontally projected. The row number of the nonzero element along with the column number is stored in the matching cluster(s).

Referring to Figure 15, suppose we want to compute the dependencies for the triangular block T_{c_1,c_2}. $HP(P_1)$ is $[I_1, I_1']$ and $VP(T_{c_1,c_2})$ is $[c_1, c_2]$. Since $HP(P_1) \bigcap VP(T_{c_1,c_2})$ is $[c_1, c_2]$, P_1 updates block $T_{c1,c2}$. The intersection also gives complete information on the portions of a block involved in a particular update. In above case, the upper shaded extent of P_1 between rows c_1 and c_2 updates all of $T_{c1,c2}$. Now consider block R_2. Since $VP(R_2)$ is $[c_2, c_3]$ and $HP(P_1)$ is $[I_1, I_1']$, $VP(R_2) \bigcap HP(P_1)$ is $[c_2, I_1']$. Further, $VP(M_1)$ matches $VP(P_1)$ and $HP(M_1)$ intersects $HP(R_2)$. Thus, the lower shaded portion of the base P_1, enclosed between rows c_2 and I_1', and all of the multiplier M_1 are used in updating the shaded portion of R_2. Figure 15 also shows R_5 being updated by the combination of the upper shaded portion of P_2 and the whole of M_2 and R_6 being updated by the lower shaded portion of P_2 and the whole of M_2. All such inter-block dependencies are determined using the method of projection described above and from this the BDAG is constructed.

6.4 Partitioner schematic

So far we have described the individual components of the partitioner. In Figure 16, we present a schematic view of the functioning of the entire partitioner. The lower triangular symbolic factor, with clusters and columns identified by the pre-partitioner, is the input to the partitioner. The partitioner sweeps left to right across this input matrix, processing the columns and clusters as they are encountered in left to right order. When a column is processed, the nonzeros in the column are projected to the diagonal and stored in the matching clusters and columns. Next, the dependencies for the column due to blocks to the left of the column, are computed. The dependencies are computed using the previously stored projections due to the columns and rectangles to the left of the column being processed. When a cluster is processed, the off-diagonal rectangular blocks are projected to the diagonal and stored in the matching clusters and columns. The triangles and rectangles are then partitioned into atomic blocks using the values of parameters T and R. The dependencies for each of atomic block in the cluster are then determined. This is done by first building a data structure called the interval tree [27]. The tree is used to store the projections

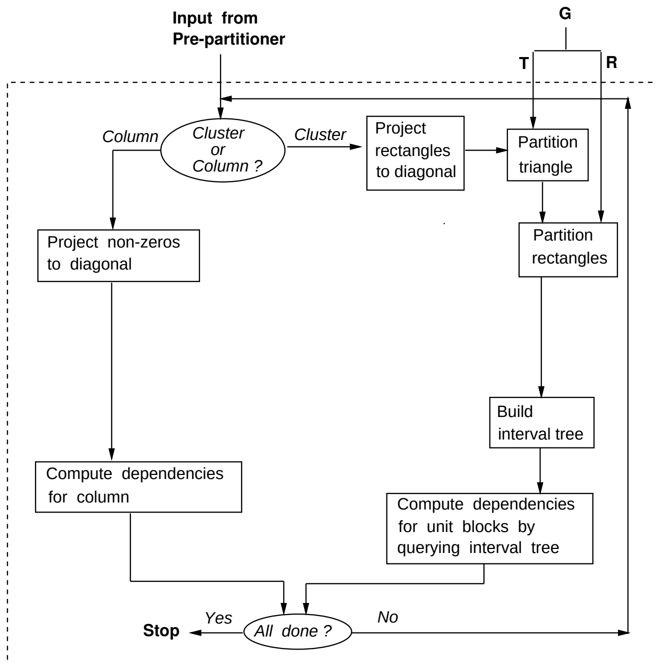


FIGURE 16
Partitioner schematic.

from the columns and rectangles that are to the left of the cluster. Dependencies for the blocks are computed efficiently by making queries on the interval tree. The computed inter-block dependencies are stored in the block dependency BDAG. All the algorithms for the partitioner appear in [37].

6.5 Parallelizing the partitioner

The inter-block dependencies generated by the partitioner and used in the numerical factorization, represent a huge amount of data (generally, polynomial in the number of blocks). Although all the computations performed by the partitioner are on the symbolic factor L, for large sparse matrices these computations are significant and if performed on a single processors, the computation time is comparable to the parallel numerical factorization performed on tens of processors. Furthermore, with the limited memory available on each node of iPSC/860 (8 MB, in our case), one is forced to use secondary storage for storing the inter-block dependencies, if the partitioner is to be run on a single node. This slows down the partitioning step considerably and is a potential bottleneck in parallel factorization. To avoid this bottleneck, we have parallelized memory intensive steps (particularly, the determination of inter-block dependencies) in the partitioner. This also allows us to take advantage of all the available resources including memory of a distributed system in a scalable fashion. Some of the steps which have limited parallelism and which are relatively insignificant, are repeated on every processor as sequential steps since, at this stage, the thrust of our parallel implementation is not towards getting good speed-ups.

The salient points of the parallel partitioner are as follows. The partitioner is parallelized in an SPMD style. The single columns and the clusters identified by the pre-partitioner are input to the parallel partitioner. Each processor is assigned a subset of these single columns and clusters, in an equitable fashion. A processor is responsible for determining the inter-block dependencies for the

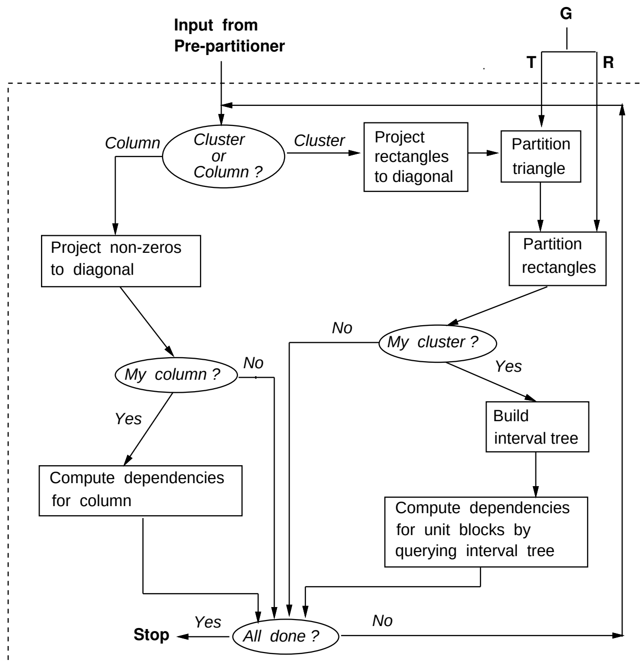


FIGURE 17
Schematic of the parallel partitioner

single columns and for all the blocks in the clusters assigned to it. Each processor scans the entire input data set while partitioning and projecting the clusters and single columns, and computes the dependencies for the assigned columns and clusters. This is shown schematically in Figure 17. Note that each processor computes the dependencies for the clusters and columns assigned to it in parallel with the other processors. As a result, the BDAG is computed and stored in a distributed form. The work in dividing the nonzero regions in rectangular and triangular blocks and computing their projections, however, is sequentialized. So far our experience is that this sequential part is relatively insignificant. However, in the future, we plan on parallelizing even this part. A feature of the parallel partitioner is that the number of processors need not be the same as the number of processors on which the numerical factorization is to be performed.

7 Allocation and scheduling

In this section, we briefly describe the allocation and scheduling procedures used in SHAPE. For details, see [37]. By allocation we mean the assignment of blocks to processors, and by scheduling we mean the ordering of computational tasks and communication within each processor. In the numerical factorization, we use the "owner computes" rule for scheduling the computations, which means that all updates to a block, including scaling, are computed in the processor to which that block is assigned.

7.1 Allocation

The clusters are sorted by workload in a decreasing order and then are picked for allocation in this order. If the cluster is a single column, then it is assigned to the processor with current minimum work load. If a cluster is not a single column, the unit blocks within the cluster are assigned to processors, with the aim of balancing the load among the processors. The allocator makes use of the cluster DAG (CDAG), which is a dependency graph that captures the inter-cluster dependencies. A cluster is dependent on another if any block in the first cluster is dependent on any block in the second cluster.

The output of the allocator is a one-to-one mapping of blocks to processors. In the current version of SHAPE, the allocator is a sequential program and runs on a single node of the iPSC/860.

7.2 Scheduling

The scheduler is a parallel program which takes the BDAG and the mapping of blocks to processors as input. It is executed on the same number of processors of the same parallel system as the one on which the sparse matrix is to be factored. It generates, as output, a task schedule for each processor, as well as a complete communication profile for the task schedule. The profile consists of the order and the sizes of blocks to be fetched and received, the processors to which each block is to be sent and other information which is useful in reducing the communication overheads during the numerical factorization. The scheduler can be run in one of two modes. In the first mode, the scheduler traverses the block dependency DAG *without actually executing the numerical computations*, thereby doing *symbolic scheduling.* In the second and alternative mode, the scheduler computes one iteration of the numerical factorization and in the process determines a schedule that can be used in the subsequent iterations.

The scheduler reports a task schedule to be used on each processor by traversing the BDAG. Tasks are symbolically executed using a dynamic scheduling policy that is guided by the data being made available by the locally computed tasks as well as by the data received from other processors. Priorities are assigned to executable tasks in such a way that remote tasks do not starve and at the same time local communication buffers do not overflow. During the actual numerical factorization, the same schedule is assumed so that the tasks are executed in a pre-determined order. This avoids the overheads associated with dynamic scheduling in the numerical factorization step. Besides, with a priori knowledge of the spatial and temporal communication pattern, the communication cost can be reduced by aggressive optimization during the numerical factorization. In the environment provided by SHAPE, the details of scheduling and communication optimization is made transparent to the user by means of the primitives described in were explained in Sections 3.3.1 and 3.3.2.

8 Implementation of *bmod* and *bdiv* tasks

In distributed block sparse Cholesky factorization, when the matrix is partitioned into blocks as described in earlier sections, a block may be operated upon by one or more *bmod* and *bdiv* tasks (see Figure 6(b)). In each such operation where a block may be updated or scaled by other blocks, the block shape may be triangular, rectangular, or it may simply be a single sparse column. For computational efficiency, it is necessary to take into account the combined effect of different types of blocks involved in a particular task. As a result, a single generic implementation of *bmod* or *bdiv* task is not sufficient. Instead we accomplish this using a small number of efficient kernels. In the following we describe the key concepts and some of the implementation details of these kernels.

First we note that the atomic blocks may combine in nine different ways in a *bmod* task and in three different ways in a *bdiv* task. The twelve operations are summarized in Table 1 using a tuple

No.	Description	Tuple Notation	Name
1	Column Updates Column	$< Col, Update, Col >$	CC
2	Column Updates Triangle	$< Tri, Update, Col >$	CT
3	Rectangle Updates Column	$< Col, Update, Rect >$	RC
4	Rectangle Updates Triangle	$< Tri, Update, Rect >$	RT
5	Column Updates Rectangle	$< Rect, Update, Col >$	CR
6	Triangle and Rectangle Update Rectangle	$< Rect, Update, Tri, Rect >$	TRR
7	Two Rectangles Update Column	$< Col, Update, Rect, Rect >$	RRC
8	Two Rectangles Update Triangle	$< Tri, Update, Rect, Rect >$	RRT
9	Two Rectangles Update Rectangle	$< Rect, Update, Rect, Rect >$	RRR
10	Triangle Scales Rectangle	$< Rect, Scale, Tri >$	TR
11	Column Scales Column	$< Col, Scale, Col >$	SC
12	Triangle Scales Triangle	$< Tri, Scale, Tri >$	ST

TABLE 1

Types of atomic block update and scale operations.

notation $< \mathcal{P}, Oper, \mathcal{P}, [\mathcal{P}] >$, where $\mathcal{P}$ defines a block partition in L, and $Oper$ is either a block update or a block scale operation; $[\mathcal{P}]$ indicates that it is an optional operand. The partitions are identified by their shape: Col, Tri, and $Rect$ indicate a column, a triangle, and a rectangle, respectively. The same physical partition may be used as a value of two partition parameters in a single tuple. An acronym for each of type of operation is given in the last column. In the rest of the paper, we will refer to these operations by their respective acronyms. Note that CC and SC are functionally the same as *cmod* and *cdiv* operations, respectively. In the tuple representation of SC, the two column partition arguments refer to one and the same column. Similarly, in the case of ST the two triangles refer to the same partition. In this case, elements in a triangular block are updated and scaled by other elements in the same block. Nine block update and scale operations, are shown schematically in Figure 18. Not shown are CC, SC, and ST operations, which are relatively straightforward. In that figure, only the portions in the lower triangular part of L are shown. A block consisting of a single sparse column is shown by a solid vertical line; R (with suffixes) and T denote triangular and rectangular blocks, respectively. Note that not all elements in a block may participate in a particular operation. In Figure 18, the shaded area in each block represents the portion of elements that participate in a particular operation. To facilitate efficient identification of such portions of elements in involved in a particular operation, we take special

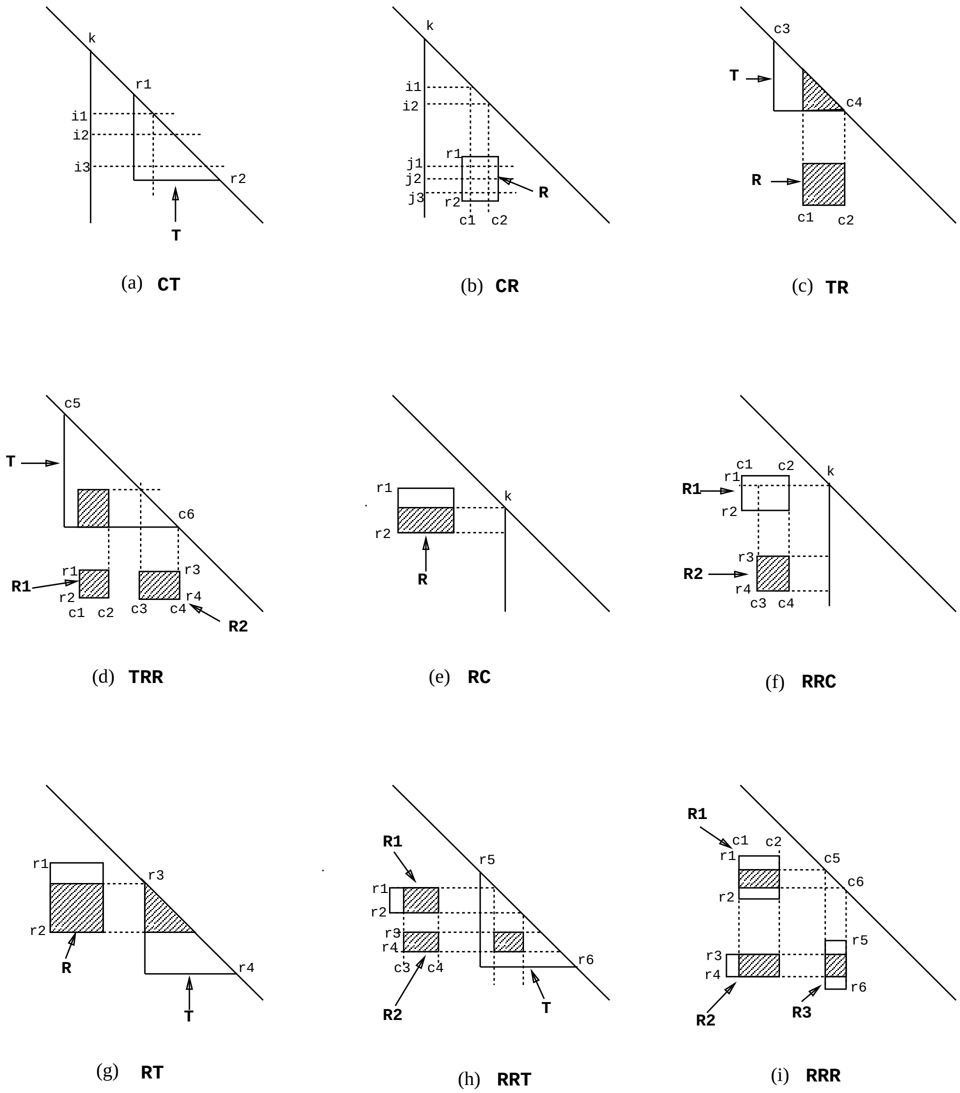


FIGURE 18

Schematic representation of nine types of block update and scale operations.

steps when the inter-block dependencies are determined (see Section 6.3).

Using the above classification of the block update and scale operations, we have developed a set of kernels that can be optimized for an individual architecture. These kernels are incorporated in the implementation of the primitives described earlier. We now describe some of the salient points of the kernels for RRR, RT, ST, and TR operations. We observe from Figure 18 that the operations of type CC, CR, TRR, RRC, and RRT can be constructed from the RRR type operation; CT and RC can be constructed from the RT type operation; and SC from ST and TR type operations (note that a sparse column is a degenerate rectangular block). However, in our implementations, all twelve kernels are implemented separately to take advantage of the special cases.

First consider the RRR type operation which captures the essence of all block update operations. In Figure 18(i), the rectangular block R3 is updated by rectangular blocks R1 and R2. Since only the elements in the shaded portions of the blocks are involved, we note that in this RRR type operation, block $R_{r3,r4,c5,c6}$ is updated by blocks $R_{r3,r4,c1,c2}$ and $R_{r1,r2,c1,c2}$. Using matrix notation, this is the same as,

$$R_{r3,r4,c5,c6} \leftarrow R_{r3,r4,c5,c6} - R_{r3,r4,c1,c2} R^T_{r3,r4,c1,c2},$$

where $R^T_{r3,r4,c1,c2}$ is transpose of block $R_{r3,r4,c1,c2}$. Similarly, in the RT type of operation shown in Figure 18(g), the triangular block T is updated by the rectangular block R. Taking into account only the elements that participate in this operation, we observe that, in the RT operation, block D is updated by block $R_{r1,r2,c1,c2}$, where block D represents the shaded region of $T_{r3,r2}$. This update can be represented by,

$$D \leftarrow D - R_{r1,r2,c1,c2} R^T_{r1,r2,c1,c2}.$$

In this operation, only the lower triangular part of D are of interest.

The block scaling operation TR is the block analog of the scaling operation of an element by the corresponding diagonal element. Consider the case shown in Figure 18(c), where the triangular block T scales the rectangular block R. Let block D be the lower triangular block at the diagonal, representing the shaded area in triangle $T_{c3,c4}$. The scaling of block $R_{r1,r2,c1,c2}$ can be written in the matrix notation as,

$$R_{r1,r2,c1,c2} \leftarrow R_{r1,r2,c1,c2} \left(D^T\right)^{-1}.$$

Note that prior to scaling by block D, the scaling block at the diagonal itself must have been updated and scaled. The triangular block T is itself scaled by an ST type operation, after all the updates are completed. This operation is again analogous to scaling of the diagonal element. Let D' represent the lower triangular block at $T_{c3,c4}$ in Figure 18(c), before it has been scaled and D be the same block after being scaled. Then block scaling operation ST can be represented by,

$$D \leftarrow D' \left(X^T\right)^{-1},$$

where X is such that $D' = X^2$. (X exists for positive definite matrices; for more details and relevant references, see [15]). For brevity, in the above, we have skipped the details and the proofs. However, these results can easily be verified using elementary block-matrix algebra.

An outline of the actual implementations of seven operations are shown in Figure 19. In that figure, block R is being updated or scaled by one or more blocks. In Figure 19(a), block R is being updated by blocks R' and R''. This is representative of the computations in RRR, TRR, RRT, RC and RRC operations. (For the RC operation, blocks R' and R'' refer to the same block.) Column extent of R and the row extent of R'' are h through h'. (For both RC and RRC operations, h equals h'.) Row extents of R and of R' are r through r'. Column extents of R' and of R'' are w through w'. In all cases, we assume a row-major data access for the blocks. (Refer also to Figure 12(b)). The variables $\mathbf{x}$ and $\mathbf{y}$ are vectors holding a row or a column from a block and the procedure *upd*, referred to in that figure, is the same as that shown as the vector code in Figure 13(b). The computations involved in RT and TR operations are shown by the kernels in Figures 19(b) and 19(c), respectively. The CC, CT, and CR type of operations are implemented as a collection of one or more *cmod* operations. Operation SC is implemented as *cdiv* operation and operation ST is implemented as a collection of *cmod* and *cdiv* operations. We note here that, for efficiency reasons, the actual implementations of the twelve operations are somewhat different from the conceptual descriptions presented above.

for $s = h$ **to** h' **do**	**for** $s = r$ **to** r' **do**	**for** $s = h$ **to** h' **do**
$\mathbf{x} \leftarrow R''_{s,s,w,w'}$	$\mathbf{x} \leftarrow R'_{s,s,w,w'}$	$\mathbf{x} \leftarrow R'_{s,s,h,s-1}$
$\mathbf{y} \leftarrow R_{r,r',s,s}$	$\mathbf{y} \leftarrow R_{s,h',s,s}$	$\mathbf{y} \leftarrow R_{r,r',s,s}$
call *upd*($\mathbf{x}$, $\mathbf{y}$, $R'_{r,r',w,w'}$)	call *upd*($\mathbf{x}$, $\mathbf{y}$, $R'_{s,r',w,w'}$)	call *upd*($\mathbf{x}$, $\mathbf{y}$, $R_{r,r',h,s-1}$)
		$v \leftarrow \sqrt{R'_{s,s,s,s}}$
		$scale(R_{r,r',s,s}, v)$
endfor	**endfor**	**endfor**
(a)	(b)	(c)

FIGURE 19
Compute kernels for
(a) RRR, RRT, TRR, RRC, and RC operations, (b) RT operation, and (c) TR operation.

9 Performance Results

In this section, we present timing results for parallel block sparse Cholesky factorization using the static task schedule generated by the scheduler. We compare these with the timings obtained from a column-based fan-out code with columns wrap-mapped to processors. We compare the scalability of the block and column methods by considering the total number of messages and total data volume. We also present timings for the preprocessing step of partitioning and scheduling in SHAPE, and relate these to the factorization time. All programs used in these experiments were written in C language using single precision arithmetic. The routine *sdot* from Kuck math library [21] was used in evaluating dot products. All the performance results are obtained from implementations and experiments carried out on the iPSC/860 at ICASE, NASA Langley Research Center. Each node of this multiprocessor system is a 40MHz i860 processor and has 8MB of main memory.

9.1 Test Matrices

The test matrices were chosen from a variety of applications. A brief description of each test case appears in Table 2. The first three cases are from the Harwell-Boeing sparse matrix collection [6]. These matrices are highly unstructured. For brevity, these matrices will also be referred to as B15, B16, and B17, respectively, and collectively, as B* matrices. The 7-point Laplacian discretizations are for structured 3-D grid problems, and the last test case is a structured 2-D grid problem. These cases will be referred to as L20, L25, and L127, respectively, and collectively, as L* matrices. The entry "nonzeros in factor" is the number of nonzeros in the lower triangular factor including the elements on the diagonal. The *Opcount* columns list the total number of floating point arithmetic operations in the numerical factorization. This is obtained by assuming the following: an add and multiply are one flop each, a reciprocal is three flops, a square root and a divide are four flops each [18].

In performing the experiments, two reordering schemes were used. These were: the recursive auto-

Matrix	Description	Order	Nonzeros in factor	Opcount
BCSSTK15	Module of an offshore problem	3,948	637,680	157,421,732
BCSSTK16	Corps of Engineers dam	4,884	749,027	153,556,767
BCSSTK17	Elevated Pressure Vessel	10,974	1,020,970	147,899,872
L7PT20	7-point Laplacian discretization ($20 \times 20 \times 20$ grid)	8,000	646,839	135,896,617
L7PT25	7-point Laplacian discretization ($25 \times 25 \times 25$ grid)	15,625	1,719,292	556,052,830
L9PT127	9-point Laplacian discretization (127×127 grid)	16,129	563,345	42,675,065

TABLE 2
Test matrices.

matic nested dissection (AND) scheme [14] and Liu's multiple minimum degree ordering (MMD) scheme [22]. In the following, unless otherwise specified, the B* matrices were reordered using MMD and the L* matrices were reordered using AND. We used SPARSKIT [32] and the Wisconsin Sparse Matrix Manipulation System [1] for generating and converting the test matrices into various formats, and for ordering and symbolically factoring most of the test matrices. For the biggest matrix, L25, we used our own parallel symbolic factorization code running on the iPSC/860, since the Wisconsin Sparse Matrix Manipulation System could not handle matrices of size L25. In all tests cases, the partitioning parameters were fixed as follows. W, the minimum acceptable cluster width was set to 5; T, the triangle grain size, was set to 400; and R, the rectangle grain size, was set to 800. For a discussion on the performance effects of these parameters, refer to [39].

9.2 Numerical Factorization

In this section, we present the parallel factorization timings for the test cases described above. To understand the costs and benefits of parallelization using SHAPE, we also discuss the costs associated with the main components of parallel factorization.

The parallel factorization time is made up of four components: the *computation* time, the *communication* time, the *idle* time, and the *overhead* time. The computation time is the time spent by an individual processor in performing floating point operations that lead directly to the computation of the Cholesky factor. On any given processor, the computation time is a function of the total number of useful floating operations assigned to that processor, the processor speed, and the effect of the data layout in utilizing the cache and the floating point unit. The communication time is the time spent in assembling and disassembling the messages, time spent in the data movement between memory and message buffers, and any system overheads (e.g., context switching) related to message passing. Other than the architectural characteristics, the communication time is a function of the number of messages sent/received and the volume of data exchanged in satisfying the inter-processor task dependencies, which in turn are directly affected by the data partitioning strategies used. The idle time results from load imbalance among processors. The load imbalance may be because of uneven distribution of work and/or because of the dependencies that lead to processor starvation. The sparsity structure of the matrix as well as the the task scheduling strategies applied affect the idle time. Finally, the overhead time is the time spent in the bookkeeping activities which are significant in sparse matrix computations. The nature of the data structures used in the sparse matrix computations invariably lead to bookkeeping overheads in performing computations and communications, as well in scheduling of these activities. The overhead time is reduced by using the scheduler generated static task list and by posting messages to be received prior to entering a parallel computation phase. Overhead time is also reduced by avoiding indirect referencing whenever possible.

Matrix	P	Fact	Comp	Comm	Ovhd	Idle	Mflops	Comp Mflops
B15	2	12541	9803	1189	894	655	12.6	16.1
	4	8342	5032	1989	428	893	18.9	31.3
	8	6251	2432	2110	236	1473	25.2	64.7
	16	5343	1308	1960	158	1917	29.5	120.4
	32	4443	667	1391	88	2297	35.4	236.0
B16	2	10740	8753	330	895	762	14.3	17.5
	4	7291	5058	653	613	967	21.1	30.4
	8	5728	2901	969	557	1301	26.8	52.9
	16	4499	1346	763	321	2069	34.1	114.1
	32	3814	668	557	113	2476	40.3	229.9
B17	4	12233	6009	3695	867	1662	12.1	24.6
	8	8963	2903	3404	451	2205	16.5	50.9
	16	6649	1561	2365	239	2484	22.2	94.7
	32	4879	731	1297	144	2707	30.3	202.3
L20	4	9932	4666	3191	1309	766	13.7	29.1
	8	7741	2438	2884	793	1626	17.6	55.7
	16	5143	1226	2150	416	1351	26.4	110.8
	32	4089	574	1099	166	2250	33.2	236.8
L25	16	15572	3716	5587	979	5290	35.7	149.6
	32	10485	1863	3346	463	4813	53.0	298.5
L127	2	10876	5231	3046	2321	278	3.9	8.2
	4	9188	2816	3962	1430	980	4.6	15.2
	8	6262	1457	2912	834	1059	6.8	29.3
	16	3771	738	1722	418	893	11.3	57.8
	32	2664	369	944	230	1121	16.0	115.7

TABLE 3
Times, in milliseconds, for block factorization (no communication optimizations).

Table 3 gives the execution times for factoring the test matrices. In these experiments, no communication optimizations were performed. (We defer this to the following section.) All timings are in milliseconds, and are measured for the processor which finished executing last. We refer to this processor as the *last-to-finish* or simply the *last* processor. In Table 3, *Fact* refers to the total parallel factorization time, *Comp* is the time spent by the last processor in arithmetic computations, *Comm* is the time spent by the last processor in communication, *Ovhd* is the time spent by the last processor in book-keeping activities associated with the scheduling of tasks, *Idle* is the idle time for the last processor. Thus, *Fact* is the sum of *Comp*, *Comm*, *Ovhd*, and *Idle*. *Mflops* is the total delivered mflops and *Comp Mflops* is the delivered mflops taking only the arithmetic computation for the numerical factorization into account. For any test matrix, $Mflops = Opcount/Fact$ and $Comp\ Mflops = Opcount/Comp$, scaled down to units of millions of floating point operations. Note from Table 3 that, in all cases, idle time is the dominant cost when 32 processors are used and, for the unstructured matrices, it is dominant even with 16 processors. This indicates that for these problems there is not enough parallelism when 16 and 32 processors are used. This reflects clearly in the total Mflops delivered.

9.2.1 Communication optimization

The communication profile generated by the scheduler may be used to optimize the communication during the numerical factorization. In [40], we have shown that for the column-based partitioning and scheduling, significant reduction in communication is possible by using force type messages on the iPSC/860. In that paper, we also discussed the limitations of the preliminary approach we used at that time. Recently, we have designed and implemented a scalable algorithm for communication optimization using force type messages, for the block-based factorization. We present preliminary performance results for the algorithm on some of the test cases and show the reduction in communication cost.

Matrix	P	Fact	Opt Comm	Idle	Unopt Comm	I_{comm} (%)	Mflops	Mf_{gain} (%)
B15	2	10879	312	415	1189	73.7	14.5	15.1
	4	7365	544	1035	1989	72.6	21.4	13.2
	8	5407	744	1722	2110	64.7	29.1	15.5
	16	4106	869	1525	1960	55.7	38.3	29.8
	32	3158	531	1763	1391	61.8	49.8	40.7
B17	4	9317	1090	1426	3695	70.5	15.9	31.4
	8	6508	1179	1758	3404	65.4	22.7	37.6
	16	4940	938	2127	2365	60.3	29.9	34.7
	32	4103	537	2582	1297	58.6	36.0	18.8
L20	4	7962	1175	815	3191	63.2	17.1	24.8
	8	6097	1302	1129	2884	54.8	22.3	26.7
	16	4291	999	1321	2150	53.3	31.7	20.1
	32	3523	530	2181	1099	51.8	38.6	16.3
L127	2	8535	1227	711	3046	59.7	5.0	28.2
	4	6407	1315	997	3962	66.8	6.7	45.7
	8	4494	1008	1083	2912	65.4	9.5	39.7
	16	3114	710	1124	1722	58.8	13.7	21.2
	32	2224	396	1103	944	58.1	19.2	20.0

TABLE 4
Times, in milliseconds, for block factorization (optimized communication).

Table 4 gives the timings for four test cases when communication is optimized. The new optimized communication timings are shown in the fourth column. The unoptimized communication timings from Table 3 are repeated in the sixth column. The percent reduction in the communication costs are shown in the column with heading I_{comm}, where,

$$I_{comm} = \frac{\text{Unopt Comm} - \text{Opt Comm}}{\text{Unopt comm}} \times 100.$$

Thus, the communication optimizations result in 50% to 70% reduction in communication costs. The communication optimization also helps in reducing the idle time, especially with 16 and 32 processors. There is little change in the computation and overhead times and we do not report those again. The total delivered Mflops with the optimized communication are shown in the next to last column of Table 4. Overall, the communication optimization results in 16% to 40% improvement in the delivered Mflops with 16 and 32 processors. This is indicated by Mf_{gain} in the table.

9.3 Comparison with column-wrap

To see the benefits of blocking, we now compare the block factorization code with a column-based factorization code. The later is derived from a distributed fan-out algorithm, with columns being wrap-mapped to processors. We will refer to this as the column-wrap code. To make a fair comparison, in column-wrap case, a parallel column-based scheduler is executed prior to the numerical factorization to generate a static, completely ordered task list for each processor. This static task schedule is then used by the column-wrap factorization code. This type of static pre-scheduling reduces run-time scheduling overheads [40]. Shown in Table 5 are the factorization timings for the test matrices. All timings are in milliseconds and are for the last-to-finish processor. For each matrix, *Fact* is the parallel factorization time, *Comp + Ovhd* is the time spent by the last-to-finish processor in arithmetic computations and overheads associated with scheduling these computations, *Comm* is the time spent by the last processor in communication, and *Idle* is the idle time in the last processor. *Mflops* is the total delivered mflops. It is computed as $Opcount/Fact$, scaled down to units of millions of floating point operations per second.

Comparing Tables 4 and 5, we see that the block code is three to five times as fast as the column-wrap code. Both computation time and communication time contribute significantly to this improvement. The improvement in computation time comes from the fact that the cache and floating-point-unit utilization capabilities of the block partitions are significantly better than that of column partitions, resulting in much higher computational rates for the block-based tasks [39]. The communication performance of the block code is also substantially superior to that observed with the column-wrap code. In fact, the reduction in the communication costs is one of the key benefits of using the block method. These cost reductions are because of reductions in the total number of messages as well as in the total volume of data communicated. We elaborate on this in the next sub-section.

9.4 Communication Costs: Block vs. Column

To get an insight into the reduced communication costs of the block method, we compare the total number of messages sent and the total volume of data transmitted in the block method with the corresponding quantities in the column-wrap method. We use B15 and L127 as representative test cases.

Matrix	P	Fact	Comp + Ovhd	Comm	Idle	Mflops
B15	4	36523	32148	3477	898	4.3
	8	21286	16916	3802	568	7.4
	16	13710	8687	4238	785	11.5
	32	10148	4620	4320	1208	15.5
B16	4	35780	30398	4334	1048	4.3
	8	21430	15870	4777	783	7.2
	16	14940	8294	5189	1457	10.3
	32	12829	4469	5300	3060	12.0
B17	8	25324	15878	7474	1972	5.8
	16	19068	8575	7944	2549	7.8
	32	16877	4609	7856	4412	8.8
L20	4	32255	26875	3905	1475	4.2
	8	19340	14148	4202	990	7.0
	16	12638	7475	4259	904	10.8
	32	9637	3844	4203	1590	14.1
L25	16	41836	29042	10415	2379	13.3
	32	28999	14815	10478	3706	19.2
L127	4	17823	9897	5281	2645	2.4
	8	13402	5359	5379	2664	3.2
	16	9900	2876	4942	2082	4.3
	32	7370	1551	4269	1550	5.8

TABLE 5
Times, in milliseconds, for column-wrap

Figures 20(a) and 21(a) compare the total number of messages for B15 and L127, respectively. Figures 20(b) and 21(b) compare the total volume of data communicated for the two test cases. In each case, results are plotted for 4, 8, 16, and 32 processors. The trends shown here are representative of all structured and unstructured matrices we have studied. To show the effect of blocking on the total number of messages, we have segregated the messages corresponding to the triangular and rectangular blocks from those corresponding to the columns. The curves marked "Block partition" in Figures 20(a) and 21(a) represent the total number of messages in communicating columns, rectangles, and triangles. The "Blocks only" curves exclude the messages that communicate single column blocks. We observe the following from the two figures. First of all, in the column-wrap method the both the number of messages and the total volume of data increase almost linearly with the number of processors. On the other hand, for the block case, the growth rates are much smaller. With large number of processors, the differences are almost an order of magnitude. This observation supports our analytic results in [38] where we show that, for the block method, the total number of messages and total volume of data transmitted are an order of magnitude less than those in the column-based methods. This result is asymptotic in the number of processors. The next important observation to make here is that in the block methods, most of the messages are associated with single column blocks. Thus, more the rectangles and triangles in the partitioning, the lesser are the number of messages transmitted. Finally, note that the communication results shown here have direct bearings on the scalability of the block and column methods.

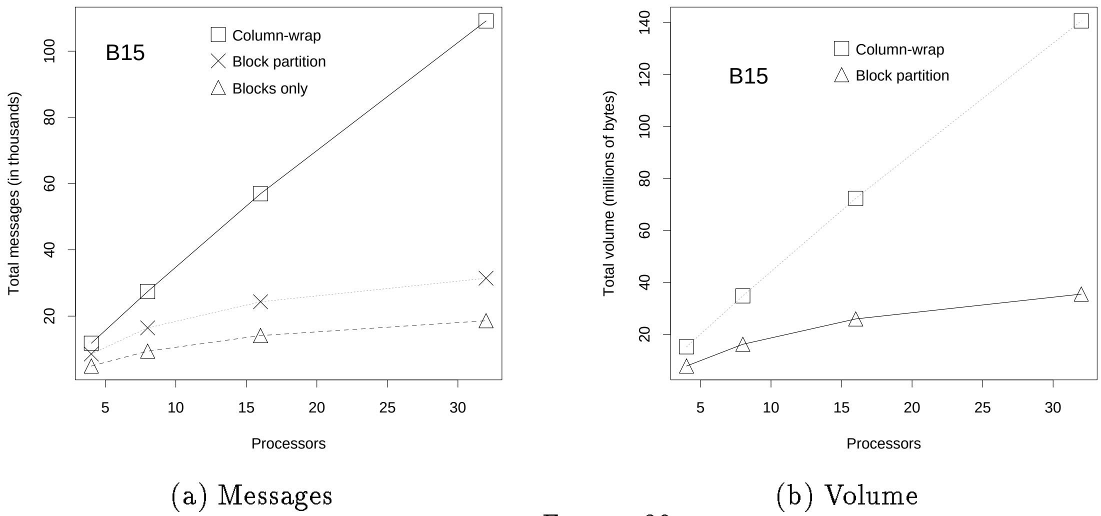


FIGURE 20
Communication growth rates: B15.

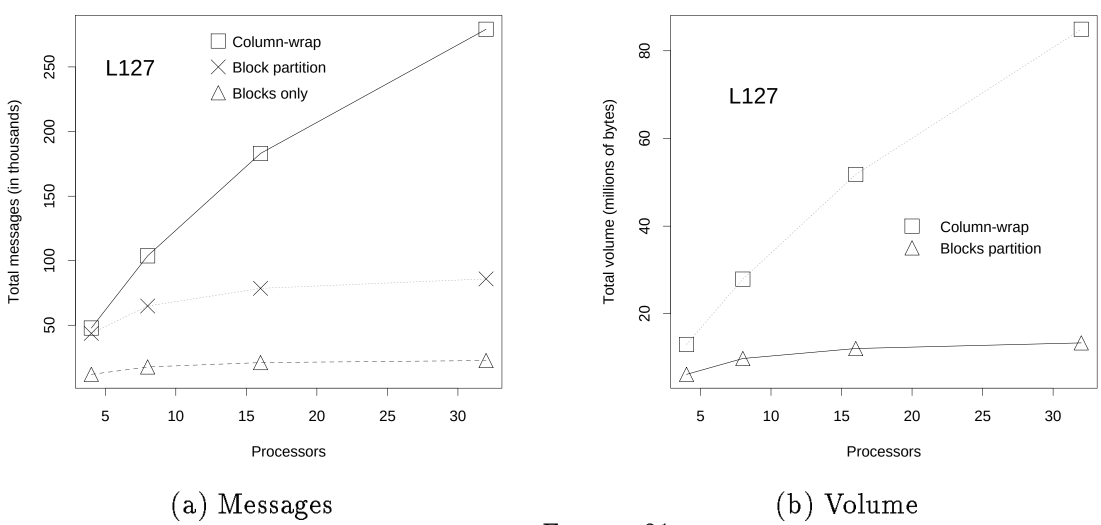


FIGURE 21
Communication growth rates: L127.

9.5 Preprocessing costs of SHAPE

In using SHAPE, extra preprocessing has to be performed. Here we quantify these costs. The overall preprocessing time is the sum of the individual times for pre-partitioning, partitioning, allocation and scheduling. The partitioner and scheduler are both parallel programs. The pre-partitioner and allocator are sequential programs and the timings for these are from the runs on a single processor of the iPSC/860. All the timings presented in this section are in seconds and do not include the IO cost.

Matrix	Partitioner						Scheduler				
	1	2	4	8	16	32	2	4	8	16	32
BCS15	9.2	5.2	4.7	5.6	5.1	5.1	4.7	4.5	4.4	4.3	4.0
BCS16	8.5	5.6	3.8	3.4	3.8	3.0	5.1	5.5	5.4	5.1	5.0
BCS17	12.5	9.4	6.8	5.1	5.2	4.8	-	7.8	6.9	5.7	4.5
L7PT20	25.1	14.9	21.2	15.2	13.6	11.4	-	8.9	7.0	5.1	4.0
L7PT25	-	-	-	14.3	9.4	8.0	-	-	-	13.7	10.5
L9PT127	17.7	10.6	7.2	5.5	5.0	4.2	11.3	9.1	7.0	4.8	2.9

TABLE 6
Partitioner and scheduler timings, in seconds.

Matrix	Clusters	Col	Tri	Rect	Total	Deps
B15	112	1,337	257	1,600	3,194	80,160
B16	156	1,319	473	1,667	3,459	80,354
B17	538	3,413	778	3,701	7,892	105,918
L20	291	3,917	576	6,460	10,953	256,475
L25	562	7,410	852	13,914	22,176	646,545
L127	678	8,580	777	6,648	16,005	182,167

TABLE 7
Statistics on block partitions

Table 6 shows the timings for the parallel partitioner and parallel scheduler (using the timings of the last-to-finish processor). Timings for the pre-partitioner and allocator are not shown here. Together, they contribute less than 10% of the parallel time required for partitioning and scheduling on a single processor. A blank entry in the table means that the corresponding test could not be run due to the lack of sufficient memory on that many processors. We first analyze the partitioner timings. The time required by the partitioner is mainly determined by the number of dependencies reported. To see this empirically, consider Table 7 where we show the size of each test case in terms of the number of clusters, number of columns, number of dense triangles, number of dense rectangles, total number of blocks (sum of columns and dense blocks), and number of dependency arcs output by the partitioner. A dependency arc from block B_i to block B_j exists if and only if B_i is used to modify B_j. From Tables 6 and 7, it is clear that on a single processor, the partitioning time is proportional to the number of dependency arcs reported by the partitioner. Also, the number of dependency arcs is an order of magnitude greater than the number of blocks, for all the test matrices. Thus, construction of dependencies is the major work performed by the partitioner. As stated in Section 6.5, the partitioner has been parallelized mainly to distribute data so as to take advantage of the available memory. The emphasis is not on efficient computational parallelization. Columns and clusters are allocated in an equitable manner to processors for partitioning, but this does not imply an equitable distribution of the computational work involved in determining the inter-block dependencies. Table 6, therefore, does not show good speed-up figures for the partitioner.

Next, we analyze the scheduler timings, which are also shown in Table 6. As described in Section 7.2, the scheduler is a tool which does not use any knowledge of the problem, and can generate a static schedule (symbolically) for any algorithm, so long as the dependencies among the tasks are known. In SHAPE itself, it is possible to completely detach the numerical factorization part from the scheduler. In our experiments, however, during the first iteration of the numerical factorization the scheduler was used to determine the task and communication schedule, i.e., the scheduler was run in the non-symbolic mode. The scheduler timings shown in Table 6, have been obtained by subtracting the computation time (i.e., the *Comp* time) from the total factorization time (i.e., the *Fact* time) for the last-to-finish processor. This results in a slight over-estimation of the scheduler timings since the idle time included is higher during the numerical factorization than in a symbolic factorization. Nonetheless we report this timing as the cost of scheduling. Like the partitioner, the speed-ups for the parallel scheduler are poor, but not unexpected since the timings represent purely symbolic computations. Non-numerical computations, such as these, typically perform poorly on message-passing machines.

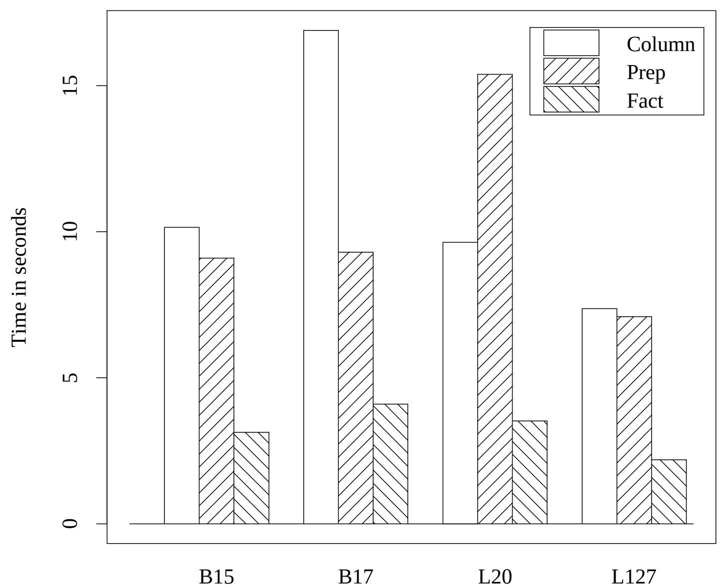


FIGURE 22
Iteration time relative to column-wrap, 32 processors.

We now compare the preprocessing cost of SHAPE with the subsequent per iteration cost for block factorization as well as with the per iteration cost in the column-wrap method. The preprocessing cost of SHAPE is the sum of the times for the pre-partitioner, the partitioner, the allocator and the scheduler. The per iteration cost is the cost of a single numerical factorization after all the preprocessing. Using bar charts, these costs are shown in Figure 22 for test cases B15, B17, L20, and L127.

The leftmost bar in each test case indicates the cost of a single factorization using the column-wrap method. In the column-wrap method, there is no preprocessing and thus, even if several numerical factorizations are to be performed using the same zero-nonzero structure, the cost of each factorization remains the same. The center bar represents the SHAPE preprocessing time. For each test case, the rightmost bar represents the time for a single factorization using the block numerical factorization method. Clearly, the combined cost of preprocessing and one iteration of factorization in the block method is less than the time spent in two iterations using the column-wrap factorization. Also, except for L20, one iteration of the numerical factorization in the block method is three to four times as fast as one iteration of the column method. Therefore, the time taken by the initial preprocessing followed by three iterations of the block method is less than the time spent by three iterations of the column-wrap method. After this point, the block method outperforms the column method in every iteration.

10 Conclusions

Sparse Cholesky factorization is an important kernel in a variety of engineering and optimization problems. It is a fitting representative of a class of applications in which the dependencies are unstructured and the communication patterns are not known until run-time. Moreover, activation of computations at execution-time is determined by the arrival of data on the processors, i.e. the computations are highly data-driven in nature. In this paper, we have presented a methodology for systematically parallelizing the unstructured computations of sparse Cholesky factorization on scalable architectures. The steps of our methodology are: *partitioning*, *allocation*, *scheduling of computation*, and *scheduling of communication*, followed by the actual numerical factorization. Using this methodology, we have built a scalable software system called SHAPE, for Sparse Hybrid Automatic Parallelization Environment. SHAPE is composed of a parallel partitioner, a parallel scheduler, a scalable communication optimization algorithm, and a run-time support system. These components are designed as modular tools that are tied together by an explicit representation for block-based unstructured data-driven computations. SHAPE can be used either as a stand-alone system, or as a run-time support tool made accessible to the user program via a set of primitives. In this paper, we have described the design of various components of SHAPE and have presented some of the implementation details with particular emphasis on the partitioner. To show the possible benefits of our methodology, we have presented performance data from experiments on a 32-node iPSC/860, using various structured and unstructured sparse matrices. In addition to explaining the preprocessing costs associated with SHAPE, we have also compared the performance of the resulting parallel block sparse factorization with a parallel column-based factorization method.

To summarize, (1) as a tool, SHAPE is flexible in partitioning a sparse matrix and offers sufficient parametric control so that the partitioning is sensitive both to the matrix sparsity structure and to the architectural granularity; (2) SHAPE is modular, so that separate investigation into different aspects of the problem such as partitioning, mapping, and scheduling may be pursued independently; (3) it offers a set of tools that alleviate the coding effort and hide the details of the architecture from the user; (4) it provides an environment for detailed investigation of parallel sparse Cholesky factorization.

The objectives of our implementations and of the experiments have primarily been to demonstrate the feasibility and practicality of our methodology and of a tool such as SHAPE for parallel block sparse factorization. Towards that end, we have been quite successful. From the point of view of absolute performance, when compared to a column-based method, our approach delivers significantly better performance at relatively low preprocessing cost. We are quite aware of the fact that there are other column-based methods (particularly, multifrontal methods) that are much more efficient than the one we have used for the purpose of comparison [35, 8]. We would like to point out that we have neither fine-tuned the implementation of SHAPE nor of the block numerical factorization for the iPSC/860. In fact, our objective here is not towards undertaking a benchmarking-like exercise, but to show the general benefits one may expect by adopting a block based parallel factorization approach and how that can be made practical without resorting to a complex programming effort. We have shown that (1) by using three simple primitives, simplicity in the parallel block factorization can be maintained without losing the scalability; (2) the communication costs are significantly reduced, both in terms of the number of messages communicated and the total volume of data transmitted; (3) the block data structures are naturally more amenable to better cache and floating-point-unit performance and thus, yielding better computational performance than a column-based method.

Currently we are working on improving the performance of SHAPE and of the block sparse Cholesky factorization algorithm on iPSC/860 as well as on other more recently available scalable systems. We are working on improving some of the functional aspects of SHAPE as well. The ordering of the sparse matrices and the preprocessing for sparse triangular solve are two efforts we would like to mention here. For the results presented in this paper, we have used two well-known ordering schemes: the multiple minimum degree (MMD) [22] and automatic nested dissection (AND) [14]. The choice of these ordering methods was based primarily on the availability of existing software rather than on the suitability of these ordering method for the parallel factorization of the test matrices. The MMD and AND schemes were developed to minimize the fill-in in the factorization so as to reduce the sequential computational work. While this is an important consideration in parallel factorization, it is equally important to use an ordering scheme that tends to increase the available parallelism by rearranging the unknowns so that the resulting elimination tree is shallow and spread-out rather than tall and skinny. In recent years, the aggressive effort in parallelizing sparse Cholesky factorization has resulted in renewed investigation for designing new ordering schemes which would generate more parallelism for factorization. However, this parallelism has to be traded off against a possible increase in the fill-in, and subsequently in the computational work. A good ordering scheme for parallel factorization would therefore be one that would create a significantly larger amount of added parallelism for factorization, without giving away too much additional computational work. Recent ordering algorithms include spectral nested dissection (SND) [30] and a Cartesian nested dissection (CND) [17]. The SND algorithm has been shown to incur fill-in that is quite close to MMD and much smaller than AND. It outperforms both MMD and AND with respect to parallelism offered by the elimination tree for factoring. The CND algorithm has been shown to incur fill-in that is somewhat worse than both MMD and AND. It also produces shorter elimination trees than MMD and AND. However, the performance of both SND and CND have been shown on only a few problems. Efficiency on a wider class of problems needs to be determined. Another question that needs to be investigated is the effect of ordering methods on the size of the blocks and ultimately on the efficiency of the dense block computations in the factorization. With SHAPE, it is now possible of us to conduct the extensive experimentations required to answer these questions. We are in the process of incorporating these newer ordering methods in SHAPE and hope to provide more insight to answer some of the questions.

The sparse triangular solves are just as important and just as difficult to efficiently parallelize as the sparse factorization. Although the dependencies are different, the methodology proposed here is applicable to the triangular solves as well. In fact, it is applicable to a wider class of problems including sparse matrix-vector multiplication. Currently, we are working on generalizing the partitioning and scheduling schemes embedded in SHAPE to handle the data-dependencies observed in sparse triangular solves and in sparse matrix-vector multiplications.

Acknowledgement: The authors would like to thank NASA Langley Research Center for allowing us the generous use of their 32 node iPSC/860. In particular, we are thankful to Bob Voigt and Yousuff Hussaini for their help in facilitating the use of this machine.

References

[1] F. L. Alvarado. The sparse matrix manipulation system users manual. Technical report, University of Wisconsin, Madison, Wisc., 1990.

[2] C. Ashcraft, S. Eisenstat, J. W. H. Liu, and A. Sherman. A comparison of three column-based distributed sparse factorization schemes. Technical Report YALEU/DCS/RR-810, Department of Computer Science, Yale University, New Haven, CT, 1990.

[3] C. Ashcraft, S. C. Eisenstat, and J. W. H. Liu. A fan-in algorithm for distributed sparse numerical factorization. *SIAM Journal of Scientific and Statistical Computing*, 11:593–599, 1990.

[4] C. Ashcraft, R. Grimes, J. Lewis, B. Peyton, and H. Simon. Progress in sparse matrix methods for large linear systems on vector supercomputers. *The International Journal of Supercomputing Applications*, 1:10–30, 1987.

[5] C. C. Chen and Y.H. Hu. A practical scheduling algorithm for parallel LU factorization in circuit simulation. *IEEE International Symposium on Circuits and Systems*, 3:1788–1791, 1989.

[6] I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Transactions on Mathematical Software*, 15:1–14, 1989.

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

[8] K. Eswar, P. Sadayappan, C. Huang, and V. Visvanathan. Supernodal sparse Cholesky factorization on distributed-memory multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, volume 3, pages 18–22, 1993.

[9] G. A. Geist and E. Ng. Task scheduling for parallel sparse Cholesky factorization. *International Journal of Parallel Programming*, 18-4:291–314, 1989.

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

[11] A. George, M. Heath, and J. W. H. Liu. Sparse Cholesky factorization on a local memory multiprocessor. *SIAM Journal of Scientific and Statistical Computing*, 9:327–340, 1988.

[12] A. George and D. McIntyre. On the application of the minimum degree algorithm to finite element systems. *SIAM J. Numerical Analysis*, 15:90–111, 1978.

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

[14] J. A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, Inc., Englewood Cliff, NJ, 1981.

[15] G. H. Golub and C. F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, Baltimore, MD, 1983.

[16] M. T. Heath, E. Ng, and B. W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33:420–460, 1991.

[17] M. T. Heath and P. Raghavan. A Cartesian parallel nested dissection algorithm. Technical Report UIUCDCS-R-92-1772, Department of Computer Science, University of Illinois at Urbana-Champaign, 1992.

[18] J. Helin and R. Berrendorf. Analyzing the performance of message passing MIMD hypercubes: A study with the Intel iPSC/860. In *Proceedings of the International Conference on Supercomputing*, pages 376–385, 1991.

[19] J. A. G. Jess and H. G. M. Kees. A data structure for parallel L/U decomposition. *IEEE Transactions on Computers*, C31:231–239, 1982.

[20] S. G. Kratzer. Massively parallel sparse-matrix computations. Technical Report SRC-TR-90-008, Supercomputing Research Center Institute for Defense Analyses, 1990.

[21] Kuck & Associates, Inc., Champaign, IL. *CLASSPACK Basic Math Library/C, User's Guide*, release 1.3 edition, 1992.

[22] J. W. H. Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 11:141–153, 1985.

[23] J. W. H. Liu. Computational models and task scheduling for parallel sparse Cholesky factorization. *Parallel Computing*, 3:327–342, 1986.

[24] J. W. H. Liu. Computational models and task scheduling for parallel sparse Cholesky factorization. *Parallel Computing*, 3:327–342, 1986.

[25] J. W. H. Liu. The role of elimination trees in sparse factorization. Technical Report CS-87-12, Department of Computer Science, York University, North York, Ontario, Canada, 1987.

[26] J. W. H. Liu. The multifrontal method for sparse matrix solution: Theory and practice. Technical Report CS-90-04, Department of Computer Science, York University, North York, Ontario, Canada, 1990.

[27] K. Mehlhorn. *Data Structures and Algorithms, Vol. 3 - Multidimensional Searching and Computational Geometry.* Springer-Verlag, 1984.

[28] V. K. Naik. *Multiprocessing: Trade-offs in Computation and Communication.* Kluwer Academic Publishers, Norwell, MA, 1993.

[29] V. K. Naik and M. L. Patrick. Data traffic reduction schemes for Cholesky factorization on asynchronous multiprocessor systems. In *Proceedings of the 1989 International Conference on Supercomputing*, pages 283–294, 1989. *Also available as* IBM Research Report RC 14500.

[30] A. Pothen, H. D. Simon, L. Wang, and S. T. Barnard. Towards a fast implementation of spectral nested dissection. In *Proceedings of Supercomputing '92*, pages 42–51, 1992.

[31] E. Rothberg and A. Gupta. An efficient block-oriented approach to parallel sparse Cholesky factorization. In *Proceedings of Supercomputing'93*, pages 503–512, 1993.

[32] Y. Saad. SPARSKIT: a basic tool kit for sparse matrix computations. Technical Report 90-20, RIACS, NASA Ames Research Center, 1990.

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

[34] R. Schreiber. Scalability of sparse direct solvers. Technical Report 92.13, Research Institute for Advanced Computer Science (RIACS), NASA Ames Research Center, 1992.

[35] C. Sun. Efficient parallel solutions of large sparse SPD systems on distributed-memory multiprocessors. Technical Report CTC92TR102, Advanced Computing Research Institute, Cornell Theory Center, Cornell University, 1992.

[36] S. Venugopal and V. K. Naik. Effects of partitioning and scheduling sparse matrix factorization on communication and load balance. In *Proceedings of Supercomputing '91*, pages 866–875, 1991.

[37] S. Venugopal and V. K. Naik. SHAPE: A parallelization tool for sparse matrix computations. Technical Report DCS-TR-290, Department of Computer Science, Rutgers University, 1992. Also available as IBM Research Report RC 17899.

[38] S. Venugopal and V. K. Naik. A comparison of block and column methods in distributed sparse Cholesky factorization. In preparation, 1993.

[39] S. Venugopal and V. K. Naik. Towards understanding block partitioning for sparse Cholesky factorization. In *Proceedings of the 7th Int'l. Parallel Processing Symposium*, pages 792–796, 1993. For a complete version of this paper, see IBM Research Report RC 18666, 1993.

[40] S. Venugopal, V. K. Naik, and J. Saltz. Performance of distributed sparse Cholesky factorization with pre-scheduling. In *Proceedings of Supercomputing '92*, pages 52–61, 1992.

[41] O. Wing and J. W. Huang. A computation model of parallel solution of linear equations. *IEEE Transactions on Computers*, C-29:632–638, 1980.

[42] E. Zmijewski and J. R. Gilbert. A parallel algorithm for large sparse symbolic and numeric Cholesky factorization on a multiprocessor. Technical Report 86-733, Department of Computer Science, Cornell University, Ithaca, NY, 1986.