SHAPE: A Parallelization Tool for Sparse Matrix Computations

Sesh Venugopal
Dept. 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: We describe the design, implementation and performance of a Sparse Hybrid Automatic Parallelization Environment (SHAPE). SHAPE partitions and schedules sparse matrix computations for Cholesky factorization with the goal of achieving good performance at low cost, while providing flexibility for use as an experimental tool. It employs efficient parallelization algorithms which reduce the communication cost without adversely affecting the load balance by using a hybrid mixture of column and block partitions. Through several parameters, SHAPE aims for portability across a diverse range of sparse matrix structures and message-passing multiprocessors with different communication cost parameters. We present preliminary timing results on the iPSC/860 and compare the performance of SHAPE with that of a commonly used column-based method. The results show that SHAPE significantly reduces computation time, number of messages, and overall communication time for a variety of test matrices.
1 Introduction

The solution of sparse linear systems of equations, in general, involves computations with nonuniform and unstructured dependencies. As a result, efficient partitioning and scheduling of these computations is a hard problem. As an example, in Figure 1, we show the distribution of nonzeros (○ and ●) in a $25 \times 25$ sparse matrix. The computations are performed only at the nonzero elements with certain inter-element dependencies dictated by the algorithm. In the case of direct solution methods, such as Cholesky factorization, some of the difficulties have been overcome in the past by partitioning and scheduling the sparse matrices using columns as the manipulation units [5], [3], [8]. In these schemes, 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. We refer to these as column-based partitioning and scheduling methods.

![Figure 1](image)

An example of zero-nonzero structure in a $25 \times 25$ matrix.

The column-based partitioning and scheduling methods are straightforward to apply and thus, are most commonly used in hand-parallelized or in compiler-directed parallel Cholesky factorization. In addition, in many cases, these schemes result in a good balance of load. A major drawback, however, is the high communication overhead both in volume and number of messages. On message passing multiprocessors, characterized by relatively high message initialization (i.e., latency) and data transmission costs, the column methods usually suffer from poor performance.

To alleviate the communication overheads associated with the column-based methods, block-based partitioning and scheduling schemes have been proposed by [12] and [2] for parallel Cholesky factorization. Here the manipulation units for partitioning and scheduling are contiguous blocks of nonzero elements which span across multiple columns, but may not contain all the nonzeros from any single column. In these schemes, the communication overhead is reduced by taking advantage of locality. A drawback of these methods is that partitioning a sparse matrix into blocks is an extremely tedious and error-prone task. For large matrices encountered in real-life applications, any type of systematic block partitioning by hand is simply impractical. In an effort to simplify the block partitioning task, in a previous paper [14], we proposed and described an automatic,
general purpose block-based partitioning and scheduling scheme. There, the sparse matrix was partitioned into several blocks of various sizes by taking maximum advantage of locality. For a variety of sparse matrices representing both structured and unstructured problems, we showed that the communication overhead is less than that in the column-based methods.

Although block-based methods reduce the communication overhead, load balancing, in general, turns out to be more difficult than in column-based schemes. This is so because there is very little flexibility in pure block-based schemes to keep the load balanced among the processors at all times. Thus, sophisticated and expensive allocation and scheduling methods are needed to realize good load balance which is comparable in quality to that obtained by using column-based methods.

In this report, we describe the design and implementation of a Sparse Hybrid Automatic Parallelization Environment (SHAPE) for sparse Cholesky factorization on message passing multiprocessor systems. SHAPE is based on a hybrid partitioning methodology that uses both blocks and columns as partitioning units. This hybrid technique, while retaining the advantages of the pure block and column methods, overcomes several of the drawbacks described above. To automate the process for sparse matrices arising from a large class of applications, SHAPE accepts parameters that characterize the matrices and the target multiprocessor systems. Using these parameters, regions of the input sparse matrix are partitioned into columns or blocks depending on the sparsity and the available parallelism. This gives SHAPE its hybrid property. The parametrization also allows for efficient portability and scalability across a diverse range of sparse matrix applications and architectures.

In the current design, SHAPE functions as a preprocessing step prior to the computationally intensive numeric factorization step of the Cholesky factorization. As a partitioner and scheduler, the output of SHAPE is a total ordering of the computations on all processor. In addition, SHAPE also provides a communication profile which can be used to schedule the communication structure on each processor. Such a scheduling of communication further reduces the run-time message passing overhead.

Compared to the commonly used column-based methods, partitioning and scheduling with SHAPE is more expensive. In this report, we show that gains achieved in the numerical factorization are significant and when the factorization is to be performed more than once on the same matrix structure, the additional cost of preprocessing with SHAPE is more than offset after just a small number of iterations. We show this by presenting performance results on the execution time of SHAPE itself, as well as the parallel execution time of numerical factorization using the partitions and schedules generated by SHAPE. We compare these with the performance of the factorization using column-based methods. All the execution times reported here are obtained from the Intel iPSC/860 system.

To describe the design and performance of SHAPE, we have organized the rest of the report as follows. In the next section, we introduce the sparse matrix related terminology used in the report and present the highlights of the steps in sparse Cholesky factorization that are relevant to the discussion in this report. In Section 3, we present an overview of the design and the functionality of SHAPE. The four functional components of SHAPE, namely pre-partitioner, partitioner, allocator, and scheduler, are described in detail in Sections 4, 5, 6, and 7 respectively. The numerical factorization code is described in Section 8. In Section 9, we give preliminary timing results for a variety of test matrices, comparing the performance of hybrid partitioning with a commonly used column-wrap scheme for Cholesky factorization. The report concludes with Section 10. In the rest
of the report, to preserve uniformity, columns in the hybrid partitions produced by SHAPE are also considered to be blocks. Thus, we always use the term “block” to refer to the hybrid equivalent.

2 Background and Terminology

SHAPE, as described in this report, facilitates the partitioning of sparse matrices and the scheduling of the resulting computation and communication so that the sparse Cholesky factorization can be efficiently performed on message passing multiprocessor systems. Cholesky factorization is a well-known method for factoring symmetric positive definite matrices. The algebraic details of Cholesky factorization can be found in [7]. For efficient application of this algorithm to sparse matrices, a few additional steps are required. The details of these steps have been extensively described in the literature; see, for example, [5] and the references therein. In the following, we bring out the dependencies in Cholesky factorization. Understanding these dependencies is crucial to understanding the functionality of SHAPE. We also describe the CSC data structure that is commonly used, and that we will be using, for storing the sparse matrices.

2.1 Sparse Cholesky factorization

Consider the direct solution of

\[ Ax = b, \]

where \( A \) is an \( N \times N \) sparse symmetric positive definite matrix. The Cholesky algorithm factors \( A \) into \( L \) and \( L^T \), where \( L \) is a lower triangular matrix. The sequential column-oriented sparse Cholesky factorization algorithm that computes \( L \) is shown below. 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 \( L \). To understand the details of the Cholesky factorization algorithm, we show the element-level

\[
\begin{align*}
\text{for } j = 1 \text{ to } N \\
\text{for each } k \in [1, j - 1] \text{ s.t. } L_{i,k} \neq 0 \\
\quad \text{for each } i \in [j, N] \text{ s.t. } L_{i,k} \neq 0 \\
\quad \quad L_{i,j} \leftarrow L_{i,j} - L_{i,k} \cdot L_{j,k} \\
\quad \quad \text{endfor} \\
\quad \end{align*}
\]

\[
\text{endfor} \\
L_{j,j} \leftarrow \sqrt{L_{j,j}} \\
\text{for each } i \in [j + 1, N] \text{ s.t. } L_{i,j} \neq 0 \\
\quad L_{i,j} \leftarrow L_{i,j} / L_{j,j} \\
\text{endfor} \\
\text{endfor}
\]

data dependencies in Figure 2. In that figure, the direction of the arrows indicates the direction of data flow. Thus, elements \( L_{j,k} \) and \( L_{i,k} \) from column \( k \) of the factor \( L \) are required in computing element \( L_{i,j} \). \( L_{i,j} = L_{i,j} - L_{i,k} \cdot L_{j,k} \) is the corresponding operation in 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 non-zero elements \( L_{j,k} \) and \( L_{i,k} \), \( 1 \leq k < j \). 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.

![Figure 2](image)

*Figure 2*

Inter-element dependencies in Cholesky factorization.

In practice, to improve the efficiency of sparse Cholesky factorization, additional steps are required. These are:

- **Ordering**: Find a good ordering of the unknowns for elimination. The ordering is given by a permutation matrix \( P \). Most often, a “good” ordering implies one which leads to a sparse factor and fewer arithmetic operations in the numerical factorization step.

- **Symbolic Factorization**: Determine the sparsity structure of the factor \( L \).

Further details on these steps can be found in any reference book on sparse matrix computations (see, for example, [6]). Following these two steps, the actual numerical factorization step is performed using the algorithm shown in Section 2.1. When a linear system is to be solved, additional steps for computing the triangular solutions are also required.

### 2.2 CSC format

Special data structures are needed for the efficient storage and manipulation of sparse matrices as well as to take full advantage of the sparsity. The CSC (compressed storage of columns) format is one such commonly used data structure and, for SHAPE, we have chosen to represent sparse matrices in this format.

In the CSC format, the \( N \times N \) sparse matrix \( A \) is represented using two “parallel” arrays: *Val* and *Struct*. The non-zeros are stored column-wise in ascending order of column numbers. Within each column, they are stored in ascending order of row numbers. For each non-zero value in the Val
array, its row number or index is stored in the Struct array. A third auxiliary array, called Index
array, gives pointers to the first non-zero element for each column. Thus, the nonzero elements in
column $j$ are stored in the Val array from location $\text{Index}[j]$ through $\text{Index}[j+1] - 1$. The row index
of each of these elements is stored in the corresponding locations of the Struct array. As an example,
consider the lower triangular sparse matrix $L$ shown in Figure 3. A non-zero is represented by an
‘x’.

$$
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
\hline \\
1 & x \\
2 & x \\
3 & x \\
4 & x & x & x \\
5 & x & x & x \\
6 & x & x & x & x \\
\end{array}
$$

Figure 3
An example showing the structure of sparse $L$.

Figure 4 shows the CSC format for this matrix. The three arrays, $\text{Lval}$, $\text{Lstruct}$, and $\text{Lindex}$ are
the Val, Struct, and Index arrays, respectively. The arrows depict pointers into the $\text{Lstruct}$ array
(and identically into the $\text{Lval}$ array) pointing to the first storage locations of the columns of $L$.

For instance, $\text{Lindex}[3] = 5$ is the index in $\text{Lval}$ and $\text{Lstruct}$ at which the first non-zero element for
column 3 is stored and $\text{Lindex}[4] = 8$ is the index in $\text{Lval}$ and $\text{Lstruct}$ at which the first non-zero
for column 4 is stored. Thus, $\text{Lval}[5]$ through $\text{Lval}[7]$ store the non-zero values in column 3 and
$\text{Lstruct}[5]$ through $\text{Lstruct}[7]$ hold the corresponding row indices - 3, 4, and 6 are the respective
row numbers of the non-zeros in column 3.

For the partitioning and scheduling of Cholesky factorization, actual nonzero values are irrelevant.
However, the symbolic representation of the factor $L$, obtained from the symbolic factorization step, is essential. In dealing with this symbolic structure, we will ignore any numeric cancellation that might occur during factorization. The two arrays, $Lindex$ and $Lstruct$, completely describe the symbolic structure of matrix $L$, and this symbolic structure is the input to SHAPE.

3 Overview of SHAPE

As pointed out in the previous section, the input to SHAPE is the symbolically factored lower triangular matrix $L$. The output is a schedule of tasks, in the form of a total ordering, assigned to each processor. A task is defined in terms of a 4-tuple, $< \mathcal{P}, Oper, [\mathcal{P}], [\mathcal{P}] >$, where $\mathcal{P}$ defines a partition in $L$ and $Oper$ is either an update or a scale type operation on a partition. Altogether there are twelve variants of these two types of operations. In some of the twelve operations, one or both of the tuple members enclosed within the square brackets are not required. The definition of the partitions and the twelve variants of the operations are described in detail in Section 5. The output of SHAPE also includes a profile of communication that can be used to automatically schedule message passing on each processor. Figure 5 gives a top-level view of how SHAPE fits into the step-wise process of solving $Ax = b$.

![Figure 5](image_url)

**Figure 5**
Solution of a sparse linear system using SHAPE.

Functionally, SHAPE operates in four stages: **pre-partitioning, partitioning, allocation** and **scheduling**. The interaction among these four functions is shown schematically in Figure 6.
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. The output of the pre-partitioner is a decomposition of $L$ into clusters of columns. Within each multiple-column cluster, the pre-partitioner identifies triangular and rectangular shaped blocks. The partitioner, using the parameter $G$, divides each cluster into unit blocks of non-zeros, producing partitions that are either single columns or are triangular or rectangular in shape. Apart from these unit blocks, the output of the partitioner consists of the block dependency DAG, which is a directed acyclic graph of inter-block dependencies and the cluster DAG, which is a directed acyclic graph of inter-cluster dependencies. The allocator uses the cluster DAG, with the number of processors, $P$, as a parameter, to produce an assignment of blocks to processors. The scheduler then symbolically executes the block dependency DAG using the block-to-processor assignment, assuming that all computations on a block are performed on the processor to which that block is mapped. It determines a schedule, which is 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.
4 Pre-partitioning

To take locality into account, in partitioning the sparse matrix $L$, SHAPE makes use of some of the general properties observed in the sparsity structure of $L$. 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. If the ordering used is a minimum-degree like ordering, then, overall, 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 these blocks is the first step in partitioning the sparse matrix.

In SHAPE, starting with the Lindex and Lstruct arrays, the pre-partitioner efficiently identifies the naturally occurring triangular and rectangular dense blocks in $L$. For this, the pre-partitioner first divides the matrix $L$ into disjoint bands of columns called clusters. 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.

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 $\{L_{i,j} \mid m \leq i, j \leq n, i \geq j\}$. We alternatively represent this triangle as $(m,n)$. $T_{m,n}$ 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 7(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.

![Figure 7](image_url)

**Figure 7**
Dense triangles and rectangles.
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 $n - m + 1$.

The following is a formal definition of a rectangle. A rectangle 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 block of elements
\[ \{ L_{i,j} \mid n < r_1 \leq i \leq r_2, m \leq c \leq j \leq n \} \]. We alternatively represent this rectangle as
\[ (r_1, r_2, c, n) \]. $R$ 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 7(b), we show a portion of some matrix below the triangle of 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 $R_{12,15,5,5}$ and $R_{14,14,4,5}$ are dense, but not maximally dense rectangles. The significance of the solid-line and dotted-line boxes in that figure will be clear shortly.

The main functions of the pre-partitioner are:

i. To identify maximally dense triangles in $L$ given $L\text{index}$ and $L\text{struct}$. From these, disjoint clusters are identified. We call this the cluster identification step.

ii. To identify a set of disjoint rectangles in each cluster. We call this the rectangle identification step.

We now present algorithms for the cluster identification and rectangle identification steps and analyze their time complexities. In the rest of the discussion, when we refer to a cluster, we mean a multi-column cluster.

4.1 Cluster identification

The following assertion is used to construct a fast algorithm for cluster identification.

Claim 1: Let $T_{k,m}$ be a dense triangle and $T_{k-1,m}$ not be a dense triangle. (i) if $L_{m+1,k}$ is a nonzero element then $T_{k,m+1}$ is a dense triangle. (ii) if $L_{m+1,k}$ is zero then $T_{k,m}$ is a maximally dense triangle.

Proof: We prove the first part of the claim by showing that $L_{m+1,j}$ is a nonzero
\[ \forall j, k \leq j \leq m + 1. \] Since $T_{k,m}$ is a dense triangle, $L_{j,k}$, $k \leq j \leq m$, is a nonzero element. Thus, when $L_{m+1,k}$ is a nonzero, in Cholesky factorization $L_{j,k}$ and $L_{m+1,k}$ combine to produce a nonzero element $L_{m+1,j}$, $\forall j, k < j \leq m + 1$.

The second part of the claim follows from the definition.

Algorithm $Id\_Clusters$ below gives a procedure to identify maximally dense triangles using which disjoint clusters are identified. The input to $Id\_Clusters$ are the $L\text{struct}$ and $L\text{index}$ arrays representing the $N \times N$ matrix $L$. For clarity, in the following algorithm, we only show the identification
of maximally dense triangles and reporting of the disjoint dense triangles. The disjoint cluster associated with each reported dense triangle can be identified in a straightforward manner.

---

procedure 

\[ Id\_Cluster \]

\[
0. \; c \leftarrow 1, \; startrow \leftarrow 0
\]

1. while (\( c \leq N \)) do
   \[ index \leftarrow \text{Lindex}[c] + startrow \]
   \[ \text{while} \ (index < \text{Lindex}[c + 1] - 1 \text{ and } \text{Lstruct}[index + 1] = \text{Lstruct}[index] + 1) \text{ do} \]
   \[ index \leftarrow index + 1 \text{ endwhile} \]
   \[ c1 \leftarrow \text{Lstruct}[index] \]
   \[ entry \leftarrow index - \text{Lindex}[c] \]
   \[ \text{if} \ (c1 = N) \text{ then} \]
   \[ \text{report triangle } T_{c,N} \]
   \[ \text{quit} \]
   \[ \text{else} \]
   \[ k \leftarrow c + 1 \]
   \[ \text{endif} \]
endwhile

3. \[ done \leftarrow \text{false} \]
   \[ \text{while} \ (k \leq c1 \text{ and } \text{not done}) \text{ do} \]
   \[ index \leftarrow \text{Lindex}[k] + entry \]
   \[ \text{if} \ (\text{Lstruct}[index] = c1 + 1) \text{ then } \text{done} \leftarrow \text{true} \]
   \[ \text{else} \]
   \[ k \leftarrow c1 + 1, \; entry \leftarrow entry - 1 \text{ endif} \]
endwhile

4. \[ \text{if} \ (k = c1 + 1) \text{ then report triangle } T_{c,c1}, \; c \leftarrow c1 + 1, \; startrow \leftarrow 0 \]
   \[ \text{else report triangle } T_{c,k-1}, \; c \leftarrow k, \; startrow \leftarrow entry \text{ endif} \]
endwhile

---

The algorithm scans columns left to right, beginning with the first column, tracing an outline of the nonzero elements that form intersecting triangular blocks along the diagonal. We explain the rest of the algorithm by referring to Figure 7(a). In the while loop of step 2, the selected column \( c \) is scanned downwards, beginning at the diagonal or from the bottom edge of a predetermined triangle. For the example in Figure 7(a), \( c \) is scanned from the diagonal. The column \( c \) is scanned until a zero in that column is encountered or until all rows are exhausted. In the latter case, the entire block of the matrix below (and including) the diagonal and to the right of (and including) column \( c \) is a dense triangle. In scanning \( c \), a zero is encountered when two successive nonzero elements in that column are more than one row apart. In the example of Figure 7(a), a zero is encountered in row \( c1 + 1 \). By claim 1, \( T_{c,c1} \) is a maximally dense triangle. Now the columns to the right of \( c \) are scanned to identify any intersecting maximally dense triangles. This is done in step 3 of the algorithm, where nonzero elements are searched on row \( c1 + 1 \) starting at column \( c1 + 1 \). In the CSC representation, this would be equivalent to checking whether, in any column \( k \), \( c < k \leq c1 \), the row number of the entry immediately succeeding an entry in row \( c1 \) is \( c1 + 1 \). If the first nonzero element is on the diagonal, then the maximally dense triangle \( T_{c,c1} \) is reported in step 4 and a new triangle is searched starting at the diagonal element on column \( c1 + 1 \). Otherwise, there is another
maximally dense triangle $T_{c',c,2}$, $c < c' \leq c1$, intersecting $T_{c,c1}$, as in the example of Figure 7(a), and the search in step 3 ends when $k$ is equal to $c'$. In this case the triangle $T_{c,c1}$ is reported and the next iteration of the algorithm begins at row $c1 + 1$ in column $c'$. When the algorithm exits, all the consecutive maximally dense triangles are identified; the non-intersecting maximally dense triangles are reported as such and when there is a sequence of intersecting maximally dense triangles, as in the example above, a list of disjoint dense triangles followed by a maximally dense triangle is reported.

**Claim 2:** The worst case time complexity of algorithm $Id\_Clusters$ is $O(N)$ comparisons for an $N \times N$ lower triangular matrix.

**Proof:** Consider one iteration of step 1. Let the scan begin in column $c1$ at row $r1$. See Figure 8. Suppose step 2 scans down the column until row $r2$. This scan makes $r2 - r1 + 1$ comparisons. Now let step 3 scan along row $r2$ until column $c2$, making $c2 - c1 + 1$ comparisons. In this iteration, therefore, the algorithm has traced an 'L'-shaped contour (a vertical section with a horizontal section at right angles to it). We will call this one step. In the next iteration, there will be another step starting at column $c2 + 1$. The algorithm begins scanning at the element $L_{1,1}$ and when all the steps are traced, each row and column is scanned exactly once. When a row or column is scanned, one comparison is performed. It is easy to see that the total number comparisons is $2N$ and hence the claim.

4.2 Rectangle identification

Here we describe the process of disjoint rectangle identification within each cluster reported by algorithm $Id\_Cluster$. The following assertion is used to construct a fast algorithm for rectangle identification.

**Claim 3:** In a cluster $C_{m,n}$, if $L_{i,j}$, $j \leq n < i$, is a nonzero element, then $\forall k$, $j < k \leq n$, $L_{i,k}$ is also a nonzero element.
**Proof:** By definition, associated with cluster $C_{m,n}$ there is a dense triangle, $T_{m,n}$. Thus, $\forall k, \ j < k \leq n$, $L_{k,j}$ is a nonzero element. If $L_{i,j}$, $j \leq n < i$, is a nonzero element, then in Cholesky factorization, $L_{k,i}$ and $L_{i,j}$ combine to produce a nonzero element in $L_{i,k}$.

Claim 3 asserts that if there is a nonzero element $L_{i,j}$ below the triangle in a cluster, then all the elements to its right in row $i$ of the cluster will be nonzeros. Also, in a cluster $C_{m,n}$, any nonzero element below the triangle $T_{m,n}$ will lie in some dense rectangle with right edge on column $n$.

Algorithm $Id\_Rects$ below gives the procedure to identify disjoint rectangles in cluster $C_{m,n}$. The input to this algorithm are Lstruct and Lindex arrays and the cluster $C_{m,n}$.

---

**procedure** $Id\_Rects$

1. **while** ($c > m$ and Lstruct[Lindex[c - 1] + last[c - 1]] = r1) **do**
   
2. **endwhile**

3. **if** ($c = m$) **then**
   
4. **else**
   
5. **endif**

6. **report rectangle** $R_{r2,r2,c,n}$

7. **for** $i = c$ to $n$ **do**
   
8. **endfor**

The algorithm $Id\_Rects$ identifies all disjoint dense rectangles that have right edge on column $n$. In each iteration of the loop of Step 1, the algorithm identifies a new dense rectangle stretching to the left. The disjoint rectangles are obtained by scanning column $n$ downwards and by breaking any intersecting maximally dense rectangles row-wise so that the resulting disjoint dense rectangles have their right edge on column $n$.

An auxiliary array, last[0 : n] is maintained whose size is equal to the cluster width of $C_{m,n}$. This array acts as an offset for each column of the cluster $C_{m,n}$, indicating the first nonzero element in each column that is not part of an already identified rectangle. This array is useful in skipping over the nonzeros in the interior of each rectangle, without examining them. This speeds up the identification process.
We now explain the details of the above algorithm using the example shown in Figure 7(b). The main while loop of step 1 checks whether there is a nonzero in column $n$ that is not contained in any rectangle reported thus far. If there is such a nonzero element, say in row $r1$, the detection of a new rectangle, say $R$, begins at that nonzero element. In the example, the first nonzero entry in column 5 is in row $r1 = 7$. In step 2, columns in the cluster are scanned going right to left across columns, checking for a nonzero in row $r1$. Since all rectangles above row $r1$ have already been reported, $Lstruct[Lindez[c] + last[c]] \geq r1, m \leq c \leq n$. When a column which does not have a nonzero in row $r1$ is encountered, the algorithm has detected the left edge of $R$, say in column $c$, and the nonzeros seen thus far in row $r1$, between columns $n$ and $c$, form the top edge of $R$. For example, in Figure 7(b), scanning row $r1 = 7$ results in detecting an edge in column 3.

Step 3 scans downward along the left edge $c$ to find the bottom edge of $R$. This is detected when (a) the bottom row of the matrix is encountered, or (b) two successive nonzeros in column $c$ are more than one row apart, or (c) two successive nonzero entries are only a row apart but the second entry (in row $r$, say) has a companion entry in row $r$ of column $c - 1$, indicating an intersection of maximally dense rectangles. When the scan terminates, by claim 3, the rectangle $R$ has been completely identified. In Figure 7(b), case (a) would be encountered in detecting rectangle $(15, 15, 2, 5)$ while scanning down column 2; case (b) would be encountered in detecting rectangle $(7, 8, 3, 5)$ while scanning column 3 or while detecting rectangle $(10, 11, 2, 5)$ while scanning column 2; and case (c) would be encountered in detecting rectangle $(12, 13, 5, 5)$ while scanning column 5 or in detecting rectangle $(14, 14, 4, 5)$ while scanning column 4.

Step 4 reports $R$. All the columns that make up the rectangle are marked as having examined $w$ more entries, where $w$ is the breadth of $R$. This marking is achieved by advancing by $w$ the entries in last corresponding to the component columns of $R$. This ensures that the nonzero elements in $R$’s interior are not examined.

Claim 4: The worst case time complexity of algorithm $Id.Rects$ is $O(Z)$ comparisons, where $Z$ is the number of nonzeros which make up the rectangles.

Proof: The algorithm traces over the top and left edges of every rectangle reported. For a rectangle $R$, the top edge is traced in step 2 and the left edge is traced in step 3. Step 5 is equivalent to tracing the bottom edge of $R$. In all edge, each element is examined exactly once. Thus the time to trace an edge is proportional to the number of nonzeros in the edge, or the length of the edge.

In the worst case, every rectangle could be of unit height, and $Id.Rects$ examines all the nonzeros which make up the rectangles, giving the above complexity. In the best case, there is a single rectangle of height $l$ and breadth $w$ and the time taken by $Id.Rects$ is $O(l + w)$.

4.3 Parameters

So far we have described the functioning of the pre-partitioner assuming that each cluster is a multi-column cluster. 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. 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 flexibility, the pre-partitioner makes use of the input parameters $W$ and $G$ in the cluster identification and rectangle identification steps. The parameter $W$ specifies the \textit{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, with $W$ equal to 5, clusters that are at least 5 columns wide are accepted for block partitioning. The parameter $G$ adds more flexibility to this process. The parameter $G$ is used primarily in the partitioning step to partition triangles and rectangles identified by the pre-partitioner into smaller blocks. The details of this mechanism are described in Section 5. This parameter is also used by the pre-partitioner in identifying the clusters. Before describing this use of $G$, we first describe its significance.

The value of $G$ specifies the minimum block size in partitioning the triangle and rectangles in a cluster. The block size is based on the number of nonzeros and/or the amount of computational work per block. For matrices that are ordered using minimum degree and other similar ordering schemes, 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 of higher priority. In the communication overhead prone 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.

In the pre-partitioning step, if, in a cluster identified using the parameter $W$, neither the triangle nor the rectangles yield to partitioning according to $G$, then that cluster is decomposed into single column clusters.

5 Partitioning

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. We refer to these as the \textit{unit} blocks, i.e., blocks which are atomic units for allocation and scheduling. When there is no ambiguity, we refer to these simply as blocks. For uniformity, a single column is also considered to be a unit block. SHAPE uses two different grain sizes for partitioning: one for triangles and the other for rectangles. Once the unit blocks are identified, block-based tasks and inter-block dependencies are defined. These dependencies are computed using the idea of \textit{projection} and a data structure called the \textit{interval tree}. The partitioner produces a block dependency directed acyclic graph (DAG) that encapsulates this inter-block dependency information. The partitioner also produces a cluster DAG based on the dependencies among the clusters. The block dependency DAG is used by the scheduler and the cluster DAG is used by the allocator.

We first summarize the different types of tasks that are produced by the partitioner and present a sequential block sparse Cholesky algorithm that uses these tasks. We then describe the details of
partitioning and formulation of the block dependency DAG used in the block Cholesky algorithm.

5.1 Block sparse Cholesky factorization

To define block Cholesky factorization, we first extend the definitions of the element-level operations presented in Section 2 to block update and scale operations. Note that all elements of $L$ referred to in the following definitions are nonzeros.

**Definition 4:** Block $B$ is updated by a base block $B'$ and a multiplier block $B''$ iff $\exists i, j, k$ such that $L_{i,j} \in B \land L_{j,k} \in B' \land L_{i,k} \in B''$. The update operation is referred to as block base-multiplier update. We say that $B$ depends on $B'$ and $B''$, denoted as $B, B'' \delta B$.

**Definition 5:** Block $B$ is updated by a base block $B'$ only iff $\exists i, j$ such that $L_{i,j} \in B' \land L_{i,i} \in B$. The update operation is referred to as block base update. We say that $B$ depends on $B'$, denoted as $B' \delta B$.

<table>
<thead>
<tr>
<th>No.</th>
<th>Task Description</th>
<th>Tuple Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Column Updates Column</td>
<td>$&lt; \text{Col, Update, Col}&gt;$</td>
</tr>
<tr>
<td>2</td>
<td>Column Updates Triangle</td>
<td>$&lt; \text{Tri, Update, Col}&gt;$</td>
</tr>
<tr>
<td>3</td>
<td>Rectangle Updates Column</td>
<td>$&lt; \text{Col, Update, Rect}&gt;$</td>
</tr>
<tr>
<td>4</td>
<td>Rectangle Updates Triangle</td>
<td>$&lt; \text{Tri, Update, Rect}&gt;$</td>
</tr>
<tr>
<td>5</td>
<td>Column Updates Rectangle</td>
<td>$&lt; \text{Rect, Update, Col}&gt;$</td>
</tr>
<tr>
<td>6</td>
<td>Triangle and Rectangle Update Rectangle</td>
<td>$&lt; \text{Rect, Update, Tri, Rect}&gt;$</td>
</tr>
<tr>
<td>7</td>
<td>Two Rectangles Update Column</td>
<td>$&lt; \text{Col, Update, Rect, Rect}&gt;$</td>
</tr>
<tr>
<td>8</td>
<td>Two Rectangles Update Triangle</td>
<td>$&lt; \text{Tri, Update, Rect, Rect}&gt;$</td>
</tr>
<tr>
<td>9</td>
<td>Two Rectangles Update Rectangle</td>
<td>$&lt; \text{Rect, Update, Rect, Rect}&gt;$</td>
</tr>
<tr>
<td>10</td>
<td>Triangle Updates Rectangle</td>
<td>$&lt; \text{Rect, Update, Tri}&gt;$</td>
</tr>
<tr>
<td>11</td>
<td>Scale Column</td>
<td>$&lt; \text{Col, Scale}&gt;$</td>
</tr>
<tr>
<td>12</td>
<td>Scale Triangle</td>
<td>$&lt; \text{Tri, Scale}&gt;$</td>
</tr>
</tbody>
</table>

Table 1
The twelve generic update and scale tasks.

Note that a block base update could, in general, include element-level base-multiplier updates. As pointed out in the Section 3, there are twelve different variants of the block update and scale operations. These variants are defined by the shapes of the unit blocks that participate in these operations; each variant results in a different computational code. Accordingly, there are twelve different generic tasks. These are summarized in Table 1 using the tuple notation $<\mathcal{P}, \text{Oper}, [\mathcal{P}],[\mathcal{P}]>$. 

15
introduced in Section 3. In that table, the partitions are identified by their shape: “Col” for column, “Tri” for triangle and “Rect” for rectangle.

In the following, we present a modified sparse Cholesky factorization algorithm that uses block partitions and the tasks defined on these blocks. We refer to this algorithm as block sparse Cholesky factorization.

```
procedure Block_Sparse_Cholesky

for C = 1 to Q do
    for each block B in C do
        $U_B \leftarrow$ list of updaters of B
        for each $B' \in U_B$ do
            $< B, Update, B' >$
        endfor
        for each $(B', B'') \in U_B$ do
            $< B, Update, B', B'' >$
        endfor
        if (B is not a rectangle) then
            $< B, Scale >$
        endif
    endfor
endfor
```

In this algorithm, $Q$ is the number of clusters in $L$ as identified by the pre-partitioner. For each cluster, computations are performed on each block in that cluster. For each block, first the updating unit blocks are identified from the block dependency DAG. Recall that the updating blocks include both the base and multiplier blocks. Next, all the base update tasks are completed. This is followed by all the base-multiplier updates and finally the scale task for that block is performed. Note that a rectangle does not have a scale task. This is because the scaling of a rectangle depends on one or more triangles which update it, since these triangles contain the diagonal elements required for the scaling. The scaling is therefore implicitly performed when these updates are executed.

In the following sections, we first describe the details of partitioning triangles and rectangles into unit blocks and then the formulation of the dependency DAGs.

### 5.2 Triangle and rectangle partitioning

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. In Figure 9(a), $k = 4$. All the strips are of the same width except, possibly, the bottom-most horizontal strip and the rightmost vertical strip which may be smaller in width. So all rectangular unit blocks are of the same size and all triangular unit blocks are of the same size, except, possibly, those at the “edges”. The triangle is thus partitioned into $k(k + 1)/2$ unit blocks: $k$ triangular unit blocks and $k(k - 1)/2$ rectangular unit blocks. The size of each partition (and hence $k$) is controlled by the grain size parameter $G$, described in Section 4.3. The parameter $G$ imposes a lower bound on
the block size and hence an upper bound on the number of partitions. Let $G$ allow at most $t$ parts
in the given triangle. Then the partitioner will choose the largest $k$ for which $k(k + 1)/2$ is at most
$t$. Procedure $\text{Partition\_triangle}$ shows how a triangle of width $w$ is partitioned into at most $t$ parts.

\begin{verbatim}
procedure Partition_triangle
1. $k \leftarrow (\sqrt{1 + 8 \times t} - 1)/2$
2. $v \leftarrow w/k$
3. if $w$ is not an integer multiple of $k$ then $v \leftarrow v + 1$ endif
4. $k' \leftarrow w/v$
5. if $w$ is not a multiple of $v$ then $k' \leftarrow k' + 1$ endif
\end{verbatim}

All divisions in that procedure are integer divisions. Step 1 computes the maximum number of
horizontal and vertical strips given that there can be at most $t$ partitions. In step 2, $v$, the size of
a strip, is initialized to $w/k$. If $w$ is not an integer multiple of $k$, $k - 1$ strips are of width $v$ each
and one strip is of width $v + (w \mod k)$. Steps 3, 4 and 5 ensure that the width of the “edge” strip
is less than that of the others and the number of strips, $k' \leq k$.

Now consider a rectangle of size $m \times n$, where $m$ is the height and $n$ is the breadth. Let the
parameter $G$ be such that a maximum of $r$ partitions are to be made for this rectangle. The
rectangle is divided into $p$ horizontal strips and $q$ vertical strips. All the horizontal strips are of
equal height except, possibly, the bottom-most one and all the vertical strips are of equal breadth
except, possibly, the rightmost vertical strip. These “edge” strips may be narrower than the rest.
Thus all unit blocks are of the same size except, possibly, at the edges. Let $p \times q$ be equal to $k$.
The partitioner chooses the largest $k$ which is at most equal to $r$. The factors $p$ and $q$ are chosen in
such that each unit block is approximately square in shape. This ensures high degree of locality for
a given amount of work. For example, in Figure 9(b), $p$ is equal to three and $q$ is five. Procedure
$\text{Partition\_rectangle}$ describes the simple heuristic used to achieve this. Let $h$ be the height of a
horizontal partition and $v$ be the breadth of a vertical partition. The algorithm will choose a $h$
and a $v$ for which $|h - v|$ is minimum. At the end of the process, $best_v$ and $best_h$ give the vertical
partition breadth and the horizontal partition height respectively.
procedure Partition\_rectangle

1. $l \leftarrow \sqrt{r}$, best\_v, best\_h, best\_d \leftarrow \text{max val}$
2. \textbf{for} $i = 1$ to $l$ \textbf{do}
3. \hspace{1em} $j \leftarrow r/i$
4. \hspace{1em} \textbf{if} $m < i$ \textbf{then} $h \leftarrow 1$ \textbf{else} $h \leftarrow m/i$ \textbf{endif}
5. \hspace{1em} \textbf{if} $n < j$ \textbf{then} $v \leftarrow 1$ \textbf{else} $v \leftarrow n/j$ \textbf{endif}
6. \hspace{1em} \textbf{if} $n < i$ \textbf{then} $v' \leftarrow 1$ \textbf{else} $v' \leftarrow n/i$ \textbf{endif}
7. \hspace{1em} \textbf{if} $m < j$ \textbf{then} $h' \leftarrow 1$ \textbf{else} $h' \leftarrow m/j$ \textbf{endif}
8. \hspace{1em} \textbf{if} $(|v - h| < |v' - h'|)$ \textbf{then} best\_v \leftarrow $v$, best\_h \leftarrow $h$, best\_d \leftarrow $|v - h|$ \textbf{endif}
9. \hspace{1em} \textbf{endfor}
10. \textbf{if} $m$ is not a multiple of $p$ \textbf{then}
11. \hspace{2em} best\_h \leftarrow best\_h + 1, $p \leftarrow m/best\_h$
12. \hspace{2em} \textbf{if} $m$ is not a multiple of $best\_v$ \textbf{then} $p \leftarrow p + 1$ \textbf{endif}
13. \textbf{endif}
14. \textbf{if} $n$ is not a multiple of $q$ \textbf{then}
15. \hspace{2em} best\_v \leftarrow best\_v + 1, $q \leftarrow n/best\_v$
16. \hspace{2em} \textbf{if} $n$ is not a multiple of best\_d \textbf{then} $q \leftarrow q + 1$ \textbf{endif}
17. \textbf{endif}
18. $r \leftarrow p \times q$

The algorithm proceeds by picking possible factors $i$ and $j$ as follows. The factor $i$ is chosen to be $1, 2, \ldots, \sqrt{r}$ in turn and for each $i$, the corresponding $j = r/i$. This is coded in the \textbf{for} loop of step 2. For each such pair of factors, the following action is performed in step 4. Two temporary pairs of values are obtained for the horizontal and vertical partition widths: one pair is obtained using $i$ for the horizontal and $j$ for the vertical and the other is obtained using $j$ for the horizontal and $i$ for the vertical. In step 5, the difference of the vertical and horizontal partition widths is compared for the two pairs. The lesser difference indicates a “more square” shape; this is then compared with the “best” difference so far. The best difference, along with the corresponding horizontal and vertical partition widths are reset. Steps 6, 7, 8 and 9 ensure that the “edge” horizontal and vertical strips are no bigger in size than the other strips.

5.3 Dependency types and definitions

After partitioning the clusters into unit blocks, the partitioner identifies the dependencies among these blocks. The inter-block dependencies are established by building on the element-level dependencies shown in Figure 2. These dependencies can be classified according to the shape and type of the unit blocks involved. We describe how these dependencies are constructed, using the idea of \textit{projection} and the intersection of projections.
For the purpose of illustration, we use the example shown in Figure 10. In that figure, $A$, $B$ 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 triangular block of cluster $C$ is partitioned into six unit blocks; three triangular units and three rectangular units. The rectangular blocks are each partitioned into three unit 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 $R_6$.

5.3.1 Projection

**Definition 6:** 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 $[m,n]$. This is denoted as $HP(T)$.

**Definition 7:** 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 $[m,n]$ and is denoted as $VP(T)$.

A single point $P : (i,j)$ 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 the first case, $HP(P)$ is $[i,i]$ and $VP(P)$ is $[j,j]$. 

![Figure 10](image-url) Inter-block dependencies in Cholesky factorization.
In the second case, \( HP(P) \) and \( VP(P) \) are both equal to \([i, i]\). In both cases the projections are degenerate zero-length intervals.

In Figure 10, \( HP(P_1) = [I_1, I'_1] \), \( HP(P_2) = [I_2, c1] \) and \( VP(R_2) = [c2, c3] \).

Projections are used by the partitioner in order to compute 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 unit 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 10, the cluster \( C \) matches the projection \( HP(P_1) \), and the matching interval is \([c1, I'_1]\). 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 non-zeros is horizontally projected. The row number of the non-zero element along with the column number is stored in the matching cluster(s).

5.3.2 Characterization of dependencies

We use Definition 5 and Definition 4 of Section 5.1 and the following two definitions to characterize the different generic task dependency types in block sparse Cholesky factorization.

**Definition 8:** If \( B' \delta B \) then \( \delta_{B', B} \) denotes the portion of \( B' \) that is transferred from \( B' \) to \( B \), i.e., \( \delta_{B', B} = (r, r', c, c') \) means that the sub-part of \( B' \) enclosed in the row extent \( r - r' \) and column extent \( c - c' \) is used to update \( B \).

**Definition 9:** If \( B', B'' \delta B \) then \( \delta_{B', B'', B} \) gives the portions of \( B' \) and \( B'' \) that are used to update \( B \), i.e., \( \delta_{B', B'', B} = (r, r', i, i', c, c') \) means that the sub-part of \( B' \) enclosed in the row extent \( r - r' \) and column extent \( c - c' \) is combined with the sub-part of \( B'' \) enclosed in the row extent \( i - i' \) and column extent \( c - c' \) to update \( B \).

The following two rules are used to determine inter-block dependencies or updates. Each of these dependencies results in a task, and will be denoted by using the tuple notation for tasks, \(< P, Oper, |P|_1, |P|_2 > \) introduced in Section 3.

1. Base update

**Claim 5:** Block \( B \) is updated by a base block \( A \) iff \( VP(B) = HP(B) \) and \( HP(A) \cap HP(B) \neq \emptyset \).

By applying this rule, we obtain the following generic dependencies based on block shapes. In each case, the corresponding task in the form of a tuple is also indicated. The block dependencies are illustrated in Figure 11.

- Column updates column, \(< Col, Update, Col > \)
  Let \( A \) be column \( k \) and \( B \) be column \( j \). For \( A \delta B \) to be true, \( \exists (P_1 \in A \land P_2 \in B) \) such
that \( VP(P_2) = HP(P_2) \) and \( HP(P_1) = HP(P_2) \). This is only satisfied for \( P_1 = L_{jk} \) and \( P_2 = L_{j,k} \). This is the base case as seen in Figure 2. Also, \( \delta_{A,B} = (j, N, k, k) \), i.e., all non-zeros below and including the non-zero in row \( j \) of column \( k \) are used in the update.

- **Column updates triangle, \( \langle Tri, Update, Col \rangle \)**
  Let block \( A \) be column \( k \) and block \( B \) be triangle \( T_{r,r'} \). By definition, \( VP(T) = HP(T) = [r,r'] \). For \( A \) \( \delta \) \( B \) to be true, then, \( \exists (P_i \in A) \) such that \( VP(P_i) \cap HP(T) \neq \emptyset \). This is only satisfied for any \( P_1 = L_{i,k}, \tau \leq i \leq \tau' \). Also, \( \delta_{A,B} = (i, \tau', k, k) \), i.e., all non-zeros in column
$k$ from row $i$ to row $r'$ are used in the update. In Figure 11(a), the non-zero elements of column $k$ that are involved in the update are in rows $i_1$, $i_2$ and $i_3$. The points of intersection of the dotted lines with each other and of the dotted lines with the diagonal are the points of triangle $T$ that are updated by column $k$.

- **Rectangle updates column, <Col, Update, Rect>**
  Let block $A$ be rectangle $R_{r,r',c,c'}$ and block $B$ be column $k$. For $A \delta B$ to be true, $\exists (P \in B)$ such that $HP(P) \cap HP(R) \neq \emptyset$. This is only satisfied for $P = L_{k,k}, r \leq k \leq r'$. Then, $\delta_{A,B} = (k,r',c,c')$, i.e., the sub-part of the rectangle enclosed between rows $k$ and $r'$ is used to update the column. In Figure 11(e), the shaded portion of the rectangle between rows $k$ and $r_2$ update the column elements between rows $k$ and $r_2$.

- **Rectangle updates triangle, <Tri, Update, Rect>**
  Let block $A$ be rectangle $R_{r,r',c,c'}$ and block $B$ be triangle $T_{m,n}$. By definition, $VP(T) = HP(T) = [m,n]$. For $A \delta B$ to be true, then, $HP(A) \cap HP(B) \neq \emptyset$. This is satisfied if and only if $[r,r'] \cap [m,n] \neq \emptyset$. Let this intersection be $[i,i']$, then $\delta_{A,B} = (i,i',c,c')$. In Figure 11(g), the shaded portion of $R$ updates the shaded portion of $T$.

2. **Base-multiplier update**

**Claim 6:** Block $C$ is updated by blocks $A$ and $B$ ($A,B \delta C$) iff $HP(A) \cap VP(C) \neq \emptyset$ and $VP(A) \cap VP(B) \neq \emptyset$.

The following block dependencies are obtained by applying this rule.

- **Column updates rectangle, <Rect, Update, Col>**
  Let $A$ and $B$ be block $k$ and $C$ be rectangle $R_{r,r',c,c'}$. For $A,B \delta C$ to be true, $\exists (P_1 \in A \wedge P_2 \in A)$ such that $HP(P_1) \cap VP(C) \neq \emptyset$ and $HP(P_2) \cap VP(C) \neq \emptyset$. This is only satisfied for $P_1 = L_{j,k}, P_2 = L_{i,k}, c \leq j \leq c'$ and $r \leq i \leq r'$. Then, $\delta_{A,B,C} = (c,c',r,r',k,k)$. In Figure 11(b), the non-zero elements in rows $i_1$ and $i_2$ of column $k$ combine with the non-zero elements in rows $j_1$ and $j_2$ to update a portion of $R$. This updated portion is the set of points given by the intersection of the dotted lines in $R$'s interior. This update can be treated as a base update for practical purposes, since both the base and the multiplier are the same block.

- **Triangle and rectangle update rectangle, <Rect, Update, Tri, Rect>**
  Let block $A$ be triangle $T_{m,n}$, block $B$ be rectangle $R_{r,r',c,c'}$ and block $C$ be rectangle $R_{i,i',j,j'}$. For $A,B \delta C$ to be true, $[m,n] \cap [j,j'] = [u,v] \neq \emptyset$ and $[m,n] \cap [c,c'] = [u,u'] \neq \emptyset$ and $[r,r'] \cap [i,i'] = [t,t'] \neq \emptyset$. Then, $\delta_{A,B,C} = (v,v',t,t',u,u')$. In Figure 11(d), the shaded rectangular portion of $T$ combines with the entire shaded rectangle $R_1$ to update the entire shaded rectangle $R_2$.

- **Two rectangles update column, <Col, Update, Rect, Rect>**
  Let block $A$ be rectangle $R_{r,r',c,c'}$, block $B$ be rectangle $R_{i,i',j,j'}$ and block $C$ be column $k$. For $A,B \delta C$ to be true, $[r,r'] \cap [k,k] \neq \emptyset$, i.e., $r \leq k \leq r'$ and $[c,c'] \cap [j,j'] = [u,u'] \neq \emptyset$; and $\exists L_{q,k}$ such that $[i,i'] \cap [q,q] \neq \emptyset$, i.e., $i \leq q \leq i'$. The fill-in property guarantees that if $[r,r'] \cap [k,k] \neq \emptyset$ then all the elements $L_{q,k}, i \leq q \leq i'$ are non-zero. Thus $\delta_{A,B,C} = (k,r',i,i',u,u')$. In Figure 11(f), the elements of $R_1$ which are in the row $k$ between the
vertical dotted lines combine with the entire shaded rectangle $R_2$ to update the elements between rows $r_3$ and $r_4$ in column $k$.

- **Two rectangles update triangle, $<\text{Tri}, \text{Update}, \text{Rect}, \text{Rect}>$**
  Let block $A$ be rectangle $R_{r,r',c,c'}$, block $B$ be rectangle $R_{i,i',j,j'}$ and block $C$ be triangle $T_{m,n}$. For $A, B \in C$ to be true, $[r, r'] \cap [m, n] = [v, v'] = \emptyset$; $[c, c'] \cap [j, j'] = [u, u'] = \emptyset$; and $[i, i'] \cap [m, n] = [t, t'] = \emptyset$. Then, $\delta_{A,B,C} = (v, v', t, t', u, u')$. In Figure 11(h), the shaded portion of $R_1$ combines with the entire shaded rectangle $R_2$ to update the shaded rectangular portion of $T$.

- **Two rectangle update rectangle, $<\text{Rect}, \text{Update}, \text{Rect}, \text{Rect}>$**
  Let block $A$ be rectangle $R_{r,r',c,c'}$, block $B$ be rectangle $R_{i,i',j,j'}$ and block $C$ be rectangle $R_{p,p',q,q'}$. For $A, B \in C$ to be true, $[r, r'] \cap [q, q'] = [v, v'] = \emptyset$ and $[c, c'] \cap [j, j'] = [u, u'] = \emptyset$. Then, $\delta_{A,B,C} = (v, v', t, t', u, u')$. In Figure 11(i), the shaded portion of $R_1$ combines with the shaded portion of $R_2$ to update the shaded part of $R_3$.

- **Triangle updates rectangle, $<\text{Rect}, \text{Update}, \text{Tri}>$**
  Let block $A$ be triangle $T_{m,n}$ and blocks $B$ and $C$ be rectangle $R_{r,r',c,c'}$. Then for $A, B \in C$ to be true, $[m, n] \cap [c, c'] = [v, v'] = \emptyset$. This implies that triangle and rectangle must belong to the same cluster and hence $[v, v'] = [c, n]$. Also, $[m, n] \cap [c, c'] = [u, u'] = \emptyset$, which by the same argument as above implies that $[u, u'] = [c, n]$. Finally, $HP(B) \cap HP(C) = \emptyset$ is trivially true by virtue of $B$ and $C$ being the same rectangle. $HP(B) = [r, r']$. We thus have $\delta_{A,B,C} = (c, n, r, r', c, n)$. In fact this can be effectively treated as a base update. In Figure 11(c), the shaded portion of $T$ updates the shaded portion of $R$.

Referring to Figure 10, suppose we want to compute the dependencies for the triangular block $T : (c_1, c_2)$. $HP(P_1) = [I_1, I_1']$ and $VP(T) = [c_1, c_2]$. $HP(P_1) \cap VP(T) = [c_1, c_2]$ and therefore, the base block $P_1$ updates the block $T$. The intersection also gives complete information about the portions of all the blocks involved in each update. In above case, the upper shaded extent of $P_1$ between rows $c_1$ and $c_2$ updates all of $T$. Now consider block $R_2$. $VP(R_2) = [c_1, c_3]$ and $HP(P_1) = [I_1, I_1']$, hence $VP(R_2) \cap HP(P_1) = [c_2, I_1']$. $VP(M_1)$ matches $VP(P_1)$. $HP(M_1)$ 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 10 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 $M_2$.

### 5.4 Computation of dependencies

In Section 5.3 we formulated rules, based on the intersection of projections, to compute the interblock dependencies and thus determine the different task types. In this section, we describe the algorithms and data structures used to compute these dependencies. In particular, we discuss the *interval tree* data structure which is used to compute the intersection of projections and is central to the efficient functioning of the partitioner.
Interval Tree

To construct the block dependency DAG, first the intersections of projections are computed. For this purpose, we make use of the interval tree data structure. The interval tree allows storage of $n$ intervals in linear space such that intersection queries can be answered in logarithmic time. Let $U \subseteq R$ be a finite set and let $S = \{ [x_i, y_i] : x_i \in U, y_i \in R, 1 \leq i \leq n \}$ be a set of $n$ closed intervals on the real line. Note that the left endpoints of the intervals in $S$ are in $U$. An interval tree for $S$ (with respect to the universe $U$) is a leaf-oriented search tree for set $U$ where each node of the tree stores a node list. The node list $NL(v)$ of a node $v$ is the set of intervals in $S$ containing the split value of $v$ but of no ancestor of $v$. The split value of a leaf representing $x_i$ is $i$. Let $\text{max}(y)$ be the maximum value of a leaf node in the subtree rooted at $y$ and $\text{min}(y)$ be the minimum value of a leaf node in the subtree rooted at $y$. The split value of a non-leaf node $v$ with left child $l(v)$ and right child $r(v)$ is $(\text{max}(l(v)) + \text{min}(r(v)))/2$. The node list of a node $v$ is stored as as two sequences: the ordered list of left endpoints and the ordered list of right endpoints.

Figure 12 shows an interval tree for the set of intervals

$$S = \{ [1, 8], [2, 5], [3, 7], [4, 5], [6, 8], [1, 3], [2, 3], [1, 2] \}$$

with respect to the universe $U = \{ 1, 2, 3, 4, 5, 6, 7 \}$. In that Figure, for example, the root node's split value is 4.5. The intervals stored at this node are all and only those intervals of $S$ which contain the split value. These intervals are $[1, 8], [2, 5], [3, 7]$ and $[4, 5]$, stored in two sequences: one sequence stores them in order of left endpoints, i.e., in the order $[1, 8], [2, 5], [3, 7], [4, 5]$ and the other sequence stores them in order of right endpoints, i.e., in the order $[2, 5], [4, 5], [3, 7], [1, 8]$. These sequences (represented by the left and right endpoints, respectively) are shown to the left and right of the root node in the figure.
Let $|U| = N$ and let $I = [x_0, y_0]$ be a query interval. Let $A = \{[x, y] \in S; [x, y] \cap [x_0, y_0] \neq \emptyset\}$ be the set of intervals in $S$ intersecting $I$. The following properties of the interval tree are important in making the computation of the intersections of projections, and hence the detection of dependencies efficient in space and time:

i. an interval tree for $S$ uses space $O(n + N)$;

ii. an interval tree for $S$ of depth $O(\log N)$ can be constructed in time $O(N + n \log N + n \log n)$;

iii. $A$ can be computed in time $O(\log N + |A|)$.

For a detailed description of the interval tree and its uses, refer to [11].

Grouping and storage of intervals

Clusters are partitioned and the dependencies involving blocks in the cluster are computed by proceeding left to right along the matrix, processing clusters one by one. When a cluster $C_{m,n}$ is taken up for processing, all the clusters to its left have been processed. Projections of the blocks to the left of $C_{m,n}$ which fall within the range $[m, n]$ have already been stored at the time of processing previous clusters, as explained in Section 5.3.1 above. An interval tree is then constructed with the universe $U$ being the set of integers in the closed interval $[m, n]$.

Before storing intervals in the tree, they are divided into groups to ensure that every reported intersection in a query on the tree results in at least one dependency. The updates to any unit block within a rectangular block in the cluster can be only of the base-multiplier type. Within the triangular block of a cluster, the updates to any rectangular unit block can be of either the base type or the base-multiplier type. To facilitate the computation of these updates, the blocks in $C_{m,n}$ are given group numbers as follows: triangle $T_{m,n}$ is numbered “-1” (for the base updates) and “0” (for the base-multiplier updates), followed by numbering the rectangular blocks starting with “1” and going top to bottom. In Figure 10, cluster $C$ has the following group numbering associated with blocks: “-1”, “0” (triangle), “1” (the rectangle composed of $R_1$, $R_2$ and $R_3$) and “2” (the rectangle composed of $R_4$, $R_5$ and $R_6$).

The projection intervals are then divided into groups. First, every base block’s projection is in group “-1”, corresponding to the update of some unit triangle(s) in $T_{m,n}$ by the base block. Next, the intervals corresponding to base-multiplier pairs are divided into groups depending on the updated block’s group number. The interval for a base-multiplier pair belongs to group “$i$”, $i \geq 0$ if the multiplier’s horizontal projection intersects the horizontal projection of block $i$ in $C_{m,n}$.

In Figure 10, consider the interval $[I_1, I_1']$ which has been stored in cluster $C_{m,n}$ as a result of a previous projection, and for which the projecting rectangle is $P_1$. First of all, the projection is in group “-1” ($P_1$ updates a unit triangle). The projection is also in group “0” (the unit rectangle ($c_2$, $c_3$, $c_1$, $c_2$) is updated by the upper shaded portion of $P_1$ as base and the lower shaded portion of $P_1$ as multiplier) and in group “1” ($M_1$’s horizontal projection intersects that of rectangle number 1). Thus the projection of $P_1$ is split into three intervals, one in each group, and the intervals in groups “1” and “2” are annotated with the respective multiplier information.
Each node in the interval tree stores $2g$ interval lists, where $g$ is the number of groups in the cluster. There are four groups in the cluster $(c_1, c_4)$ of Figure 10. Each group has two interval lists, one list in ascending order of left end-points and the other in descending order of right end-points. Hence, there will be eight interval lists at each node of the interval tree.

Procedure $Build\_interval\_tree$ lists the algorithm for grouping and storing the intervals in the interval tree.

\begin{verbatim}
procedure Build_interval_tree
1. for each updater u do
   if u is a column c then
      store interval $[c, c]$ in group “-1”
      $z' \leftarrow$ row number of first non-zero in c s.t. $z' \in [m, n]$
      $z \leftarrow$ row number of non-zero immediately following $z'$ in c
      do a binary search of Cl to find group g for z
      store interval $[c, c]$ in group g
      $r \leftarrow$ bottom row of matching rectangle or triangle
      while there are unprocessed non-zeros in c do
         do a binary search of c for first non-zero beyond row r
         $z \leftarrow$ row number of this non-zero
         do a binary search of Cl to find group g for z
         store interval $[r, r']$ in group g
      endwhile
   else
      $u$ is a rectangle updater $R$, $[r, r']$ is interval or projection
      store interval $[r, r']$ in group “-1”
   for each rectangle below $R$
      do a binary search of Cl to find group g for $R$
      store interval $[r, r']$ in group g
   endfor
endif
endfor
2. sort the stored intervals in ascending order with left endpoint
   as primary key and group number as secondary key
3. store the intervals in the interval tree
\end{verbatim}

The main loop of step 1 goes through all updaters of the cluster $C_{m,n}$ or $Cl$. The projection interval for each updater has been already stored with the cluster. If the updater is a column $c$, $z'$ is the non-zero with the least row number which is contained in the extent $[m, n]$. The column updates the unit triangle in $T_{m,n}$ which contains $L_{z', z'}$ and the interval $[c, c]$ is therefore stored in group “-1”. Then, starting with the first non-zero following $L_{z', c}$ in column $c$, non-zeros are matched against the rectangles in $C_{m,n}$ to identify groups. Suppose the non-zero $L_{z, c}$ matches with a rectangle in group $g$, whose bottom-most row number is $r$; $z$ is the least row number of a non-zero in $c$ for which such a match can occur. Then, the non-zeros $L_{i, c}, z \leq i \leq r$ are all used in updating this
rectangle and can be skipped over in looking for the next matching group. This is achieved by doing a binary search of $c$ for a non-zero with the least row number $> r$. If the updater is a rectangle $R$, it becomes the base block for each multiplier block below it. Each of these multiplier blocks is matched against some rectangle in $C_{m,n}$ by doing a binary search of the rectangles in $C_{m,n}$. Steps 2 and 3 build and store the intervals in the interval tree. For details on how to store the intervals in an interval tree, refer [11].

**Querying the interval tree**

When the dependencies for a unit block are to be constructed, the interval tree can be queried with (a) the query interval which is the vertical projection of the unit block and (b) the group number, $g$, which is the group number of the block to which the unit belongs. The intersections reported will be either a set of base updaters (if $g$ is “-1”) or a set of base-multiplier pairs (if $g$ is not “-1”). Since a unit rectangle needs a base-multiplier pair for its update, group number “-1” will be only used for the unit triangle.

When $g$ is “-1”, the base updaters returned will each update the unit triangle for which the query was made. Since the base updaters themselves may be partitioned into unit blocks, each of the unit blocks in each base updater will update the unit triangle. Thus every intersection reported results in at least one dependency. For other values of $g$, there is potential inefficiency if queries are handled naively. Suppose a query is made for a unit rectangle $R$ which belongs to a rectangular block $D$ whose group number is $g$. Let $S$ be the set of base-multiplier pairs returned. Corresponding to each pair of intervals in the intersection, there is a pair of base and multiplier blocks, $(S, M)$, such that $HP(S)$ intersects $VP(R)$, $VP(S)$ intersects $VP(M)$ and $HP(M)$ intersects $HP(D)$. The inefficiency arises in the situation where $HP(M)$ does not intersect $HP(R)$ and therefore the pair $(S, M)$ does not update $D$, resulting in a wasted output interval.

To overcome this inefficiency by guaranteeing that every intersection reported on a query results in at least one dependency, one query is made for each vertical strip in the triangle and each vertical strip in each of the rectangles. In fact, this also reduces the number of queries as compared to making one query for each unit block. Using one query for a vertical strip, dependencies for all the unit blocks in that strip can be computed efficiently.

Figure 13 shows a general query situation. Algorithm Query_interval_tree shows how the dependencies are computed for a given vertical strip.

---

**procedure** Query_interval_tree

1. Query the interval tree with $[c,c']$ and $g$
2. **if** $g$ = “-1” **then**
   - $S$ ← set of base updaters, $T$ ← query triangle
   - **for each** $U \in S$ **do** Base_deps($U, T$) **endfor**
   **else**
   - $S$ ← set of base-multiplier pairs of updaters, $V$ ← vertical strip of query
   - **for each** $(U, M) \in S$ **do** Base_mult_deps($U, M, V$) **endfor**
3. **endif**
In step 1, a query is made on the interval tree with the appropriate group number. If the group number is “-1”, the query returns a set of base updaters for the unit triangle at the top of the vertical strip. In step 2, $S$ is the set of base updaters which correspond to the intervals of intersection. Each base updater block $U$ may be itself partitioned into unit blocks (if $B$ is not a column). In this case, the procedure $Base\_deps$ sets up dependencies $B \delta T$ for all unit blocks $B \in U$ for which $HP(B)$ intersects $HP(T)$.

---

**procedure** $Base\_deps$

0. $T \leftarrow$ input triangle  
1. **if** $U$ is a column **then** $U \delta T$  
   **else**
   2. compute $S$, the topmost horizontal strip in $U$ for which $HP(S) \cap HP(T) \neq \emptyset$
   3. **while** $(HP(S) \cap HP(T) \neq \emptyset)$ **do**
      **for** each $B \in S$ **do** $B \delta T$ **endfor**
      $S \leftarrow$ next horizontal strip after $S$
   **endwhile**
   **endif**

For example, in Figure 13, the block containing the $P_1$’s updates the unit triangle $T$. The topmost horizontal strip in this block whose horizontal projection intersects $HP(T)$ is the second horizontal strip i.e. the one which is partitioned into $P_1$, $P_2$. This strip gives rise to the dependencies $P_1 \delta T$ and $P_2 \delta T$. The next horizontal strip also updates $T$, and gives rise to the dependencies $P_3 \delta T$ and $P_4 \delta T$.

Before we discuss how the dependencies are computed for group numbers which are not “-1”, we
describe a procedure which we refer to as $\text{Merge\_scan}$ that is repeatedly used in the computation of these dependencies. This procedure takes two vertical (resp. horizontal) strips of blocks and returns a set of pairs of blocks whose horizontal (resp. vertical) projections intersect each other. Let $S_1$ and $S_2$ be two strips, with extents $(e_1, e_1')$ and $(e_2, e_2')$ respectively. If $S_1$ and $S_2$ are horizontal strips, $e_1$ and $e_2$ are the leftmost edges and $e_1'$ and $e_2'$ are the rightmost edges; if they are vertical strips, $e_1$ and $e_2$ are the topmost edges and $e_1'$ and $e_2'$ are the bottommost edges. If one of $S_1$ or $S_2$ (say $S_1$) is a part of a column then $e_1$ is the row number of the first non-zero and $e_1'$ is the row number of the last non-zero in the part under consideration. Each of $S_1$ and $S_2$ (if they are not columns) has been partitioned into unit blocks.

The procedure essentially scans the strips in parallel, much like the algorithm for merging two sorted lists of values. The strips are matched against each other starting at $e_1$ and $e_2$ respectively. One pointer is maintained for each strip, pointing to the current unit block in the scan of that strip. The pointers are initially respectively set to $p_1 = b_{11}$ and $p_2 = b_{21}$, where $b_{11}$ and $b_{21}$ are the first unit blocks in $S_1$ and $S_2$ respectively whose projections intersect each other. This initial pointer setting can be done in constant time. The loop invariant ensures that the projections of the respective blocks to which the pointers point intersect each other.

Suppose in the $i$th iteration, the pointers are at $p_1 = b_{1i}$ and $p_2 = b_{2i}$. Then the pair $(b_{1i}, b_{2i})$ is reported. Next, there are three cases for advancing the respective pointers. We will describe these cases assuming horizontal strips; the discussion is symmetric for vertical strips. Let $r_1$ and $r_2$ be the locations of the right edges of $b_{1i}$ and $b_{2i}$ respectively.

- a) If $r_1 < r_2$, $p_1$ is advanced to the unit block immediately to the right of $b_{1i}$ in $S_1$;
- b) If $r_1 > r_2$, $p_2$ is advanced to the unit block immediately to the right of $b_{2i}$ in $S_2$;
- c) If $r_1 = r_2$, $p_1$ and $p_2$ are both advanced to respectively point to the unit block immediately to the right of $b_{1i}$ in $S_1$ and the unit block immediately to the right of $b_{2i}$ in $S_2$.

The scan ends when both pointers are advanced beyond $e'$, where $e'$ is the minimum of $e_1'$ and $e_2'$. Obviously, the pairs of blocks are reported in time linear in the total number of pairs reported.

If one of the strips is a column, a unit block is replaced by a row extent. Suppose $S_2$ is a column. Let $(b_1, b_2)$ be one of the reported pairs. Then $b_1$ is essentially the strip of non-zeros in the column such that the row numbers of the first and last non-zeros are within the top and bottom edges of $b_2$, respectively. ($S_2$ must be a vertical strip). Also, one or both of $S_1$ and $S_2$ could be a strip of strips, i.e., instead of being comprised of unit blocks, they could be comprised of strips of unit blocks. For instance $S_1$ could be a vertical strip, with each horizontal partition in it being a horizontal strip instead of a unit block, and $S_2$ could be a vertical strip. A pair $(b, b')$ reported by $\text{Merge\_scan}$ would mean that the horizontal projection of the horizontal strip $b$ intersects the horizontal projection of the unit block $b'$.

If the group number is other than "-1", the query returns a set of base-multiplier updater pairs. For each such base-multiplier pair, the procedure $\text{Base\_mult\_deps}$ is called to compute the dependencies for the vertical strip.
procedure $\text{Base\_mult\_deps}$

1. if $U = M$ is a column then
   
   $P \leftarrow \text{Merge\_scan}(U, V)$
   for each $(b, b') \in P$ do $U \delta b'$ endfor
   
   else

2. compute $S_u$, the topmost horizontal strip in $U$ for which $HP(S_u) \cap VP(V) \neq \emptyset$

3. while $(HP(S_u) \cap VP(V) \neq \emptyset)$ do
   
   $P_u \leftarrow \text{Merge\_scan}(M, V)$
   for each $(s_m, B) \in P_u$ do
     
     $P_a \leftarrow \text{Merge\_scan}(S_u, s_m)$
     
     for each $(b, b') \in P_a$ do $b, b' \delta B$ endfor
   
   endfor

   $S_u \leftarrow$ next horizontal strip after $S_u$ in $U$

   endwhile

endif

The working of this procedure is best described by means of an example. In Figure 13, suppose we want to compute the dependencies for the vertical strip which contains $R$. On doing a query, let one of the pairs of base-multipliers returned be the rectangle containing the $P_i$'s and the rectangle containing the $M_i$'s respectively.

In step 2 of procedure $\text{Base\_mult\_deps}$, $S_u$ corresponds to the strip containing $P_1$ and $P_2$. In step 3, one of the pairs returned in $P_3$ is $(Q, R)$, where $Q$ is the horizontal strip containing $M_1$, $M_2$, $M_3$. The corresponding $P_a$ contains the pairs $(P_1, M_4)$, $(P_1, M_5)$, $(P_2, M_5)$, $(P_2, M_3)$, all of which update $R$. Another pair in $P_4$ is $(Q', R)$ where $Q'$ is the horizontal strip containing $M_4$, $M_5$, $M_6$. The corresponding $P_a$ contains the pairs $(P_1, M_4)$, $(P_1, M_5)$, $(P_2, M_5)$, $(P_2, M_6)$ which update $R$.

In the next iteration of the while loop, $S_u$ corresponds to the strip containing $P_3$ and $P_4$. By a similar process in the for loop of statement 3, the pairs $(P_5, M_1)$, $(P_3, M_2)$, $(P_3, M_4)$, $(P_3, M_5)$, and $(P_4, M_2)$, $(P_4, M_3)$, $(P_4, M_5)$ are identified and all of these update $R$.

Dependencies internal to cluster

The discussion above described how dependencies of the type $A \delta C$ or $A, B \delta C$ are computed, where $C$ is in a cluster and $A$ and $B$ are external to the cluster to which $C$ belongs. We now describe how dependencies between blocks entirely within a cluster are computed. These dependencies are computed without using the interval tree.

Consider first the triangle $T_{m,n}$ in a cluster $C_{m,n}$. Let some unit triangle $B$ in $T_{m,n}$ belong to the horizontal strip $H$ in $T_{m,n}$. Then, $A \delta B$ for all unit rectangles $R$ such that $R \in H$. This is computed in step 1 of the procedure $\text{Internal\_tri\_deps}$.
procedure Internal_tri_deps

1. for each unit triangle $T$ do
   
   $H \leftarrow$ horizontal strip to which $T$ belongs
   
   for each unit rectangle $R$ in $H$ do $R \in T$ endfor

endfor

2. for each horizontal strip $H$ do
   
   for each unit rectangle $R$ in $H$ do
       
       $V \leftarrow$ vertical strip to which $R$ belongs

       $T \leftarrow$ unit triangle contained in $V$

       $T \in R$

       $H_t \leftarrow$ horizontal strip to which $T$ belongs

       for each unit rectangle $R_m$ to the left of $R$ in $H$
           
           $R_s \leftarrow$ unit rectangle in $R_m$'s vertical strip and in $H_t$

           $R_s \in R_m \in R$

       endfor

   endfor

endfor

To compute the dependencies for unit rectangles in $T_{m,n}$, let $R$ be one such unit rectangle in $T_{m,n}$, and let $R$ belong to the vertical strip $V$ and horizontal strip $H$. Then, $A \in R$ where $A$ is the unit triangle in $V$. Let $A$ belong to the horizontal strip $H_a$. For every vertical strip $V_i$ to the left of $V$, $B, C \in R$ where $B \in H_a$, $B \in V_i$ and $C \in H, C \in V_i$. The dependencies for $R$ are computed in step 2 of the procedure Internal_tri_deps.

Next, consider the rectangles below $T_{m,n}$. For each of these rectangles, the following dependencies are computed. Let $R$ be a unit rectangle which belongs to the vertical strip $V$ and horizontal strip $H$ in some rectangle below $T_{m,n}$. Let $H_s$ be a horizontal strip in the triangle $T_{m,n}$ such that $HP(H_s) \cap VP(V) \neq \emptyset$. Then, $A, B \in R$ where $A \in H_s, B \in H$ and $VP(B) \cap VP(A) \neq \emptyset$. $B$ is in some vertical partition to the left of $V$ in the rectangle.

procedure Internal_rect_deps

1. compute $S_u$, the topmost horizontal strip in $T$ for which $HP(S_u) \cap VP(V) \neq \emptyset$

2. while $HP(S_u) \cap VP(V) \neq \emptyset$ do

3. for each unit rectangle $R \in V$ do

   $H \leftarrow$ part to the left of $R$ of the horizontal strip to which $R$ belongs

   $P_a \leftarrow$ Merge_scan($S_a, H$)

   for each $(b, b') \in P_a$ do $b, b' \in R$ endfor

endfor

$S_u \leftarrow$ next horizontal strip after $S_u$ in $U$

endwhile
Procedure *Internal\_rect\_deps* gives the algorithm to compute these dependencies. \( S_u \) in step 1 can be computed in constant time since the sizes of the partitions in the triangle and rectangle are known.

Procedures *Process\_column* and *Process\_cluster* are the top-level partitioning routines which tie everything together. Columns and clusters are “processed” going left to right in the matrix. When a column is encountered in this left-to-right scan, the procedure *process\_column* is called to process it; when a cluster is encountered, the procedure *process\_cluster* is called. The data structure *ColTree* is a dynamic balanced binary search tree, which stores those columns which depend on at least one block, but which have not yet been processed. A column is independent if it has no updaters. The data structure *ClipTree* is an array which stores the clusters in left to right order of their column extents.

```
procedure Process\_column

if c is not an independent column then
  delete c from ColTree
  for each updater u in c’s list of updaters do u δ c endfor
endif

for each non-zero in row j in column c do
  do a binary search of ColTree for j
  if (match is found) then
    Cl ← matching cluster
    store interval \([j, j]\) in Cl
  else
    do a binary search of ColTree for j
    if found then
      add c as j’s updater
    else
      insert j (with c in it’s updater list) into ColTree
    endif
  endif
endfor

procedure Process\_cluster

1. call Build\_interval\_tree

2. \( W \leftarrow \) work in triangle, \( S \leftarrow \) size of triangle
   \( Z \leftarrow \) minimum non-zeros, \( A \leftarrow \) minimum work
   \( P \leftarrow S/Z \) or \( P \leftarrow W/A \)
   call Partition\_triangle

3. for each vertical partition in the triangle do
   call Query\_interval\_tree, group “1”
   call Query\_interval\_tree, group “0”
```
call Internal_tree.deps
endfor

4. for each rectangle $R$ below triangle do
   $g \leftarrow g + 1$
   $W \leftarrow$ work in $R$, $S \leftarrow$ size of $R$
   $P \leftarrow S/Z$ or $P \leftarrow W/A$
   call Partition_rectangle
   for each vertical partition in $R$ do
      call Query_interval_tree, group $g$
      call Internal_rect.deps
   endfor
endfor

5. do a binary search of ColTree to find match with $R$
   if $R$ projects to a set of columns $S$ then
      for each column $c \in S$ do
         do a binary search of $c$ in ColTree
         if found then
            add $R$ as $c$'s updater
         else
            insert $c$ (with $R$ as an updater) in ColTree
         endif
      endfor
   endif
else
   $S \leftarrow$ set of clusters matching $R$'s projection
   for each cluster $C \in S$ add $R$ as updater endif
endfor

5.5 Block and cluster dependency DAGs

The partitioner produces two output directed acyclic graphs (DAGs), the block dependency DAG and the cluster dependency DAG. In the following, we capture the main properties of these two DAGs, leaving out the details.

Block dependency DAG

Informally, the block dependency DAG (BDAG) is a DAG that describes the dependencies among the unit partitions or blocks. The BDAG differs from the task dependency DAG commonly used in the literature to represent computations with precedence relationships. The following features characterize the block dependency DAG:

i. Each node $n_i$ in a BDAG represents a block $B_i$. The node is not a computational task.

ii. The weight on each node $n_i$ is a measure of the total computational work associated with the corresponding block $B_i$. In other words, this weight is a measure of the sum of the computations performed during the updating and scaling of $B_i$. 

33
iii. In a BDAG, an update task of the form $< B_i, Update, B_j >$ is represented by the directed edge $(n_j, n_i)$. An update task of the form $< B_i, Update, B_j, B_k >$ is represented by the pair of edges $(n_j, n_i)$ and $(n_k, n_i)$. In the latter case, the edge $(n_j, n_i)$ (respectively $(n_k, n_i)$) is annotated with the information that block $B_k$ (respectively $B_j$) is paired with block $B_j$ (respectively $B_k$) to update $B_i$. We call this the pairing information.

iv. Each edge $(n_j, n_i)$ carries all the information needed to determine exactly what portion of $B_j$ is used to update $B_i$. This information is also necessary to schedule the communication arising from the resultant flow of data from $B_j$ to $B_i$.

v. Scaling tasks are not explicitly represented in a BDAG. A column or a triangle is scaled once all the updates to it have been executed. Scaling a triangle is the same as the dense Cholesky factorization of that triangle. A rectangle does not have a scaling task associated with it. The scaling occurs implicitly during the execution of one or more tasks of the type $< Rect, Update, Tri >$ which update that rectangle.

vi. For a columnar or a triangular block, a BDAG does not impose any precedence order among the tasks which update that block. For a rectangular block, there is a weak restriction on the order in which some of the update tasks are to be performed.

Cluster dependency DAG

The cluster dependency DAG (CDAG) can be considered to be a “supergraph” of the block dependency DAG. It is obtained from the BDAG by collapsing all nodes that belong to the same cluster, into a “supernode”. All the edges among the nodes which are collapsed to form a supernode are eliminated. The remaining edges in the BDAG form the directed edges among the supernodes; all duplicate edges between any pair of supernodes are merged. Each supernode in the CDAG is annotated with a weight equal to the total workload in the cluster represented by that supernode. This is the sum of the weights of all the nodes that collapsed to form that supernode. Each directed edge in the CDAG is annotated with a measure of the volume of data communicated from the source cluster supernode to the dependent cluster supernode, as a result of the dependencies from the unit blocks of the source cluster to the unit blocks of the dependent cluster.

6 Allocation

The allocation of unit partitions to processors proceeds in a cluster by cluster manner. We describe two allocation strategies. These differ in the order in which the clusters are chosen for allocation. In the following, we first describe the allocation policy for blocks within a given cluster and then describe the two strategies used for ordering the selection of clusters.

Allocation of cluster

If a cluster $C_l$ is not a single column, processors are assigned to the component partitions using the procedure $Allocate\_cluster$, with the aim of balancing the load among processors. In that procedure, $P$ is the total number of processors available. A priority queue, $Pheap$, stores all processors, with the highest priority given to the processor which has the least amount of computational work
procedure allocate_cluster

0. \( k \leftarrow \text{number of partitions in triangle, } r \leftarrow \min(P, k), \ CHeap \leftarrow \text{empty} \)

1. \( PA \leftarrow \text{top } r \text{ processors from } Pheap \text{ in heap order} \)
   delete these from \( Pheap \) and insert into \( CHeap \)

2. for each unit block in triangle
   \( p' \leftarrow \text{top of } CHeap \)
   assign \( p' \) to the block
   increase the key for \( p' \) in \( CHeap \) with work in block and reheap \( CHeap \)
  endfor

3. for each rectangle below the triangle
   for each unit block in rectangle
   \( p' \leftarrow \text{top of } CHeap \)
   assign \( p' \) to the block
   increase the key of \( p' \) in \( CHeap \) with work in block and reheap \( CHeap \)
  endfor
  endfor

4. while \( CHeap \) is not empty do
   \( p' \leftarrow \text{next processor in } CHeap \)
   delete \( p' \) from \( CHeap \)
   insert \( p' \) into \( Pheap \) with current work
  endwhile

assigned so far. Let \( k \) be the number of partitions in \( CT \)'s triangle. Let \( r \) be \( \min(P, k) \). Then the top \( r \) processors in \( Pheap \), i.e., those with the least computational work, are picked for assignment to the unit blocks in the cluster. Let the set of these processors be \( PA \). The processors in \( PA \) are maintained in a temporary priority queue, \( CHeap \), with the same priority criterion as for \( Pheap \). Using \( CHeap \), first, the unit blocks in the triangle are allocated by going through the unit triangles first and then going through the unit rectangles top to bottom, and left to right along each horizontal strip, i.e., in block row major order. For every unit, the processor in \( PA \) with current minimum load is assigned to it. Next, for each rectangle below the triangle, the unit blocks are traversed in block row major order, and for every unit block, the processor in \( PA \) with current minimum workload is assigned to it.

Selection of cluster and allocation of column

As mentioned earlier, the clusters may be selected for allocation in two ways. In the first, which is a simpler version, the clusters are sorted by workload in a decreasing order and then are picked for allocation in this order. If the cluster is a column, then it is assigned the processor with current minimum work load.

In the second version, procedure \texttt{GlobalAllocate} is followed.

A cluster is \texttt{ready} for allocation only after all its predecessors in the CDAG have been allocated.
procedure global_allocate

0. Insert all ready clusters (no predecessors) into Clheap

1. while Clheap is not empty
   
   $C_l \leftarrow$ cluster at top of Clheap, delete top of Clheap
   
   if ($C_l$ is a column) then allocate_column
   
   else allocate_cluster
   
   endif
   
   for each child $C_i$ of $C_l$ in the CDAG
   
   decrement predecessor count $\text{pred}(i)$ by 1
   
   if ($\text{pred}(i) = 0$)
   
   insert $C_i$ in Clheap
   
   endif
   
   endfor
   
endwhile

processors. A heap, Clheap, maintains a priority queue for the ready clusters, in descending order of work in the clusters. Along with Pheap, it works towards balancing the workload among the processors. In each iteration of the allocation process, the cluster, $C_l$, at the top of ClHeap (i.e., the ready cluster with maximum work) is picked next for allocation. After allocation, the new ready clusters are inserted in Clheap.

In the above scheme, if $C_l$ is a column, the procedure process_column identifies the predecessor $C_r$ which has maximum communication with $C_l$. If $C_r$ is a column, the same processor $p$ that is assigned to $C_r$ is assigned to $C_l$. Otherwise, if $C_r$ is a cluster, one processor is arbitrarily picked from the processors that are assigned to $C_r$, and is assigned to $C_l$.

The two strategies presented here have advantages and disadvantages. Among other parameters, the structure of $L$ affects the performance of two strategies presented here. We leave the quantification of this effect to future study. The performance results presented in Section 9 use the first version where prior to allocation the clusters are sorted in decreasing order of workload.

7 Scheduling

The scheduler is a parallel program which takes as input the block dependency DAG, the mapping of blocks to processors and the symbolic factor. The scheduler 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. We refer to the scheduler as a symbolic scheduler because it traverses the block dependency DAG without actually executing the numerical computations. For the generated task schedule, the communication profile is the exact schedule of communication on each processor. 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 runtime communication overheads during the numerical factorization phase. In [15] we describe how the communication profile can be used to improve the performance of sparse column Cholesky factorization. The results in this report, however, do not utilize the communication profile.
As pointed out above, on each processor, the symbolic scheduler traverses the same block dependency DAG as the actual numerical factorization step. Tasks are 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. The task schedule reported by the scheduler is the schedule obtained by dynamically traversing the block dependency DAG. During the actual numerical factorization, the same schedule is assumed so that the tasks are executed in a predetermined order. This avoids the run-time overheads associated with dynamic scheduling in the numerical factorization step. In Section 9, where we present performance results, these savings are found to be substantial indicating the cost effectiveness of the symbolic scheduling phase.

In the following, we describe the details of the block symbolic scheduler. We also describe a symbolic scheduler for the column version. In Section 9, a comparison is made between the performance of the block and column Cholesky factorization algorithms, implemented using these two scheduling policies, respectively.

7.1 Block symbolic scheduler

Figure 14 is the pseudo-code for the parallel block symbolic scheduler. Each processor is assigned the list of tasks generated by the allocator. Using the block dependency DAG, the scheduler traces a parallel version of the sequential block code for Cholesky factorization, given in Section 5.1. The pseudo-code omits several details of data structures and only presents a top-level view. The following data structures are used for scheduling: (a) \textit{scale}$_q$ : a queue of blocks to be scaled; (b) \textit{base\_update}$_q$ : a queue of local blocks to be used in updating other local blocks; (c) \textit{base\_mult\_update}$_q$ : a queue of (base\_multiplier) block pairs to be used in updating local blocks; (d) \textit{ready}$_q$ : a queue of blocks ready to be sent off; (e) \textit{wait\_list} : list of blocks that are waiting for one or more blocks to pair with for base-multiplier updates. The variable \textit{myblk}$_s$ is the number of blocks assigned to the processor. A block is \textit{independent} if it has no updaters. The output of the symbolic scheduler is the task sequence, \textit{task}$_q$ for each processor.

The updates are divided into base updates and base-multiplier updates, as described in Section 5. The computations associated with the scaling and update tasks are not carried out - they are simply added to the output \textit{task}$_q$ at the appropriate time. Blocks are sent off to the processors in the order in which they become ready to be sent. In every iteration of the outer loop, greater priority is given to the clearing of all messages - notice that once a message is received, execution proceeds only after all waiting messages are cleared. This goes toward preventing the message buffers from getting choked up with too many waiting messages. Although not shown, it is possible to assign different priorities to the base update tasks and the base-multiplier update tasks. If there are many more blocks than columns, there are a larger number of base-multiplier type updates. It may then be beneficial to favor these updates over the single base updates. On the other hand, if the symbolic factor is extremely sparse, there are a greater number of single columns, and hence a greater number of column-updates-column (base update) type of tasks. In this case, it may be advantageous to give a higher priority to these than to the base-multiplier update tasks. All such priorities can be incorporated in the scheduler and the resulting schedule can be used without any modifications to the performing numerical factorization code.
done ← 0, task_q ← empty
for each independent block \( k \) do add \( k \) to scale_q, ready_q endfor

while (done < myblks) do
1. \( k \) ← block at head of scale_q
   delete \( k \) from scale_q, add \( < k, \text{Scale} > \) to task_q

2. \( k \) ← block at head of ready_q
   send \( k \) to processors that have dependent blocks, done ← done + 1
   if (\( k \) is a base used to update local blocks) add \( k \) to base_update_q endif
   if (\( k \) pairs with \( j \) to update local blocks and \( j \) is in wait_list)
      add \( (k, j) \) to base_mult_update_q
   else add \( k \) to wait_list endif
endfor

3. \( k \) ← block at head of base_update_q
   delete \( k \) from base_update_q
   for each local block \( j \) that is updateable by \( k \) do
      add \( < j, \text{Update}, k > \) to task_q
      if (all updates to \( j \) are done) add \( j \) to scale_q, ready_q endif
   endfor
endwhile

4. while (base_mult_update_q is not empty) do
   \( (k, j) \) ← pair at head of base_mult_update_q
   delete \( (k, j) \) from base_mult_update_q
   for each local block \( i \) that is updateable by \( (k, j) \) do
      add \( < i, \text{Update}, k, j > \) to task_q
      if (all updates to \( i \) are done) add \( i \) to scale_q, ready_q endif
   endfor
endwhile

Output task_q

\textbf{Figure 14}
\textit{Parallel block symbolic scheduler.}
7.2 Column symbolic scheduler

To compare the performance of the block partition with the column partition, we developed a parallel column symbolic scheduler. This scheduler is analogous to the block symbolic scheduler given above. Figure 15 is the pseudo-code for the parallel column symbolic scheduler. This code is

\begin{verbatim}
for each independent column k do add k to done_q endfor, task_q ← empty

while (done_q is not empty or some column has not been factored) do

1. k ← column at head of done_q
   delete k from done_q
   add cdinv(k) to task_q
   add k to update_q
   send k to processors which have dependent columns

2. k ← column at head of update_q
   delete k from update_q
   for each local column j which is updateable by k do
      add cmod(j,k) to task_q
      if (all updates to j are done) add j to done_q endif
   endfor

3. while (there is a column k in the message buffer) do
   receive column k from message buffer
   for each local column j which is updateable by k do
      add cmod(j,k) to task_q
      if (all updates to j are done) add j to done_q endif
   endfor

endwhile

Output task_q
\end{verbatim}

Figure 15

Parallel column symbolic scheduler.

a version of the fan-out algorithm for parallel distributed sparse Cholesky factorization in [5], with some differences in the way the computation and communication on a processor are interleaved. A version of this code also appears in [15]. There are two main data structures used in the scheduling process: (a) done_q: a queue of columns to which all updates have been done and (b) update_q: a queue of columns which are to be used to update local columns. The input to the scheduler is the symbolic factor and the column-to-processor mapping and the output is the sequence of tasks, task_q for each processor. The main while loop cycles until all local columns are factored and sent off to the processors which own the dependent columns. The work inside the loop is divided into scaling or cdinv tasks and update or cmod tasks. The computations associated with these tasks are not carried out - they are simply added to the output task_q at the appropriate time. Again, like the block version, once a message arrives at a processor, all messages or incoming columns are cleared before continuing with other local work.
8 Numerical Factorization

In the previous four sections, we have described in detail the functioning and capabilities of SHAPE. The partitions and the task schedule generated by SHAPE are used in computing the actual numerical factor, $L$. To perform the sparse factorization in parallel, the block sparse Cholesky algorithm presented in Section 5.1 must be suitably modified. In this section, we present the parallel block sparse Cholesky algorithm that uses the partitions and the task schedule produced by SHAPE. Performance results of this algorithm from the iPSc/860 are presented in the next section.

To bring out the performance gains realized by subscribing to the hybrid partitioning methodology of SHAPE, in the next section, we also show the performance of a Cholesky factorization algorithm that uses pure column-based partitioning and scheduling methods. The column-based Cholesky algorithm used here is similar to the one presented in [10]. To present a fair comparison, we have modified that algorithm so that the tasks on each processor are scheduled ahead of the numerical factorization using a symbolic scheduler, just as is done in the case of block sparse Cholesky factorization.

The two algorithms are presented in the following. Each of these codes takes as input the task sequence $(\text{task}_q)$ obtained by running the appropriate symbolic scheduler.

```
while (\text{task}_q \text{ is not empty}) \text{ do }
\quad T \leftarrow \text{ task at head of } \text{task}_q
\quad \text{delete } T \text{ from } \text{task}_q
\quad \text{if } (T \text{ is } \text{cdiv}(k))
\quad \quad \text{cdiv}(k)
\quad \quad \text{send}_{\text{out}}(k)
\quad \text{else } \{ T \text{ is } \text{cmmod}(j,k) \}
\quad \quad \text{if } (k \text{ is not available locally})
\quad \quad \quad \text{wait for } k \text{ to arrive}
\quad \text{endif}
\quad \text{receive}(k)
\quad \text{cmmod}(j,k)
\text{endwhile}

(a) Parallel column code
```

```
\text{while } (\text{task}_q \text{ is not empty}) \text{ do }
\quad T \leftarrow \text{ task at head of } \text{task}_q
\quad \text{delete } T \text{ from } \text{task}_q
\quad \text{if } (T \text{ is } < B, \text{Scale }>)
\quad \quad \text{scale}(B)
\quad \quad \text{send}_{\text{out}}(B)
\quad \text{else if } (T \text{ is } < B, \text{Update }, A >)
\quad \quad \text{if } (A \text{ is not available locally})
\quad \quad \quad \text{wait for } A \text{ to arrive}
\quad \quad \text{endif}
\quad \text{receive}(A)
\quad < B, \text{Update }, A >
\quad \text{else if } (T \text{ is } < C, \text{Update }, A, B >)
\quad \quad \text{if } (A \text{ is not available locally})
\quad \quad \quad \text{wait for } A \text{ to arrive}
\quad \quad \text{endif}
\quad \text{receive}(A)
\quad \text{if } (B \text{ is not available locally})
\quad \quad \text{wait for } B \text{ to arrive}
\quad \text{endif}
\quad \text{receive}(B)
\quad < C, \text{Update }, A, B >
\text{endif}
\text{endwhile}

(b) Parallel block code
```

To get an insight into the behavior of the two codes, we timed the execution of various computation and communication sections on the iPSc/860. The computation time on a processor is the sum of the times spent in the update and scaling tasks, and the task scheduling overheads. The idle
time on a processor is the sum of the times spent in waiting for off-processor data to arrive. The communication time on a processor is the sum of the times spent in the send’s and receive’s. The total factorization time on a processor is the sum of the computation, communication and idle times. The parallel time for factorization is the factorization time on the processor which finished last. The timings on 8 and 16 processors are tabulated in the next section.

9 Preliminary performance results

In this section, we compare the performance of the two versions of the Cholesky factorization algorithm described in Section 8. The first version is the parallel block sparse Cholesky algorithm that uses the partitions and the task schedule generated by SHAPE. The other version implements the most commonly used column-based Cholesky algorithm. We refer to the former version as the block algorithm and the latter version as the column algorithm. The performance comparison is made in terms of the execution time for computing the numerical factor, the communication cost, the processor idle times, and also the extra cost of using SHAPE for partitioning and scheduling tasks. All the performance results, including those for SHAPE, 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.

The results presented here are preliminary, in the sense that we have devoted very little effort in optimizing any of the our implementations to the cpu architecture. The intent of presenting these results is to show the gains in performance obtained by using the hybrid partitioning methodology for sparse factorization and also to show that the extra cost of using SHAPE is acceptable for a variety of sparse matrices.

All timings reported in this section are in milliseconds.

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 five cases are from the Harwell-Boeing sparse matrix collection [4]. These matrices are highly unstructured. The remaining four test cases arise in structured 2-D and 3-D grid problems. The entry “nonzeros in factor” refers to the number of nonzeros in the lower triangular factor including the elements on the diagonal. All these matrices were reordered using Liu’s modified multiple minimum degree ordering scheme [9]. We used SPARSKIT [13] and the Wisconsin Sparse Matrix Manipulation System [1] for generating and converting the test matrices into various formats, and for ordering and symbolically factoring the matrices.

9.2 Performance of numerical factorization

For each test matrix, for both the block and column cases, we first ran the appropriate symbolic scheduler immediately followed by the numerical factorization code. The tests were conducted on 8 and 16 processors. In this sub-section, we present the timings for the numerical factorization step. All the block cases were partitioned with the partitioning parameters minimum cluster width \( W \) set to 15, triangle grain size \( T \) set to 200 and rectangle grain size \( R \) set to 200. Unless otherwise stated, all execution times reported are the maximum over all the processor execution times.
<table>
<thead>
<tr>
<th>Matrix</th>
<th>Description</th>
<th>Order</th>
<th>Nonzeros in factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCSSTK13</td>
<td>Fluid flow problem</td>
<td>2003</td>
<td>265575</td>
</tr>
<tr>
<td>BCSSTK15</td>
<td>Module of an offshore problem</td>
<td>3948</td>
<td>637680</td>
</tr>
<tr>
<td>BCSSTK16</td>
<td>Corps of Engineers dam</td>
<td>4884</td>
<td>749027</td>
</tr>
<tr>
<td>BCSSTK24</td>
<td>Calgary Olympics Saddledome Arena</td>
<td>3562</td>
<td>278922</td>
</tr>
<tr>
<td>BCSSTK28</td>
<td>Solid element model, linear statics</td>
<td>4410</td>
<td>352261</td>
</tr>
<tr>
<td>L5PT127</td>
<td>5-point Laplacian discretization</td>
<td>16129</td>
<td>390643</td>
</tr>
<tr>
<td>L7PT16</td>
<td>7-point Laplacian discretization</td>
<td>4096</td>
<td>290195</td>
</tr>
<tr>
<td>L9PT100</td>
<td>9-point Laplacian discretization</td>
<td>10000</td>
<td>309940</td>
</tr>
<tr>
<td>L9PT127</td>
<td>9-point Laplacian discretization</td>
<td>16129</td>
<td>544785</td>
</tr>
</tbody>
</table>

**Table 2**
Test matrices.

Table 3 compares the block and column timings for 8 processors.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Column</th>
<th>Block</th>
<th>$\eta_{fact}$</th>
<th>$\eta_{comp}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fact</td>
<td>Comp</td>
<td>Idle</td>
<td>Fact</td>
</tr>
<tr>
<td>BCS13</td>
<td>7966</td>
<td>5852</td>
<td>338</td>
<td>4648</td>
</tr>
<tr>
<td>BCS15</td>
<td>21451</td>
<td>16548</td>
<td>881</td>
<td>9900</td>
</tr>
<tr>
<td>BCS16</td>
<td>21593</td>
<td>15884</td>
<td>870</td>
<td>10451</td>
</tr>
<tr>
<td>BCS24</td>
<td>6334</td>
<td>3637</td>
<td>375</td>
<td>3921</td>
</tr>
<tr>
<td>BCS28</td>
<td>8070</td>
<td>4316</td>
<td>706</td>
<td>4837</td>
</tr>
<tr>
<td>L5PT127</td>
<td>9830</td>
<td>3923</td>
<td>2330</td>
<td>7916</td>
</tr>
<tr>
<td>L7PT16</td>
<td>9008</td>
<td>6462</td>
<td>592</td>
<td>5571</td>
</tr>
<tr>
<td>L9PT100</td>
<td>7626</td>
<td>2648</td>
<td>1773</td>
<td>6264</td>
</tr>
<tr>
<td>L9PT127</td>
<td>13848</td>
<td>5407</td>
<td>3208</td>
<td>10405</td>
</tr>
</tbody>
</table>

**Table 3**
Execution time in milliseconds for Cholesky factorization on 8 processors.

In the table, *Fact* is the total time taken to factor the input sparse matrix; *Comp* is the average time per processor spent on actual numerical computations and on task scheduling; *Idle* is the
average per-processor idle time spent in waiting for data to arrive from other processors. The total

time, Fact, includes times reported in Comp, Idle, and the time spent in communicating data. The
column $\eta_{fact}$ is the percentage improvement in the factorization time of the block code over the
column code, computed as:

$$\eta_{fact} = \frac{Fact_{\text{column}} - Fact_{\text{block}}}{Fact_{\text{column}}} \times 100$$

The column $\eta_{comp}$ is the percentage improvement in the computation time of the block code over
the column code, computed as:

$$\eta_{comp} = \frac{Comp_{\text{column}} - Comp_{\text{block}}}{Comp_{\text{column}}} \times 100$$

Clearly, the block version performs consistently better than the column version, in some cases by
more than a factor of two. Notice also that the gains are even more significant in the ‘Comp’ times.
This is especially true in the case of BCS* matrices. This is because, for these matrices, SHAPE
produces many more dense block partitions than single columns. The locality of computations in
these dense blocks results in improved cache performance of the i860 processor. However, the idle
times are generally worse in the block code. The computations in the blocks are of a much larger
than sparse columns. Thus, compared to the column partitioning, the block partitioning idle
times are much more sensitive to the allocation and scheduling schemes.

Table 4 summarizes the average communication time ($Comm$) on each processor and the average
number of messages ($Msgs$) sent and received on each processor. The column $\eta_{msgs}$ is the
percentage improvement of block messages over column messages, i.e.,

$$\eta_{msgs} = \frac{Msgs_{\text{column}} - Msgs_{\text{block}}}{Msgs_{\text{column}}} \times 100$$

and the column $\eta_{comm}$ is the percentage improvement of block communication time over column
communication time.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Comm (msec.)</th>
<th>Msgs</th>
<th>Comm (msec.)</th>
<th>Msgs</th>
<th>$\eta_{msgs}$</th>
<th>$\eta_{comm}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCS13</td>
<td>1775</td>
<td>3428</td>
<td>1356</td>
<td>3574</td>
<td>-4.3</td>
<td>23.61</td>
</tr>
<tr>
<td>BCS15</td>
<td>4021</td>
<td>6840</td>
<td>2868</td>
<td>7204</td>
<td>-5.3</td>
<td>28.67</td>
</tr>
<tr>
<td>BCS16</td>
<td>4837</td>
<td>8408</td>
<td>3386</td>
<td>8148</td>
<td>3.1</td>
<td>30.00</td>
</tr>
<tr>
<td>BCS24</td>
<td>2320</td>
<td>6204</td>
<td>1609</td>
<td>4694</td>
<td>24.4</td>
<td>30.65</td>
</tr>
<tr>
<td>BCS28</td>
<td>3047</td>
<td>7704</td>
<td>1791</td>
<td>5196</td>
<td>41.22</td>
<td>32.6</td>
</tr>
<tr>
<td>L5PT127</td>
<td>3576</td>
<td>18778</td>
<td>3210</td>
<td>15674</td>
<td>16.5</td>
<td>10.23</td>
</tr>
<tr>
<td>L7PT16</td>
<td>1952</td>
<td>5534</td>
<td>1621</td>
<td>5786</td>
<td>-4.6</td>
<td>16.96</td>
</tr>
<tr>
<td>L9PT100</td>
<td>3204</td>
<td>15436</td>
<td>2929</td>
<td>11854</td>
<td>23.2</td>
<td>8.58</td>
</tr>
<tr>
<td>L9PT127</td>
<td>5232</td>
<td>25012</td>
<td>4701</td>
<td>19184</td>
<td>23.3</td>
<td>10.15</td>
</tr>
</tbody>
</table>

**Table 4**

Communication cost in Cholesky factorization using 8 processors.

It is interesting that in the BCS13, BCS15 and L7PT16 cases, the block partition results in more
messages than the column partition. The number of messages is a function of the partitioning and
allocation. In the block case, we have not optimized the partitioning parameters. Since different problems have different matrix structures, the partitioner would need a different set of width and grain size parameters for better performance.

Table 5 compares the block and column factorization timings for 16 processors and Table 6 compares the communication cost for 16 processors.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Column</th>
<th>Block</th>
<th>( \eta_{\text{fact}} )</th>
<th>( \eta_{\text{comp}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCS13</td>
<td>5425</td>
<td>3084</td>
<td>452</td>
<td>4420</td>
</tr>
<tr>
<td>BCS15</td>
<td>13748</td>
<td>8632</td>
<td>858</td>
<td>7531</td>
</tr>
<tr>
<td>BCS16</td>
<td>14964</td>
<td>8352</td>
<td>1386</td>
<td>8342</td>
</tr>
<tr>
<td>BCS24</td>
<td>5000</td>
<td>1972</td>
<td>616</td>
<td>3314</td>
</tr>
<tr>
<td>BCS28</td>
<td>6248</td>
<td>2379</td>
<td>712</td>
<td>4048</td>
</tr>
<tr>
<td>L5PT127</td>
<td>6910</td>
<td>2144</td>
<td>1542</td>
<td>5459</td>
</tr>
<tr>
<td>L7PT16</td>
<td>5758</td>
<td>3416</td>
<td>400</td>
<td>4422</td>
</tr>
<tr>
<td>L9PT100</td>
<td>5340</td>
<td>1469</td>
<td>1056</td>
<td>4070</td>
</tr>
<tr>
<td>L9PT127</td>
<td>9754</td>
<td>2936</td>
<td>2129</td>
<td>6826</td>
</tr>
</tbody>
</table>

Table 5
Execution time in milliseconds for Cholesky factorization on 16 processors.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Column</th>
<th>Block</th>
<th>( \eta_{\text{msgs}} )</th>
<th>( \eta_{\text{comm}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCS13</td>
<td>1888</td>
<td>3522</td>
<td>1341</td>
<td>3212</td>
</tr>
<tr>
<td>BCS15</td>
<td>4257</td>
<td>7106</td>
<td>2816</td>
<td>6544</td>
</tr>
<tr>
<td>BCS16</td>
<td>5226</td>
<td>8942</td>
<td>3416</td>
<td>7270</td>
</tr>
<tr>
<td>BCS24</td>
<td>2410</td>
<td>6396</td>
<td>1454</td>
<td>3908</td>
</tr>
<tr>
<td>BCS28</td>
<td>3155</td>
<td>8138</td>
<td>1558</td>
<td>4232</td>
</tr>
<tr>
<td>L5PT127</td>
<td>3223</td>
<td>14950</td>
<td>2539</td>
<td>11208</td>
</tr>
<tr>
<td>L7PT16</td>
<td>1941</td>
<td>4862</td>
<td>2161</td>
<td>8610</td>
</tr>
<tr>
<td>L9PT100</td>
<td>2813</td>
<td>13208</td>
<td>3475</td>
<td>13780</td>
</tr>
<tr>
<td>L9PT127</td>
<td>4688</td>
<td>21542</td>
<td>36.0</td>
<td>25.87</td>
</tr>
</tbody>
</table>

Table 6
Communication cost in Cholesky factorization using 16 processors.

9.3 Performance of SHAPE

Recall that SHAPE functions as a preprocessing step prior to numeric factorization. One measure of the performance of SHAPE is the performance improvements observed in Cholesky factorization. The results in the previous section are indicative of this performance. Another measure of the performance is the actual cost of executing SHAPE. This cost must be taken into account when it pays to use SHAPE and when it does not. In this section, we present the
timings for the various stages of SHAPE.

The pre-partitioner, partitioner and allocator are all sequential programs that run on a single processor of the iPSC/860, making use of the distributed memory of the system for storage purpose only. The symbolic scheduler is a parallel program that runs on as many processors as the numerical factorization code. Table 7 gives the timings for these components of SHAPE. The pre-partitioner and partitioner times are independent of the number of processors. For the partitioning, the parameters \( W \), \( T \) and \( R \) set to 15, 200 and 200 respectively. The allocator and symbolic scheduler timings are given for 8 and 16 processors; in the case of the allocator the number of processors refers to the parameter \( P \), while in the case of the scheduler it is the number of processors on which the scheduler is executed.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BCS13</td>
<td>349</td>
<td>23331</td>
<td>49</td>
<td>51</td>
<td>3945</td>
<td>3859</td>
</tr>
<tr>
<td>BCS15</td>
<td>293</td>
<td>30937</td>
<td>91</td>
<td>95</td>
<td>6514</td>
<td>6134</td>
</tr>
<tr>
<td>BCS16</td>
<td>512</td>
<td>28752</td>
<td>98</td>
<td>103</td>
<td>7452</td>
<td>7074</td>
</tr>
<tr>
<td>BCS24</td>
<td>197</td>
<td>8613</td>
<td>82</td>
<td>86</td>
<td>3299</td>
<td>2827</td>
</tr>
<tr>
<td>BCS28</td>
<td>232</td>
<td>9047</td>
<td>96</td>
<td>100</td>
<td>3879</td>
<td>3448</td>
</tr>
<tr>
<td>F5PT127</td>
<td>570</td>
<td>15680</td>
<td>327</td>
<td>344</td>
<td>6763</td>
<td>5357</td>
</tr>
<tr>
<td>F7PT16</td>
<td>309</td>
<td>26794</td>
<td>88</td>
<td>92</td>
<td>4575</td>
<td>4472</td>
</tr>
<tr>
<td>F9PT100</td>
<td>639</td>
<td>8977</td>
<td>202</td>
<td>216</td>
<td>5724</td>
<td>4152</td>
</tr>
<tr>
<td>F9PT127</td>
<td>1211</td>
<td>18131</td>
<td>330</td>
<td>346</td>
<td>9471</td>
<td>6881</td>
</tr>
</tbody>
</table>

**Table 7**

Timings in milliseconds for SHAPE.

The total time taken by all these components is summarized in the SHAPE column of Table 8. To evaluate the relative gain in using SHAPE and the block numerical factorization code instead of the column code, we compare the total time taken by SHAPE and the block numerical factorization code against the time taken by the column numerical factorization code. Specifically, the column \( \text{Iter} \) is the minimum number of iterations of the block code needed in order that the time to run SHAPE once, followed by the time to run \( \text{Iter} \) iterations of the block code is less than the time to run \( \text{Iter} \) iterations of the column code. \( \text{Iter} \) is computed as:

\[
\text{Iter} = \left[ \frac{\text{SHAPE}}{F_{\text{col}} - F_{\text{block}}} \right]
\]

To compute \( \text{Iter} \), we have not taken into account the one time cost of column symbolic scheduling for the column code, which is carried out before the column numerical factorization.

The results presented in Table 8 indicate that to truly benefit from using SHAPE, the problem to be solved should be such that the same matrix structure is used in repeated matrix factorization. Problems solving non-linear PDEs as well as eigen value problems fall in this category. In several of these cases, the matrix may be factored many more times than the tradeoff number indicated by the quantity \( \text{Iter} \). In all these cases the extra cost of using SHAPE is amortized within a few iterations.

Note also that currently SHAPE is not optimized for execution time performance. This results in higher values for \( \text{Iter} \). The pre-partitioner, partitioner, and allocator are “parallelized” solely
for the sake of taking advantage of the distribute memory. In general, the symbolic factor of the input sparse matrix and the block dependency graph generated by the partitioner require much more memory than is available on a single processor. To overcome this limitation, the symbolic factor input to the partitioner is distributed among several processors. We distribute it to all the processors that are used in the numerical factorization. Although “parallelization” of SHAPE may also be used to reduce its execution time, we have left this aspect out for future study.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>SHAPE</th>
<th>$F_{\text{block}}$</th>
<th>$F_{\text{col}}$</th>
<th>Iter</th>
<th>SHAPE</th>
<th>$F_{\text{block}}$</th>
<th>$F_{\text{col}}$</th>
<th>Iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCS13</td>
<td>27674</td>
<td>4648</td>
<td>7966</td>
<td>9</td>
<td>27590</td>
<td>4420</td>
<td>5425</td>
<td>28</td>
</tr>
<tr>
<td>BCS15</td>
<td>37835</td>
<td>9900</td>
<td>21451</td>
<td>4</td>
<td>37459</td>
<td>7531</td>
<td>13748</td>
<td>7</td>
</tr>
<tr>
<td>BCS16</td>
<td>36814</td>
<td>10451</td>
<td>21593</td>
<td>4</td>
<td>36441</td>
<td>8342</td>
<td>14964</td>
<td>6</td>
</tr>
<tr>
<td>BCS24</td>
<td>12191</td>
<td>3921</td>
<td>6334</td>
<td>6</td>
<td>11723</td>
<td>3314</td>
<td>5000</td>
<td>7</td>
</tr>
<tr>
<td>BCS28</td>
<td>13254</td>
<td>4837</td>
<td>8070</td>
<td>5</td>
<td>12827</td>
<td>4048</td>
<td>6248</td>
<td>6</td>
</tr>
<tr>
<td>F5PT127</td>
<td>23340</td>
<td>7916</td>
<td>9830</td>
<td>13</td>
<td>21951</td>
<td>5459</td>
<td>6910</td>
<td>16</td>
</tr>
<tr>
<td>F7PT16</td>
<td>31766</td>
<td>5571</td>
<td>9008</td>
<td>10</td>
<td>31667</td>
<td>4422</td>
<td>5758</td>
<td>24</td>
</tr>
<tr>
<td>F9PT100</td>
<td>15542</td>
<td>6264</td>
<td>7626</td>
<td>11</td>
<td>13984</td>
<td>4070</td>
<td>5340</td>
<td>12</td>
</tr>
<tr>
<td>F9PT127</td>
<td>29143</td>
<td>10405</td>
<td>13848</td>
<td>9</td>
<td>26569</td>
<td>6826</td>
<td>9754</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 8
Total cost: block vs. column factorization.

10 Conclusions

In this report, we have described the design, implementation and performance of SHAPE. The preliminary performance results presented here are intended to show the feasibility and the general trends in the effectiveness of the methodology proposed in SHAPE. The test matrices used in the performance studies represent a diverse range of applications and in all cases it is observed that SHAPE is effective in significantly improving the performance of parallel sparse Cholesky factorization as compared to the more commonly used column-based factorization algorithm. We expect to see further improvements for larger problems.

A key feature of SHAPE is that it allows partitioning of the sparse matrices into individual columns as well as dense blocks of matrices. Furthermore, using the parameters $W$ and $G$, the partitioning can be controlled. Having individual columns as schedulable units allows for balancing of load at a finer grain of computation, whereas the dense blocks exploit locality, thereby reducing the communication overheads. It has been our experience that $W$ and $G$ serve as sensors for detecting the amount of useful parallelism in an application. In our future study we intend to show that by having some insight into the problem that gives rise to the sparse matrix, one can choose values for these parameters so that a right-mix partitioning is obtained. However, we would like stress here that even without such knowledge one can get reasonable performance improvements for a variety of sparse matrices.

The parallel block numerical factorization code used to obtain the results in this report is not tuned for efficient communication. Besides, it currently does not use the structure of the communication pattern provided by the symbolic scheduler. This information would be beneficial in reducing the
run-time communication overhead.

Another aspect of SHAPE not described in this report is its ability to predict performance of the system for which the partitioning is targeted. During partitioning and scheduling, SHAPE achieves this by gathering statistics on computation times, load imbalance, number of messages, communication volume, and idle times of the processors. Such performance estimates are valuable in characterizing the sparse matrices as well as for selecting the right parameters $W$ and $G$.

Finally, we would like to point out that although SHAPE as described in this report is a preprocessor in parallel Cholesky factorization, it has several components that can be used in partitioning sparse matrix computations other than Cholesky factorization. In fact, if the element level operations can be quantified, as done for Cholesky factorization in this report, then that sparse matrix operation can also be partitioned and scheduled in a similar fashion. Again, this aspect will be considered in a future study.

Acknowledgement: The authors would like to thank Joel Saltz and Yousuff Hussaini of ICASE, NASA Langley Research Center for generously granting the use of their 32 node iPSC/860.

References


