# Sequential and Parallel Algorithms for Cholesky Factorization of Sparse Matrices

Nerma Baščelija
Sarajevo School of Science and Technology
Department of Computer Science
Hrasnicka Cesta 3a, 71000 Sarajevo
Bosnia and Herzegovina
nerma.bascelija@ssst.edu.ba

*Abstract:* The purpose of this paper is to discuss and present the options for building an efficient algorithm for Cholesky factorization of sparse matrices. Both sequential and parallel algorithms are explored. Key ingredients of a symbolic factorization as a key step in efficient Cholesky factorization of sparse matrices are also presented. The paper is based on the author's master thesis [1], defended at Sarajevo School of Science and Technology in 2012.

*Key–Words:* Cholesky factorization, parallel algorithms, sequential algorithms

## 1 Introduction

The most common methods for solving large systems of linear equations $Ax = b$ is to decompose the original matrix into factors of lower or upper triangular matrices. Cholesky factorization is based on decomposing the original symmetric positive definite matrix $A$ into the product of a lower triangular matrix $L$ and its transpose $L^T$. The matrix L is called Cholesky factor.

Based on the approach taken in computing $L$, Cholesky factorization can be row or column oriented. In both of these approaches previously computed rows or columns are used for the computation of the current row or column. Matrix sparsity is utilized in Cholesky algorithms for sparse matrices in recognizing the fact that a row or column does not necesserily depends on all previously computed rows or columns (due to the sparsity structure). That is why, in the case of the sparse matrix, a pre step, known as symbolic factorization, of predicting the nonzero pattern of Cholesky factor L before computing the actual values is a good practice. After the nonzero structure of L is known, computation can be focused only on computing nonzero entries. To implement this we use a structure called elimination tree, whose computation is part of symbolic factorization and which shows column dependencies in Cholesky factor $L$.

Cholesky factorization of sparse matrices offers a number of possibilities for parallel computation. The structure of the elimination tree can serve as a guide in choosing the best task decomposition strategy needed for designing parallel algorithms.

## 2 Sparse Matrices

A matrix whose most of the entries are zero elements is called sparse matrix. Exploring the advantage of sparse matrices structure is mostly based on the fact that operations are done only on its nonzero entries.

### 2.1 Sparse matrix data structure

Sparse matrix data structure is based on storing only nonzero elements of a matrix. Some of the storage forms that are commonly used are: compressed row and column storage, block compressed row storage, diagonal storage, jagged diagonal storage, and skyline storage. Compressed row and column storage will be explained briefly.

The material from [2] was used in writing this section. Both column and row version of compressed storage for sparse matrices are based on forming three arrays: two arrays which serve as pointers to nonzero values, and one array for the actual nonzero values. In the case of compressed column storage, one of the pointer arrays (which will be referred to as $index$ array) will hold indexes of positions of nonzero entries in each column; other pointer array (which will be referred to as $pointer$ array) will point to the positions in the $index$ array where new column starts and the third array (which will be referred to as $values$ array) will hold nonzero values of a matrix in a column order.

Instead of using two dimensional array and going through all entries in a column, with compressed column structure nonzero entries of a column $i$ can be accessed relatively easy and in less time. For each column $i$, its values are at: values[pointer[j]] to values[pointer[j+1]-1]. The similar rules hold for compressed row storage, the only difference is that instead of listing items by columns, they are listed by rows. In this paper, the column compressed storage format is assumed.

## 3 Cholesky factorization

Cholesky factorization is a method which solves $Ax = b$, with $A$ a symmetric positive definite matrix, by factorizing the original matrix $A$ into the product of the lower triangular matrix $L$ and its transpose $L^T$:

$$A = LL^T$$

Factor matrices $L$ and $L^T$ are then used to get unknown vector $x$ by solving the following equations:

$$Ly = b \tag{1}$$
$$L^T x = y \tag{2}$$

which is now straightforward computation task, since both $L$ and $L^T$ are triangular. The column version of the algorithm is:

**Algorithm 1** Solving triangular system accessing by columns

```
1: $x \leftarrow b$
2: for each column $j$ do
3:     $x_j \leftarrow x_j / l_{jj}$
4:     for each row i with index greater than j, for which $l_{ij} \neq 0$ $(j...n)$ do
5:         $x_i \leftarrow x_i – l_{ij} x_j$
6:     end for
7: end for
```

For sparse matrices a pre step is needed in which it is needed to predict the nonzero structure of the vector $x$. Using the predicted nonzero structure of $x$, the sparse version of the algorithm (Algorithm 2) will compute the actual values.

The nonzero structure of $L^T$ is represented by $G_L$, the graph defined as follows: $G_L = (V, E)$, where $V = \{1, ..., n\}$ the set of nodes is set of columns of L, and $E = \{(j,i) | l_{ij} \neq 0\}$.

Which nodes are reachable from a given node is determined using depth first search. After DFS is run for each node $i$ of $G_L$ for which $b_i \neq 0$, the union of all obtained Reach sets is the nonzero structure of the solution $x$.

**Theorem 1.** *Define the directed graph $G_L = (V, E)$ with nodes $V = \{1, \ldots, n\}$ and edges $E = \{(j,i) \mid l_{ij} \neq 0\}$. Let $Reach_L(i)$ denote the set of nodes reachable from node i via paths in $G_L$, and let $Reach(B)$, for a set B, be the set of all nodes reachable from any node in B. The nonzero pattern $X = \{j | x_j \neq 0\}$ of the solution x to the sparse linear system $Lx = b$ is given by $X = Reach_L(B), where B = \{i | b_i \neq o\}$, assuming no numerical cancellation [3].*

**Algorithm 2** Solving triangular system , x is sparse

```
1: input: nonzero structure of x
2: $x \leftarrow b$
3: for each column $j$ for which $x_j \neq 0$ do
4:     $x_j \leftarrow x_j / l_{jj}$
5:     for each row i with index greater than j, for which $l_{ij} \neq 0$ $(j \ldots n)$ do
6:         $x_i \leftarrow x_i – l_{ij} x_j$
7:     end for
8: end for
```

**Algorithm 3** Lx=b, nonzero structure of x

```
1: for each node $i$ for which $b_i \neq 0$ do
2:     mark node $i$
3:     perform depth first search
4: end for
```

Algorithms for computing Cholesky factor are based on one of the three approaches: left, right or up looking approach. Each method uses previously computed rows (in the case of left and right looking) and columns (in the case of an up looking) to compute current row or column. An up looking Cholesky algorithm computes L one row at a time. When computing $k$th row, this algorithm uses all rows from 0 to $k - 1$ which are assumed to be already computed:

**Algorithm 4** Up looking Cholesky factorization

```
1: $l_{11} \leftarrow \sqrt{a_{11}}$
2: for each row $k$ such that $0 < k < n$ do
3:     $k^T \leftarrow L_{(k-1)(k-1)} k^T = a_{*k}$
4:     $k \leftarrow$ transpose of $k^T$
5:     compute diagonal entry $l_{kk} \leftarrow \sqrt{a_{kk} - l}$
6: end for
```

The sparse triangular solve algorithm in Algorithm 2 is based on predicting the nonzero structure of the unknown vector $x$. This means that for each row $k$, its nonzero structure is predicted first, and then the sparse triangular algorithm is performed. Thus, the third line of the Algorithm 4 consists of the steps that are listed in the Algorithm 5.

**Algorithm 5**

```
1: set $K$ is empty
2: for each $i$ for which $a_{ik} \neq 0$ do
3:     $K \leftarrow K$ union depth first search on $L_{(k-1)(k-1)}$ starting at node $i$
4: end for
5: perform algorithm 2 with set $K$ as nonzero pattern
```

If the nonzero structure of the whole matrix L is known in advance, the Algorithm 5 would have the simpler structure. This is why Cholesky factorization algorithms are based on two major steps: symbolic and numerical factorization. Symbolic factorization stands for a process during which only the structure of $L$ is predicted without computing the actual values of its entries. Cholesky factor $L$ will have nonzero entries at positions where original matrix $A$ has nonzero entries plus additional nonzeros which are called fill in. After the nonzero structure of $L$ is known, computation of each row by triangular solve would be faster since the time for predicting the nonzero structure of rows would be reduced. A structure which is commonly used as a reflection of the nonzero structure of $L$ is elimination tree.

Theorem 2 states that the Cholesky factor $L$ will have all nonzero positions as its original matrix $A$. Theorem 3 defines positions on which additional nonzero entries, called fill in, will appear. Following these rules, we can construct the nonzero pattern of $L$.

**Theorem 2.** *For a Cholesky factorization $LL^T = A$ and neglecting numerical cancellation, $a_{ij} \neq 0$ implies that $l_{ij} \neq 0$. That is, if $a_{ij}$ is nonzero, then $l_{ij}$ will be nonzero as well [4].*

**Theorem 3.** *For a Cholesky factorization $LL^T = A$ and neglecting numerical cancellation $i < j < k, l_{ji} \neq 0, l_{ki} \neq 0 \implies l_{kj} \neq 0$ . That is, if both $l_{ji}$ and $l_{ki}$ are nonzero where $i < j < k$, then $l_{kj}$ will be nonzero as well [4].*

The elimination tree is obtained from $G_L$ by removing certain edges: If there is a path from $i$ to $k$ via $j$ that does not traverse the edge $(i, k)$, the edge $(i, k)$ is not needed to compute Reach(i). Reach(t) for a node $t$, if there is a path from $t$ to $i$ is also not affected if $(i, k)$ is removed from $G_L$. If $j > i$ is the least numbered node for which $l_{ij} \neq 0$, all other $l_{ki} \neq 0$, where $k > j$ are not needed.

To get the nonzero structure of rows in $L$, the elimination tree is traversed as described in the following algorithm.

**Algorithm 6**

```
1: input: elimination tree
2: for each $i < k$ for which $a_{ik} \neq 0$ do
3:     mark node $i$
4:     climb up the tree and mark each node, until a marked node or node $k$ is reached
5: end for
6: marked nodes are nonzero pattern of $k$
```

## 4 Parallel Cholesky factorization for sparse matrices

There are several issues that need to be considered when talking about parallel algorithms for Cholesky factorization. The level of task decomposition and strategy which will be used to map tasks to computing resources are probably the most important ones. Task can be seen as a process in the computation, while the computing resource is a runner of the task. Several tasks may be assigned to a single computing resource. This is another important issue that influences algorithm efficiency: the strategy and the level used to map tasks to computing resources available. Depending on the architecture available, this assignment can be done dynamically (in shared memory architecture) or statically (in distributed memory architecture). Either way, in most algorithms for parallel Cholesky which are based on column approach the level at which the tasks are assigned to processors is column level. This means that one processor is responsible for performing all tasks related to one or more columns assigned to it.

We consider column level task decomposition. For discussion on entry level decomposition and block level decomposition we refer the reader to [1].

For sparse matrices, a column $j$ does not necessarily depends on all columns $i$, $0 < i < j$. By its structure, elimination tree reflects column dependencies of $L$ in a way that a node $i$ in the elimination tree (column $i$) depends on all nodes that are its descendants in the elimination tree. In other words, all columns that are below it in the elimination tree need to be computed first and are used in computing a column $i$. According to this we can conclude that columns which are leaves in the elimination tree are those which do not depend on any other columns.

After the structure of the elimination tree is known, the dependency set can be built for each column $j$ and then it can be used to construct parallel Cholesky algorithms whose task decomposition is on column level in a way that columns which do not depend on each other can be computed in parallel. At each stage of the computation, columns that correspond to leaves in the elimination tree can be computed simultaneously. An algorithm designed on this basis is presented in [6]. The algorithm is based on repeatedly performing the following steps until all columns are computed:

1. Compute columns which correspond to leaves in elimination tree

2. Delete leaves from the elimination tree

Obviously the running time depends on the elimination tree height, which means that wider and low height elimination tree is preferable. The way in which the structure of the elimination tree is affected is by reordering the original matrix A. The algorithm itself relies a lot on the symbolic factorization and nonzero structure of Cholesky factor L.

Depending on the nature of the communication model, the algorithm can be implemented in several ways. One of them is to assign a processor to each nonzero in $L$. The assumption is that each processor $P_{ij}$ have access to the structure of the elimination tree.

Each column is computed only when it becomes a leaf in the elimination tree, and is then deleted from the elimination tree so that a node which depends on it becomes a leaf and can be computed next. Again, as a reminder, the operations which need to be done on each column $j$ (once it becomes a leaf):
1. Modification of each nonzero $l_{ij}$ value by nonzero values in columns which are direct descendants of a column $j$ in the elimination tree:
1.1. Each $l_{ij}$ will be computed by subtracting each multiple $l_{jz}l_{iz}$ for each $z$ which is direct descendant of $j$ in elimination tree
1.2. The multiples can be computed and subtracted one by one or can be summed up an subtracted at once
1.3. Once a column $j$ becomes a leaf in elimination tree, each processor $P_{ji}$ (which are in charge of values in row $j$ – since its multiples are affecting column ) is signaled and modifications are made (either one by one or after summing them up)
2. Division of each nondiagonal nonzero value by the square root of its diagonal value
2.1. When a node $j$ becomes a leaf, all processors which are in charge of nodes in column $j$, $P_{ij}$ have to divide its value $l_{ij}$ by the square root of diagonal value

The assumption is that each processor can access the values of other processors. Each processor will access only those values that are needed for the computation. Since each of them will start only after all values on which it depends are computed, it will not happen that a processor accesses the value whose computation is not finished.

Another assumption is that each processor $P_{ij}$ is signaled to start once all values in $j$'s dependency set are computed - that is, when $j$ becomes a leaf in the elimination tree. The signalization may also be done by a processor which would be in charge of the scheduling. On the other hand, since each processor can access the structure of the elimination tree by itself, each can know when its column is a leaf and can start its computation without any signalization. However, since this is the case when a processor is assigned to a single value and thus a single column is computed by multiple processors - they each would have to regularly check if their column is a leaf, it is better for the efficiency that there is a single processor dedicated to scheduling and signalization.

The signalization processor communicates with each processor $P_{ij}$ signalling them to start. Each processor $P_{ij}$, on the other hand, informs the signalization processor once it is done. This is how the signalization processor knows when each value in column is computed, so it can delete it from the elimination tree and inform new leaves to start.

**Algorithm 7** Schedule processors

```
1: while there are nodes in elimination tree do
2:     signal each processor $P_{ij}$, for which $j$ is a leaf to start computation
3:     if all values in column $j$ are computed then
4:         delete $j$ from the elimination tree
5:     end if
6: end while
```

**Algorithm 8** Computation processor $P_{ij}$

```
1: if $i \neq j$ then
2:     for each descendant $z$ of $j$ in elimination tree do
3:         $l_{ij} \leftarrow l_{ij} - l_{jz}l_{iz}$
4:     end for
5:     $l_{ij} \leftarrow l_{ij}/l_{jj}$
6: else
7:     for each descendant $z$ of $j$ in elimination tree do
8:         $l_{jj} \leftarrow l_{jj} - l_{jz}l_{jz}$
9:     end for
10:    $l_{jj} \leftarrow \sqrt{l_{jj}}$
11: end if
12: inform signalization processor that $l_{ij}$ is computed
```

Requiring one processor per nonzero in L could lead to too many processors needed. Instead, this same logic could be implemented by assigning the whole column to a single processor. The level of task decomposition and basic algorithm structure would remain the same, but each processor would have more entries to be in charge of. Again, columns which are current leaves of elimination tree can be computed at the same time:

- Each processor $P_j$ would know when column $j$ is a leaf
- When its column becomes a leaf, $P_j$ collects all multiples needed to perform updates on its values – it needs updates only from columns which are the direct descendants of its column in the elimination tree. Since the elimination tree structure is known to each processor, $P_j$ knows which values it needs
- Alternatively, these updates could be summed up by sourcing processors and then applied to values in $j$
- Finally, processor $P_j$ divides its values by the diagonal entry

The efficiency would not suffer in this case if each processor knows by itself when it becomes a leaf, without the signalization processor. The logic is thus that each processor $P_j$ regularly checks the structure of the elimination tree. Once $j$ is a leaf, the computation of its values starts. After all values are computed, $j$ is deleted from the elimination tree.

**Algorithm 9** Computation processor $P_j$

```
 1: for each descendant $z$ of $j$ in elimination tree do
 2:     $l_{jj} \leftarrow l_{jj} - l_{jz}l_{jz}$
 3: end for
 4: $l_{jj} \leftarrow \sqrt{l_{jj}}$
 5: for each nondiagonal value $i$ in column $j$ do
 6:     for each descendant $z$ of $j$ in elimination tree do
 7:         $l_{ij} \leftarrow l_{ij} - l_{jz}l_{iz}$
 8:     end for
 9: end for
10: $l_{ij} \leftarrow l_{ij}/l_{jj}$
11: delete $j$ from the elimination tree
```

## 5 Running sample algorithms

### 5.1 Sample matrix and tools

In this section, the results and graphs obtained in running different algorithms on sample matrices and comparing their performance will be presented.
The sample matrix used in this section is one of the matrices that can be found in Sample Gallery of the University of Florida Sparse Matrix Collection [7]. MATLAB and its built in functions are used for building and running tests. tic and toe MATLAB commands were used to measure the running time of the functions.
The matrix chosen for the examples in this section is called mesh2e1. It is a real symmetric positive definite matrix of size 306 by 306. Total number of nonzero entries is 2018 (out of total 93636), which place it in the group of sparse matrices.

### 5.2 Full versus sparse

As a first test case, comparison is made on performance of the Cholesky factorization algorithm on the same matrix - once presented as a full and once presented as a sparse matrix.
Cholesky factorization algorithm that is used for the test is MATLAB's function chol.
Part of the MATLAB code that was used to perform these steps is below. The first line converts the matrix into the full form; following with the code section which performs cholesky factorization on it and measures the running time. The other part of the code converts the matrix back to the sparse form and performs the same algorithm on it.

```
F = full(A);
tic
CF = chol(F)';
toc;
%Result: Elapsed time is 0.070510 seconds.

S = sparse(A);
tic
CS = chol(S)';
toc
%Result: Elapsed time is 0.003222 seconds.
```

Notice that Cholesky factorization of the matrix represented as a sparse one takes around 21 times less time then performing the factorization on the same matrix represented as a full one.

The methods for reordering the original matrix before computing the actual Cholesky factor are very common. We consider the minimum degree ordering and Reverse Cuthill-McKee ordering method.

The minimum degree ordering algorithm is a widely used heuristic for finding a permutation P so that $PAP^T$ has fewer nonzeros in its factorization than A[4]. Its MATLAB implementation is the function symamd. To test the effects which this ordering has on the original matrix and fill in of its Cholesky factor, matrix mesh2e1 is reordered and the time needed for Cholesky factorization is measured again. The result shows that in this case, matrix reordering improved the speed of Cholesky factorization.

```
% minimum degree ordering and its effect on
Cholesky factor ;
%A is mesh2e1 matrix
S = sparse(A);
MD = symamd(S);

%perform Cholesky factorization on ordered matrix
tic
MDC = chol(S(MD,MD))';
toc
%Result: Elapsed time is 0.000625 seconds
```

Reverse Cuthill-McKee ordering method moves all nonzero elements closer to the diagonal. Its corresponding MATLAB function is symrcm, which returns the permutation vector which is then applied to the original matrix.

```
S = sparse(A);
p = symrcm(S);
tic
L = chol(S(p,p))';
toc
%Result: Elapsed time is 0.001047 seconds.
```

## 5.3 Symbolic factorization

Figure 1 shows the elimination tree of the Cholesky factor obtained out of the original matrix (the elimination tree figure is generated in MATLAB).

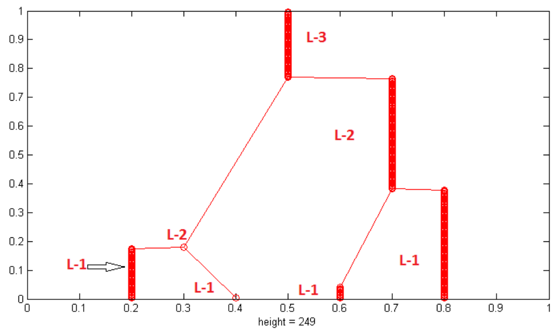

Figure 1: Elimination tree

It shows that there are not many computations which could be done in parallel. Several large leafchains mean that there are many nodes that depens on many other nodes (all nodes that are below it in the chain) and thus can not be computed without them.The options for the possible parallelism are marked in the figure. Leafchains marked with the same mark could be computed in parallel. Leafchains marked with L-1 could be computed independently of each other, following by the nodes and leafchains at the level L-2 which depend on the nodes computed in L-1 level. Finally, level L-3 can be computed only after nodes marked with L-1 and L-2 are computed. Parallelism at the entry level could be added inside the leafchains themselves.

For the examples of the elimination trees of the Cholesky factor of a matrix reordered using minimum degree ordering and also using Reverse Cuthill-McKee ordering the reader is referred to [1].

*References:*

[1] N. Bascelija. *Sequential and Parallel Algorithms for Cholesky Factorization of Sparse Matrices*. Master Thesis, SSST, 2012.

[2] Zhaojun Bai, James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst. (2000). *Templates for the Solution of Algebraic Eigenvalue Problems:a Practical Guide*. [Online]. Available: `http://web.eecs.utk.edu/~dongarra/etemplates/`

[3] J. R. Gilbert and T. Peierls. *Sparse partial pivoting in time proportional to arithmetic operations*. SIAM J. Sci. Statist. Comput, 1988.

[4] Timothy A. Davis. *Direct Methods for Sparse Linear Systems*. University of Florida, Gainesville, Florida: SIAM, 2006.

[5] Ananth Grama, Anshul Gupta, George Karypis and Vipin Kumar. (2003). *Principles of Parallel Algorithm Design*. [Online]. Available: `http://www-users.cs.umn.edu/~karypis/parbook/Lectures/AG/chap3_slides.pdf`

[6] John R. Gilbert, Hjalmtyr Hafsteinsson. *Parallel Cholesky factorization of sparse matrices*, 87-983. New York: Department of Computer Science, Cornell University Ithaca, 1987.

[7] The University of Florida. (2012) The University of Florida Sparse Matrix Collection. [Online]. Available: `http://www.cise.ufl.edu/research/sparse/matrices/`