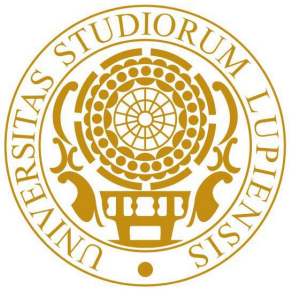
UNIVERSITÀ DEL SALENTO



Facoltà di Ingegneria

Corso di Laurea Magistrale in Computer Engineering

Parallel Algorithms

Parallel Cholesky Factorization of a SPD Matrix

Professor: **Massimo Cafaro**

Student:
Salvatore Corvaglia

Academic year 2020/2021

Contents

List of Figures

Introduction

In this paper we present parallel Cholesky factorization. This method is widely used for numerical solutions inter alia of linear equations. Realization of decomposition will be done with a use of the Message Passing Interface (MPI).
The Cholesky decomposition (or Cholesky factorization) is a decomposition of a Hermitian (complex-valued matrix that coincides with its own conjugate transpose) and positive-definite into the product of a lower triangular matrix and its conjugate transpose (it can be considered as a special case of the more general LU decomposition).
An LU decomposition is a factorization of a matrix into a lower triangular matrix L, an upper triangular matrix U, and a permutation matrix P, used in numerical analysis to solve a system of linear equations, to calculate the inverse of a matrix or to calculate the determinant of a matrix.

Chapter 1

Cholesky decomposition

If matrix A is hermitian and positive definite its Cholesky decomposition is of the form:

$$A = LL^* \tag{1.1}$$

where L is a lower triangular matrix and L^* is hermitian conjugate of L. For positive-definite A the Cholesky decomposition is unique, so the transpose conjugate of L coincides with the transpose and we have:

$$A = LL^T \tag{1.2}$$

1.1 Cholesky Decomposition Algorithm

The Cholesky algorithm, used to compute the L decomposition matrix, is a modified version of the Gauss algorithm. One may figure out Cholesky factorization algorithm simply solving the equation:

$$\begin{pmatrix} a_{11} & a_{21} & \dots & a_{n1} \\ a_{21} & a_{22} & \ddots & a_{n2} \\ \vdots & \ddots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} = \begin{pmatrix} l_{11} & 0 & \dots & 0 \\ l_{21} & l_{22} & \ddots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ l_{n1} & l_{n2} & \dots & l_{nn} \end{pmatrix} \begin{pmatrix} l_{11} & l_{21} & \dots & l_{n1} \\ 0 & l_{22} & \ddots & l_{n2} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & l_{nn} \end{pmatrix} =$$

$$\begin{pmatrix} l_{11}^2 & l_{11}l_{21} & \dots & l_{11}l_{n1} \\ l_{11}l_{21} & l_{21}^2 + l_{22}^2 & \dots & l_{21}l_{n1} + l_{22}l_{n2} \\ \vdots & \ddots & \ddots & \vdots \\ l_{11}l_{n1} & l_{21}l_{n1} + l_{22}l_{n2} & \dots & l_{n1}^2 + ... + l_{nn}^2 \end{pmatrix}$$

Therefore if we compare the terms in general we have

$$a_{ii} = \sum_{j=1}^{i} l_{ij}^2 \tag{1.3}$$

$$a_{ki} = \sum_{j=1}^{i} l_{ij} l_{kj} \tag{1.4}$$

For L matrix terms we obtain (we take the convention that second index of the term is always less or equal first that is for every l_{ij} we have $j \leq i$):

$$l_{ii} = \sqrt{a_{ii} - \sum_{j=1}^{i-1} l_{ij}^2} \tag{1.5}$$

$$l_{ij} = \frac{1}{l_{jj}} (a_{ij} - \sum_{k=1}^{j-1} l_{jk} l_{ik}) \tag{1.6}$$

From above equations one immediately notices that calculation must be done in a sequential order:

$$l_{11} = \sqrt{a_{11}}$$

$$l_{21} = \tfrac{a_{21}}{l_{11}}, ..., l_{n1} = \tfrac{a_{n1}}{l_{11}}$$

$$l_{22} = \sqrt{a_{22} - l_{21}^2} \tag{1.7}$$

$$l_{32} = \tfrac{1}{l_{22}} (a_{32} - l_{21} l_{31}), ..., l_{n2} = \tfrac{1}{l_{22}} (a_{n2} - l_{21} l_{n1})$$

$$l_{nn} = \sqrt{a_{nn} - \sum_{j=1}^{n-1} l_{nj}^2}$$

Let us now emphasize that solving for L must be done sequentially-column after column (from left to right) but in a given column, subtracting terms apparent in sums can be done concurrently which is the most important source of parallelism in column-oriented factorization Cholesky algorithm. In other words, looking at the above formulae calculations are done line by line but for a given line all calculations may be done concurrently. The computation is usually arranged in either of the following orders:

- The Cholesky-Banachiewicz algorithm gives a formula to directly calculate the revenues of the lower triangular matrix L. It begins by forming the upper left corner of the L matrix and proceeds to calculate the matrix row by row

- The Cholesky-Crout algorithm provides a somewhat different procedure for calculating the revenues of the lower triangular matrix L. It starts by forming the upper left corner of the L matrix and proceeds to calculate the matrix column by column.

Cholesky decomposition is relatively demanding in terms of computational operations. Indeed, the computation time of the algorithm scales quite poorly with respect to matrix size, with a complexity of order $O(n^3)$. Luckily, much of the work can be parallelized, allowing for higher performance in less time.
The algorithm is iterative and recursive, such that after each major step in the algorithm, the first column and row of the L factor are known and the algorithm is applied to the remainder of the matrix. Hence, the computational domain of the algorithm successively reduces in size.

1.2 Example

In a 3x3 example, we have to solve the following system of equations:

$$A = \begin{pmatrix} a_{11} & a_{21} & a_{31} \\ a_{21} & a_{22} & a_{32} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{pmatrix} \begin{pmatrix} l_{11} & l_{21} & l_{31} \\ 0 & l_{22} & l_{32} \\ 0 & 0 & l_{33} \end{pmatrix} \equiv LL^T$$

$$= \begin{pmatrix} l_{11}^2 & l_{21}l_{11} & l_{31}l_{11} \\ l_{21}l_{11} & l_{21}^2 + l_{22}^2 & l_{31}l_{21} + l_{32}l_{22} \\ l_{31}l_{11} & l_{31}l_{21} + l_{32}l_{22} & l_{31}^2 + l_{32}^2 + l_{33}^2 \end{pmatrix}$$

We can see that for the diagonal elements (l_{jj}) of L there is a calculation pattern:

$$l_{11} = \sqrt{a_{11}}$$

$$l_{22} = \sqrt{a_{22} - l_{21}^2}$$

$$l_{33} = \sqrt{a_{33} - (l_{31}^2 + l_{32}^2)}$$

For the elements below the diagonal (l_{ij}, where $i > j$) there is also a calculation pattern:

$$l_{21} = \frac{a_{21}}{l_{11}}$$

$$l_{31} = \frac{a_{31}}{l_{11}}$$

$$l_{32} = \frac{a_{32} - l_{31}l_{21}}{l_{22}}$$

Chapter 2

Implementation

2.1 Serial Cholesky Algorithm

Serial algorithm helps to understand how the algorithm works and how we can improve the algorithm by parallelism.
There are three popular implementations of the Cholesky decomposition algorithm, each based on which elements of the matrix are computed first within the innermost for loop of the algorithm:

1. Row-wise Cholesky (Row-Cholesky)

2. Column-Cholesky (Column-Cholesky)

3. Block Matrix-Cholesky (Block-Cholesky)

Each choice of i, j, or k index in outer loop yields different Cholesky algorithm, named for portion of matrix updated by basic operation in inner loops:

1. Row-Cholesky: with i in outer loop, inner loops compute current row by solving triangular system involving previous rows
2. Column-Cholesky: with j in outer loop, inner loops compute current column using matrix-vector product that accumulates effects of previous columns
3. Block-Cholesky: with k in outer loop, inner loops perform $rank - 1$ update of remaining unreduced submatrix using current column In the case of the serial implementation of the algorithm, was considered the column-wise version to limit the number of cache misses on the system.

The column-wise algorithm is shown below:

for $j = 1$ **to** n

  for $k = 1$ **to** $j - 1$

    for $i = j$ **to** n

      $a_{ij} = a_{ij} - a_{ik}a_{jk}$;

    end

  end

  $a_{jj} = \sqrt{a_{jj}}$;

  for $i = j + 1$ **to** n

    $a_{ij} = \frac{a_{ij}}{a_{jj}}$;

  end

end

We can modify/compactify above algorithm declaring functions: **cdiv(j)** which takes a square root of a diagonal element of the column j and divides non-diagonal elements by a diagonal one and **cmod(j, k)** which subtracts from columns j elements multiples of columns k elements.
Using above definitions algorithm for Cholesky decomposition may be rewritten as:

for $j = 1$ **to** n

  for $k = 1$ **to** $j - 1$

    cmod(j,k)

  end

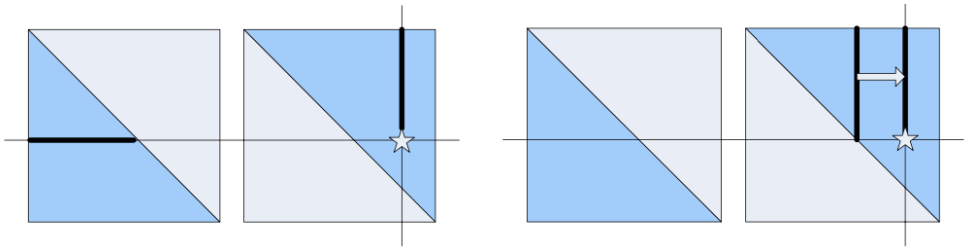
cdiv(j)

end

Studying data dependencies of the above one sees that $cmod(j, k)$ functions for a given j and all $k \in 1, 2..., n$ can be done simultaneously, but $cdiv(j)$ function for a given j depends on $cmod(j, k)$ for $k \in 1, 2..., n$ so this part must be done in an ordered way.

2.2 Parallel Cholesky Algorithm

We need to analyze first on how Cholesky factorization works.



(a) (b)

Figure 2.1: Analysis of Parallel Cholesky Algorithm

To get the value located at the star mark, (a) multiplying the row (bold horizontal line) of the leftmost matrix and the column (bold vertical line above the star mark) of the second leftmost matrix. However, by using the symmetry property of Cholesky factorization, this multiplication equals (b) multiplying two columns (bold vertical line) of the rightmost matrix.
As the figure above shows, to get the value of the star mark, two rows must be previously calculated. This information, say $A[0 : i, i]$ and $A[0 : i, j]$, should be broadcasted to the processors which wishes to compute $A[i, j]$.

2.3 Realization

In order to have an efficient MPI implementation of Cholesky decomposition, it is necessary to distribute computations in a way that minimizes communications between processes. Before we can figure out what the best solution is, we have to identify the dependencies. By analyzing the algorithm, we notice two things: previous columns need to be updated before any subsequent column and, when updating entries of the same column, the new value of diagonal entry has to be computed first. This is easier to understand with an example.
Suppose we have a 5x5 matrix and we wish to update entries in column 3 (entries in previous columns have already been updated). Then, the dependencies for each entry in column 3 are the following:

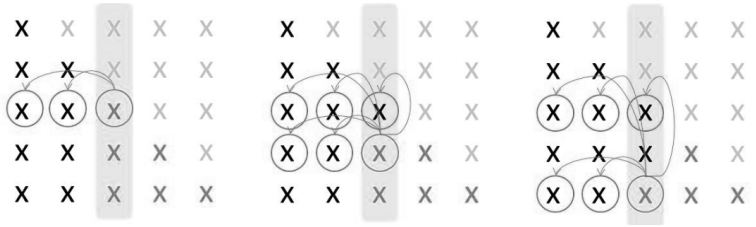


Figure 2.2: Dependency chart for the message passing implementation

Notice that the values that are needed to update an entry are either located on the same row as the entry or on the row that contains the entry on the diagonal. Hence, decomposing the domain into rows is probably the best solution to minimize communications between processes. Cyclic assignment of rows to processes can easily be achieved by taking the remainder of the division of the column number to the total number of processes (i.e. $j\%nprocessors$). Note that we can take the column number instead of the row number because the matrices on which the algorithm is applied are always square.
Let us recapitulate how Cholesky decomposition works. For every column:

1. Replace the entries above the diagonal with zeros
2. Update the entry on the diagonal
3. Update entries below the diagonal

1. The first step is pointless to parallelize because it involves no computations.
2. The second step could be parallelized. We could start by broadcasting the row that contains the entry on the diagonal, then use $MPI\ Reduce()$ to compute the new value of the entry on the diagonal and broadcast again to send the updated value to all processes. However, this would be terribly inefficient because processes would spend a lot of time sending and receiving messages. It is more efficient to simply let the process assigned to that row compute the new value for the entry on the diagonal and broadcast the whole row once.
3. The third step is where we gain performance by parallelizing tasks. Since processes already have access to the data in their respective row(s), we only need to broadcast the row that contains the entry on the diagonal. As soon as a process gets this data, it can proceed and update the entry(entries) that correspond to its row(s). Thus, at this point in the program, processes all compute in parallel.

2.4 Results

We performed experiments measuring the runtime and speedup as the number of processes increases but with a variable workload on the machine (see specifications of the machine used).
We observe that the execution time with np 1 increases in function of the matrix size. This result corresponds to our expectations since Cholesky decomposition is composed of three nested for loops resulting in a time complexity of $O(n^3)$. Similarly, we note that the MPI runtime increase with respect to the matrix size, but less dramatically. This results illustrates that parallelizing computations improves execution time.
As expected, the parallel implementations result in a much higher performance than their serial counterpart; obviously, parallelizing independent computations results in much smaller runtimes than executing them serially.

	Number of processes			
Matrix size	*1*	*2*	*3*	*4*
50	0.000100	0.000242	0.000300	0.000790
150	0.002175	0.002192	0.003154	0.002925
250	0.010879	0.007688	0.008647	0.009107
500	0.092777	0.049040	0.054558	0.051301
750	0.286443	0.141824	0.163909	0.145831
1000	0.753350	0.420643	0.412198	0.367997
1500	2.607116	1.324154	1.415673	1.429420
2000	6.087206	3.049646	3.389080	3.080965
3000	17.791296	8.855257	9.888016	8.924254

Figure 2.3: Runtime in function of the matrix size and number of processes for the MPI implementation with debugger

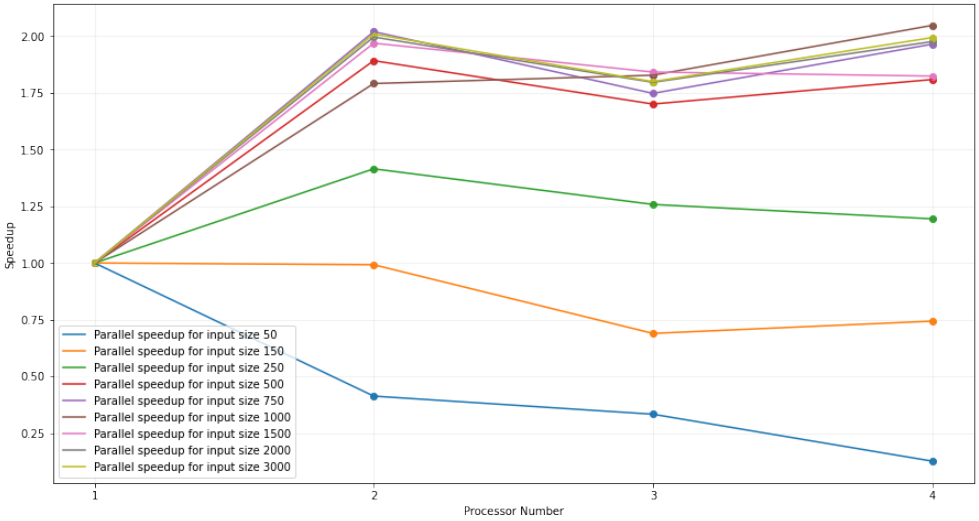


Figure 2.4: Speedup in function of the matrix size and number of processes for the MPI implementation with debugger

2.5 Execution

Follow these steps to run the program:
1. *make*
2. *mpirun −np* [*number of processes*] *./cholesky* [*matrix size*]

Test Machine Specifications:
MacBook Pro (13-inch, 2017)
2,3 GHz Intel Core i5 dual-core
8 GB 2133 MHz LPDDR3
- sysctl -n hw.ncpu: 4
- sysctl -n hw.logicalcpu: 4
- sysctl -n hw.physicalcpu: 2

2.6 Conclusions and future developments

In the future, we would like to experiment new modes in the hope of discovering an even more efficient parallelization scheme for Cholesky factorization. Moreover, we would like to conduct further tests by running program on computers with various amounts of CPUs. This project improved our knowledge regarding the different parallelization tools that can be used to parallelize a program. In fact, we were able to apply the parallel computing theories learned at classes.