ENHANCING MATRIX INVERSION EFFICIENCY

A PARALLEL IMPLEMENTATION OF CHOLESKY AND LU DECOMPOSITION ALGORITHMS

Ramya Rajaraman, Revathi Sekar
Dept of Electrical and computer engineering
School of Engineering and Computer Science
Oakland University, Rochester Hills, Michigan.
emails: ramyarajaraman@oakland.edu, rsekar@oakland.edu

ABSTRACT
Matrix inversion is a critical computational task in various scientific and engineering applications, such as embedded systems, signal processing, control systems, machine learning, real-time systems etc., Traditional sequential matrix inversion techniques can be computationally intensive, especially with large matrices, leading to significant delays, more resource consumption, and inefficiencies in performance-critical systems. In this project, we implemented parallelized versions of the two matrix inversion decomposition algorithms- Cholesky and LU decomposition algorithms and we achieved performance efficiency through different Intell TBB parallelization strategies, which makes it suitable for using it in a real time applications. Through this parallelization techniques, scalability issue has been resolved, which helped in making use of all the available resources.
Keywords- Parallel Programming, Performance Optimization, Matrix Inversion, LU and Cholesky Decomposition, Multithreading, Sequential vs. Parallel Processing, Performance Comparison.

I. Introduction

Our project focus on applying the parallel computing to two important matrix inversion methods - LU and Cholesky decomposition. we explained what these algorithms are, how it works, and why they are useful for solving equations and inverse matrices. Due to limitations of Sequential approach, which solve problems one step at a time, become slow when working with large matrices and why we need a quicker approach. By applying Parallelization in LU and Cholesky decomposition. The tasks will get divided into multiple threads or subranges and these threads are assigned to different cores present in the system to get processed simultaneously to speed up the process. We also compared the speed of the parallel program with the Sequential ones and measured how much improvement we can achieve. The main motivation is to make solving large matrix problems faster and more efficient. As technology advances and the size of data grows, traditional sequential methods become inefficient and slow when dealing with real time applications. By implementing parallelization, we can handle these larger problems in less time, making important applications like simulations, data analysis, and machine learning more efficient. By improving the speed of LU and Cholesky decompositions, this project can save time and computing resources in industries like data science, engineering, and research. Faster computations lead to improved performance in systems that handle complex math, allowing faster results in real-world tasks which will be suitable for real time applications as well.

We have implemented our project with LU and Cholesky decomposition inverse algorithms by using different parallelization frameworks like Intel TBB Parallel for and Parallel Pipeline that we have learned in this

course in order to parallelize our algorithms to achieve performance efficiency.

Parallel programming is used in many applications to make tasks faster and more efficient, especially when dealing with complex computations. In Graphics Transformation, matrix inversion is used to compute transformations in 3D graphics, such as perspective corrections and rigid body motions. In cybersecurity, matrix inversion plays a crucial rule for encryption and decryption techniques. In autonomous Vehicles, the concept of Kalman filter is used for object detection, lanes detection etc., matrix inversion plays a significant role in this Kalman filter, so by paralleling this matrix inversion algorithms to achieve performance efficiency will make it suitable for using it in real time systems.

II. METHODOLOGY

A. LU DECOMPOSITION MATRIX INVERSION

An LU decomposition of a matrix A is the product of a lower triangular matrix and an upper triangular matrix that is equal to A. A=LU [1]; A -Input Matrix L-Lower triangular matrix - A square matrix in which all the elements above the main diagonal are zero. In LU Decomposition matrix inversion, the main diagonal elements in lower triangular matrix being equal to 1.U-Upper triangular matrix- A square matrix in which all the elements below the main diagonal are zero. To find the inverse of a matrix using LU decomposition then the input matrix needs to be square (number of rows and columns should be equal) and a non-singular matrix (determinant of a input matrix shouldn't be zero) By using LU, we can find inverse of A through forward and backward substitution method.

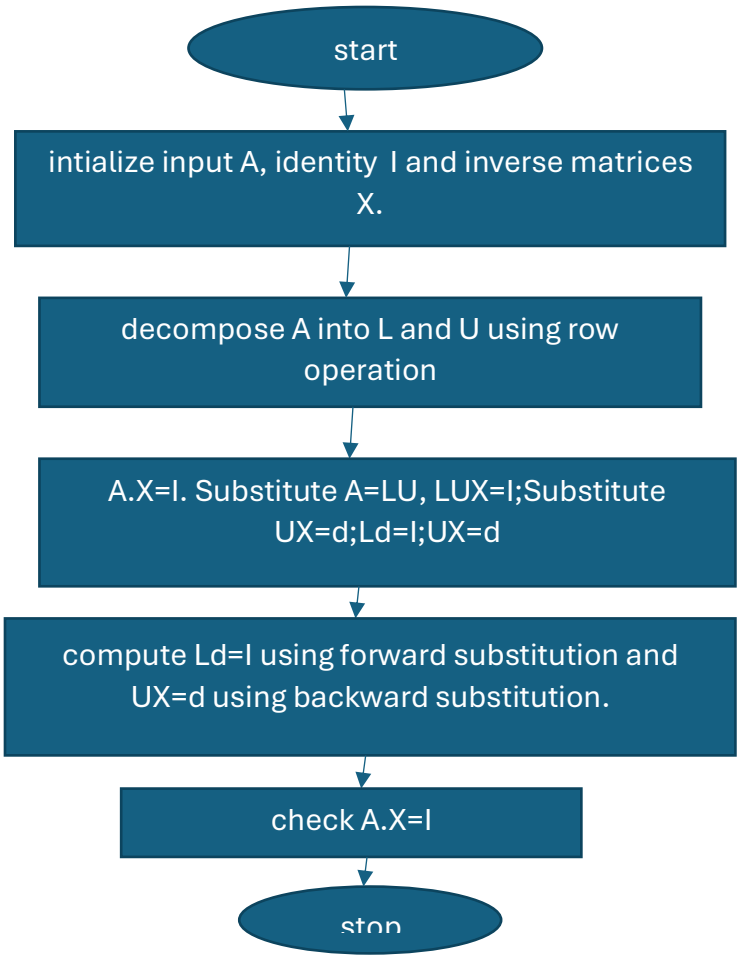


Fig: Flowchart of LU Decomposition

LU DECOMPOSITION MATRIX INVERSION- ALGORITHM EXPLANATION

A-Input matrix, X- Inverse of A matrix, I-identity matrix (diagonal elements are 1).

A*X=I.

For instance, let's consider A, I as 3*3 matrix.

Decompose A into L and U.

Find L, U using Gaussian elimination technique. The steps are as follows,

Step 1: Do Row operations [4], in row 2 and row 3 using row 1 to get the first element of row 2 and row 3 as 0, and while doing this, substitute the number what we used to get first element of row 2 and 3 as zero using row 1, into the first element of

lower triangular matrix row 2 and 3.

Step 2: Do same Row operation [4], in row 3 using row 2 to get the second element of row 3 as 0, and while doing this, substitute the number what we used to get second element of row 3 as zero, into lower triangular matrix row3 second element.

Step 3: By doing step 1 and 2, we can get upper and lower triangular matrix.

Once we compute L and U, we can use this to find inverse of matrix A.

A.X=I; where X is inverse of A

Substitute A=LU, in above equation,

LU.X=I.

LUX=I; Let's assume UX=d (solve using backward substitution method), then Ld=I [3] (solve using forward substitution method); where d is also a 3*3matrix which is unknown so far, we can find d matrix using L and I.

Once we find d matrix, we can use d to solve the equation UX=d [3]; U and d are known, we can use this to compute X, where X is also 3*3 matrix used to store result (which is inverse of input matrix).

FORWARD SUBSTITUTION METHOD

To compute Ld=I[3], we can use this forward substitution method (solve the linear equation from top to bottom), to split the I and d (3*3 matrix) into 3 columns, where the coefficients of first row of L (1*3), getting multiplied with first column of d (3*1) and these Ld is equals to first column of I (1*3) i.e.) [1 0 0] to compute first column of d matrix value [1].

Repeat the above step for each row of L with corresponding column of unknown d by using each column of I, to compute d matrix.

BACKWARD SUBSTITUTION METHOD

To compute UX=d [3], we can use this backward substitution method (solve the linear equation from bottom to top), to split the d and X (3*3) into 3 columns, where the the coefficients of first row of U (1*3) getting multiplied with last column of X (3*1) and these UX is equals to last column of d (1*3) to compute X matrix last column values [1].

Repeat the above step for each row of U with corresponding column of unknown X by using each column of I, to compute X matrix.

So, by using the above Forward and backward substitution, we can find the inverse of input matrix, which is getting stored in X matrix.

Finally, we can check our output by multiplying input matrix with the X (resultant matrix) as a result we should get I-identity matrix.

B. CHOLESKY DECOMPOSTION MATRIX INVERSION

The Cholesky decomposition, is a process of breaking down of positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose, which is important for quick numerical solutions in linear algebra. The goal of Cholesky decomposition is to find a matrix L such that: A = L*L(pow)T. where A is the original matrix (symmetric and positive-definite), L is a lower triangular matrix (a matrix where all elements above the diagonal are zero). L(pow)T is the transpose of matrix L. The key points are to start with a symmetric, positive-definite matrix A (input matrix). The matrix L is lower triangular, means all elements above the main diagonal are zero. calculate the lower triangular matrix L, such that A=L×L(pow)T. Once we have the lower triangular matrix L from Cholesky decomposition, we can find the inverse of A by using Forward and Backward substitution.

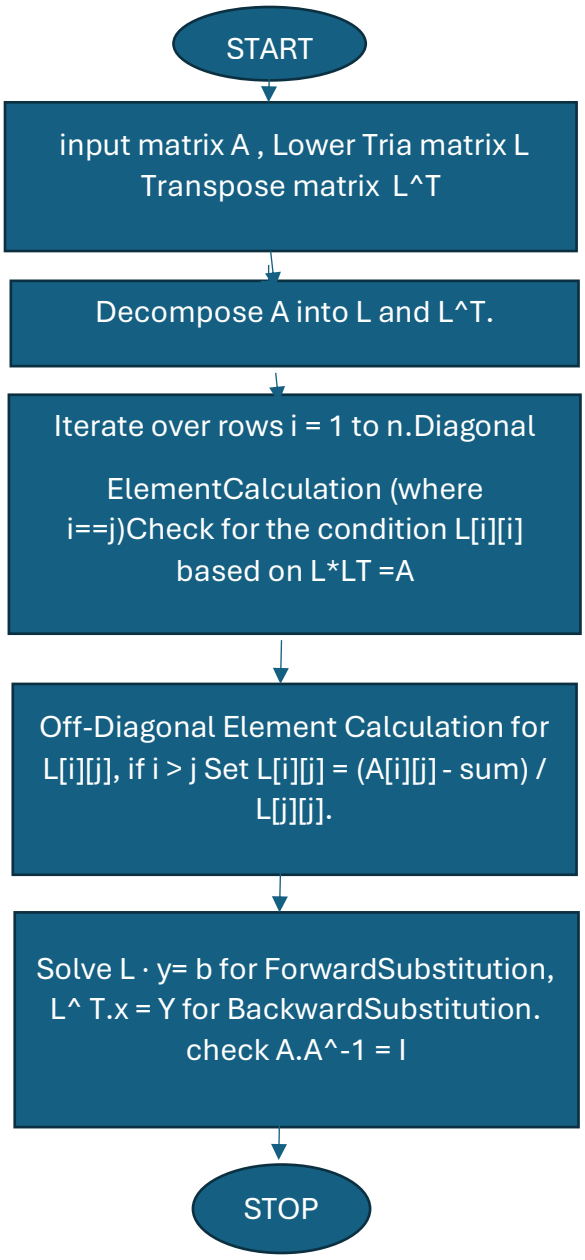


Fig: Flowchart of Cholesky Decomposition

CHOLESKY DECOMPOSTION ALGORITHM EXPLANATION

A – Input matrix, L – Lower Triangular matrix, L(pow)T – Transpose of matrix L

For example, consider A matrix – 3*3 matrix, Factorized A into L and L(pow)T.

Steps to find the Lower Triangular Matrix as follows

Step 1: The input matrix A must be symmetric (A=A(pow)T) and positive-definite (all eigenvalues are positive). A positive definite matrix is required in Cholesky decomposition because it ensures that all values along the diagonal of the resulting lower triangular matrix L are real, positive numbers. To check if this matrix is positive definite, test whether x(pow)T. Ax > 0 for any non-zero vector x.

For example: x= [1,1] ; x(pow)T = [1,1] ; A = [{2,1},{1,2}]

A. x = [{2,1}, {1,2}]. [1 ,1] = [3,3];

x(pow)T. A. x = [1,1]. [3,3] = 6 (which is positive.)

Step 2: To find the Lower Triangular Matrix, create an empty L matrix of size 3*3, initially filled with zeros. This matrix will eventually contain the lower triangular elements. To Fill the matrix L, begin with the first row and move downwards, filling in the elements of L row by row:

For diagonal elements (where row index equals column index where i == j), L[i][i] must be chosen such that when L is multiplied by L^T, it produces the element A[i][i] in the corresponding position. This means summing the products of all the previous values in the ith row and ith column of L (all values up to L[i−1] [i−1] and earlier), and then subtracting that from A[i][i]. This guarantees the diagonal element is correctly set.

For off-diagonal elements (where row index is greater than column index (where i>j), calculate it by subtracting the sum of products of the elements of row i and column j in L from A[i][j], and then divide by L[j][j] (the previously computed diagonal element).

For instance: 3*3 Lower triangular matrix L = [{L11, 0, 0}, {L21, L22, 0}, {L31, L32, L33}]

First Row (i = 0): You calculate only L11 because other entries in this row would be upper triangular and are therefore zero.

Second Row (i = 1): You calculate L21 (off-diagonal) and L22 (diagonal).

Third Row (i = 2): You calculate L31, L32, and L33. After filling in all elements of L, multiply L with its transpose LT. If the result equals the original matrix A, the decomposition is complete and correct.

Step 3: Once we have the lower triangular matrix L from the Cholesky decomposition of A, we can use it to find the inverse of A which involves solving two systems of equations (forward and backward substitution) for each column of the identity matrix to build the inverse of A.

Forward Substitution - Let us consider the equation AX = b, where A = L.L(pow)T So L. L(pow)T. X = b where L(pow)T = y. Therefore **L*y = b** to get b**.** where L is a lower triangular matrix, y is the vector we want to solve for, and b is a column vector from the identity matrix I. Since L is lower triangular, forward substitution allows us to solve for each element of y in sequence, from top to bottom, because each row only depends on the elements that have already been calculated. Suppose we have matrix L = [{2,0,0}; {3,4,0}, {1, -1,5}]; b = [1,0,0]

solve y1 = 2y1.1 => **y1 =½**, similarly y2 = 3y1+4y2 =0; 3.1/2 + 4y2 = 0 => **y2 = -3/8**

Solve y3 **=** 1y1 −1y2 +5y3 =0; ½ + 3/8 + 5y3 = 0 =>**y3 = -7/40**

Backward Substitution – L(pow)T*x = y once we have the vector y from the forward substitution, we solve for x in the equation **L(pow)T*x=y** to get x**.** This is done using backward substitution, which is similar to forward substitution but starts from the last element and works upwards. By performing these steps for each column of the identity matrix, we get the inverse of A. Now we have matrix **L^T** = [{2,3,1}; {0,4, -1}; {0,0,5}]; **y** = [½, -3/8, -7/40]; L^T*x =y for **x** = [x1, x2, x3] based on the above input matrices, we solve each row from the bottom up, to get the values of x.

C. PARALLELISATION STRATEGY FOR LU DECOMPOSITION MATRIX INVERSION-MAP PATTERN

After initialization and decomposition of A into L and U, I am planning to use **Intel TBB parallel_for (map pattern) [5]** for **Forward substitution** to solve L*d=I-> to get d, **backward substitution** to solve UX=d->to compute X(output); and **verification of A*X=I.**

DECOMPOSING A INTO L AND U: parallel_for-row

FORWARD AND BACKWARD SUBSTITUTION: parallel_for-column

Let's consider the scheduler divides task A [3][3] into 3 subranges-3 elements in each subrange in case of using parallel_for(map pattern) [5].

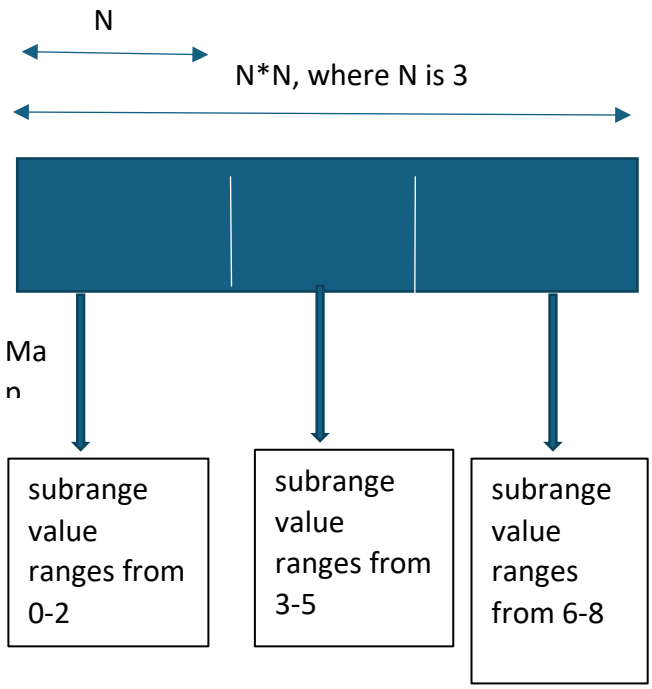


Fig. 2 depicts one possible parallel implementation of the map pattern for LU and Cholesky Decomposition.

For forward, backward substitution, refer above diagram, main task may get divided into 3 tasks (theses 3 tasks will run in parallel however each task will run sequentially inside it), so each task may handle each column values (as explained in STEP 3: flowchart explanation) to compute d*X from L*d=I; UX=d; where L, I- for forward and U, d-for backward, matrices are shared among 3 tasks, but these tasks won't modify/update the L, I in forward and U, d in backward, it will just read values from L and I, and U and d to compute d [3][3] and X [3][3]so that we can prevent the race condition.

D. PARALLELISATION STRATEGY FOR LU DECOMPOSITION MATRIX INVERSION-PARALLEL PIPELINE

After initialization and decomposition of A into L and U, we used Intel TBB parallel pipeline, Forward substitution to get d, backward substitution to compute X(output).

We have implemented the forward and backward substitution by using parallel pipeline.

- **First stage:** Giving columns indices as input
- **Second stage:** Forward substitution to compute intermediate matrix(d)
- **Third stage:** Backward substitution to compute inverse matrix(x) using intermediate matrix(d) from forward substitution.

PIPELINE IMPLEMENTATION

We have given number of token as 16, in the first stage 16 columns indices will be processed and given as an input to the second stage where the forward substitution will solve 16 columns in parallel to compute the intermediate matrix (d) and once the each columns is processed it will be send as an input to the final stage where backward substitution takes place to compute inverse matrix(x) using intermediate matrix(d) values from previous stage like this till the column values reaches the size of given matrix this process will continue to compute the inverse.

Fig. 2 depicts one possible parallel implementation of the parallel pipeline for LU Decomposition.

E. PARALLELISATION STRATEGY FOR CHOLESKY DECOMPOSITION MATRIX

After initialization and decomposition of A into L and L^T, I am planning to use **Intel TBB parallel_for (map pattern)** for **Foward substitution** to solve **L.y = b ->** to get y, **backward substitution** to solve **L(pow)T.x=y** to get x; and **verification of A*A^-1=I.**

For forward, backward substitution , refer above diagram, main task may get divided into 3 tasks (theses 3 tasks will run in parallel however each task will run sequentially inside it), so each task may handle each column values (as explained in STEP 5: flowchart explanation) to compute y, x where L, b for forward and L(pow)T, y for backward matrices are shared among 3 tasks, but these tasks won't modify/update the L, b in forward and L(pow)T, y in

backward, it will just read values from L and b, and L(pow)T and y to compute y [3][3] and X [3][3]so that we can prevent the race condition.

For verification of A*A^-1=I, matrix multiplication of A and inverse of A may get split into 3 tasks and it may in parallel if we use the map pattern.

F. PARALLELISATION STRATEGY FOR CHOLESKY DECOMPOSITION MATRIX INVERSION-PARALLEL PIPELINE

After initialization, we used Parallel pipeline for computing Foward substitution to get Y, backward substitution to get X(inverse). We use 2 main stages in the Parallel pipeline implementation.

Stage 1: Generates column indices col represents the column of the inverse matrix A^-1.

Stage 2: Compute the inverse of the given matrix by doing forward and backward substitution by getting the column indices from stage1, and it will store the inverse.

IMPLEMENTATION OF PIPELINE

The pipeline implementation involves 2-stage pipeline. The stage-1 is serial stage, which generates col indices, that represents the current column to process. The stage-2 is parallel stage, it takes the column index (col) from Stage 1 and performs the calculations to compute the inverse of the column using forward and backward substitution. Combine the work of forward substitution, backward substitution, and assignment to the inverse matrix A^-1 in a single stage. Col 1 - solve L* y = e, col2- solve L^T*X = Y, col 3 - store X in the col (column of A^-1).

For verification of A*A^-1=I, matrix multiplication of A and A^-1 are used in parallel_for , and these tasks will run in parallel and executed simultaneously.

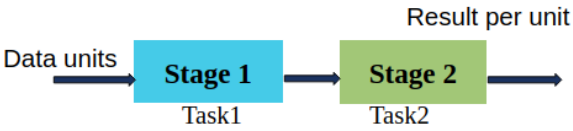


Fig. Two stage Parallel Pipeline

III. EXPERIMENTAL SETUP

Once we compute the inverse of the given matrix via LU and Cholesky Decomposition separately, we have done matrix multiplication for the inverse and the input matrix, which needs to produce identity matrix as an output. For smaller size of matrices, we can directly display the inverse and the identity matrices and we can check but for the larger data it's difficult to do that so, we have used MATLAB to verify our project implementation output. Once we got the identity matrix we have stored that identity matrix in a binary file. For LU and Cholesky Decomposition, we have created the MATLAB script, once the inverse is found via MATLAB, by using the inverse and the input matrix we have checked whether we are getting the identity matrix by doing matrix multiplication of those and store it in an binary file- result file, and this MATLAB script will also get the binary file from our implementation-Received file as an input and it will check the whether the binary file of what it generated is equal to the binary file what it received. We are checking SAD-sum of absolute difference, if both are same, then we can conclude that our result is correct if it shows some difference then result is not correct. In addition of Sum of absolute differences, we checked it via figure too, which will display the result of Matlab file, received binary file from our code and the difference between these two files in a figure. We expected to get sum of absolute difference as 0.

IV. RESULTS

We have implemented the LU Decomposition in sequential method initially, and it took 22 seconds to compute

the inverse of the given matrix for the size of 2000*2000.

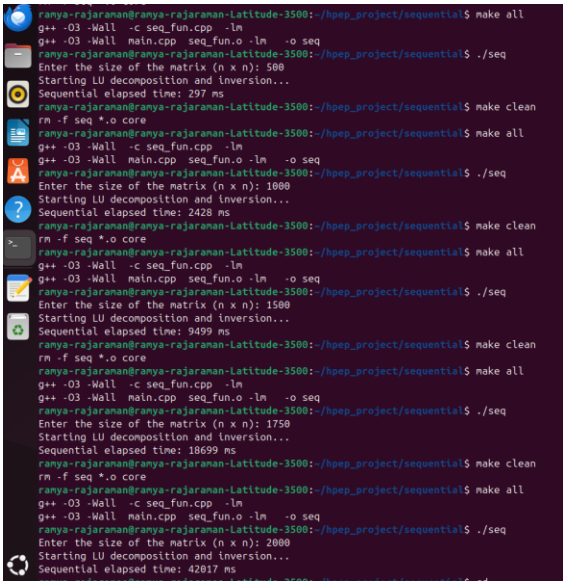


Fig: Sequential implementation of LU Decomposition- Time to compute inverse for different matrix sizes-terminal output.

We checked our output via MATLAB. Below are the figures verifying the result

.

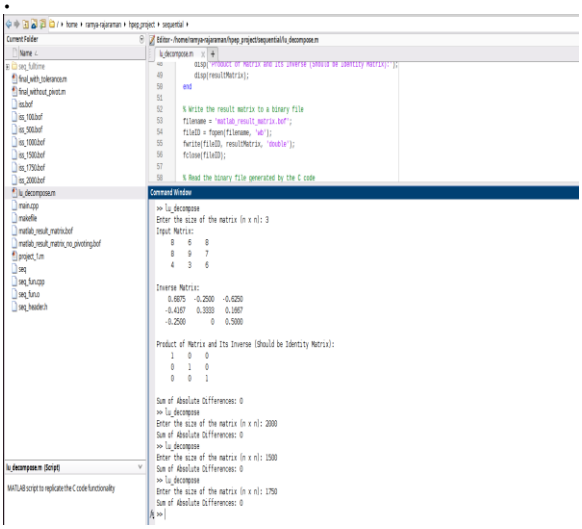


Fig: Sequential implementation of LU Decomposition- MATLAB-Command Prompt output shows SAD as 0 for different matrix sizes.

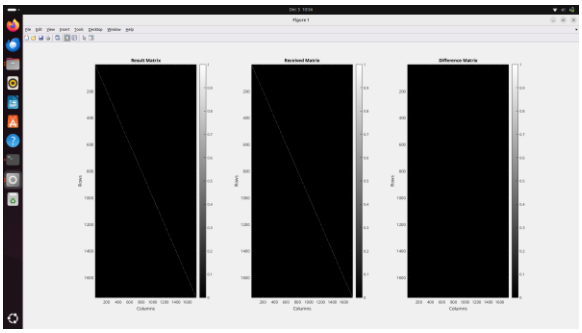


Fig: Sequential implementation of LU Decomposition- MATLAB- Figure displays, MATLAB, code-Identity matrix and its difference of matrix size 1750*1750.

We implemented LU Decomposition using different parallelization strategies such as Intell TBB- Parallel_for, and Parallel Pipeline which we have learned in our course. For LU Decomposition, via parallelization strategy we have achieved 65% performance efficiency compared to sequential implementation. The implementation via parallel_for took only 7 seconds which is more efficient than the sequential implementation. Through the above different implementation, we have achieved the performance via parallelization and we gained the output what we have expected (the sum of absolute difference as 0, for different size of square matrices). We found that the sequential method of implementation works better for the smaller size, but for the larger data sizes parallelization method shows much better.

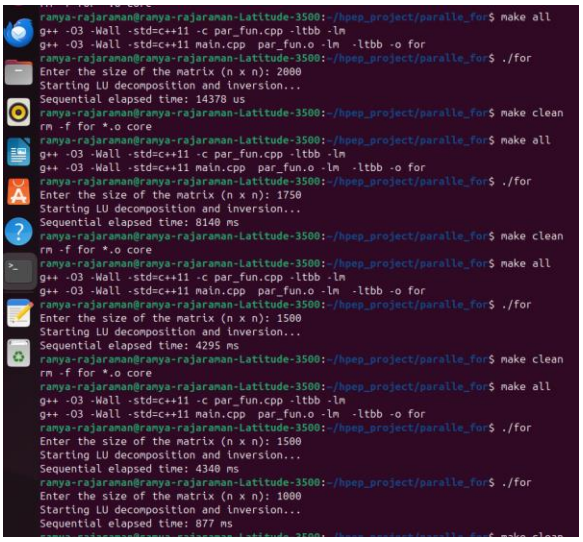


Fig: Parallel implementation using Parallel_for -LU Decomposition- Time to compute inverse for different matrix sizes-terminal output.

We checked our output via MATLAB. Below are the figures verifying the result

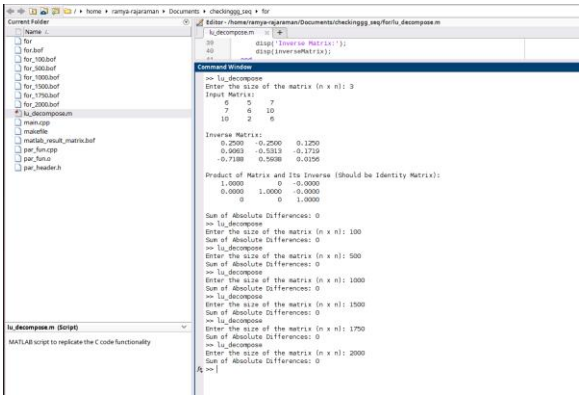


Fig: Parallel implementation using Parallel_for -LU Decomposition- MATLAB-Command Prompt output shows SAD as 0 for different matrix sizes.

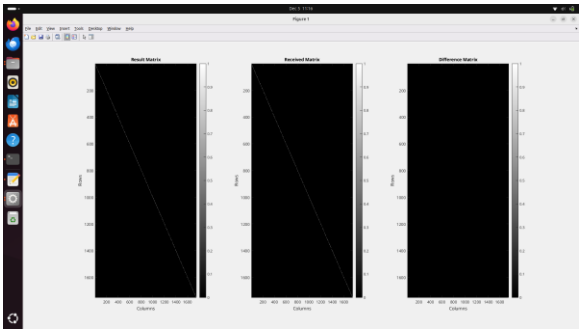


Fig: Parallel implementation using

Parallel_for -LU Decomposition - MATLAB- Figure displays, MATLAB, code-Identity matrix and its difference of matrix size 1750*1750.

We have used larger matrix size for our implementation. By analyzing the below table, we can say that the performance is achieved through parallel processing by dividing the instruction stream into a set of smaller streams i.e.) thread and distribute those threads to multiple cores present in the system to gain the efficiency of the system. In our system, 4 cores are available we have made use of this cores and achieved performance efficiently via parallel programming.

Programming methods	Time to compute inverse (in milli-seconds)	Speed improvement via parallelisation in %
Sequential	42017	-
Parallel using parallel_for	14378	65
Parallel using parallel pipeline	14257	66

Table: Timing and performance comparison between sequential and different parallel implementation of LU Decomposition

Code style	comments	Execution time(in milliseconds)

Sequential	Straightforward sequential implementation of LU decomposition to find inverse for size 2000	42017
Intel-TBB-Parallel_for	Parallelized version of LU decomposition to find inverse for size 2000	14378
Intel-TBB-Parallel pipeline and parallel_for.	Parallelised version of LU decomposition to find inverse for size 2000. Below are implemented in pipeline stages after decomposition of A into L and U Stage 1: Processing and giving Column indices Stage 2: Forward substitution to find intermediate matrix Stage 3: Backward substitution to find inverse matrix.	14257

Table: Comparison of various parallel approaches of LU Decomposition Matrix inversion implementation

We have implemented the **Cholesky Decomposition** in sequential method to compute the inverse matrix with different matrix sizes.

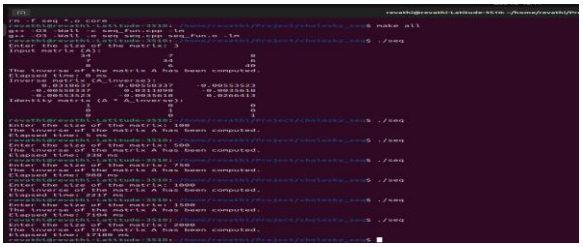


Fig: Terminal Output - Sequential implementation of Cholesky Decomposition with different matrix sizes.

We checked our output via MATLAB. Below are the figures to verifying the result.

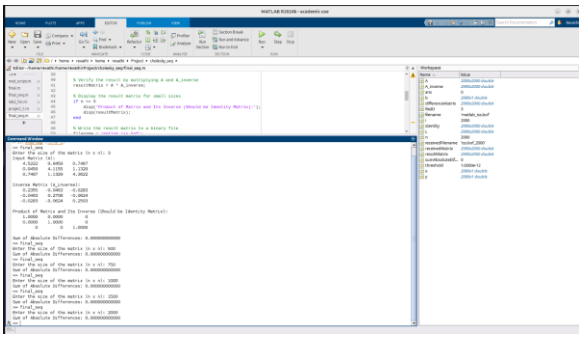


Fig: Sequential implementation of Cholesky Decomposition- MATLAB- Command Prompt output shows SAD as 0 for different matrix sizes.

The below figure shows the MATLAB Identity matrix (Result), Code Identity matrix (Received) and its difference for 2000*2000 matrix

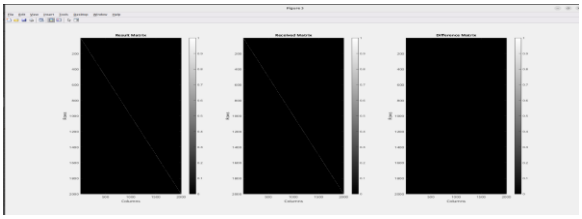
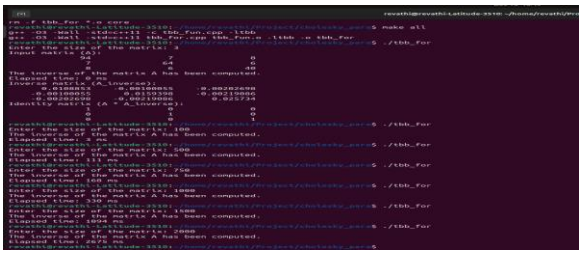


Fig: Sequential implementation of Cholesky Decomposition. MATLAB & Code-Identity matrix and its difference with matrix size of 2000*2000

We have implemented our project with LU and Cholesky decomposition inverse algorithms. We use different parallelization frameworks like **Intel TBB Parallel for and Parallel Pipeline that we have learned in our course.** For implementing Cholesky decomposition, I used parallelization techniques which allows multiple operations to be performed simultaneously, significantly reducing the time required to compute the inverse of large matrices. Parallel algorithms efficiently distribute the

computations across available cores minimizing the idle time for processors. By using Parallelization in our code, we achieved 84% of performance efficiency compared to sequential implementation. Overall, the sequential method works better for smaller workloads but for larger data size Parallelization works effectively.

We have implemented the Cholesky Decomposition in sequential method to compute the inverse matrix with different matrix sizes.



. **Fig: Terminal Output - Parallel implementation of Cholesky Decomposition with different matrix sizes**

We checked our output via MATLAB. Below are the figures verifying the result

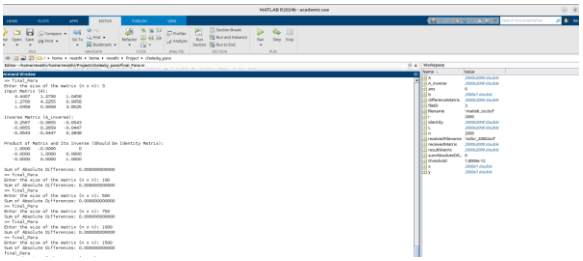


Fig: Parallel implementation using Parallel_for -Cholesky Decomposition-MATLAB-Command Prompt output shows SAD as 0 for different matrix sizes.

The below figure shows the MATLAB Identity matrix (Result), Code Identity matrix (Received) and its difference for 2000*2000 matrix

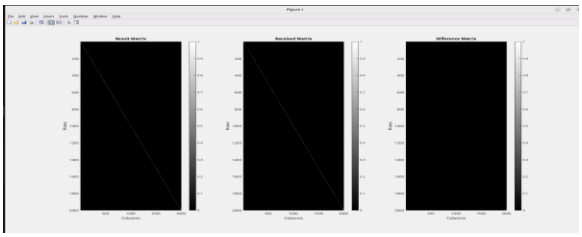


Fig: Parallel implementation of Cholesky Decomposition. MATLAB & Code-Identity matrix and its difference with matrix size of 2000*2000

We have used larger matrix size for our implementation. By analyzing the below table, we can say that the performance is achieved through parallel processing by dividing the instruction stream into a set of smaller streams i.e.) thread and distribute those threads to multiple cores present in the system to gain the efficiency of the system. In our system, 4 cores are available we have made use of this cores and achieved performance efficiently via parallel programming.

Programming methods	**Time to compute inverse (in milli-seconds)**	**Speed improvement via parallelization in %**
Sequential	17180	-
Parallel using parallel_for	2675	84

Parallel using parallel pipeline	3015	82

Table: Timing and performance comparison between sequential and different parallel implementation of Cholesky Decomposition

Code style	comments	Execution time(in milliseconds)
Sequential	Straightforward sequential implementation of Cholesky decomposition to find inverse for size 2000	17180
Intel-TBB-Parallel_for	Parallelized version of Cholesky decomposition to find inverse for size 2000	2675
Intel-TBB-Parallel pipeline and parallel_for.	We implemented the pipeline in 2 main stages. Stage 1: Generates column indices, col represents the column of the inverse matrix A^-1. Stage 2: Compute the inverse of the given matrix by doing forward and backward substitution by getting the column indices from stage1, and it will store the inverse	3015

Table: Comparison of various parallel approaches of Cholesky Decomposition Matrix inversion implementation

CONCLUSIONS

Finally, our project demonstrates the huge performance advantages obtained through parallelizing the LU and Cholesky Decomposition matrix inversion algorithms using Intell-TBB-parallel_for and pipeline. We used parallel programming approaches to optimally leverage multi-core computers, resulting in significant reductions in execution time, when compared to sequential methods. The implementation successfully solved issues like load balancing and synchronization overhead, resulting in optimal performance and scalability. This initiative not only improves the efficiency of inversion processing operations but also provides significant insights and a solid platform for future developments in parallel matrix inversion computing. The performance evaluation results outlined in this project have demonstrated that our project is suitable for achieving performance efficiency in matrix inversion algorithms through parallel programming.

REFERENCES

[1] "Matrix Inversion and Eigenvalue," SRM Institute of Science and Technology. Available: https://webstor.srmist.edu.in/web_assets/srm_mainsite/files/2018/MatrixInversionandeigenvalue.pdf

[2] "Matrix Multiplying," *Math is Fun*. Available: https://www.mathsisfun.com/algebra/matrix-multiplying.html

[3] Y. Zhang, "LU Decomposition," CAAM, Rice University, Fall 2009. Available: https://www.cmor-faculty.rice.edu/~zhang/caam335/F09/handouts/lu.pdf

[4] "Matrix Row Operations," *Khan Academy*. Available: https://www.khanacademy.org/math/algebra-home/alg-matrices/alg-elementary-matrix-row-operations/a/matrix-row-operations

[5] “ECE-5772-Lecture notes unit 4-Map Pattern”- https://moodle.oakland.edu/pluginfile.php/9

501748/mod_resource/content/5/Notes%20-%20Unit%204.pdf

[6] A. Ziefert, "Cholesky Decomposition," Matrix Algebra, Oct. 13, 2020. [Online]. Available: https://zief0002.github.io/matrix-algebra/cholesky-decompostion.html

[7] "Triangular matrix," Wikipedia, The Free Encyclopedia. [Online]. Available: https://en.wikipedia.org/wiki/Triangular_matrix