TEAM 1: LU Decomposition

Daniel Schlatter   
College of Engineering, Design and Computing  
University of Colorado at DenverDenver, USA  
daniel.schlatter@ucdenver.edu  
  
Benjamin Yee  
College of Engineering, Design and Computing  
University of Colorado at DenverDenver, USA  
benjamin.yee@ucdenver.edu  
  
Jazmin Barraza  
College of Engineering, Design and Computing  
University of Colorado at DenverDenver, USA  
jazmin.barraza@ucdenver.edu  
  
Maria Rodriguez-Yanez  
College of Engineering, Design and Computing  
University of Colorado at DenverDenver, USA  
maria.rodriguez-yanez@ucdenver.edu

*Abstract*— This project aims to design and implement parallel frameworks for solving systems of linear equations using LU Decomposition for different parallel computer platforms. The three parallel frameworks for this project were the following: OpenMP, MPI, and CUDA. In this report, in-depth analysis and description will be discussed for each of the parallel programming frameworks where each algorithm will be compared to a sequential algorithm.

# Introduction

In this paper, we will be discussing our design and implementation parallel frameworks for solving systems of linear equations in the form of Ax=b using LU Decomposition with forward and backward substitution which will be discussed in Section II.

We will design and implement three different parallel frameworks to solve this problem by comparing runtimes of each algorithm to the sequential version as our baseline. The three different parallel frameworks that we will be using for this implementation are OpenMP [shared memory], MPI [distributed memory], and CUDA [GPU-based]. In section III, we will conduct an in-depth analysis on our implementations in regard to how we solved this problem with the parallel frameworks used. In section IV, results of the three different parallel frameworks and figures of our test runs. These results will include tables and a graphical representation of our runtimes and speedups.

In our conclusion, we will conclude our paper based on the comparisons of our results and discuss why some parallel frameworks performed better than others.

# Background

Lower-Upper decomposition, otherwise known as, LU Decomposition solves systems of linear equations, Ax = b, where A is a n×n square matrix of coefficients and x is the vector of n unknown, and b is the solution vector of dimension matrix. In this implementation, all parallel frameworks that were implemented in this project, was to find the solution vector for x. To do so, matrix A is factored into L which is the lower triangular matrix and U which corresponds to the upper triangular matrix. Which leads to:

For instance, let us say we have a 3x3 where its decomposition is as follows:

**Figure 1 (LU Decomposition)**

In order to solve LUx=b, we need to solve two triangular systems of equations as seen in Figure I. To solve the triangular system, Ly= b, we first have to find the solution vector for y which is of dimension n. Then we need to solve the upper triangular system to find the solution vector y. These steps are known as forward and back substitution respectively. Once we are able to solve for L and U, it leads us to Matrix A where it was factored, and L and U were used to solve the system of equations for several right hand-sided b vectors.

**Figure 1. Matrix A**

The relevance of LU Decomposition has an advantage over the Gaussian Elimination method due to this and is used in computers where they often implement this method to solve square systems of linear equations. It is also a key step when inverting or computing a matrix.

# implementation

## Sequential

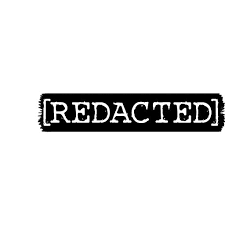
The sequential implementation of LU Decomposition consisted of implementing the forward and backward substitution. The program has three major functions InitializeMatrix, where it takes in the size and value and allocates a 2D square matrix. It then generates input matrices (a and b) and initializes the resulting matrix. DeleteMatrix and PrintMatrix are trivial as all they do is delete or show the matrix.

The function LUDecomposition takes in the three matrices and the size. Within the function we have a for loop that iterates n times to then have a nested for loop where if I is greater than j, initialize the u matrix to 0. Otherwise, the u and a matrix equal and execute the matrix.

## MPI

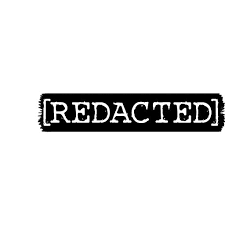
The MPI implementation of LU decomposition used the MPI methods of Scatter() and Gather() for data communication, and therefore block partitioning for our work distribution. A master process sends equal chunks of the matrix A to each worker process for performing row reductions with the Scatter method, which in turn means that the number of worker processes must be able to divide the size n of the matrix evenly (i.e., a matrix of size 8 would be distributed in 4 row chunks to two processes). Then, after each process has performed their row reductions, the master process gathers them with Gather(), into either the lower or upper matrix. Forward and backward substitution are then subsequently calculated with L and U.

When the program begins, it initializes three matrices: the randomly filled matrix A, and the two matrices filled with zeros: L and U.



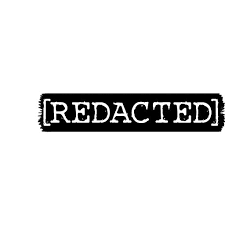
**Figure 5 (MPI Initialization)**

L and U are scattered to each process. The new master process ID is calculated, and then that process creates a pivot, and subsequently Bcast() that pivot to the other processes. The master process will also store the results of its row reductions into the lower matrix.



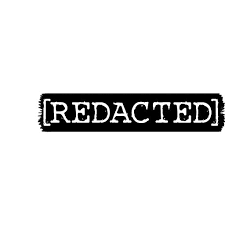
**Figure 6 (MPI Master process work)**

The other processes now receive that pivot, and compute their row reductions, and then the master gathers the results for the matrices L and U into L and U.



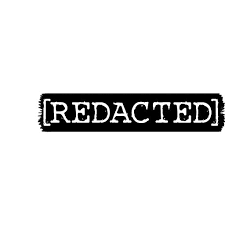
**Figure 7 (MPI worker process reductions and final Gather)**

For the forward and backward substitution, they would progress their solution step by step, and when one row gets a solution, it is broadcasted to every other row so every row can make another step forward. Forward substitution is done first with the lower matrix, starting at the bottom, each solution being broadcast one at a time and then the resulting vector of solutions y is broadcast to be used in the backward substitution.



**Figure 8 (MPI Forward substitution)**

Backward substitution progresses much the same, starting from the bottom of the upper matrix and sending out solutions one by one to each process as they come so they can all be progress step by step. The resulting solution vector is x, which is what we were looking for originally.



**Figure 8 (MPI Backward substitution)**

Now, our L and U have been calculated, and the system of linear equations has been solved using the forward and backward substitution, in MPI.

## OpenMP

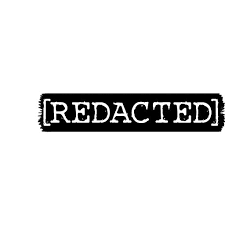
OpenMP was implemented using static scheduling and we used our sequential algorithm as a baseline. The program initializes a matrix based on the desired size input and have a function, intializeMatrix(), where it initializes the values of both the upper and lower triangular matrix. After the matrices are initialized, we move to the LUDecomposition() function where it computes the LU decomposition of the upper and lower triangular matrices.

Within the LUDecompositon function, we have a for loop the size of the matrix and have a nested for loop where we parallelized the sequential method by sharing the variables a, l, and u.

## CUDA

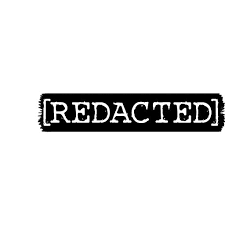
The CUDA implementation is based upon our sequential implementation. We wanted to make sure that there weren’t too many threads per block, thus we went about out implementation as follows. Our implementation prepares the respective matrices, and then adds inputs and configuration to the GPU by reference. Then, our functions to compute L and U are added to the GPU kernel.

Blocks were assigned rows and elements were assigned to block threads, we used block scheduling in which we chose a matrix block size and then rewrite matrix A of size LU. We then limited the number of threads that were allocated by the GPU.



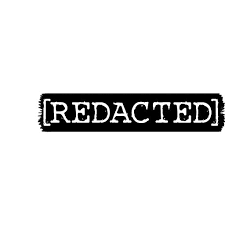
**Figure 8 (CUDA computing L and U)**

When the computations are completed, the results are copied back to the host, and it displays its matrices.

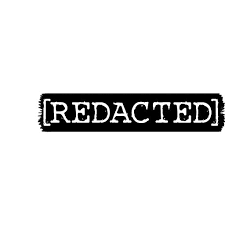


**Figure 8 (CUDA sending results)**

We then perform the forward and back substitutions upon the L and U still within the GPU:



**Figure 8 (CUDA Forward substitution)**



**Figure 8 (CUDA Back substitution)**

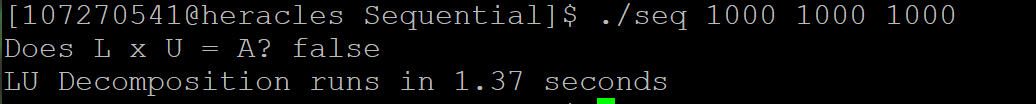
Thus, we have solved our system of linear equations.

# Analysis/results

Here we present the runtimes we received for running our programs. We have tables for each implementation, and a graph for comparison at the end.

## Sequential

Proof of correctness

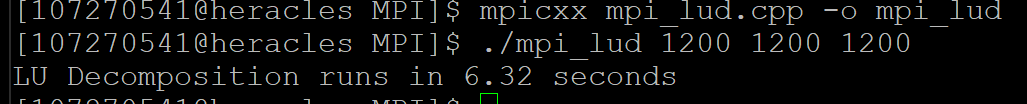


|  |  |
| --- | --- |
| Matrix Size | Runtime |
| 1200 x 1200 | 2.32[s] |
| 2400 x 2400 | 19.27[s] |
| 4800 x 4800 | 225.06[s] |
| 9600 x 9600 | 1675.38[s] |

**Table I. Sequential Runtimes**

## MPI

ntasks:24, ntasks per node: 24

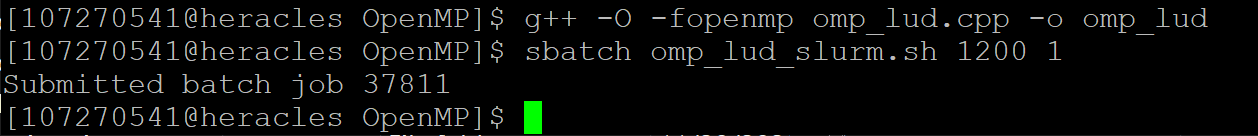


|  |  |
| --- | --- |
| Matrix Size | Runtime |
| 1200 x 1200 | 0.39[s] |
| 2400 x 2400 | 2.43[s] |
| 4800 x 4800 | 19.27[s] |
| 9600 x 9600 | 148.36[s] |

**Table II. MPI Runtimes**

## OpenMP

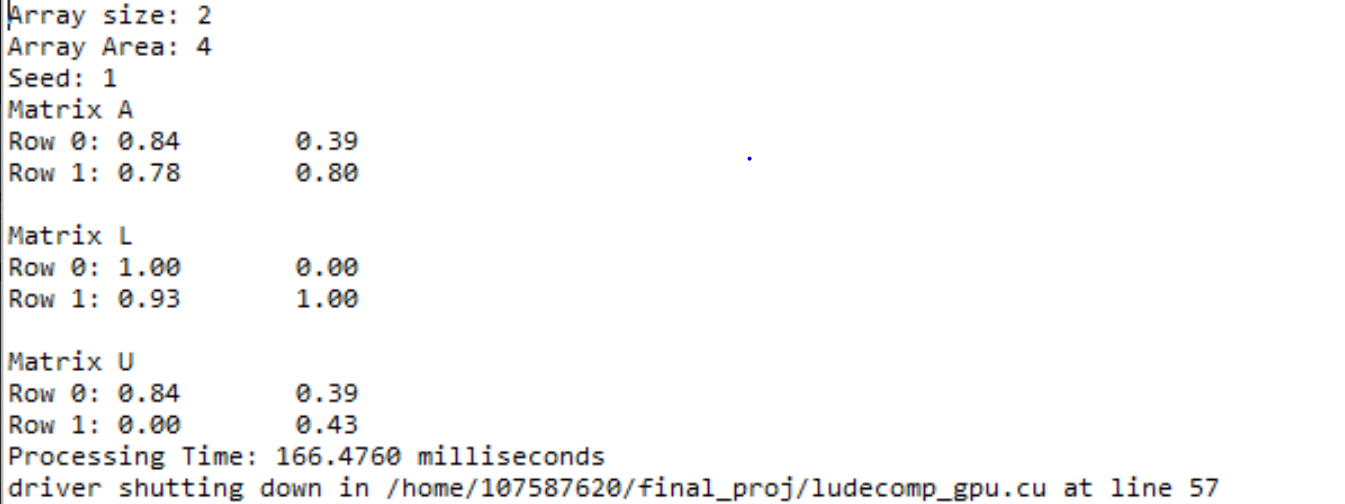
ntasks:24



|  |  |
| --- | --- |
| Matrix Size | Runtime |
| 1200 x 1200 | 2.05[s] |
| 2400 x 2400 | 16.34[s] |
| 4800 x 4800 | 7.18[s] |
| 9600 x 9600 | 150.70[s] |

## **Table 3. OpenMP Runtimes**

## CUDA



|  |  |
| --- | --- |
| Matrix Size | Runtime |
| 1200 x 1200 | 0.32[s] |
| 2400 x 2400 | 0.46[s] |
| 4800 x 4800 | 1.22[s] |
| 9600 x 9600 | 2.92[s] |

**Table 4. CUDA Runtimes**

## Comparison

# Conclusion

This project showed that the utilization of parallel processing is a must for its efficiency and effectiveness regarding processing large datasets, in this case matrices.

In conclusion, we implemented LU Decomposition using a sequential program to build the basis for MPI, OpenMP, and CUDA. In our analysis, we were able to find that all three parallel implementations had a faster runtime showing their efficiency. However, when comparing the three programs with their runtimes closely, OpenMP seemed to be the fastest with a smaller dataset but as the size of the matrices increased starting with the size 2400, it was clear that CUDA efficient in processing larger datasets. As the datasets grew, MPI had a decrease in runtime followed by OpenMP. This proved that CUDA had more efficiency regarding runtime.