# Exploring MPI and CUDA Algorithms for Parallel LU Decomposition on Large Dense Matrices

Cooper Werner
Xenia Khusid
wernec6@rpi.edu
khusix@rpi.edu
Rensselaer Polytechnic Institute
Troy, NY, USA

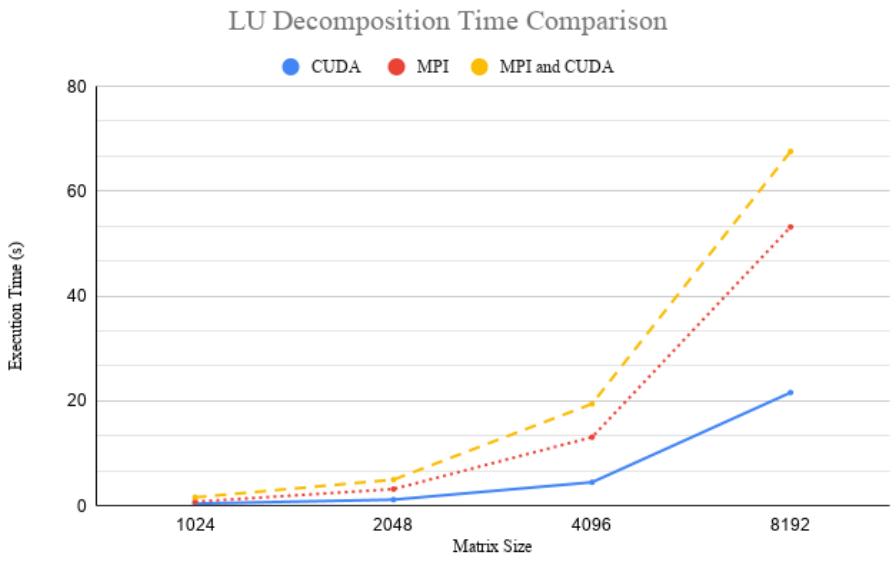


**Figure 1: LU Decomposition time in seconds for CUDA and MPI implementations.**

## Abstract

The ability to solve linear systems of equations is fundamental to a number of fields from engineering, to chemistry and computer science. The most popular method for solving systems of linear equations is through LU decomposition for its numerical stability, better performance, and reusability for a variety of input vectors. Existing research focused on parallelizing LU decomposition for smaller matrices using one parallelization strategy. As simulations become more complex, there becomes a growing need to solve larger systems of equations faster. In this paper we present a comparative performance study utilizing high performance computing for LU decomposition across MPI and CUDA implementations to evaluate their performance and scalability[1]. Our results indicate for matrices small enough to fit inside one GPU's memory ( 32GB for our setup), utilizing one CUDA device was the fastest way to perform LU Decomposition, performing 10% faster than MPI-Based Solution and 361% faster than our MPI and CUDA hybrid strategy. The poor performance of our MPI-CUDA strategy demonstrates that for matrices small enough to fit in a single CUDA device's memory, single device computation is still the best way to perform LU decomposition. For larger matrices, hybrid or multi-GPU computation could still prove to be a faster method for performing LU decomposition provided more work occurs to optimize performance.

## CCS Concepts

• **Computing methodologies** → **Massively parallel algorithms**;
• **General and reference** → *Performance.*

## Keywords

Parallel algorithm, performance

## 1 Introduction

Various tasks in engineering and the physical sciences depend on the ability to solve systems of linear equations: $\mathbf{A} \cdot x = b$ quickly and efficiently. Solving systems of linear equations is foundational for nodal analysis [13], the finite element method [1] [15], and solving partial differential equations often found in chemistry [7] and mechanical engineering [1]. The most common method for solving systems of linear equations is through LU decomposition [12]. LU decomposition provides a number of advantages over other

[1]Source code is available at: https://github.com/CooperW824/PCP-2026-Final-Project

methods. While matrix inversion ($x = \mathbf{A^{-1}b}$) takes $2n^3$ FLOPS, LU decomposition only requires $\frac{2}{3}n^3$ FLOPS [9] [12]. LU decomposition also produces a more accurate final solution than that of matrix inversion [9] [10]. The residual bound for LU decomposition is given by:

$$|\mathbf{b} - \mathbf{Ax_{inv}}| \leq \gamma|\mathbf{L}||\mathbf{U}||x|$$

The residual bound for matrix inversion is given by:

$$|\mathbf{b} - \mathbf{Ax_{inv}}| \leq \gamma|\mathbf{A}||\mathbf{A}^{-1}||x|$$

The values for $\gamma$ for the two bounds are approximately equal, and $|\mathbf{L}||\mathbf{U}|$ is approximately equal to $|\mathbf{A}|$ [10]. Since LU decomposition removes the $|\mathbf{A}^{-1}|$ term from the residual bound, it produces significantly more results. Lastly, once we have the **L** and **U** matrices, performing substitution for a new vector of $b$ is an $O(n)$ operation, compared to an $O(n^2)$ matrix multiplication required for matrix inversion.

LU decomposition breaks the matrix **A** into two, $A = \mathbf{LU}$ [12]. Where **L** is a lower triangular matrix and **U** is an upper triangular matrix. By breaking the matrix into two triangular matrices we reduce the problem from having $n$ unknowns, to one which we can solve using substitution [9].

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ \ell_{2,1} & 1 & 0 & \dots & 0 \\ \ell_{3,1} & \ell_{3,2} & 1 & \dots & 0 \\ \vdots & \vdots & \ell_{4,3} & \ddots & \vdots \\ \ell_{n,1} & \ell_{n,2} & \dots & \ell_{n,n-1} & 1 \end{bmatrix} \begin{bmatrix} u_{1,1} & u_{1,2} & \dots & u_{1,n-1} & u_{1,n} \\ 0 & u_{2,2} & \dots & u_{2,n-1} & u_{2,n} \\ & & \ddots & \vdots & \vdots \\ 0 & 0 & & & \\ 0 & \dots & 0 & u_{n-1,n-1} & u_{n_1,n} \\ 0 & 0 & \dots & 0 & u_{n,n} \end{bmatrix}$$

Once we have our matrix in the form $\mathbf{A} = \mathbf{LU}$ we can easily solve our system of linear equations, $\mathbf{A}x = b$ by finding a vector, $y$ such that $\mathbf{L}y = b$, then finding $x$ such that $\mathbf{U}x = y$. We can find our vector $y$ such that $\mathbf{L}y = b$ through forward substitution. Forward substitution finds $y$ by setting $y_1 = b_1/\ell_{1,1}$. We then substitute $y_1$ into the equation formed by the second row of the matrix, $b_2 = \ell_{2,1}y_1 + \ell_{2,2}y_2$, to reduce the equation to one unknown, which can then be solved for. Repeatedly substituting values of $y_1$ to $y_{m-1}$ allows us to find $y_m$ until we solve for the fully vector $y$. We can use a similar process, but starting from the last row of the matrix **U** and moving upwards to find the vector $x$, this is called backwards substitution.

To find the matrices **L** and **U** we can use Gaussian elimination. For each of the columns in the matrix, $i = 1..n$ we use row operations, to set all the values of $\mathbf{A}_{j,i}^{(i-1)}$ to 0 for $j = i+1..n$, where $\mathbf{A}^{(i)}$ refers to the matrix after columns $1..(i-1)$ have already been reduced. This continues until all values below the main diagonal equal 0. This reduced matrix forms the matrix **U**. To reduce the value of $\mathbf{A}_{j,i}^{(i-1)}$ to 0, we use row operations to perform $row_j = row_j - (\ell_{j,i}) \cdot row_i$. Where $\ell_{j,i} = \mathbf{A}_{j,i}^{(i-1)}/\mathbf{A}_{i,i}^{(i-1)}$ These values of $\ell_{j,i}$ create the matrix **L** [11].

To ensure numerical stability, we want to avoid dividing by very small numbers [12]. To do this, before reducing each column we want to swap the rows such that $\mathbf{A}_{i,i}^{(i-1)}$ has the largest absolute value in column $i$ of $\mathbf{A}^{(i-1)}$. This strategy is called partial pivoting [12]. This is the strategy we employ in this paper. When using partial pivoting we also must remember the order in which we swap rows. This is preserved in the matrix **P**, which is an identity matrix which has the same rows swapped during LU decomposition.

Given the importance of fast and efficient LU decomposition in a variety of fields, we wanted to analyze the performance of LU decomposition when we combined multiple parallelization strategies. This paper is organized as follows, we first start by analyzing the current state of the art for performing LU decomposition. We then discuss the algorithms that we have developed and our methodology for testing them. We analyze the performance of our algorithms. Lastly, we conclude with a summary of our findings and a few potential areas for future research.

## 2 Related Work

Due to the importance of LU decomposition in engineering and scientific computing, there has been an extensive amount of research into parallelizing Gaussian elimination for LU decomposition. Buttari, et al. [5] explored tiling algorithms to exploit instruction-level and thread-level parallelism for LU, QR, and Cholesky decomposition. They demonstrated that for their tiling algorithm, LU decomposition had an asymptotically similar performance in GFLOPS as the Intel Math Kernel Library. Their findings contributed to the tiling algorithm we used to distribute the data among MPI ranks.

Knowing that parallelization of LU decomposition was an active field of research, we sought to find which parts of the algorithm yielded the most promising parallelizability. S. Venugopal [17] highlighted that there are three parallelizable loops in Gaussian elimination with partial pivoting. These loops are the pivot selection loop, the loop that loops through each row during elimination, and the loop that performs the arithmetic for eliminating each value in a row. Venugopal also provides a number of parallel implementations individually utilizing OpenMP, MPI, and CUDA for Gaussian elimination. These implementations helped serve as the foundation for our implementations.

Once we had our tiling strategy, we also needed to decide on how we were going to perform pivoting on the matrix. Although this paper is a comparative performance study, we still wanted to achieve reasonable numerical stability and accuracy. S. Donfack et al. [8] explored different pivoting algorithms and whether different pivoting algorithms had an impact on the parallelism achievable in computing LU decomposition. In their results they found that partial pivoting had asymptotic performance as good as any other pivoting strategy. They also found the partial pivoting had the best numerical stability on random and synthetic matrices. For this reason, in our paper we choose to look at only partial pivoting LU decomposition.

Once we had the details of our algorithm defined, we worked to find existing parallel implementations that we could compare results to. Noveski and Hadzieva [13] looked at parallelizing LU decomposition for use in nodal analysis on impedance networks. In their use case the matricies were fairly sparse, this meant that they could do a pre-processing step to reduce the amount of processing needed to perform LU decomposition. In their experiment they used OpenMP to parallelize their LU decomposition. They were able to harness an impressive amount of parallelism, demonstrating very good strong and weak scaling results for a small number of

threads. Since the matrices we are testing with are dense matrices, their reduction step would not show as much benefit for our use case.

Fadi N. Sibai [14] analyzed the scaling performance of Gaussian elimination, across a large number of CPU cores when paired with single instruction multiple data (SIMD) instructions and a parallel back substitution algorithm called meet in the middle. Sibai demonstrated that Gaussian elimination demonstrates very good strong scaling and weak scaling results as its parallelized among many CPU cores. Sibai also found that the most intensive part of solving a linear equation using Gaussian elimination was the back substitution required to find the final results. For our purposes, this is not a concern as we are only computing the **U** matrix. The use of SIMD instructions and meet in the middle back-substitution does represent an interesting future field of research for large system of linear equation solvers.

S. Tomov et al. [16] developed a hybrid LU decomposition using both the CPU and GPU. Their implementation uses one GPU and up to 8 CPU cores to perform LU decomposition using a tiled technique. One of the observations they highlight is that there is significant overhead incurred from partial pivoting since memory has to move so frequently. They highlight that its possible to pre-condition the matrix such that pivoting is not required, this method reduced the precision of the result by approximately 4 decimal places for a $7000x7000$ matrix, but created a 30% increase in GFLOPS compared to LU decomposition with partial pivoting.

Existing research in this field demonstrated that LU decomposition was a highly parallelizable algorithm that could be distributed across many processors and CUDA devices. We utilized the existing research to inform the design of our implementations and give a baseline for performance. Existing work also gave plenty of optimization ideas and areas for exploration and optimization of our algorithm in the future.

# 3 Methodology

## 3.1 Overview

In this section, we introduce the various ways we implemented parallel algorithms for LU decomposition with partial-pivoting. Given an $n \times n$ matrix, we perform LU decomposition to transform the matrix into upper-triangular row echelon form. For all of our implementations, the input and output files are binary files, they simply store the raw bytes for each 64-bit float. In the command line arguments the user specifies the size of the matrix, and the implementation is responsible for allocating and reading the correct number of bytes from the file into memory. The focus of our implementations is to parallelize different aspects of this algorithm and analyze the relative performance. We used CUDA and MPI to accomplish this parallelism, creating CUDA only, MPI only, and MPI-CUDA implementations, as well as a serial algorithm to compare their execution times to.

## 3.2 Serial Implementation

The serial algorithm, with $O(n^3)$ time complexity, provides a baseline for comparison to the parallel implementations. The C serial code closely follows the pseudo-code shown in Algorithm 1, going sequentially through the columns of the matrix, finding the pivot row, swapping it with the current row, and applying the pivot to each later row. To input and output the matrix files, we use the $read()$ and $write()$ system calls, which are relatively optimized but slow down significantly on larger matrices.

**Algorithm 1:** Pseudocode for LU Decomposition

```
   Input  : Input size n, input file with 2^n × 2^n matrix
   Output : Upper triangular matrix written to output file
1  matrix_size ← 2^n;
2  matrix ← read_matrix(input, matrix_size);
3  for i = 0 to matrix_size do
4      for j = i to matrix_size do
5          if pivot_value < |matrix[j][i]| then
6              pivot_value ← matrix[j][i];
7              pivot_index ← j;
8          end
9      end
10     pivot_row ← matrix[pivot_index] ;
11     matrix[pivot_index] ← matrix[i] ;
12     matrix[i] ← pivot_row ;
13     pivot ← matrix[i][i] ;
14     for j = i to matrix_size do
15         factor ← matrix[j][i] ;
16         for k = i to matrix_size do
17             matrix[j][k] −= factor×(matrix[i][k]÷pivot)
18         end
19     end
20 end
21 output ← write_matrix(matrix, matrix_size) ;
```

## 3.3 CUDA Implementation

The CUDA-only implementation utilizes the parallel threads of a GPU to speed up the time it takes to calculate the maximum pivot and apply it to the matrix. A change between the serial implementation and the CUDA one is that rather than storing an actual 2D matrix, we flatten it out into a 1D shape for easier distribution across threads. This also simplifies the matrix read and matrix write implementation.

We parallelized calculating the pivot row loop at line 4 by replacing it with a CUDA kernel. The pivot kernel uses block striding to search through all of the indexes in the target column, as the matrix size can get very large ($2^{15}$). We use shared memory to store the maximum values that each thread finds and the index they were at to avoid the latency of accessing global memory, as these values are read and written to often. We then use a parallel reduction over the shared memory to find the global maximum value, and the thread with index 0 writes the maximum value and its index to global memory.

The row swapping step of partial pivoting is done on the CPU, similarly to the serial implementation. Although it would be possible to optimize the CUDA program further by parallelizing this step as well, we have chosen not to, as this takes up a less significant amount of the elimination step's time compared to updating each row entry with elimination.

The second CUDA kernel executes the elimination step (line 14 of the pseudocode) in a massively parallel way. Each thread block is mapped to one row of the matrix below the pivot row. We have 1024 threads in each block, with the threads striding over the columns in the target row. This parallelizes the process of performing the row-reduction update.

## 3.4 MPI Only Implementation

To parallelize the LU decomposition algorithm across multiple MPI ranks, we must have a loop where each loop iteration is independent of every other iteration. The loop at line 3 of the psuedocode is not a good candidate for parallelization as the $n$'th iteration relies on the $n-1$'th iteration having been fully finished. The loops at lines 4 and 14 make great candidates for parallelization since they operate on independent rows of the matrix. For this reason, the MPI implementation looks to parallelize these loops by splitting the entire $n \times n$ matrix into $\lfloor \frac{n}{p} \rfloor \times n$ sub-matrices, where $p$ is the number of MPI ranks. This implementation distributes contiguous sections of rows to different MPI ranks, each rank responsible for performing the computation on its subsection of rows. This implementation of the algorithm has one significant limitation, the number of rows in the matrix must be evenly divisible by the number of MPI ranks, although this could be overcome by adapting the algorithm to allow for differently sized matrices.

We implemented two methods of distributing the rows to the different MPI ranks. The serial IO method reads the entire matrix in to rank 0's memory, and uses `MPI_Scatter` to distribute the matrix data to the ranks for processing. Once processing is complete, this implementation calls `MPI_Gather` to collect the upper-triangular matrix to rank 0 so that it can be written to a file. The parallel IO implementation uses the MPI collective file operations (`MPI_File_read_at` and `MPI_File_write_at`) to parallelize file operations.

For the MPI-Only implementations, since the data for each row is split across MPI ranks, special attention needs to be paid to how we perform our partial pivoting operation on each iteration. Each MPI rank copies their pivot candidates into a separate array (pseudocode line 3). Each rank's candidates are gathered by the rank that is responsible for the current GE iteration. This rank performs an argmax operation to find the row that has the greatest pivot value. If both rows to swap are in the same submatrix, the rank swaps them locally in memory, broadcasting $\{-1, -1\}$ to the rest of the ranks to tell them that the pivoting operation is complete (line 16). If the rows are in different submatrices, then MPI send and receive operations are used to swap the two rows between the ranks' submatrices ( line 29). The pivoting algorithm could likely be sped up significantly by having each of the MPI ranks find the maximum pivot candidate of their submatrix individually and report only the value and row number. This would parallelize a portion of the argmax operation and reduce the amount of data being sent via MPI. We did not have enough time to implement and test this operation, and would be valuable to explore in future work.

**Algorithm 2:** Distributed Partial Pivoting for Gaussian Elimination

**Input:** $n$: number of rows; $r$: current GE step; $A$: matrix after step $r-1$;
$k$: rank responsible for step $r$; $i$: first row owned by this rank;
$x$: number of rows owned by this rank

1 $arr \leftarrow$ array of length $x$;
2 $arr\_out \leftarrow$ array of length $n$;
3 **for** $j \leftarrow i$ **to** $i + x - 1$ **do**
4  **if** $j < r$ **then**
5  $arr[j-i] \leftarrow 0$;
6  **else**
7  $arr[j-i] \leftarrow A[j][r]$ `// Pivot candidate at row j, column r`
8  **end**
9 **end**
10 `MPI_Gather`($arr$, $x$, $arr\_out$, $n$, dest: $k$)`// Gather all pivot candidates to rank k`
11 $best\_pivot \leftarrow -1$;
12 $rows\_to\_swap \leftarrow [-1, -1]$;
13 **if** $rank = k$ **then** `// Rank k selects the best pivot`
14  $best\_pivot \leftarrow \arg\max_j |arr\_out[j]|$;
15  **if** $best\_pivot \geq i$ **and** $best\_pivot < i + x$ **then**
16  **swap**($r$, $best\_pivot$);
17  $rows\_to\_swap \leftarrow [-1, -1]$;
18  **else**
19  $rows\_to\_swap \leftarrow [r, best\_pivot]$;
20  **end**
21 **end**
22 `MPI_Bcast`($rows\_to\_swap$, src: $k$)`// Broadcast swap decision to all ranks`
23 **if** $rows\_to\_swap = [-1, -1]$ **then**
24  **return** `// Pivot already local to rank k; no communication needed`
25 **end**
26 $best\_pivot \leftarrow rows\_to\_swap[1]$;
27 $requests \leftarrow$ [`MPI_REQUEST_NULL`, `MPI_REQUEST_NULL`];
28 $statuses \leftarrow$ [`MPI_STATUS_NULL`, `MPI_STATUS_NULL`];
29 **if** $rank = k$ **then** `// Rank k exchanges row r with owner of best pivot`
30  $owner \leftarrow \lfloor best\_pivot / x \rfloor$;
31  `MPI_Isend`($A[r]$, dest: $owner$, $requests[0]$);
32  `MPI_Irecv`($A[r]$, src: $owner$, $requests[1]$);
33 **end**
34 **if** $best\_pivot \geq i$ **and** $best\_pivot < i + x$ **then** `// Owner of best pivot exchanges with rank k`
35  `MPI_Isend`($A[best\_pivot]$, dest: $k$, $requests[0]$);
36  `MPI_Irecv`($A[best\_pivot]$, src: $k$, $requests[1]$);
37 **end**
38 `MPI_Waitall`($requests$, $statuses$)`// Wait for both non-blocking transfers to complete`
39 **return**;

### 3.5 MPI and CUDA Parallel Implementation

The MPI and CUDA parallel implementation utilizes MPI to distribute data among many different GPUs that then use CUDA to perform the reduction operation. We use cudaMallocManaged [4] to allocate the memory such that it is unified between CPU and GPU. We use MPI collective file IO operations to read the matrix from the input file and to write the resulting **U** matrix to the output file. We do this because as we will discuss later, it creates a significant speedup in IO times as compared to serial IO.

Of the three parallelizable loops, the GE process, we use CUDA to parallelize two of them. We use the CPU to find the max pivots and perform the MPI communication for swapping the rows. We then use CUDA to perform the elimination step (line 14), utilizing one block per row to eliminate, and up to 1024 threads for performing the elimination on each row.

Since we are constantly transferring the matrix data between the CPU and GPU, we incur a significant performance penalty from memory transfer overhead. We attempted to use CUDA for finding the max pivot using a parallel reduction, as in the CUDA only implementation. We unfortunately ran into a significant memory corruption bug that significantly impacted the correctness of the implementation when trying to utilize this method of identifying the pivot. We were unable to resolve this bug before the deadline of this paper, so this kernel is not present in the final version of the MPI and CUDA implementation, but we believe that this method for identifying the pivot could provide a significant speedup and is worth additional research.

In summary, our serial implementation provides us with a baseline to calculate the speedup of our parallel implementations, and the CUDA implementation aims to dramatically decrease the computation time by using thousands of parallel GPU threads. Both MPI only implementations utilize parallel ranks to speed up the LU decomposition execution time, and the MPI parallel IO improves on the MPI serial IO implementation by also spreading out the workload for read and write. The MPI-CUDA implementation explores whether we can achieve further improvements by applying the benefits of both CUDA and MPI parallelism. We evaluate the relative success of our implementations with a performance analysis.

## 4 Performance Evaluation

### 4.1 Experimental Design

We ran all of our experiments on Rensselaer Polytechnic Institute's AiMOS supercomputer. Each node is configured with two IBM POWER-9 processors with 40 cores each, 512GiB of RAM, and 6 NVIDIA V100 GPUs each equipped with 32GB of VRAM [6]. For all experiments, we allocate all GPUs on the node to avoid our performance results being interfered with by node-sharing with other users. For the MPI experiments we allocated two nodes and ran 32 processes on each node, leaving 8 cores free for handling other operating system related processes. For the MPI + CUDA experiments we allocated up to 11 nodes, for a total of 66 accessible GPUs, although in our experiments, we used a maximum of 64 GPUs at once.

To ensure accurate timing for our performance results, we use the time base registers (mftb and mftbu) available on the IBM POWER-9 system processors [6]. The time base registers are incremented at a clock rate of 512MHz [6], providing accurate timing down to 2 nanoseconds. We are timing three different sections of the algorithm: matrix read, Gaussian elimination, and matrix write.

We are comparing five distinct implementations: serial, CUDA, MPI parallel IO, MPI serial IO, and MPI and CUDA. It should be noted that the only difference between the two MPI only implementations is whether one rank is responsible for all reading and writing or whether the parallel MPI collective file operations are used for reading and writing. This means that they share the same LU decomposition code and that the elimination step times will be nearly identical for these two implementations, and we are really only comparing their read and write times. The MPI and CUDA implementation uses the same read and write as the MPI parallel IO implementation, and the same CUDA kernel as the CUDA only implementation, so we can explore whether the benefits of each outweigh the overhead of copying the memory from the CPU to GPU and back.

*4.1.1 Weak Scaling.* The matrix size for our experiments is the number of rows in the square matrix, with our experiments varying from $2^{10} = 1024$ to $2^{15} = 32768$. The serial implementation is a baseline for our parallel speedups. We performed a weak scaling performance study by increasing the number of MPI ranks for each of the MPI implementations as matrix size increases, as shown in Table 1. The implementations using CUDA are by default weak scaling as the matrix size increases, since the number of thread blocks is equal to the number of rows.

**Table 1: Number of MPI Ranks for each Matrix Size**

| Matrix Size | 1024 | 2048 | 4096 | 8192 | 16384 | 32768 |
|---|---|---|---|---|---|---|
| # MPI Ranks | 2 | 4 | 8 | 16 | 32 | 64 |

*4.1.2 Strong Scaling.* For the strong scaling experiments, we applied the same range of the number of MPI Ranks as in the weak scaling experiments (Table 1) to the matrix size $2^{14}$. By maintaining the same task size, we are able to see how much the number of processors affects the length of time our computation takes.

### 4.2 IO Performance

One of the benefits provided by MPI is collective file operations that can read from and write to a parallel file system, like the one equipped on RPI's AiMOS supercomputer, faster than a single node can. The largest matrix we tested with was a matrix of $2^{15} \times 2^{15}$ doubles, resulting in a matrix larger than 8GB in size. Our IO experiments sought to analyze both strong and weak scaling performance.

*4.2.1 Weak Scaling Performance.* When we compare the Serial IO implementation to the Parallel IO implementation at different matrix sizes and available resources, we see very strong weak-scaling performance. For smaller matrices, the performance between serial

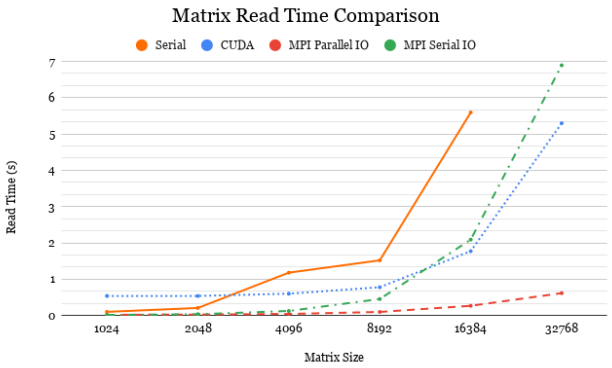


**Figure 2: Comparison of matrix read times for all of the implementations of LU decomposition, with weak scaling for the MPI versions.**

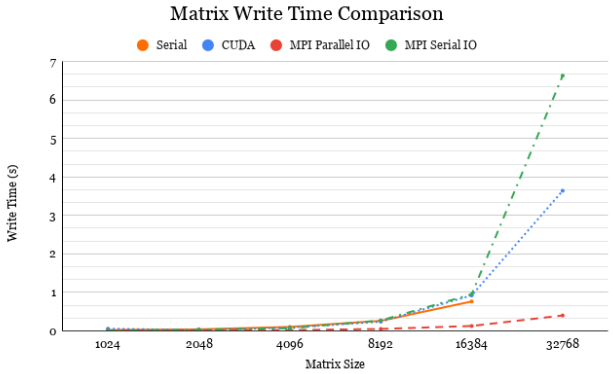


**Figure 3: Comparison of matrix write times for all of the implementations of LU decomposition, with weak scaling for the MPI versions.**

**Table 2: Input Matrix Read and Distribute Time For Each Implementation**

| Matrix Size | Serial | CUDA | MPI Parallel IO | MPI Serial IO |
|---|---|---|---|---|
| 1024 | 0.098 | 0.536 | 0.016 | 0.013 |
| 2048 | 0.205 | 0.536 | 0.024 | 0.032 |
| 4096 | 1.179 | 0.598 | 0.041 | 0.123 |
| 8192 | 1.517 | 0.776 | 0.096 | 0.452 |
| 16 384 | 5.591 | 1.770 | 0.265 | 2.088 |
| 32 768 | – | 5.295 | 0.613 | 6.891 |

IO and parallel IO is approximately the same. However, as the matrices get larger, there is a significant speedup in the time required to load and distribute the submatrices to each of the MPI ranks. Once the matrix is large, there was an 11.2 times speedup (Table 2) for loading the input matrix using MPI parallel IO as compared to loading the matrix serially using the Linux `read()` [2] system call.

**Table 3: Output Matrix Gather and Write Time for Each Implementation**

| Matrix Size | Serial | CUDA | MPI Parallel IO | MPI Serial IO |
|---|---|---|---|---|
| 1024 | 0.024 | 0.057 | 0.011 | 0.011 |
| 2048 | 0.040 | 0.021 | 0.015 | 0.033 |
| 4096 | 0.103 | 0.085 | 0.019 | 0.068 |
| 8192 | 0.261 | 0.239 | 0.050 | 0.271 |
| 16 384 | 0.764 | 0.922 | 0.128 | 0.952 |
| 32 768 | – | 3.640 | 0.400 | 6.628 |

When we look at the time it took to write the resulting **U** matrix to disk, we see very similar results to the time required to read the matrix. The write operations followed the same pattern as the read operations. For small matrices, MPI parallel write performed similarly or worse than a serial read operation. For larger matrices and more processors, there was a significant speedup of over 16.5 times (Table 3) compared to Serial write operations.

Both the read and write results demonstrate an interesting result regarding IO with MPI. Both serial IO and CUDA implementations utilize the same method to read or write the matrix to or from disk. They both allocate a buffer the size of the full matrix and use the `read()` and `write()` system calls to read the buffer from the binary matrix files. The only difference between the CUDA and the MPI implementations is that the MPI implementation then needs to scatter / gather the matrix to / from the ranks. By subtracting the CUDA implementation matrix read/write times from the MPI Serial IO matrix read/write times, we can see that for reading, about 22.1% of the time is spent scattering the matrix to each of the ranks. We also see that about 45.1% of the time that the MPI Serial IO implementation spends writing the output matrix do disk is spent gathering the results from all of the ranks.

Usually for weak scaling results, the optimal result is that the amount of time required to load twice as much data with twice as many resources is constant. We don't see that trend in our parallel IO results, instead we see that the amount of time required to read or write the matrices approximately doubles. This makes sense when we consider how fast matrices grow, for a side length of size $n$, the number of elements in a matrix grows at $n^2$. So if we are double the size of the matrix, we should actually expect four times as many elements that we need to load. Since we have four times as many elements to load, and only two times the number of processors performing the matrix load operation. For this reason, matrix loading times doubling on as we double the matrix size and number of resources represents very good weak scaling results.

As Figure 4 demonstrates, there are decreasing returns to adding more resources to reading in the input matrix. We found that for both read and write, the speedup as compared to serial IO is approximately equal to $O(\sqrt{r})$ ($R^2 = 0.869$ for reading, and $R^2 = 0.948$ for writing) where $r$ is the number of MPI ranks performing the IO. This result highlights that while for the sizes of matrices we tested,

[2]The `read()` and `write()` system calls have a maximum of 2GB of data read / written per call [2] [3]. For the largest matrices, we needed to call `read()` /`write()` at least 4 times.

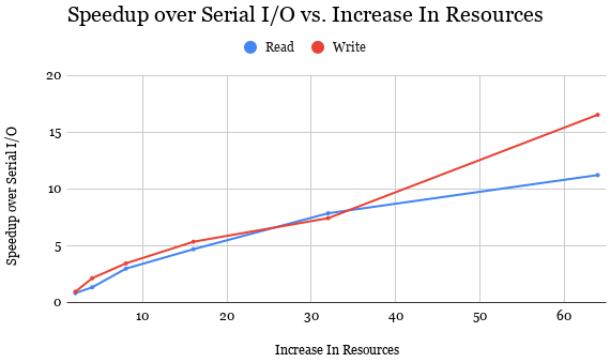


**Figure 4: Comparison of Parallel IO Speedup and the amount of resources allocated for performing the parallel file IO.**

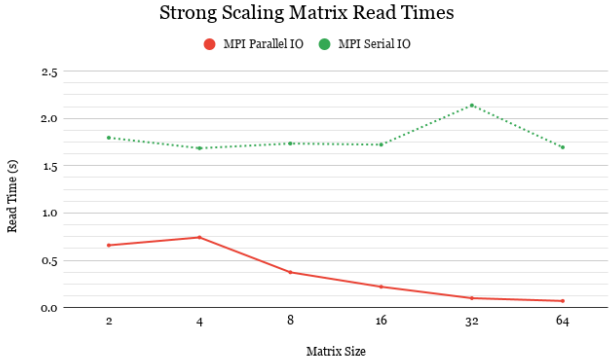


**Figure 5: Strong Scaling Matrix Read for Matrix Size $2^{14}$.**

we got good scaling results, there are still diminishing returns to adding more ranks for file IO.

*4.2.2 Strong Scaling Performance.* Strong scaling performance with MPI Parallel IO was also very good. Read operations benefited more from an increased number of MPI ranks with diminishing returns arriving around 32 ranks. For write operations, there were diminishing returns after 16 ranks. Its important to note that even through write operations reached diminishing returns sooner than the read operations, there were not any significant slow down to writing back to disk from adding more MPI ranks.

Overall for both the read and write operations, the strong scaling results were consistent with what we expected. Introducing any amount of parallel IO resulted in a 2.73 times speedup to read and distribute the submatrices to each of the MPI ranks and a 1.89 times speedup to write the resulting matrix back to disk. At 64 MPI ranks there was a 24.6x speedup to read the input matrix from disk, and a 9.91x speedup to write the resulting matrix back to disk.

The MPI parallel IO performance was very strong for our use case, especially once the matrices started to gigabyte scale. For this reason we implemented the parallel IO into the MPI and Cuda implementation. Its important to note two things here, the time spent reading or writing the matrices from / to disk does not impact the time required to compute the LU decomposition of the matrix. It only impacts the total execution time.

The other important note is that the time spent loading the matrix is a small fraction of the total execution time, most time is spent computing the LU decomposition instead of performing IO. For this reason, Amdahl's law suggests that further optimization efforts be focused on optimizing the time required to perform the LU decomposition rather than optimize IO operations.

## 4.3 LU Decomposition Performance

*4.3.1 Weak Scaling.* The execution times for the elimination step of LU decompositions show in Table 6 that any of our parallelization attempts are significant improvements compared to the serial runtime, with greater improvements as the matrix size increases. The serial implementation timed out and was not able to complete execution on the $2^{15}$ matrix, although we predict that it would have

**Table 4: Time (in seconds) required to read and distribute the submatrices to the different ranks (Matrix Size = $2^{15} \times 2^{15}$)**

| # MPI Ranks | MPI Parallel IO | MPI Serial IO |
|---|---|---|
| 2 | 0.658 | 1.795 |
| 4 | 0.741 | 1.684 |
| 8 | 0.372 | 1.734 |
| 16 | 0.219 | 1.721 |
| 32 | 0.098 | 2.138 |
| 64 | 0.069 | 1.694 |

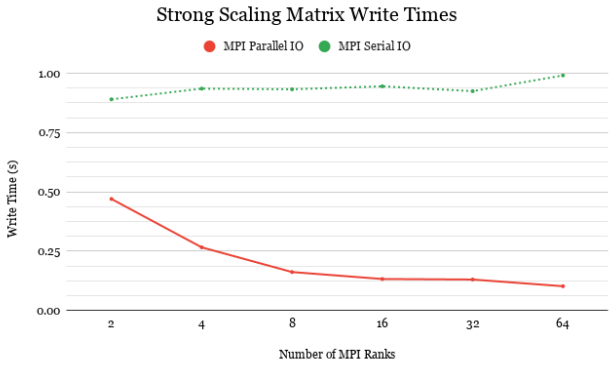


**Figure 6: Strong Scaling Matrix Write for Matrix Size $2^{14}$.**

taken about 53 hours to run. The CUDA implementation achieved between 15× and 172× speedup as compared to the serial implementation, and the MPI only implementations reached a nearly 100× speedup for matrix size $2^{14}$.

The CUDA implementation provided the highest speedup compared to the serial version across all matrix sizes. The MPI implementations performed almost as well as the CUDA one, and the gap between the two decreases as the matrix size increases. This is likely because the CUDA kernel is able to have many more threads than the MPI versions, even at smaller matrix sizes, but the MPI implementation catch up as the matrix size and the number of ranks

**Table 5: Time (in seconds) to gather and write result matrix to disk. (Matrix Size = $2^{15} \times 2^{15}$)**

| # MPI Ranks | MPI Parallel IO | MPI Serial IO |
|---|---|---|
| 2 | 0.470 | 0.890 |
| 4 | 0.265 | 0.935 |
| 8 | 0.161 | 0.932 |
| 16 | 0.132 | 0.945 |
| 32 | 0.130 | 0.924 |
| 64 | 0.101 | 0.991 |

increases. The execution time for both the CUDA and MPI implementations increases at about the same rate as each other when the matrix size increases (Figure 1). However, the MPI parallel IO seems to be closing the gap between its computation's execution time and the CUDA implementation's by matrix size $2^{15}$, so it may scale better if run on even larger matrixes. Interestingly, the MPI+CUDA implementation that uses both MPI ranks and the GPUs, performs worse than the MPI and CUDA implementations each do separately because of the context switching necessary to move the memory from the CPUs to the GPUs and back. Even though this strategy does increase the possible parallelism of the computation, the additional overhead that was introduced cancels out the potential benefits.

**Table 6: Execution Times for Elimination Step**

| | Runtime of Elimination Step (s) | | | | | |
|---|---|---|---|---|---|---|
| Matrix Size | 1024 | 2048 | 4096 | 8192 | 16384 | 32768 |
| Serial | 5.63 | 48.1 | 379 | 2997 | 24244 | – |
| CUDA | 0.365 | 1.13 | 4.45 | 21.6 | 141 | 1006 |
| MPI Parallel IO | 0.732 | 3.14 | 13.1 | 53.2 | 250 | 1118 |
| MPI Serial IO | 0.731 | 3.14 | 13.0 | 52.9 | 247 | 1109 |
| MPI and CUDA | 1.57 | 4.94 | 19.4 | 67.6 | 1847 | 4638 |

*4.3.2 Strong Scaling.* Measuring the LU decomposition time in the strong scaling experiment allows us to compare the benefit of an increased number of MPI ranks in the MPI only implementations to the MPI and CUDA implementations (Figure 7). For the MPI only implementations, we can see a clear improvement in execution time as the number of MPI ranks increases. Unlike the clear trend of the MPI only execution time, the MPI and CUDA implementation execution time gets worse before improving again. Although for MPI and CUDA, 4 processors is a performance improvement compared to 2 processors, the execution time for the same sized matrix increases up through 16 processors before decreasing again. There is increased overhead to sending out the matrix data between the higher numbers of processes that does not get compensated by the increased parallelism. This result could indicate a potential avenue for optimizing this algorithm in the future.

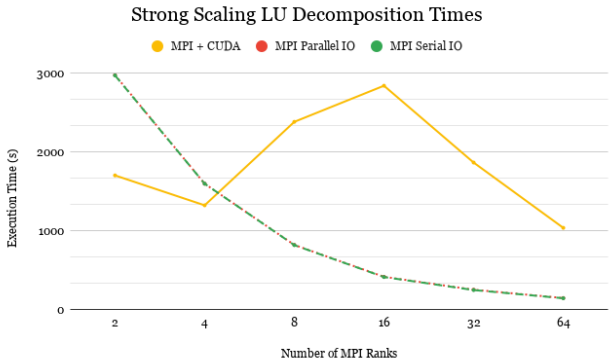


**Figure 7: Strong Scaling LU Decomposition Execution Times for Matrix Size $2^{14}$.**

## 5 Conclusion

We developed five distinct massively parallel implementations of the LU decomposition algorithm for large matrices. A simple serial implementation serves as a basis for comparison. The CUDA implementation utilizes thousands of concurrent threads to parallelize finding the maximum pivot and performing the elimination computation. We wrote two MPI only implementations, both of which use the many ranks for parallelizing the computation, but the MPI serial IO implementation has one rank handle all reading and writing, while the MPI parallel IO implementation has all the ranks participate in reading and writing via collective operations. Our final implementation combines both MPI parallel IO reading and writing and MPI rank intercommunication with the CUDA elimination kernel.

We have found that the CUDA implementation provides the fastest speedup for the computation time relative to the serial implementation for all matrix sizes, due to the sheer volume of parallel threads performing the computations. The MPI parallel IO implementation has the second fastest overall time, matching the CUDA implementation time for $2^{15} \times 2^{15}$ matrices, and taking the lowest amount of time to read and write the matrices. The MPI parallel IO implementation may scale better than the CUDA implementation for larger matrix sizes than those we have tested, but this would rely on the ability to provide many more processors for the MPI to work with, while this CUDA implementation uses only one GPU. Although the combined MPI and CUDA implementation performed significantly worse than the separate implementations, it was still an improvement on the serial implementation and has the potential for further optimizations in the future.

### 5.1 Future Work

There are many possible routes one could take to continue this project. The combined MPI and CUDA implementation especially may have the potential to achieve a much faster runtime if optimized, something we were not able to accomplish due to the constraints of time. As we noted in the description of the MPI and CUDA algorithm, there is the opportunity for a significant

performance improvement if we could utilize CUDA for pivot selection. We believe there are a number of opportunities to reduce the amount of data being transferred between the CPU and GPU which could also result in a potential speedup over our current results.

For the CUDA implementation, we used a relatively simple block-striding algorithm that does not necessarily scale the best for very large matrices, so it would be very interesting to experiment with tiling, to see whether this improves performance and makes the implementation more scalable. The MPI only solutions would likely benefit from code profiling to assess which sections of the code are the slowest and exploring how that runtime can be optimized. The MPI portions of our code would also likely benefit from other parallelization strategies like the use of SIMD vector instructions for finding the maximum pivot candidate or for performing the reduction step with more parallelism.

It would also be fascinating to see the runtime of these algorithms extended to even larger matrices, to for example see whether the MPI parallel IO algorithm execution would surpass the CUDA algorithm's runtime for matrix sizes $2^{16}$ or higher. We did not conduct this experiment because each $2^{16} \times 2^{16}$ matrix would be 32GiB, larger matrices would take up even more space on the supercomputer's storage system and we wanted to be mindful of other users of the supercomputer. Similarly, in our experiments we used synthetic randomly generated matrices, it would be exploring whether our results on generated matrices are consistent with the results for matrices from real world engineering applications like finite element analysis. Future research could determine whether the read, write, and computation time of our generated matrices is reflective of the real matrices, and whether different optimizations of our algorithms are necessarily.

Although our implementations failed to create any significant speed up when compared to existing approaches, we demonstrated performing LU decomposition on larger matrices than prior research. There are also a significant number of optimizations that could be done to improve the performance of our implementations allowing for even larger matrices to be decomposed in the future. More efficient and effective LU decomposition algorithms will be crucial as engineering simulations continue to grow in detail and complexity.

## 6 Acknowledgments

We thank Rensselaer Polytechnic Institute's Center for Computational Innovations for allowing us to utilize their AiMOS supercomputer to perform our experiments. We also thank Dr. Chris Carothers for his advice on how to approach debugging our programs and for his informative lectures for the parallel programming and computing course this semester.

## References

[1] [n. d.]. Introduction to Finite Element Analysis (FEA) or Finite Element Method (FEM). https://www.engr.uvic.ca/~mech410/lectures/FEA_Theory.pdf
[2] Linux Contributors . 2025. read(2) - Linux manual page. https://man7.org/linux/man-pages/man2/read.2.html
[3] MPICH Contributors . 2024. Manpages | MPICH. https://www.mpich.org/documentation/manpages/
[4] NVIDIA . 2026. NVIDIA Documentation Hub. https://docs.nvidia.com/
[5] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. 2018. A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. *ArXiv* 0709 (10 2018). https://arxiv.org/pdf/0709.1272
[6] RPI CCI. 2024. DCS Cluster - Center for Computational Innovation - Documentation. https://docs.cci.rpi.edu/clusters/DCS_Supercomputer/
[7] A. B. Coon and M.A. Stadtherr. 1989. Parallel implementation of sparse LU decomposition for chemical engineering applications. *Computers Chemical Engineering* 13 (08 1989), 899–914. doi:10.1016/0098-1354(89)85063-X
[8] Simplice Donfack, Jack Dongarra, Mathieu Faverge, Mark Gates, Jakub Kurzak, Piotr Luszczek, and Ichitaro Yamazaki. 2014. A survey of recent developments in parallel implementations of Gaussian elimination. *Concurrency and Computation: Practice and Experience* 27 (06 2014), 1292–1309. doi:10.1002/cpe.3306
[9] Gregory Gundersen. 2020. Why Shouldn't I Invert That Matrix? https://gregorygundersen.com/blog/2020/12/09/matrix-inversion
[10] Nicholas Higham. 2002. *Matrix Inversion.* SIAM.
[11] Nicholas J. Higham. 2011. Gaussian Elimination. *Wiley Interdisciplinary Reviews: Computational Statistics* 3 (04 2011), 230–238. doi:10.1002/wics.164
[12] Mark Holmes. 2016. *Matrix Equations.* Springer.
[13] Filip Noveski and Elena Hadzieva. 2026. Parallel Gauss-Jordan Elimination and System Reduction for Efficient Circuit Simulation. https://arxiv.org/abs/2603.28792
[14] Fadi N Sibai. 2013. Performance modeling and analysis of parallel Gaussian elimination on multi-core computers. *Journal of King Saud University - Computer and Information Sciences* 26 (03 2013), 41–54. doi:10.1016/j.jksuci.2013.03.002
[15] Mehdi Toloo and Rahele Jalili. 2015. LU Decomposition in DEA with an Application to Hospitals. *Computational Economics* 47 (06 2015), 473–488. doi:10.1007/s10614-015-9501-z
[16] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. 2010. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Comput.* 36 (06 2010), 232–240. doi:10.1016/j.parco.2009.12.005
[17] Sesh Venugopal. 2026. Gaussian Elimination. https://people.cs.rutgers.edu/~venugopa/parallel_summer2012/ge.html

Received 28 April 2026