LU decomposition can be programmed from scratch for different computer platforms.

Before we begin talking about how we parallelized LU decomposition, it is important to understand what LU decomposition is exactly and how it is calculated and that is exactly what we will discuss in the next few paragraphs.

LU decomposition of a matrix is the factorization of a given square matrix into two triangular matrices, one upper triangular matrix and one lower triangular matrix, such that the product of these two matrices gives the original matrix. A square matrix A can be decomposed into two square matrices L and U such that A = LU where U is an upper triangular matrix formed as a result of applying Gaussian Elimination Method on A; and L is a lower triangular matrix with diagonal elements being equal to 1.

For A = $\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$, we have L = $\begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix}$ and U = $\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$; such that A = L U.

Figure 1. Square Matrix A

One of the most common methods to solve triangular matrices is by using forward and backward substitution. For forward substitution, we consider a set of equations in a matrix form Ax=b , where A is a lower triangular matrix with non-zero diagonal elements. The lower triangular matrix can be visualized and solved as follows:

$$\begin{bmatrix} a_{11} & 0 & \dots & 0 \\ a_{21} & a_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mm} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

Figure 2. Lower Triangular Matrix

$$\begin{aligned} x_1 &= b_1/a_{11} \\ x_2 &= (b_2 - a_{21}x_1)/a_{22} \\ x_3 &= (b_3 - a_{31}x_1 - a_{32}x_2)/a_{33} \\ &\vdots \\ x_m &= b_m - a_{m1}x_1 - a_{m2}x_2 - \dots - a_{m,m-1}x_{m-1})/a_{mm} \end{aligned}$$

Figure 3. Solving U

For backward substitution, we consider a set of equations in a matrix form Ax=b , where A is a upper triangular matrix with non-zero diagonal elements. The upper triangular matrix can be visualized and solved as follows:

$$\begin{bmatrix} a_{11} & \dots & a_{1,m-1} & a_{1m} \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \dots & a_{m-1,m-1} & a_{m-1,m} \\ 0 & \dots & 0 & a_{mm} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_{m-1} \\ x_m \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_{m-1} \\ b_m \end{bmatrix}$$

Figure 4. Upper Triangular Matrix

$$\begin{aligned} x_m &= b_m/a_{mm} \\ x_{m-1} &= (b_{m-1} - a_{m-1,m}x_m)/a_{m-1,m-1} \\ x_{m-2} &= (b_{m-2} - a_{m-2,m-1}x_{m-1} - a_{m-2,m}x_m)/a_{m-2,m-2} \\ &\vdots \\ x_1 &= b_1 - a_{12}x_2 - a_{13}x_3 - \dots - a_{1m}x_m)/a_{11} \end{aligned}$$

Figure 5. Solving L

III. IMPLEMENTATION

In order to get a baseline to compare the runtime of LU Decomposition on different computer platforms, we implemented the sequential version so that we could clearly see how much of a difference parallelizing LU decomposition made. Then we implemented the OpenMp, MPI and CUDA and recorded the runtime for all of them in order to compare and contrast to find out which model performed the best out of all 3. Below is a detailed discussion of how each version was implemented.

A. Sequential

```
input:   A : n × n matrix, A = A^(0).
output:  A : n × n matrix, A = L − I_n + U, with
         L : n × n unit lower triangular matrix,
         U : n × n upper triangular matrix,
         I_n : n × n identity matrix,
         such that LU = A^(0).

for $k := 0$ to $n-1$ do
    for $i := k+1$ to $n-1$ do
        $a_{ik} := a_{ik}/a_{kk}$;
    for $i := k+1$ to $n-1$ do
        for $j := k+1$ to $n-1$ do
            $a_{ij} := a_{ij} - a_{ik}a_{kj}$;
```

Figure 6. Sequential LU Algorithm

A.1. Sequential Implementation

GetUserInput(): This gets the user to input matrix size and the option to print it

InitializeMatrix(): Initialize matrices A, L and U.

```
float** InitializeMatrix(int n, float value)
{
    // allocate square 2d matrix
    float **x = new float*[n];
    for(int i = 0 ; i < n ; i++)
        x[i] = new float[n] ;
    // assign random values
    srand (time(NULL));
    for (int i = 0 ; i < n ; i++)
    {
        for (int j = 0 ; j < n ; j++)
        {
            if (value == 1)  // generate input matrices (a and b)
                x[i][j] = (float)((rand()%10)/(float)2);
            else
                x[i][j] = 0;  // initializing resulting matrix
        }
    }
    return x ;
}
```

LU Decomposition(): Computes the lower and upper triangular matrix by finding the pivot rows, swapping the rows, and computing LU decomposition. The row reducing result is the upper triangular matrix. The negated coefficient of multiplier becomes the slots in the lower matrix.

```
// Upper Triangular matrix
for (int k = i; k < size; k++)
{
    // Summation of L(i, j) * U(j, k)
    int sum = 0;
    for (int j = 0; j < i; j++)
        sum += (lower[i][j] * upper[j][k]);

    // Evaluating the upper trangular matrix U(i, k)
    upper[i][k] = matrix[i][k] - sum;
}
```

```
// Lower Triangular matrix
for (int k = i; k < size; k++)
{
    if (i == k)
        lower[i][i] = 1; // Diagonal as 1
    else
    {
        // Summation of L(k, j) * U(j, i)
        int sum = 0;
        for (int j = 0; j < i; j++)
            sum += (lower[k][j] * upper[j][i]);

        // Evaluating the lower trangular matrix L(k, i)
        lower[k][i]
        = (matrix[k][i] - sum) / upper[i][i];
    }
}
```

```
//Initialize the value of matrix a, b, c
float **matrix = InitializeMatrix(n, 1.0);
float **lower = InitializeMatrix(n, 0.0);
float **upper = InitializeMatrix(n, 0.0);

LU_Decomposition(matrix,lower,upper,n);
```

A.2. Sequential Runtime Output

Matrix Size	Runtime
1200	2.21 seconds
2400	17.75 seconds
4800	195.97 seconds
6000	394.80 seconds
7920	871.66 seconds
9600	1462.86 seconds

Figure 7. Sequential Runtime

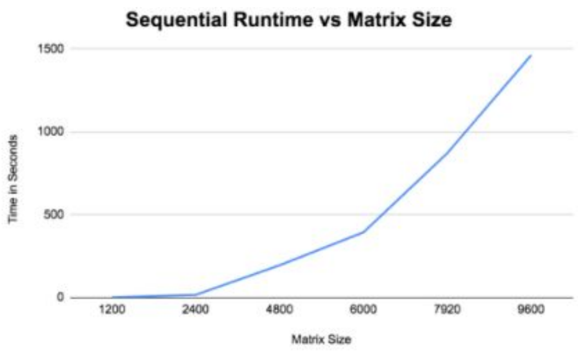


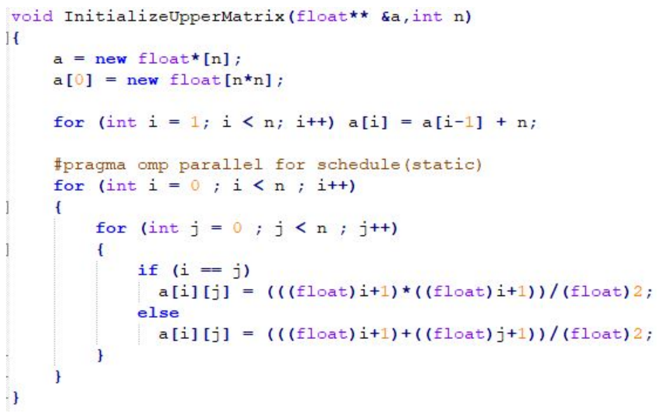
Figure 8. Sequential Runtime Graph

B. OpenMP

We used shared memory which allows immediate access to all data from all processors without explicit communication and dynamic scheduling which solves the issues of load balancing which occurs in static scheduling. In dynamic scheduling. each thread is allocated a chunk of iterations at the beginning of the loop, but the exact set of iterations that are allocated to each thread is not known. The loop iterations are divided into pieces of size chunk and dynamically scheduled among the threads. When the thread finishes one chunk, it is then dynamically assigned another.

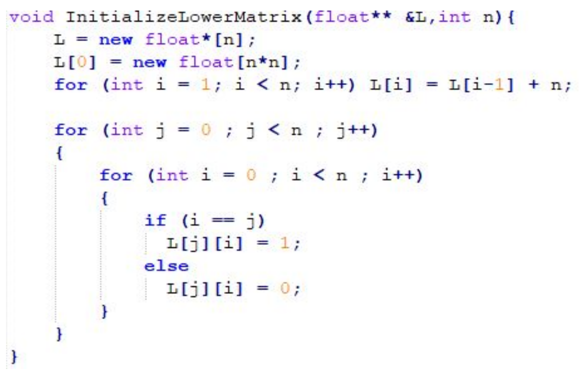
B.1. OpenMP Implementation

InitializeUpperMatrix(): Initialize the value of matrix a[n x n] for upper triangular matrix.



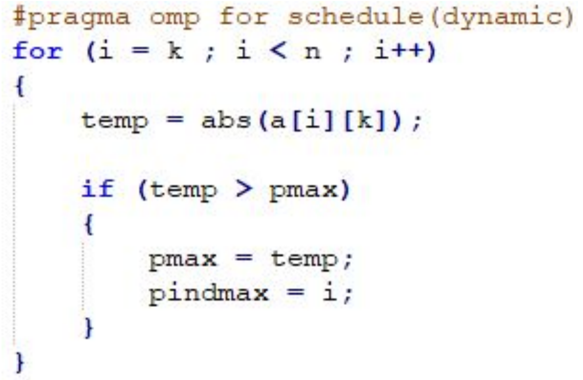
```
void InitializeUpperMatrix(float** &a,int n)
{
    a = new float*[n];
    a[0] = new float[n*n];

    for (int i = 1; i < n; i++) a[i] = a[i-1] + n;

    #pragma omp parallel for schedule(static)
    for (int i = 0 ; i < n ; i++)
    {
        for (int j = 0 ; j < n ; j++)
        {
            if (i == j)
              a[i][j] = (((float)i+1)*((float)i+1))/(float)2;
            else
              a[i][j] = (((float)i+1)+((float)j+1))/(float)2;
        }
    }
}
```

InitializeLowerMatrix(): Initialize the value of matrix L[n x n] for lower triangular matrix.

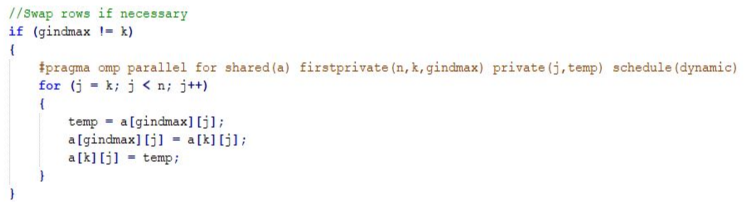


```
void InitializeLowerMatrix(float** &L,int n){
    L = new float*[n];
    L[0] = new float[n*n];
    for (int i = 1; i < n; i++) L[i] = L[i-1] + n;

    for (int j = 0 ; j < n ; j++)
    {
        for (int i = 0 ; i < n ; i++)
        {
            if (i == j)
              L[j][i] = 1;
            else
              L[j][i] = 0;
        }
    }
}
```

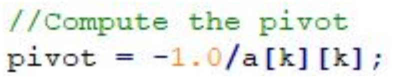
ComputeLUDecomposition(): computes the LU decomposition for the upper and lower triangular matrix. It first defines the necessary variables, finds the column max, swaps the rows in the k loop, places the pivots in a and L and finally finds L and U.



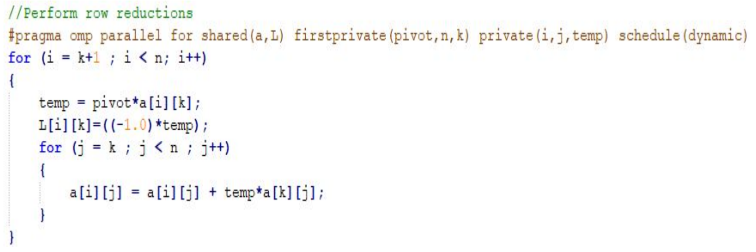
```
#pragma omp for schedule(dynamic)
for (i = k ; i < n ; i++)
{
    temp = abs(a[i][k]);

    if (temp > pmax)
    {
        pmax = temp;
        pindmax = i;
    }
}
```



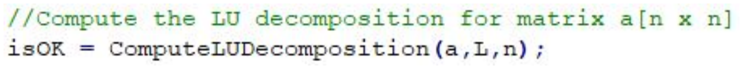
```
//Swap rows if necessary
if (gindmax != k)
{
    #pragma omp parallel for shared(a) firstprivate(n,k,gindmax) private(j,temp) schedule(dynamic)
    for (j = k; j < n; j++)
    {
        temp = a[gindmax][j];
        a[gindmax][j] = a[k][j];
        a[k][j] = temp;
    }
}
```



```
//Compute the pivot
pivot = -1.0/a[k][k];
```

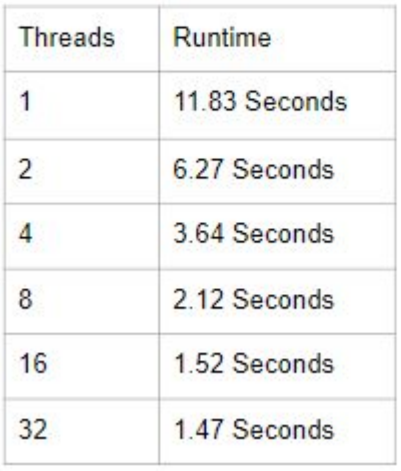


```
//Perform row reductions
#pragma omp parallel for shared(a,L) firstprivate(pivot,n,k) private(i,j,temp) schedule(dynamic)
for (i = k+1 ; i < n; i++)
{
    temp = pivot*a[i][k];
    L[i][k]=((-1.0)*temp);
    for (j = k ; j < n ; j++)
    {
        a[i][j] = a[i][j] + temp*a[k][j];
    }
}
```



```
//Compute the LU decomposition for matrix a[n x n]
isOK = ComputeLUDecomposition(a,L,n);
```

B.2. OpenMP Runtime Output



Threads	Runtime
1	11.83 Seconds
2	6.27 Seconds
4	3.64 Seconds
8	2.12 Seconds
16	1.52 Seconds
32	1.47 Seconds

Figure 10. OpenMP Runtime Version 1

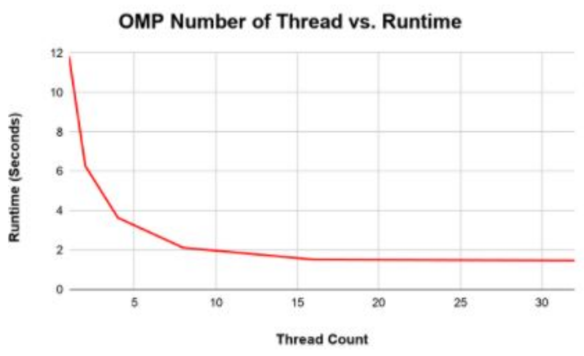
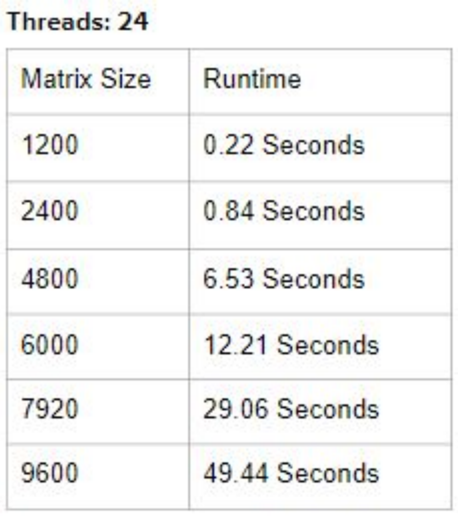


Figure 11. OpenMP Runtime Graph 1



Threads: 24

Matrix Size	Runtime
1200	0.22 Seconds
2400	0.84 Seconds
4800	6.53 Seconds
6000	12.21 Seconds
7920	29.06 Seconds
9600	49.44 Seconds

Figure 12. OpenMP Runtime Version 2

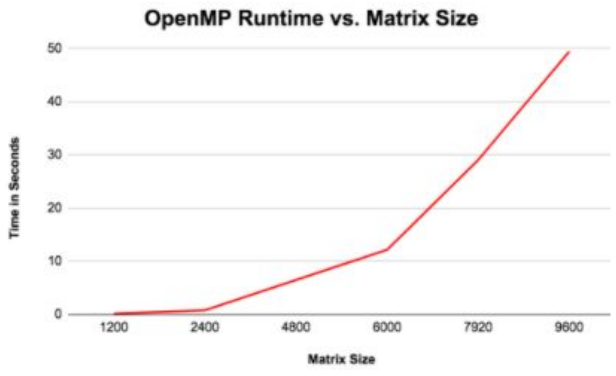


Figure 13. OpenMP Runtime Graph 2

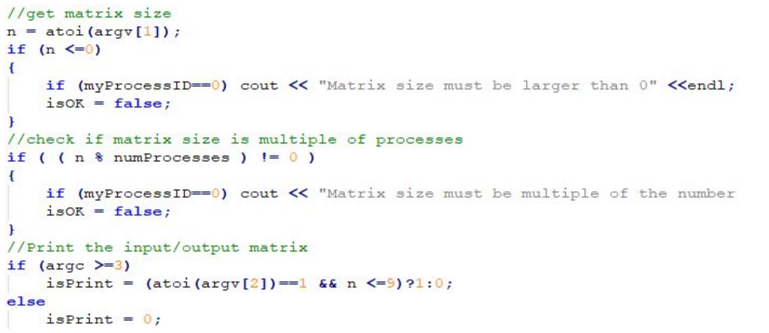
The implementation was evaluated for an input matrix of size 3000 with a combination of 1,2,4,8,16,32 threads executing in parallel for the first graph and the implementation was evaluated for an input matrix size of 1200, 2400, 4800, 6000, 7920, and 9600 with a total of 24 threads executing in parallel for the second graph. It produces the best single node performance and best single thread performance. The LU decomposition was faster when more threads were executing in parallel so the more threads we have the better the runtime because different processes are executing different instructions on different pieces of data. The runtime is comparatively better for this version compared to the sequential version because of shared memory which allows immediate access to all data from all processors.

C. MPI

We implemented MPI using the send/receive data communication and distributed memory. We used MPI_Bcast to send coefficient multipliers to processes (for indices). We also use MPI_Send and MPI_Recv to send data portions of the matrix to the processes (for array). After finding the pivot and swapping rows in the k loop, L and U are found with MPI_Bcast and performing row reduction. MPI_Bcast broadcasts a message from the process in the root to all other processes of the communication. MPI_Send and MPI_Recv are used to send the work processes and receive them.

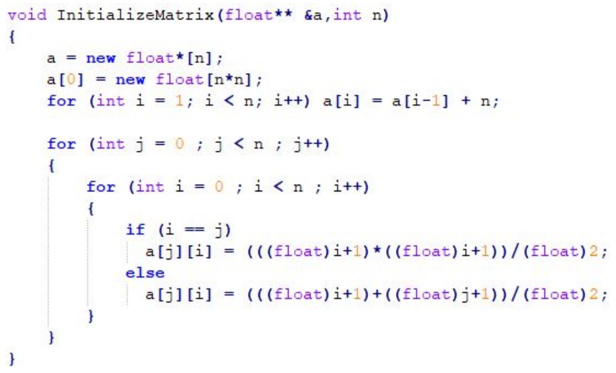
C.1. MPI Implementation

GetUserInput(): gets the user input for the matrix dimension, checks if the matrix size is multiple of processes, and gets the printing option of 0 or 1.



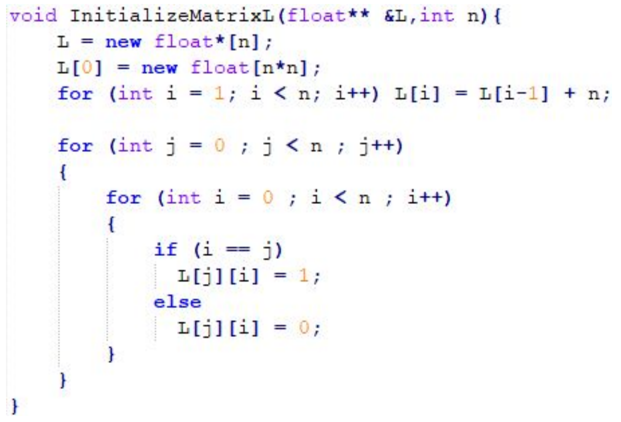
```
//get matrix size
n = atoi(argv[1]);
if (n <=0)
{
    if (myProcessID==0) cout << "Matrix size must be larger than 0" <<endl;
    isOK = false;
}
//check if matrix size is multiple of processes
if ( ( n % numProcesses ) != 0 )
{
    if (myProcessID==0) cout << "Matrix size must be multiple of the number
    isOK = false;
}
//Print the input/output matrix
if (argc >=3)
    isPrint = (atoi(argv[2])==1 && n <=9)?1:0;
else
    isPrint = 0;
```

InitializeMatrix(): initializes the value of matrix a[n x n].



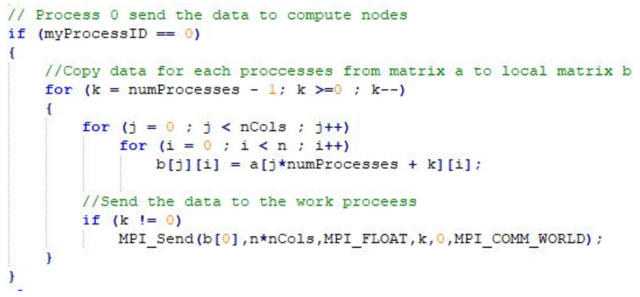
```
void InitializeMatrix(float** &a,int n)
{
    a = new float*[n];
    a[0] = new float[n*n];
    for (int i = 1; i < n; i++) a[i] = a[i-1] + n;

    for (int j = 0 ; j < n ; j++)
    {
        for (int i = 0 ; i < n ; i++)
        {
            if (i == j)
              a[j][i] = (((float)i+1)*((float)i+1))/(float)2;
            else
              a[j][i] = (((float)i+1)+((float)j+1))/(float)2;
        }
    }
}
```

InitializeMatrixL(): initializes the value of matrix L[n x n]

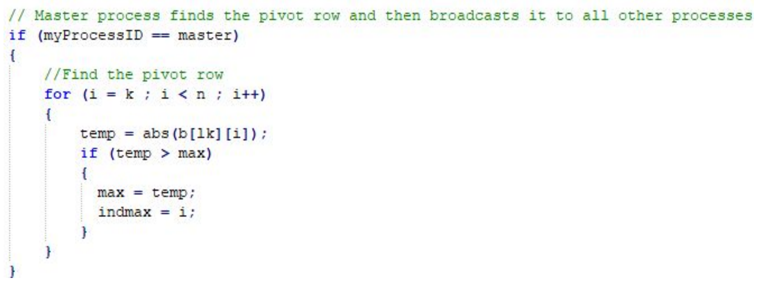


```
void InitializeMatrixL(float** &L,int n){
    L = new float*[n];
    L[0] = new float[n*n];
    for (int i = 1; i < n; i++) L[i] = L[i-1] + n;

    for (int j = 0 ; j < n ; j++)
    {
        for (int i = 0 ; i < n ; i++)
        {
            if (i == j)
              L[j][i] = 1;
            else
              L[j][i] = 0;
        }
    }
}
```

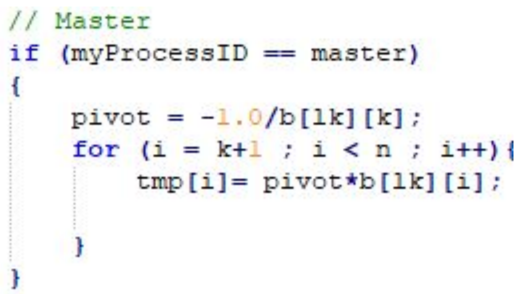
ComputeLUDecomposition(): process 0 sends the data to compute nodes by copying it from the matrix to **a** local matrix **b.** Then it performs the row wise elimination where the master process finds the pivot row and then broadcasts it to all other processes, finds the pivot rows, swaps rows if necessary and finally performs an row reduction. Process 0 collects the results from the worker processes after it receives the data from the worker processes.



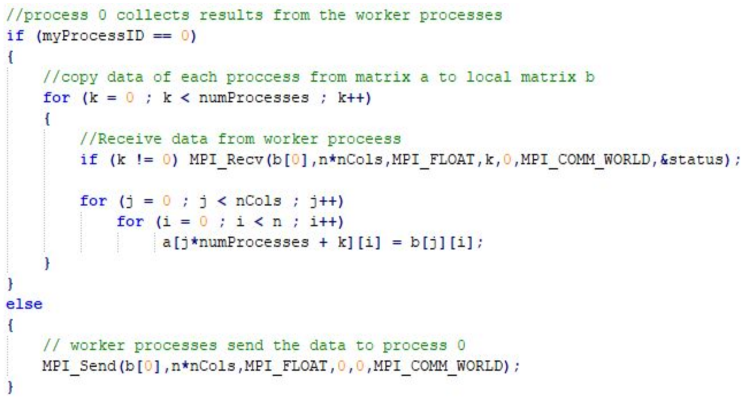
```
// Process 0 send the data to compute nodes
if (myProcessID == 0)
{
    //Copy data for each proccesses from matrix a to local matrix b
    for (k = numProcesses - 1; k >=0 ; k--)
    {
        for (j = 0 ; j < nCols ; j++)
            for (i = 0 ; i < n ; i++)
                b[j][i] = a[j*numProcesses + k][i];

        //Send the data to the work proceess
        if (k != 0)
            MPI_Send(b[0],n*nCols,MPI_FLOAT,k,0,MPI_COMM_WORLD);
    }
}
```



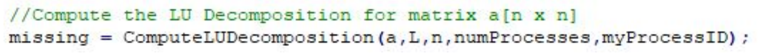
```
// Master process finds the pivot row and then broadcasts it to all other processes
if (myProcessID == master)
{
    //Find the pivot row
    for (i = k ; i < n ; i++)
    {
        temp = abs(b[lk][i]);
        if (temp > max)
        {
          max = temp;
          indmax = i;
        }
    }
}
```



```
// Master
if (myProcessID == master)
{
    pivot = -1.0/b[lk][k];
    for (i = k+1 ; i < n ; i++){
        tmp[i]= pivot*b[lk][i];

    }
}
```

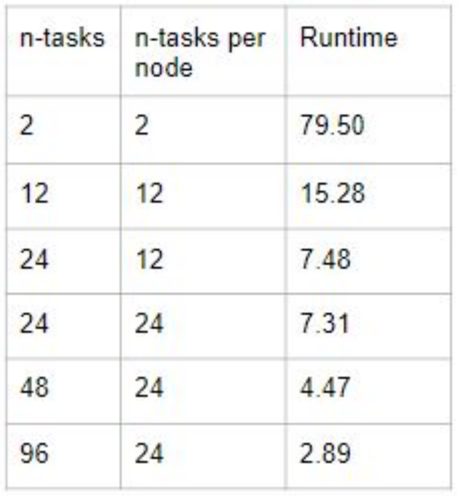


```
//process 0 collects results from the worker processes
if (myProcessID == 0)
{
    //copy data of each proccess from matrix a to local matrix b
    for (k = 0 ; k < numProcesses ; k++)
    {
        //Receive data from worker proceess
        if (k != 0) MPI_Recv(b[0],n*nCols,MPI_FLOAT,k,0,MPI_COMM_WORLD,&status);

        for (j = 0 ; j < nCols ; j++)
            for (i = 0 ; i < n ; i++)
                a[j*numProcesses + k][i] = b[j][i];
    }
}
else
{
    // worker processes send the data to process 0
    MPI_Send(b[0],n*nCols,MPI_FLOAT,0,0,MPI_COMM_WORLD);
}
```



```
//Compute the LU Decomposition for matrix a[n x n]
missing = ComputeLUDecomposition(a,L,n,numProcesses,myProcessID);
```

C.2. MPI Runtime Output



n-tasks	n-tasks per node	Runtime
2	2	79.50
12	12	15.28
24	12	7.48
24	24	7.31
48	24	4.47
96	24	2.89

Figure 14. MPI Runtime 1

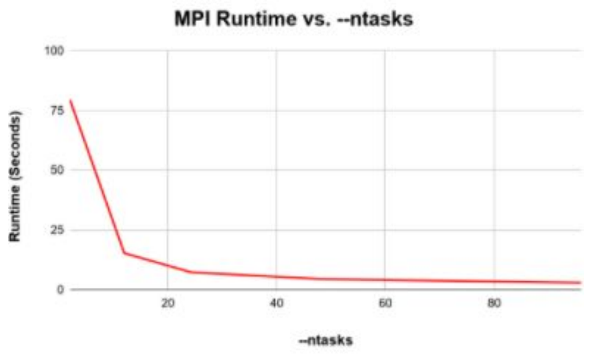
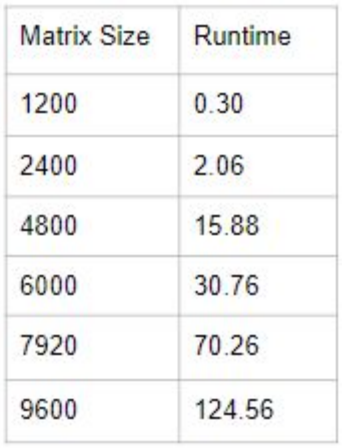


Figure 15. MPI Runtime Version 1



Matrix Size	Runtime
1200	0.30
2400	2.06
4800	15.88
6000	30.76
7920	70.26
9600	124.56

Figure 16. MPI Runtime 2

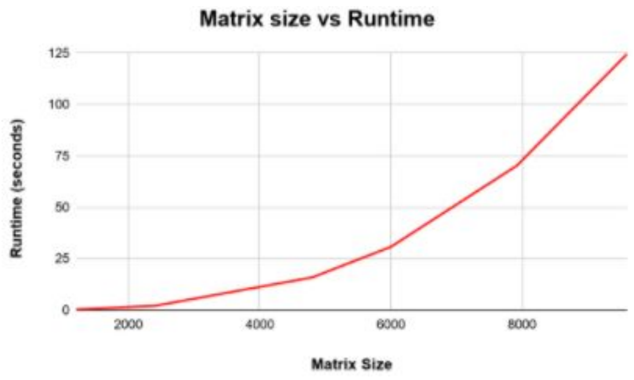


Figure 17. MPI Runtime Graph 2

The implementation was evaluated for an input matrix of size 3840 with a combination of 2,12,24,48,96 ntasks and 2,12,24 ntasks per node executing in parallel for the first graph obtaining MPI runtime. The implementation was evaluated for an input matrix size of 1200, 2400, 4800, 6000, 7920, and 9600 with a total of 24 ntasks and 24 ntasks per node executing in parallel for the second graph to compare matrix size vs runtime. The LU decomposition was faster when more threads were executing in parallel so the more ntasks per node we have the better the runtime because different processes are executing different instructions on different pieces of data. The runtime is comparatively better for this version compared to the sequential version and OpenMP because of distributed memory which allows immediate access to all data from all processors.

D. CUDA

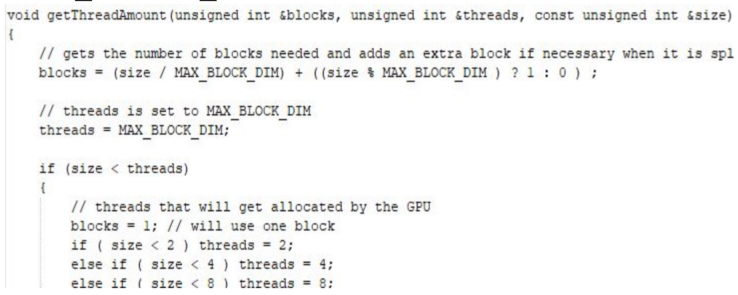
We implemented the CUDA using shared Memory wherein blocks with static size are used. The thread copies the pivot row into shared memory and after that the rest of the threads in the block start reducing. We also used block scheduling and we choose a sub matrix block size and rewrite the matrix A of size M*N as a block matrix.

D.1. CUDA Implementation

handleError(): handles and reports CUDA errors

checkCUDAError(): Checks cuda error and prints the result if necessary.

getThreadAmount(): Gets the number of blocks needed and adds an extra block if necessary when it is split by MAX_BLOCK_DIM.



```
void getThreadAmount(unsigned int &blocks, unsigned int &threads, const unsigned int &size)
{
    // gets the number of blocks needed and adds an extra block if necessary when it is spl
    blocks = (size / MAX_BLOCK_DIM) + ((size % MAX_BLOCK_DIM ) ? 1 : 0 ) ;

    // threads is set to MAX_BLOCK_DIM
    threads = MAX_BLOCK_DIM;

    if (size < threads)
    {
        // threads that will get allocated by the GPU
        blocks = 1; // will use one block
        if ( size < 2 ) threads = 2;
        else if ( size < 4 ) threads = 4;
        else if ( size < 8 ) threads = 8;
```

FindThreadAmount(): finds the thread amount

```
void FindThreadAmount( unsigned int &blocks, unsigned int &threads, const unsigned int &size )
    blocks = (size / MAX_2D_BLOCK_DIM) + ((size % MAX_2D_BLOCK_DIM ) ? 1 : 0 ) ;
    threads = MAX_2D_BLOCK_DIM;
    if (size < threads)
    {
        threads = size;
    }
}
```

ComputeGaussianElimination(): computes the Gaussian elimination by swapping the rows of the sub matrix A[row][row] because everything above and to the left of row will be data that we have already dealt with and loads the top row into shared memory.

```
if ((column > switch_row && column < N) && (row > switch_row && row < N) ) {
    coefficient = A[row*N + switch_row];
    coefficient *= device_invert_sign(A[row * N + column]); // inverts the c
    lower = A[row * N + column];
    upper = A[switch_row * N + column];
    A[row * N + column] = lower + coefficient * upper;
}
```

FindCoefficients(): finds the coefficient multiplier

```
unsigned int tx_id = threadIdx.x;
if ( tx_id == 0 ) {
    switch_row = *switch_row_ptr;
    N = *size;
    denominator = A[switch_row * N + switch_row];
}
unsigned int row = blockDim.x * blockIdx.x + tx_id;
if (row > switch_row && row < N)
{
    data[tx_id] = A[row * N + switch_row];
    coefficient = data[tx_id] / denominator;
    A[row * N + switch_row] = coefficient;
}
```

FindMaxColumn(): sets absolute position within the matrix, stores the matrix index, and gets the absolute value of the left and right values.

```
unsigned int tx_id = threadIdx.x;
if ( tx_id == 0 ) {
    partition = blockDim.x;
    N = *size;
    column = *in_column;
}
int row = blockDim.x * blockIdx.x + tx_id;
if (row >= column && row < N)
{
    data[tx_id] = A[row * N + column];
    data[tx_id + partition] = row;
}
```

```
for (int s = blockDim.x / 2; s > 0; s >>= 1) {
    if ( tx_id < s ) {
        double left_value = device_abs(data[tx_id]);
        double right_value = device_abs(data[tx_id + s]);
        if (left_value < right_value)
        {
            data[tx_id] = data[tx_id + s];
            data[tx_id + partition] = data[tx_id + s + partition];
        }
    }
    __syncthreads();
}
```

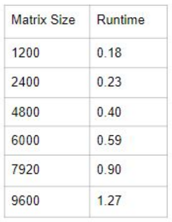
main(): determines the min, maz, mean, mode and standard deviation of the array, gets the size of the array to process, gets the seed to be used, makes sure to record the start time, allocates the array to be populates, initializes the 2D array, writes the 2D matrix array to the GPU, records the end time, performs the LU decomposition, calculates the runtime and prints everything
To compute LU decomposition, we set the variables for the number of blocks and threads assigned to the kernel, and copy the column we are looking at to the GPU. Then we find the max for the column in order to pivot, create an array in order for the kernel to return the index and max value found in the block. We call the kernel to find the column max. Finally we call the previous functions to swap the rows and perform the LU calculations.

```
// copy the current column from host to gpu
HANDLE_ERROR(cudaMemcpy(current_row_ptr, &i, sizeof(unsigned int), cudaMemcpyHostToDevice));

unsigned int find_index = max_threads[0].index;
double find_value = max_threads[0].value;
for (int j = 0; j < numblocks; ++j)
{
    if (fabs(max_threads[j].value) > fabs(find_value) && max_threads[j].index != -1.0)
    {
        find_index = max_threads[j].index;
        find_value = max_threads[j].value;
    }
}
```

D.2. CUDA Runtime Output

Matrix Size	Runtime
1200	0.18
2400	0.23
4800	0.40
6000	0.59
7920	0.90
9600	1.27

Figure 18. CUDA Runtime



Matrix Size	Runtime
1200	0.18
2400	0.23
4800	0.40
6000	0.59
7920	0.90
9600	1.27

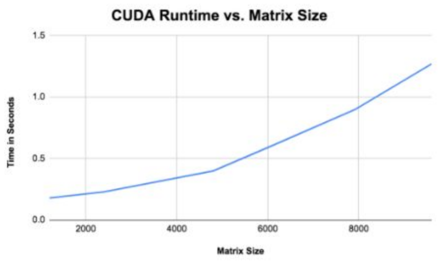


Figure 19. CUDA Runtime Graph

The implementation was evaluated for an input matrix size of 1200, 2400, 4800, 6000, 7920, and 9600. The runtime is comparatively better for this version compared to the all the other three versions we implemented and the use of shared memory along with the implementation of GPGPU computing and parallelization definitely played a key role in the increased runtime.

IV. RUNTIME COMPARISON

Sequential: 1 thread, OpenMp: 24 threads, MPI: 24 threads, CUDA: 1024 threads

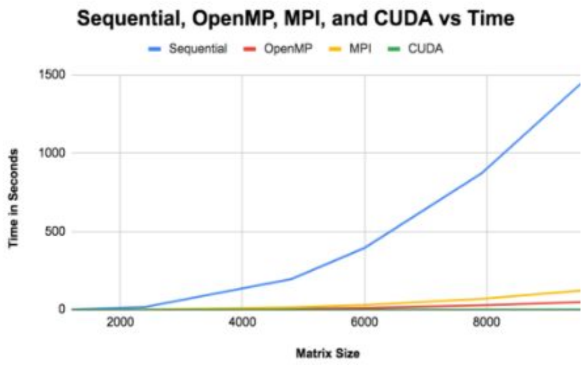


Figure 20. Runtime Comparison Graph

V. RUNTIME ANALYSIS

We have implemented a sequential version of LU decomposition to use as a baseline for our project. This sequential version allowed us to see how much faster our parallel versions turned out to be. We calculated the speedup of both the sequential runtime and the parallel runtime. If you look at the graph above, you can clearly see how different parallel algorithms performed and why they performed as they did. As expected you can see that the sequential runtime gets longer and longer as the matrix size increases. For MPI you can see that when you run 24 threads, it greatly outperforms the sequential version. MPI utilizes send and receive communication, distributed memory and multiple threads to process different parts of the matrix in parallel thus reducing runtime. For OpenMP, you can see that when you run 24 threads, it greatly outperforms the sequential version and slightly outperforms the MPI version. This is because OpenMP utilizes shared memory, dynamic scheduling and multiple threads to process different parts of the matrix in parallel thus reducing runtime. The threads are able to access the shared memory faster than distributed memory which is why OpenMP performs better than MPI. Finally, CUDA performed the best among all the different implementations that we explored for this project because it produced the fastest run time out of all our models and the reason for this is probably the fact that it has a very fast shared memory region that can be shared amongst threads. Not only that, but CUDA is optimized for parallel data and throughput computation so it definitely makes sense that CUDA outperformed the other 2 models.

VI. CONCLUSION

In conclusion, parallel processing is a must if you want to process huge amounts of data in a short amount of time. OpenMP, MPI and CUDA will not fit every case perfectly, but understanding the difference will definitely help you as a programmer which algorithm will perform better in your situation. There are different ways to implement parallel processing and even though they all utilize parallel processing, the way they are implemented will affect their runtimes. We showed how OpenMP is faster than MPI even though they both use 24 threads to process the same size matrix and that is because OpenMP uses shared memory which is faster than distributed memory that MPI uses. But, above all CUDA seems like an ideal choice if we the aim is to get the shortest runtime since CUDA is designed to handle very big matrices with minimal runtime as we saw in our runtime comparison graphs. We learned that there is improvement for all of our algorithms, as an example, we could implement the Doolittle algorithm for better efficiency results.

VII. REFERENCES

https://www.hindawi.com/journals/mpe/2019/3720450/
https://webspace.science.uu.nl/~bisse101/Book/PSC/psc2_1.pdf
https://www.gaussianwaves.com/2013/05/solving-a-triangular-matrix-using-forward-backward-substitution/