Project Report

Parallel LU Factorization

Project Team:

1. Yoshita Kondapalli - CO21BTECH11008
2. P Gayathri Shreeya - CO21BTECH11010
3. Ishaan Jain - CO21BTECH11006

LU factorization is a numerical method to solve linear equations. Suppose we have an equation, $Ax = b$. We can split A as a product of a lower diagonal matrix L, and an upper diagonal matrix U. This makes it easier to calculate the vector x as we can split the expression as $L(Ux) = b$. Or simply $Ly = b$, where $y = Ux$. Solving the previous two equations is easier than $Ax = b$.

This project aims to parallelize the LU factorization in three different ways and compare them.

Ring of Processors

The processors are arranged in a ring-like structure, where each processor is connected to the next processor unidirectionally. The data transfer can be done only in one direction. To send a message to the next processor, the processor executes *send()*, while to receive the message, the processor executes *receive()*. The message here is referred to an array of elements called the *buffer*. If a thread receives (), its preceding thread must *send()* to terminate the program. The *receive()* is blocking to keep the synchronization.

(a) The First Version:

Broadcasting is the process where some information(a buffer array here) is being transferred to every other processor. It is a sequential process since the

sending and receiving executed by each processor are not independent. The further algorithm is defined in the program design itself.

Program Design:

The processors are implemented in a circular singly linked list structure where the nodes are of type *ProcessorNode*.

1. ***ProcessorNode* Class*:***

   The *ProcessorNode* is a class that consists of the following variables, methods, and constructors:

- *proc_id* - It is an integer type variable that acts as an identifier for the processor; it goes from 0 to *nprocs*-1.
- *next* - This variable is a pointer to the next *ProcessorNode* in the linked list.
- *buffer* - This is a vector of type double. This is the vector that is being communicated with other processors.
- *localCols* - This is a 2D vector that stores the columns assigned to that particular processor.
- *received* - This is a bool variable that tracks whether a processor has received a buffer from its previous processor.
- *ProcessorNode(Constructor)* - This constructor initializes the node with *id,* the next pointer is NULL, and all the vectors have their sizes.
- *Send* - This method sends a buffer from this processor to the next processor in the list.
- *Receive* - This method receives the buffer from the previous processor with synchronization.
- *Prep* - This method takes the column number and divides all the column elements below by the diagonal element and multiplies by -1. Then, this column is stored in the buffer.
- *Update* - This method performs the following operation:

  $$A_{ij} = A_{ij} + A_{ik}A_{kj}$$

  Which in our algorithm is given as

  *localCols[i][j] = localCols[i][j] + buffer[i] * localCols[k][j]*
- *BROADCAST* - This method is for the inter-thread communication. It has two methods defined within its definition: *Send* and *Receive*.

- *FACTORISATION* - This method performs the operations on the columns of the matrix depending on which thread is allocated to it.

2. ***CircularLinkedList* Class:**
   The CircularLinkedList class is for constructing the linked list. The class consists of the following variables and methods:-

- *numprocs* - This variable stores the number of threads.
- *head* - This is a pointer to a processor node. This will be used in the constructor to construct the list.
- *procs* - This is a vector of pointers to the processor nodes.
- Constructor - The constructor takes arguments as the number of processors and matrix size. This first creates a head node and later makes a circular linked list with the last node pointing to the head itself.
- *DivideAmongThreads* - The columns are divided among the threads as per the rule *k%p,* where *k* is the column number, and *p* is the number of threads.

3. ***main* function:**
   In the main function, a function reads the inputs:

- *n_procs*
- *mat_size*
- A *mat_size*mat_size* matrix

A circular linked list is created, as explained above. Now, these threads are executed in the *runner* function with the thread ID as the only argument. The runner function makes a pointer to the processor node with that ID. A barrier is kept so all the threads go to the factorization function together.

```
1  var A:  array[0..n − 1,0..r − 1] of real
2  var buffer:  array[0..n − 1] of real
3  q ←My_Num()
4  p ←Num_Procs()
5  l ← 0
6  for k = 0 to n − 2 do
7      if Alloc(k) = q then
8          Prep(k) :
9              for i = k + 1 to n − 1 do
10                 buffer[i − k − 1] ← A[i, l] ← −A[i, l]/A[k, l]
11         l ← l + 1
12     Broadcast(Alloc(k), buffer, n − k − 1)
13     for j = l to r − 1 do
14         Update(k, j) :
15             for i = k + 1 to n − 1 do
16                 A[i, j] ← A[i, j] + buffer[i − k − 1] × A[k, j]
```

Fig 1. Pseudocode for the First Version of the Ring of Processors Algorithm

(b) Pipelining on the Ring

The previous algorithm has its drawbacks, such as using broadcasting so extensively. For a ring structure, we can send the buffer to the next processor as soon as it gets updated rather than broadcasting. This is done before it executes the *update()*. When column *k* is traveling on the ring, the previous processors have already started their execution, but those after that processors are still waiting to receive column *k*. As a result, processor *i* is computationally ahead of processor $i + 1$, making the pipeline phases. There are pipeline bubbles in each *n_procs* step. The fewer bubbles there are, the more concurrency there is.

Program Design:

The program design is similar to the first version. The major difference lies in the *FACTORISATION* function in the *ProcessorNode* class. Instead of calling the *BROADCAST* function, we call the *send()* function just after executing the *Prep() if that column is allocated to that processor,* or if the column is not allocated to that processor, execute *receive() and send()* unlike the first version where the buffer is sent after updating the columns.

This is just to make more overlaps with other processors' computation and minimization of the pipeline bubble. Sending before updating enables them not to wait for every thread to complete the basic operations. Whoever completes can go ahead with the further computations.

```
1  var A:  array[0..n − 1,0..r − 1] of real
2  var buffer:  array[0..n − 1] of real
3  q ←My_Num()
4  p ←Num_Procs()
5  l ← 0
6  for k = 0 to n − 2 do
7      if k = q mod p then
8          Prep(k) :
9              for i = k + 1 to n − 1 do
10                 buffer[i − k − 1] ← A[i, l] ← −A[i, l]/A[k, l]
11         l ← l + 1
12         Send(buffer, n − k − 1)
13     else
14         Recv(buffer, n − k − 1)
15         if q ≠ k − 1 mod p then
16             Send(buffer, n − k − 1)
17     for j = l to r − 1 do
18         Update(k, j) :
19             for i = k + 1 to n − 1 do
20                 A[i, j] ← A[i, j] + buffer[i − k − 1] × A[k, j]
```

Fig 2. Pseudocode for the Pipelining on the Ring Algorithm

(c) Look Ahead Algorithm:

The previous pipeline algorithm can be modified further to make the pipeline bubbles thinner. This time again, the only difference will be in the *FACTORISATION* function. All other things can be kept the same. This can be done by making separate cases for both cases when a column is allocated to a thread and not to a thread.

- For the first case, execute *prep()* and immediately execute *send(),* and then update all the columns.
- For the other case, receive the columns, and if the processor isn't the processor preceding this node, execute send(). After this, update all the columns.

Below is a pseudocode for the same.

```
1  var A:  array[0..n − 1,0..r − 1] of real
2  var received_buffer:  array[0..n − 1] of real
3  var sent_buffer:  array[0..n − 1] of real
4  q ←My_Num()
5  p ←Num_Procs()
6  for k = 0 to n − 2 do
7      if k = q mod p then
8          Prep(k, sent_buffer)
9          Send(sent_buffer, n − k − 1)
10         if k > 1 then
11             forall j such that j mod p = q, j > k do
12                 Update(k − 1, j, received_buffer)
13         forall j such that j mod p = q, j > k do
14             Update(k, j, sent_buffer)
15     else
16         Recv(received_buffer, n − k − 1)
17         if q ≠ k − 1 mod p then
18             Send(received_buffer, n − k − 1)
19         if q = k + 1 mod p then
20             Update(k, k + 1, received_buffer)
21         else
22             forall j such that j mod p = q, j > k do
23                 Update(k, j, received_buffer)
```

Fig 3. Pseudocode for the Look Ahead Algorithm

The main idea here is to perform the communication between the threads as early as possible. All this is to accomplish the reduction in the pipeline bubbles so that there is no or lesser time that is being wasted while waiting to receive the buffers sent by the previous processors.

Due to some reasons, this part is ending in a deadlock.

Results:

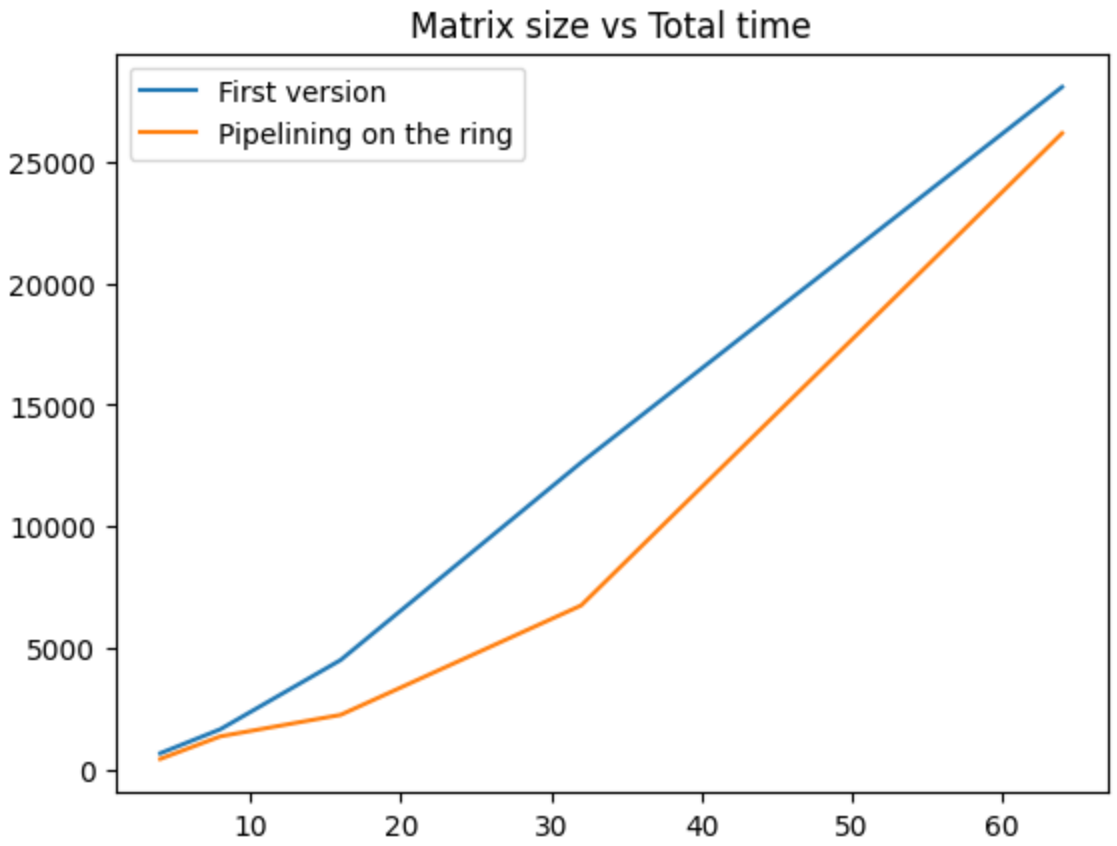
For comparing the performance of the three algorithms, we are using three different metrics - Total time taken by the algorithm, throughput, and average time spent waiting in the receive method. However, the third algorithm *Look Ahead* resulted in frequent deadlocks during testing

Plots:

Plot 1)

Size of matrix vs Total Time(ms)

Size of matrix	First Version	Pipelining the ring	Look Ahead
4	692.6	459.16666667	
8	1678.8	1387.2	
16	4521.4	2268.8	
32	12626.666667	6760.8333333	
64	28048.833333	26149.8333333	

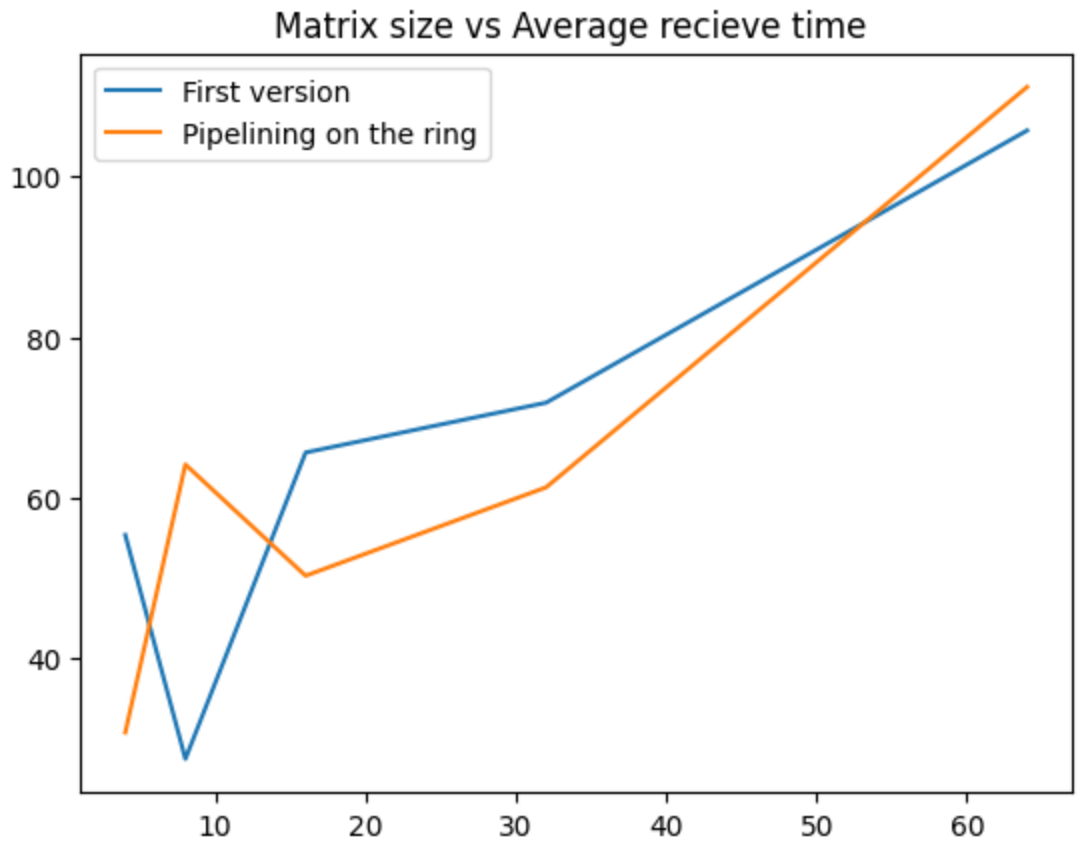


We can see that the total time for the first version is more than the pipeline on the ring version. The difference is more evident as the size of matrix increases. Sending the received buffer before updating could be one of the reasons for better performance of Pipelining algorithm.

Plot 2)

Size of matrix vs Average wait time in receive

Size of matrix	First Version	Pipelining the ring	Look Ahead
4	55.33334	30.72223833	
8	27.42858	64.1143	
16	65.6089667	50.25334	
32	71.81725	61.26883333	
64	105.7301667	111.15605	



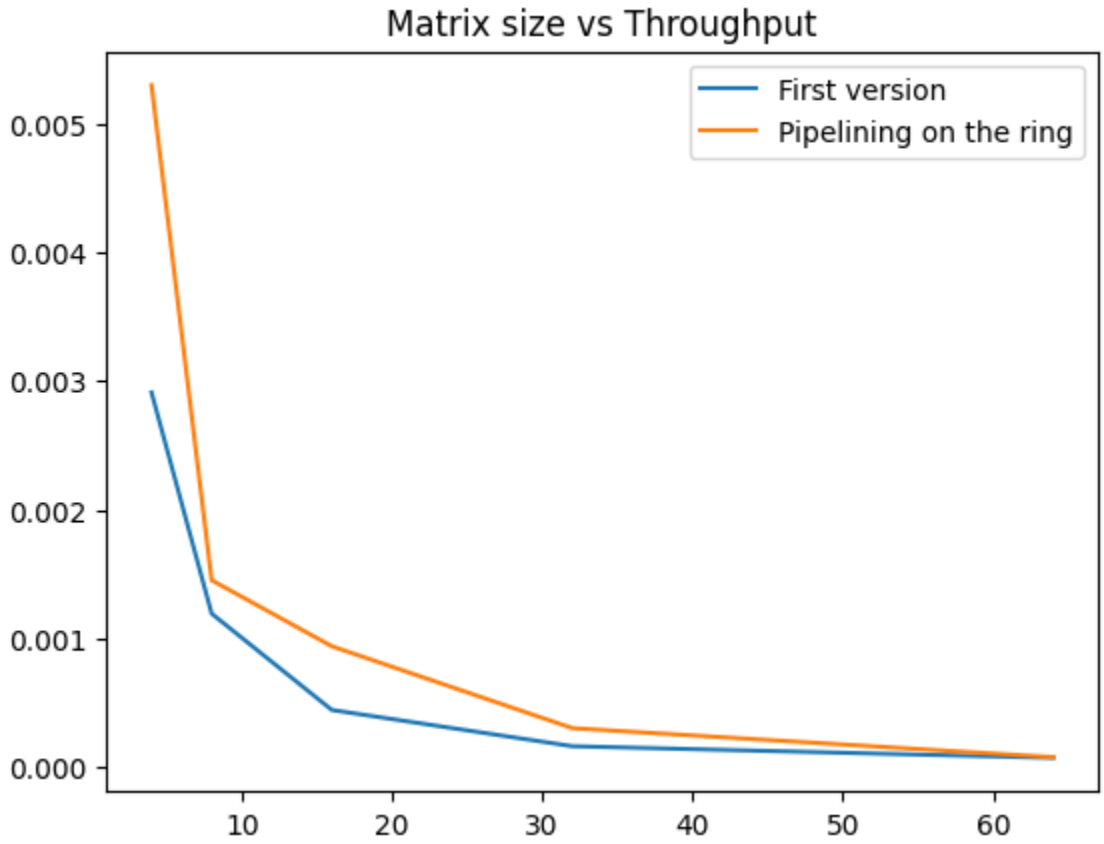
The average wait time in the First version is slightly more than the average wait time in the pipelining on the ring because the threads don't have to wait for the previous threads to update their values for them to send the received buffer. The waiting time also increases as the matrix size increases since the amount of data that needs to be communicated increases, leading to a larger communication overhead.

Plot 3)

Size of matrix vs Throughput

Size of matrix	First Version	Pipelining the ring	Look Ahead
4	2.91199e-03	5.30071e-03	
8	1.19263e-03	1.45193e-03	
16	4.43547e-04	9.38557e-04	

32	1.619225e-04	3.022587e-04	
64	7.178993e-05	7.69816e-05	



As the matrix size increases, throughput decreases for both versions as larger matrices require more computation. The pipelining on the ring algorithm is better suited for matrices with large sizes because of better parallelization.

References:

Parallel Algorithms

Matrix generator used: Generate Random Matrices – Online Math Tools