Solving Linear Equations using Gaussian Elimination

Introduction:

Gaussian elimination, also known as row reduction, is an algorithm in linear algebra for solving a system of linear equations. It is usually understood as a sequence of operations performed on the corresponding matrix of coefficients. This method can also be used to find the rank of a matrix, to calculate the determinant of a matrix, and to calculate the inverse of an invertible square matrix.

Implementation:

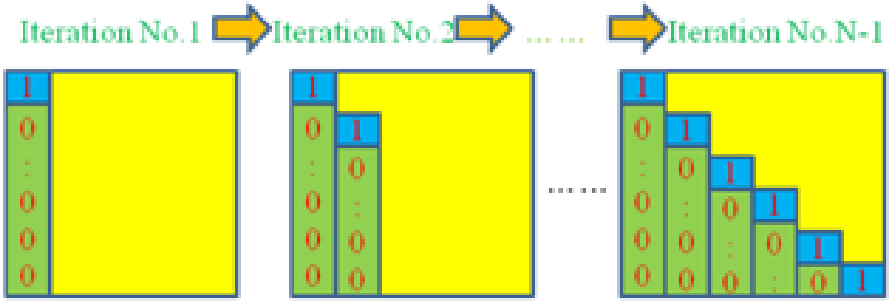
The algorithms implemented here aims to transform a system of linear equations into augmented matrix and then convert that into an upper-triangular matrix and the apply back-substitution method to get the solution using Gaussian elimination and Partial Pivot Algorithms. We observed these algorithms using below implementations.

- Serial
  - Gaussian Elimination using Sequential Algorithm
  - Gaussian Elimination using Partial Pivot
- Parallel
  - Gaussian Elimination using OpenMP
  - Gaussian Elimination using MPI
  - Gaussian Elimination using MPI Partial Pivot (Pipelined)

Serial Implementation:

Gaussian Elimination using Sequential Algorithm:

Gaussian elimination proceeds by performing elementary row operations to produce zeros below the diagonal of the coefficient matrix to reduce it to upper triangular form. Once this is done, inspection of the bottom row(s) and back-substitution into the upper rows determine the values of the unknowns.

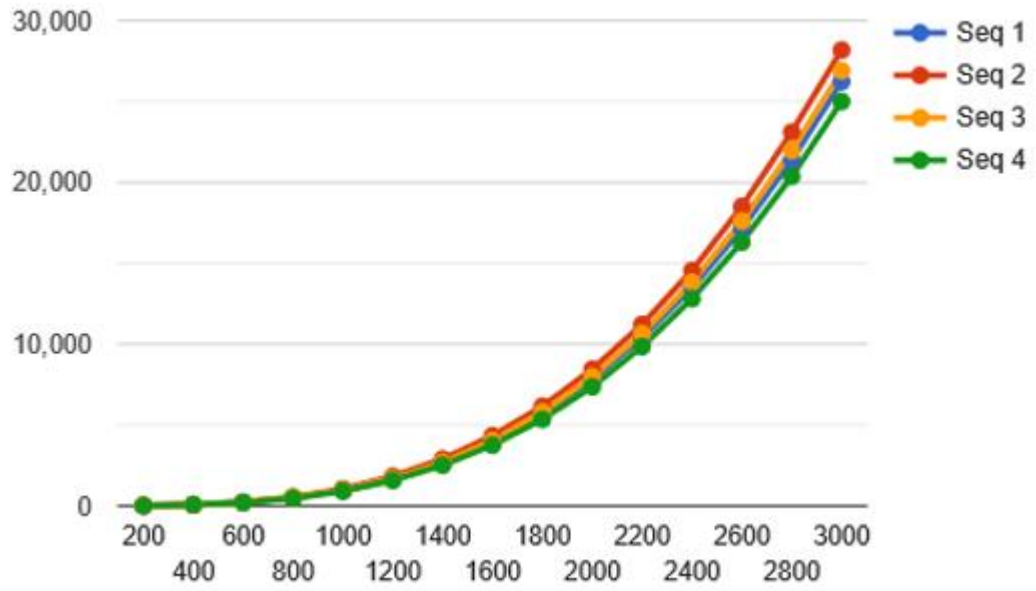


Pseudocode for Gaussian Elimination:

```
/*      Forward Elimination      */
For k= 1, n-1
   For i = k+1, n
      factor =  a(i,k) / a(k,k)
      for j = k+1, n
         a(i,j) = a(i,j) – factor * a(k,j)
      b(i) = b(i) – factor * b(k)

/*      Backward Substitution    */
x(n) = b(n) / a(n,n)
   For  i =n-1, 1, -1
      sum = b(i)
      For j = i+1, n
         sum=sum – a (i,j) * x(j)
x(i) = sum / a(i)
```

Observations:



On X-axis: Matrix Size

On Y-axis: Time in ms

Sequential Optimization with Partial Pivoting:

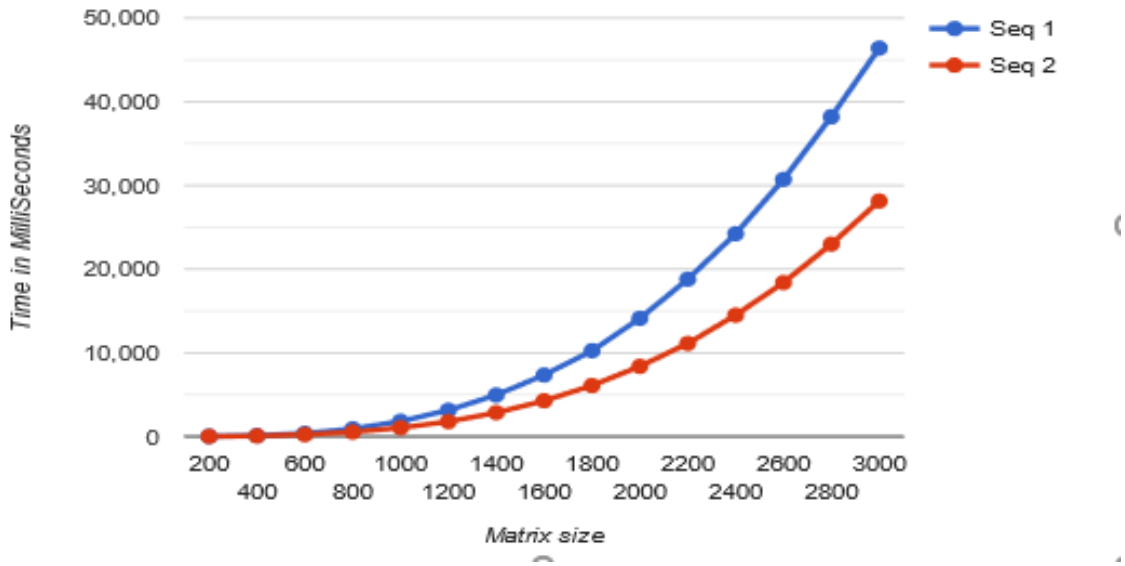
The partial pivoting technique is used to avoid roundoff errors that could be caused when dividing every entry of a row by a pivot value that is relatively small in comparison to its remaining row entries.

In partial pivoting, for each new pivot column in turn, check whether there is an entry having a greater absolute value in that column *below* the current pivot row. If so, choose the entry among these having the *maximum absolute value*. (If two or more entries have the maximum absolute value, choose any one of those.) Then we switch rows to place the chosen entry into the pivot position before continuing the row reduction process

Pseudocode for Gaussian Elimination with Partial Pivot:

```
for i = n to n – 1:
    find and record k where |A(k,i)| = max{i<=j<=n}|A(j,i)|
    i.e., largest entry in the rest of column i
       if k != i
           swap rows i and k of A
       end if
    A(i+1:n,i) = A(i+1:n,i)/A(i,i) …each quotient lies in [-1,1]
    A(i+1:n,i+1:n) = A(i+1:n, i+1:n) – A(i+1:n,i)*A(i,i+1:n)
```

Observation:



Parallel Implementation:

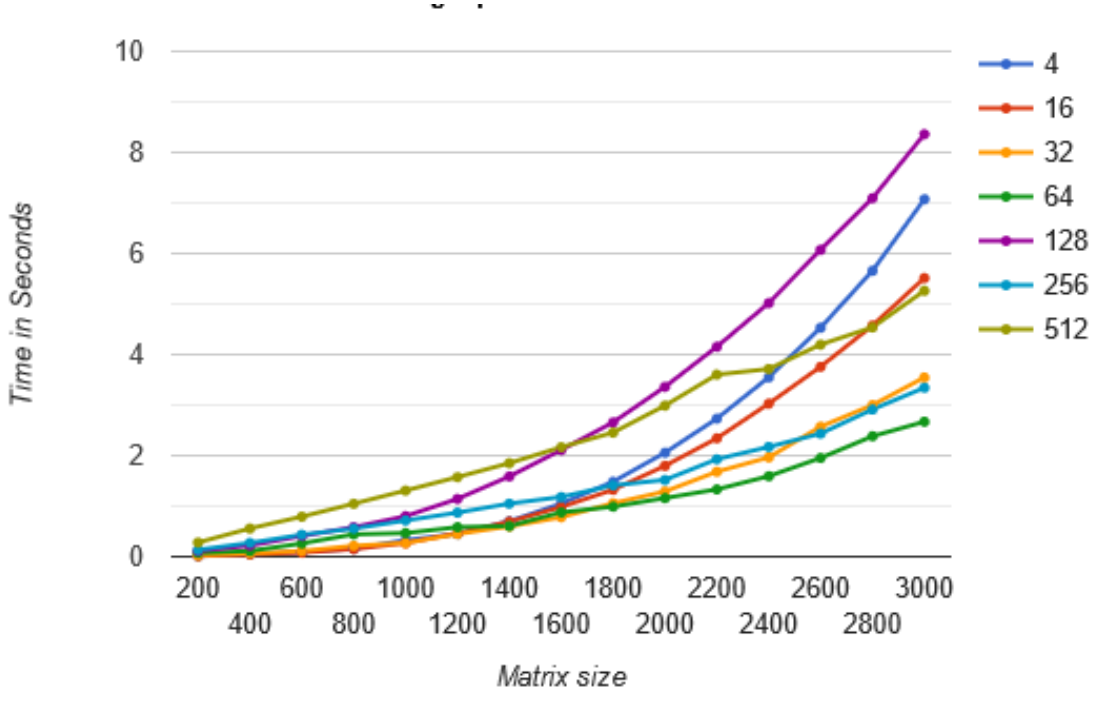
Gaussian Elimination using OpenMP:

Here we are trying to parallelize the Gaussian Elimination algorithm. But there is some date dependency across the outer loop, hence we cannot parallelize outer loop. There is no data dependency for the middle loop, so we can parallelize the middle loop.

```
Loop 1 : for (norm = 0; norm < N - 1; norm++) {
            Loop 2 : for (row = norm + 1; row < N; row++) {
                                multiplier = A[row][norm] / A[norm][norm];
                     Loop 3 : for (col = norm; col < N; col++) {
                                        A[row][col] -= A[norm][col] * multiplier;
                                }
                                B[row] -= B[norm] * multiplier;
                        }
                }
```

In first iteration of loop 1 we need to initialize all entries in column[0] starting from row[1] to '0'. This happens in loop 2 by "Add to one row a scalar multiple of another" such that corresponding column value becomes 0. This operation happens in parallel where all the threads performs this process on different rows. After this is done, the next column will be picked up in loop 1 and the operation is repeated. Once the matrix is in row echelon form, back substitution is performed to calculate the result.

Observation:



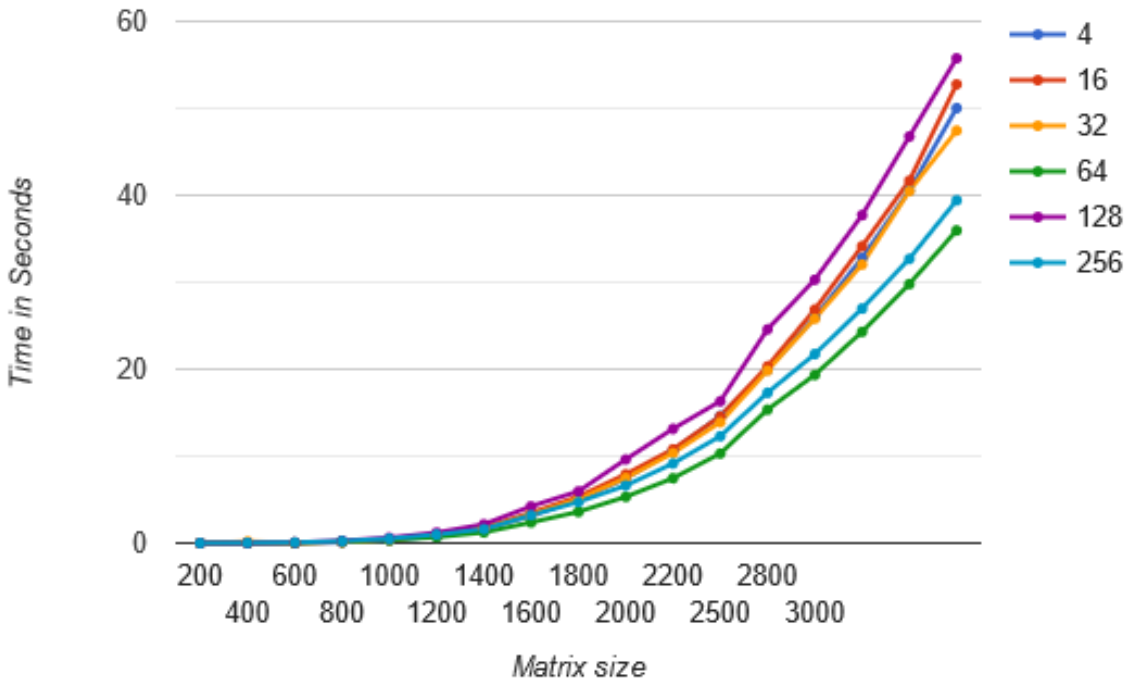
Gaussian Elimination using MPI:

There exists a data dependency across iterations of Loop 1, hence it was not parallelised. There is no data dependency across iterations of Loop 2, hence it was parallelised using data parallelism. Process with rank_id 0 is responsible for input and output.

```
Serial code which was parallelised :
Loop 1 : for (norm = 0; norm < N - 1; norm++) {
            Loop 2 : for (row = norm + 1; row < N; row++) {
                                multiplier = A[row][norm] / A[norm][norm];
                     Loop 3 : for (col = norm; col < N; col++) {
                                        A[row][col] -= A[norm][col] * multiplier;
                                }
                                B[row] -= B[norm] * multiplier;
                        }
                }
```

First Process 0 will broadcast a[outerloopindex][0] and b[outerloopindex] and send than send (MPI_Send) to each process,a[row] and b[row] corresponding to the iteration it has to compute using static interleaving logic. Then process 0 will compute the iteration assigned to it and wait for send of all processors to complete. Process 0 will recieve(MPI_Recv) from each processor the computer result. Other processors will recieve(MPI_Recv) from processor 0 the iterations assigned to them. They will compute the iteration assigned to it. They will synchronously send(MPI_Send) the result of their computation to processor 0. Then Processor 0 will now perform backsubstitution and finally Processor 0 will output the result.

Observations:

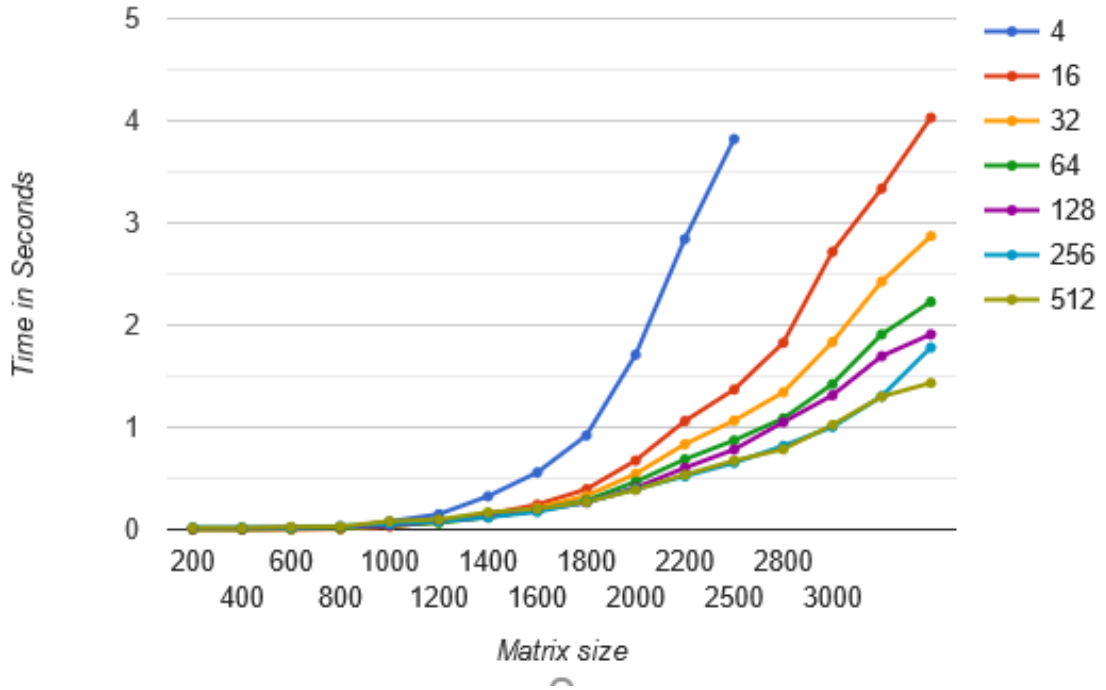


Gaussian Elimination using MPI Partial Pivot (Pipelined):

In the parallel implementation, since the data is split into multiple processes, the technique becomes slightly more involved, due to the communication design and ordering of the data. The technique now additionally involves data partitioning before the sequential steps, and gathering the results at the end, and is coupled with multiple communication steps in each phase to achieve the desired result. In this algorithm consists of three phases

1. Partitioning of Data: after the input is read from file, the data is portioned to all the processes in a cyclic manner, by cyclic partition the workload on each processor remains even.
2. Gaussian Elimination along with partial pivoting - This phase follows a pipelined communication architecture, such that each process receives a row of data only from its preceding process, and sends any data (either a received row, or one of its rows) only to the next process, by rank. When a process receives a row, it first communicates ahead to the next waiting process, then performs elimination of all rows that it contains whose indices (according to the original matrix) are greater than that of the received row. When it is ascertained that some row of the current process can be used to perform elimination on other rows (this is when row index number of 0s lie before the element of that row whose index matches the row index), it first places a pivot, then performs a division of the pivot element with all other elements of the same row, then communicates this row forward.
3. Back substitution - This phase is like the previous phase in pipelined communication design, except that the order is reversed, and instead of complete rows, only the values of pivot elements of rows are communicated upward. Using this technique, the values of all variables are computed such that the system of equations can be satisfied.

Observation:



Conclusion:

For all the implementations used above we have taken the matrix input size ranging from 10 to 3000. For parallel implementation, the number of processors/threads used are 4, 16, 32, 64, 128, 256 and 512.

For Serial Implementation, we have observed that with partial pivot we can avoid roundoff errors when the diagonal value of a matrix is zero. For OpenMP implementation, with increase in threads, the time decreases but we have optimal execution time for 64 threads. For MPI implementation, with the increase in processors, there is a huge decrease in execution time but when the processor size increases, difference in time decrement is very small, this is due to the huge communication costs involved in between the processors. Communication cost dominates the computation cost at higher processor values.

Group Members

Vinay Kumar Dupati - 16330346

Aravind Chilakamarri – 12588306

Bhanu Prakash Banala - 12588132