HW3 Performance Analysis

MPI-

Correctness argument: At the beginning of each iteration of the “for” loop, the entire matrix A and B is divided into equal parts and scattered across all the processes. The row on which normalization is done, along with matrix size and some other data is also broadcasted. At the end of each iteration the normalized data is gathered back at process 0, divided again and again the remaining part is scattered. This process is repeated until the entire matrix is normalized. At the end of it, the process 0 does the task of back substitution and displaying the result.

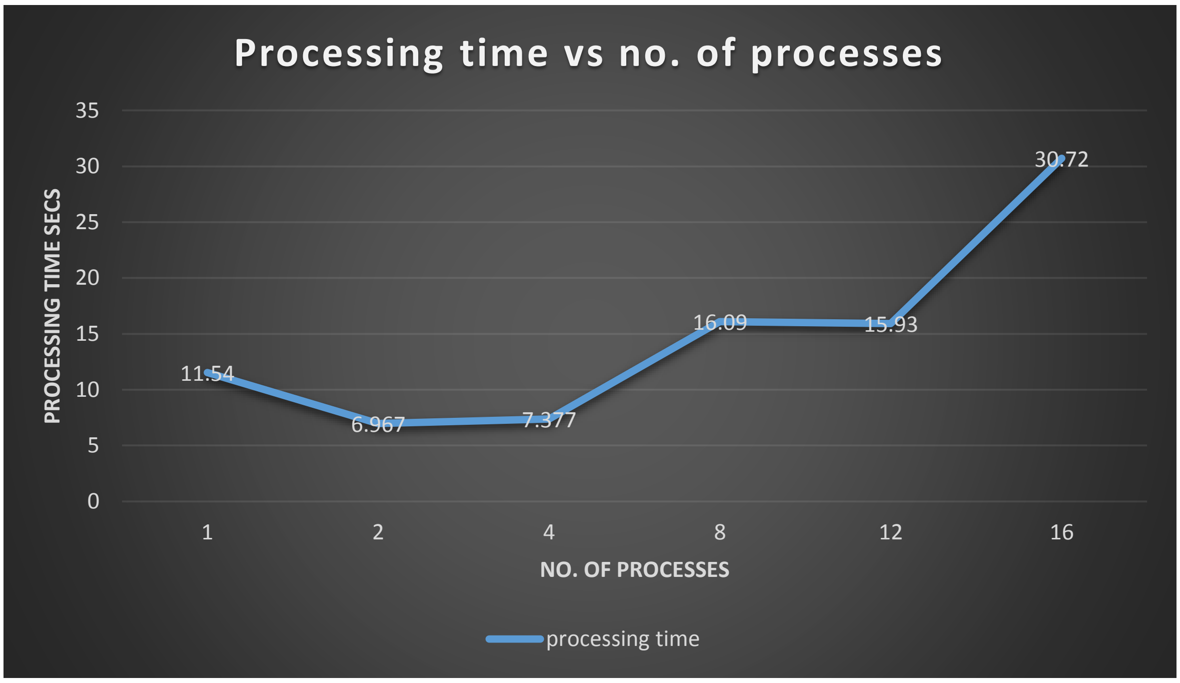
Different Versions tried:

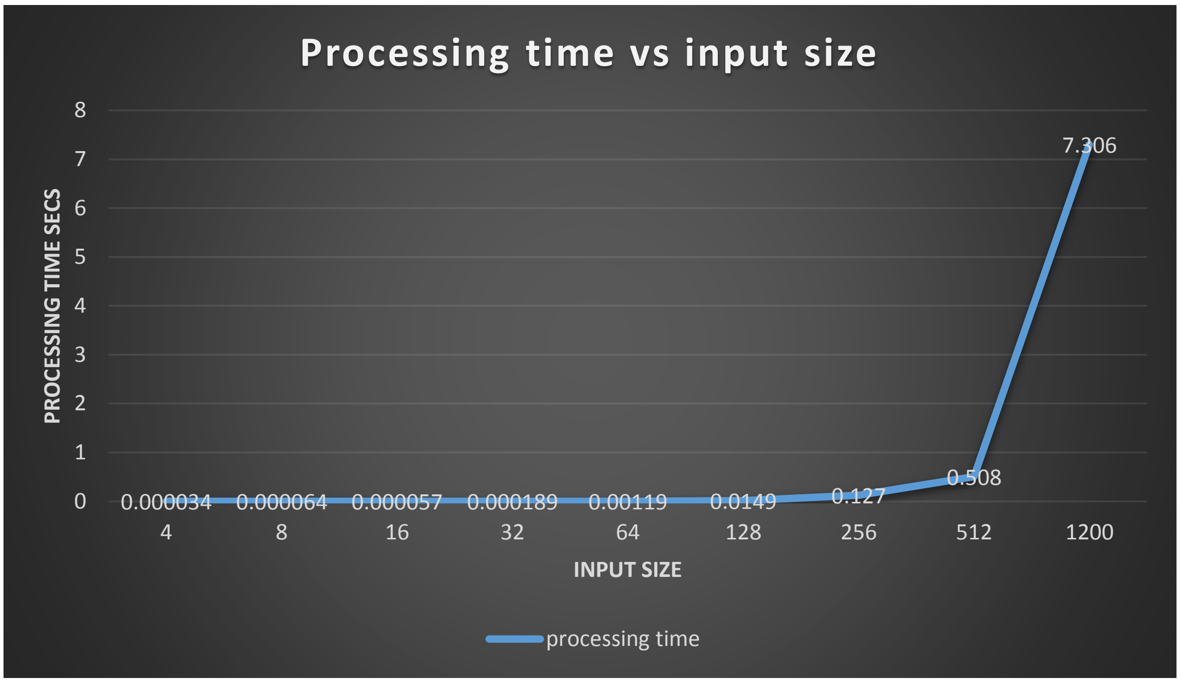
1. The first version of the program used to use MPI_send() and MPI_receive() for sending the entire data. It was not very efficient and was making the code look ugly.

2. In the second version, I replaced the MPI_send() and MPI_ receive() operations inside the loops, by some system provided methods like MPI_GatherV(), MPI_ScatterV(), MPI_Broadcast() etc, which made the code a little bit cleaner. I could not replace all of them though, due to time constraints.

3. In the third and final version, the logic of randomly generating the input matrices, and the functionality to monitor performance (timings) using MPI_wtimer() was added.

Performance analysis:





Following table shows the comparison of the time required for the task to complete for an input size of 1200*1200 matrix, for the number of processes in 1, 2, 4, 8, 12 &16. Initially, the processing time drops by almost half when the number of processes are changed from 1 to 2. While the number of processes is changed to 4, the time almost remain same. On further increasing the number of processes, the processing time increases continuously. This could be due to a very large communication cost among those processes. Also, the processes may have started getting executed on different physical machines (nodes), as the number of processes increases by 8 (as Jarvis could run 8 processes per node) and as the communication cost between 2 different node is way more, the processing time increases by a big factor.

The second graph shows the plot of processing time vs the input size of the matrix for 4 processes.

Number of processes	**Processing time (secs)**
1	11.54
2	6.967
4	7.377
8	16.09
12	15.93
16	30.719

Input Size	**Processing time (secs)**
4	0.000034
8	0.000064
16	0.000057
32	0.000189
64	0.00119
128	0.0149
256	0.127
512	0.508
1200	7.306

Possible Improvements:

1. Avoid broadcasting input matrix in every iteration: We can avoid broadcasting the input matrix in each and every iteration. Only the normalizing row can be sent in every iteration and each process can keep its share of input matrix and keep processing it. This may make the program a little more complicated, as all of the processes will have their turn to send the normalizing row, once it reaches to their share of matrix. Also the number of active processes actually doing some work on matrix will reduce as the matrix size grows smaller, but still it will have better performance due to reduced cost of communication.

2. Using MPI_scatterV and MPI_gatherv() instead of MPI_send() and MPI_receive(): There are some places where I still have the old fashioned MPI_send() and MPI_receive() inside loops for data scattering. I did replace it at some places, but left the others as it is, due to time constraint. I can replace that part as well, so that the code looks far lucid and clear. It will also increase the performance a little bit.

Speed up:

Sp = Ts/Tp

For input size N=1200,

no. of process =2

Ts = 11.54 s

Tp = 6.967 s

Sp =11.54/6.967

Sp = 1.658

So it's a sub linear speedup