Matrix Multiplication using Message Passing Interface

Hardware Specifications

All experiments given in this report are done on MacBook Pro i7th 9th Gen Processor with 16GB Ram and 6 cores.

1. Blocking P2P Communication

Blocking P2P communication directly implies that the process won't proceed with next instruction until the message buffer being sent/recieved is safe to use. Meaning, the message buffer must be passed completely before proceeding.

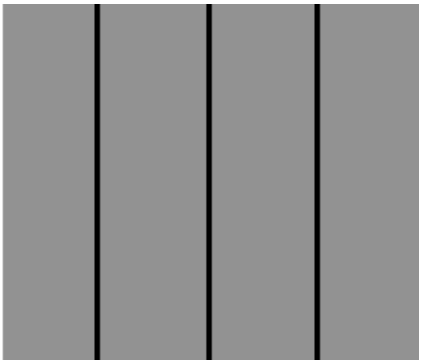
We have had many options for communication mode and matrix multiplication algorithm for multiple processes.

- Design Principles

  1. We used 1D row major array to store 2D matrices which has global pointers.

  2. We divided A vector coloumn wise and B, row wise, shown as below. This way it was easier to pass mutually exclusive blocks to all slave processes. Matrix B was passes easily as we had to divide contiguos chunks of row major vector.

  3. Finally, all processes will perform serial matrix multiplication on $A_{Nx(32/np)}$, $B_{(32/np)xN}$



- Working Analysis

1. Master process allocates memory for input vectors; divides the input vectors, into blocks.

2. Each block is sent to other processes using MPI_Rsend( ) which assumes that each process is already ready to recieve the message (which is justified because this is the first message recieved by the processes)

3. Each block is sent with a unique MPI_TAG (rank*13 for A_block and rank*97 for B_block, to make ensure that tags are unique).

4. Each slave process recieves the message using MPI_Recv( ) with their respective unique MPI_TAG; performs serial multiplication onto these submatrices; and send the result using MPI_Send( ) (not using MPI_Rsend( ) because master process might not be ready to recieve any message, since it could be sending data to other slave processes).

5. After sending submatrices, master process recieves result from indivisual slave process using MPI_Recv(C_block, MPI_Float, MPI_ANY_TAG, comm, &status) and add to resultant matrix C.

6. This recieving is made sequential because of processes might might not be sending in the right order.

7. The only component in this algorithm which is parallel, per say, is the step where each process is multiplying submatrices. This should happen at the same time.

- Restrictions

1. No. of processes multiplying the submatrices are $(n-1)$ out of n processes.

2. Number of processes should one greater than the number of processes that are going to compute the multiplication of the submatrices. There is a need of num_processes to be divisible by N i.e. the order of size of matrix; otherwise Segmentaion fault is trapped (Signal 11).

3. -O3 flag is used for improvement of performance.

2. Non-blocking P2P Communication

Here, MPI_Isend( ) doesnt wait for the complete transaction of message before executing the next instruction.

For two consecutive MPI_Isend( ) sending A and B to two different processes, the same MPI_Return variable is used.

- Design Choice: It is no different from Blocking P2P. The only difference is in the explicitly stating the MPI_Wait(&request, &status) just after sending all the matrix blocks.
- Also, each slave process have to check

A	B	C	D
Matrix Size	Blocking	Non Blocking	Collective
1000(1)	0.0222	0.0229	0.0231
1000(2)	0.012	0.0118	0.0112
1000(4)	0.0102	0.0103	0.0066
2000(1)	0.0923	0.1017	0.1102
2000(2)	0.0477	0.0523	0.0463
2000(4)	0.0387	0.0394	0.0252
4000(1)	0.4197	0.4155	0.4318
4000(2)	0.2129	0.2107	0.2033
4000(4)	0.1802	0.1782	0.1073
8000(1)	1.8681	1.8895	1.8997
8000(2)	0.9649	0.9564	0.9224
8000(4)	0.6995	0.6943	0.4805
16000(1)	7.7842	7.7485	7.7716
16000(2)	4.161	4.0422	3.92
16000(4)	2.9768	2.8955	2.0509

if master process is ready to recieve the partial result or not.

3. Collective Communication

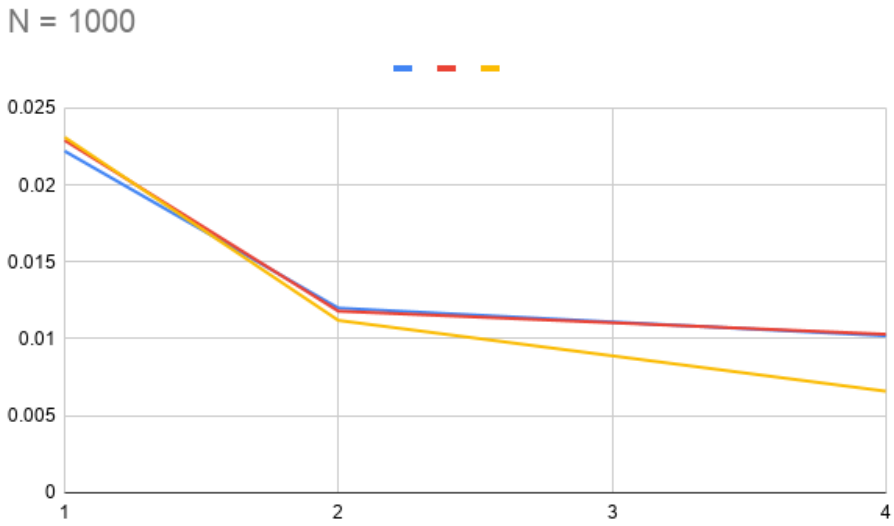
Here MPI_Scatter( ), MPI_Gather( ) and MPI_Bcast( ) are used which Scatter Matrices as Blocks to processes, do the computation onto these and then gather the results in one single process.

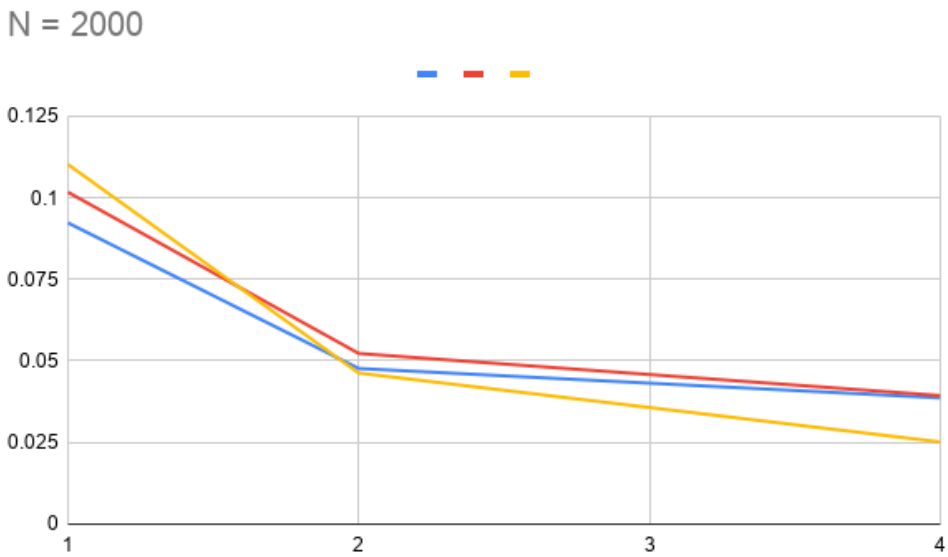
For two consecutive MPI_Isend( ) sending A and B to two different processes, the same MPI_Return variable is used.

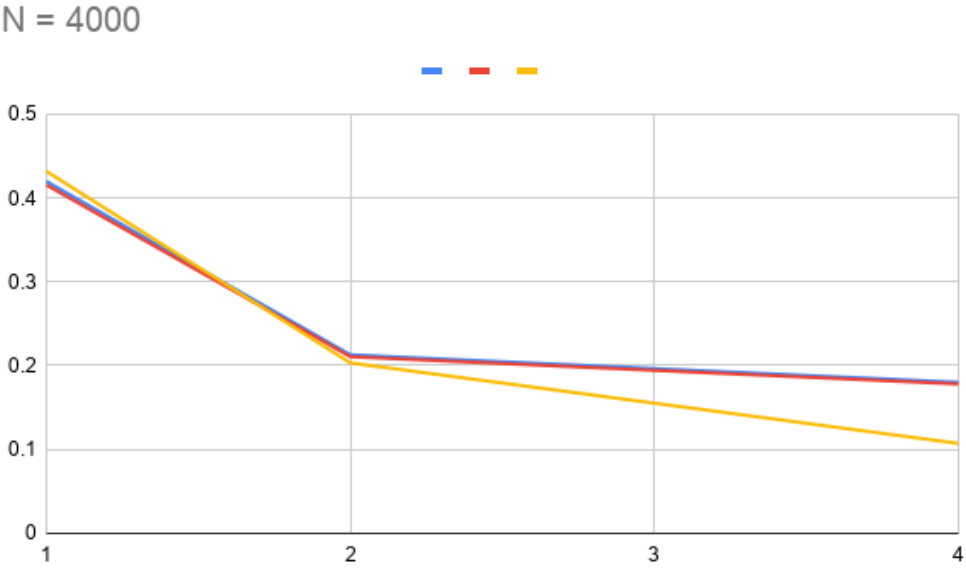
Restrictions-

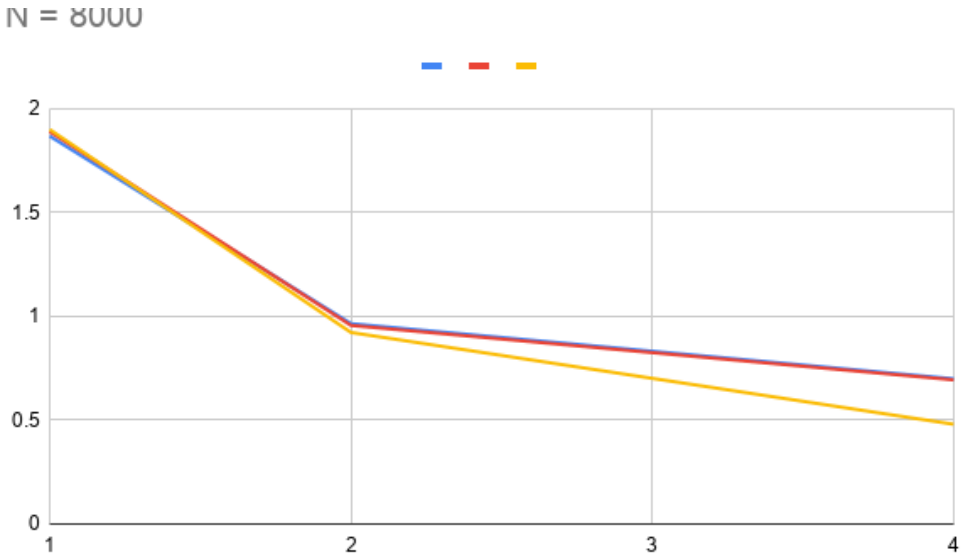
1. The size of the matrix (N) should be divisible by np(number of processes) otherwise, this will result in an uneven share of matrices resulting in wrong dimensions of matrix, hence the segmentation fault was occuring (MPI_ABORT was invoked)
2. Matrix B Is broadcasted to all the processes because here matrix A is scattered in chunks of rows and multiplied with complete B, because we cannot scatter non-contiguos elements from a array.
3. -O3 flag is used for performance enhancement but we can get a better idea of time affected by MPI APIs if we dont use the flags optimisation

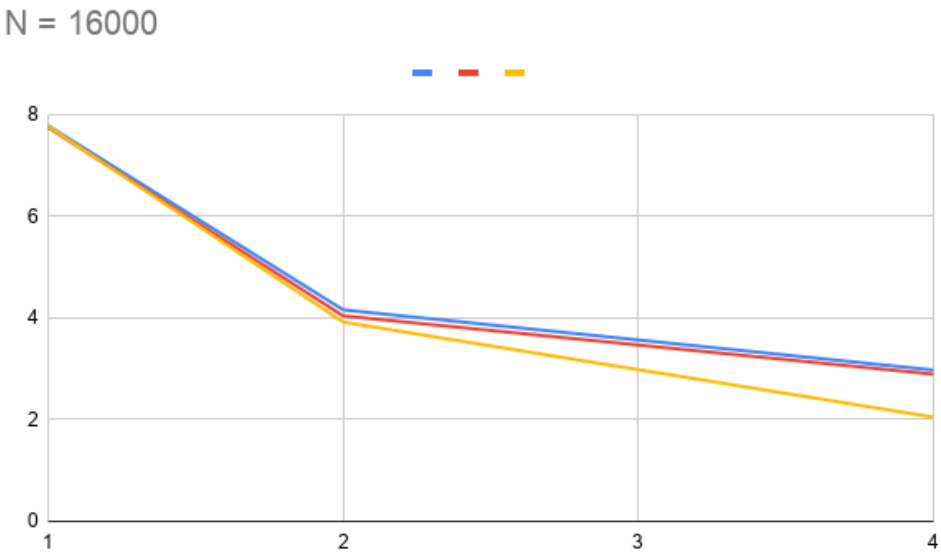
4. Results | Time of Executions

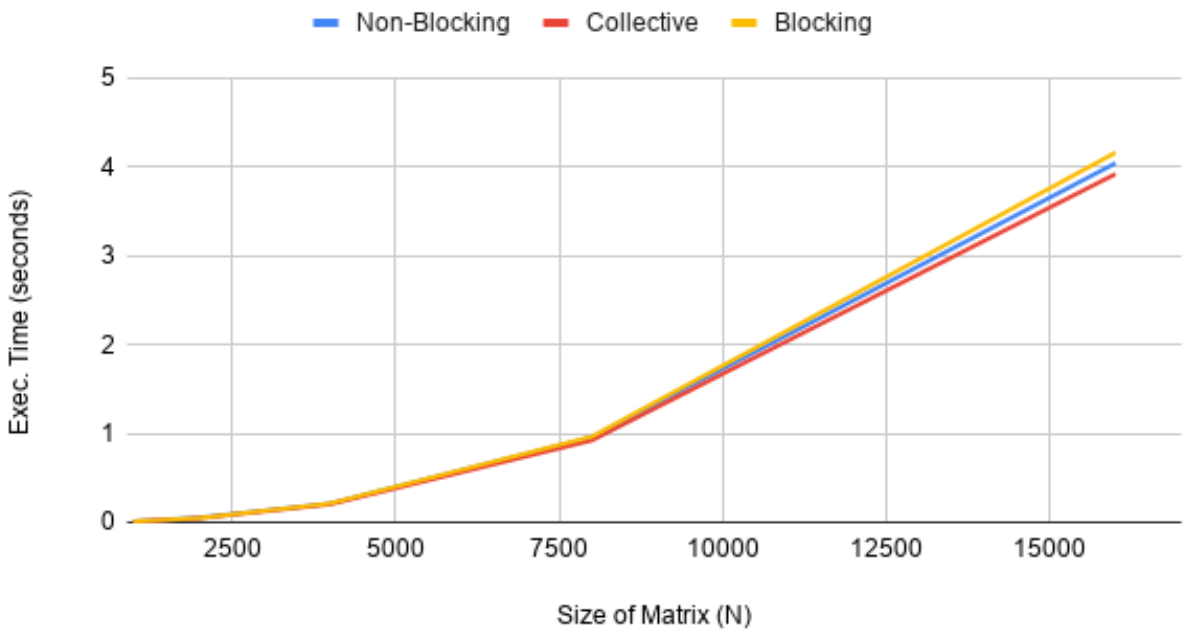







N = 8000
2
1.5
1
0.5
0
1
2
3
4


N = 16000
8
6
4
2
0
1
2
3
4



The following graph shows quantitative analysis of time of execution for different size of matrices. This clearly shows the asymptotic time to be in the quadratic order i.e. $O(n^2)$.

The order $O(n^2)$ is justified theoretically because we have number of coloumns in A and number of rows in B as a constant i.e. 32. We are exploiting this information in our algorithms to divide matrix data such that each process gets a constant number of either number of rows or coloumns.