COL381 A2 Part 1 — Report

Matrix multiplication using MPI with different communication methods

Matrices storage pattern

The matrices used in the computation—namely the matrices A, B and C where C = A x B—have been stored in memory as a contiguous block of memory, i.e. a simple 1D array. Also, to note, it's stored in row-major format, so to access the (i, j)-th element of the matrix A, we'd access the corresponding array as A[i * m + j].

Algorithm description :: Serial

The base algorithm, the serial one is as simple as it could get. The algorithm loops through the rows first, and then the columns, of the resulting matrix C, and calculates it's value by looping through the appropriate values in matrix A and B. To note —

- The matrix A and C are accessed in row-major format, so there won't be many cache misses while accessing them.
- Whereas, for the matrix B, there will be cache misses. This might slow the program.

Parallelization

As we discussed in the previous section, if we access the matrices in row-major order, that results in the best utilization of the matrix storage pattern in memory. So, I parallelized it that way, each worker gets a number of rows of the matrix C to process. So to that end, every worker had to be sent some rows of the Matrix A, the whole of matrix B, and some rows of the matrix C. Below is a brief step by step description of the algorithm with parallelization —

- MPI environment initialization.
- The first node creates the matrix A, and the second node creates the matrix B, randomly.
- The first node, i.e. the master node, sends parts of the total computation load to the workers.
- The master node does it's share of the workload. In the meantime, the worker nodes do their shares of the workload too.
- The master node collects the different parts of the matrix C to itself.

Differences among different communication modes

In principle, all of the 3 implementations are the same, as described in the section above. There are some differences when we get to the specifics, like the following —

- In blocking P2P mode, the master node has to wait before all the data to be sent is sent completely, before it can start working on its share of the workload.
- In non-blocking P2P mode, the master node starts working on its share of the workload while the data is being sent.
- In collective communication mode, the master node scatters the matrices that are to be divided among all the nodes, and once everyone is done with their part of the computation, the matrices are gathered back. The communications are done in blocking mode only, so there isn't really much performance gain here, compared to blocking P2P mode.

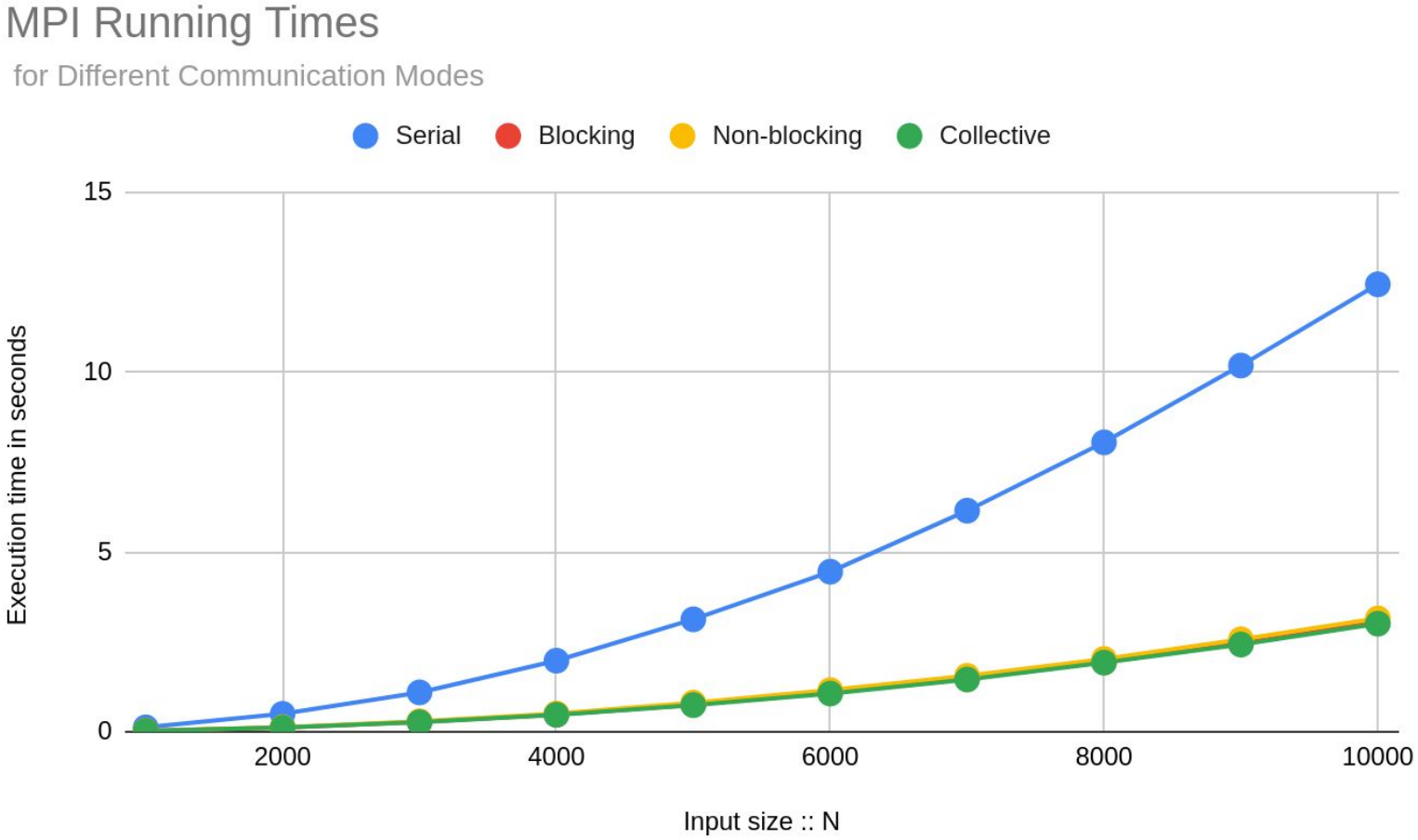
Restriction of N in the program

In the program I've assumed, or restricted, N to a multiple of 4. I am running 4 nodes with MPI to do these computations, and so in the scatter and gather operations the data couldn't be distributed or collected properly if it's not that way.

Performance testing

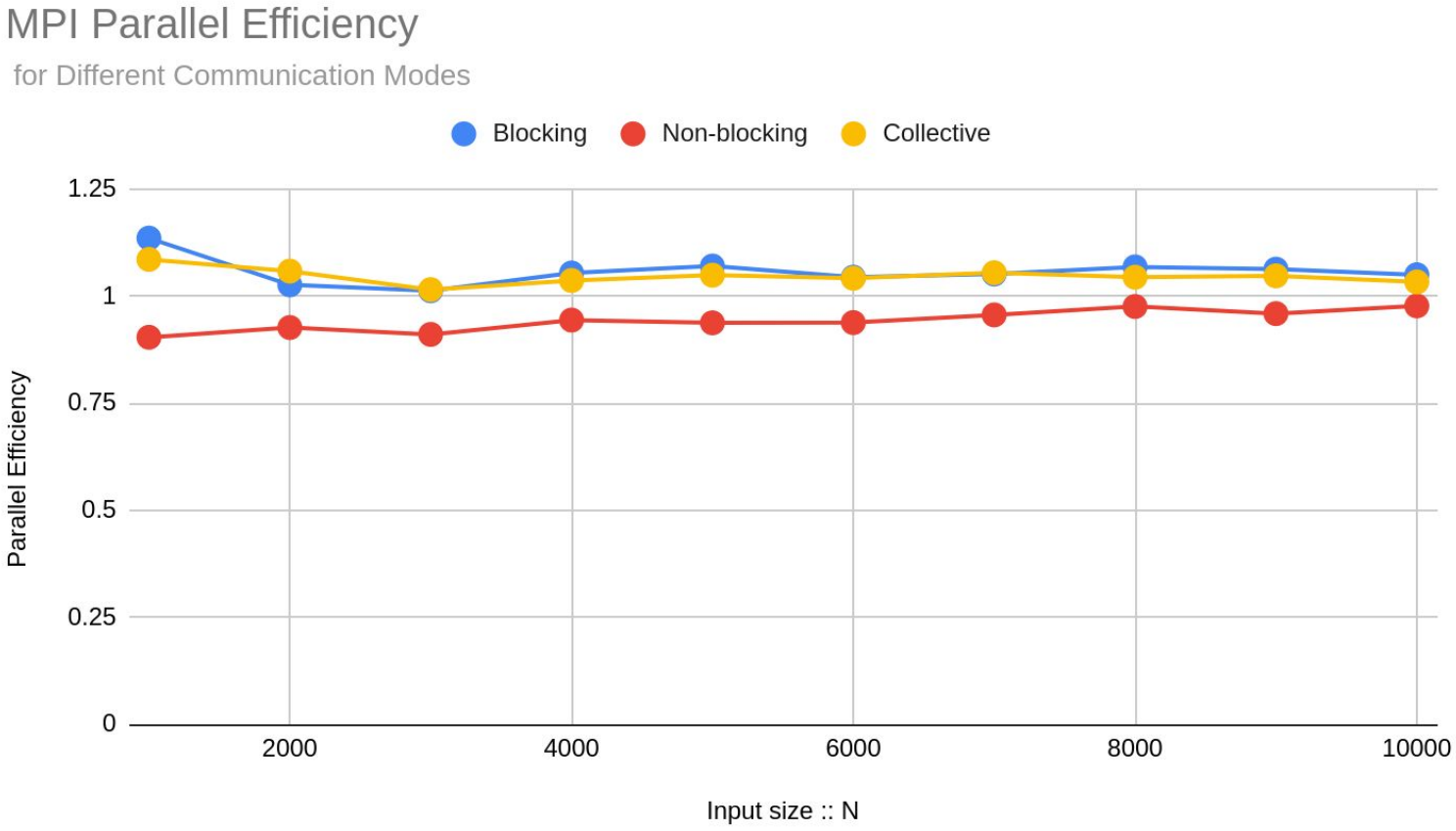
Running times

Running Times										
N	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Serial	0.128372	0.50407	1.100393	1.982082	3.13301	4.457145	6.154918	8.05506	10.192181	12.450277
Blocking	0.029454	0.122735	0.280313	0.481833	0.762302	1.099096	1.49212	1.945866	2.46311	3.052491
Non-blocking	0.034557	0.132006	0.297683	0.51303	0.81552	1.163981	1.560559	2.030939	2.57778	3.157778
Collective	0.029546	0.119855	0.27087	0.477889	0.746472	1.069289	1.458487	1.927931	2.431924	3.011838



Parallel Efficiency

N	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Blocking	1.13589835	1.026744612	1.012842786	1.05440578	1.070713116	1.044676034	1.051971189	1.06865298	1.063328982	1.050281229
Non-blocking	0.9041872848	0.9270241504	0.9109421767	0.9444428201	0.9379776094	0.9385449591	0.9564942754	0.9762174049	0.9594969315	0.9775609622
Collective	1.086204562	1.058841934	1.015609887	1.036894551	1.049272444	1.042081467	1.055017631	1.044521303	1.047748717	1.033445109



Findings and observations

- We can see that the running times increases quadratically with the input size, and this does agree with the theoretical time complexity of our algorithm, which is $O(N^2)$.
- We can also see that the parallel efficiency is lowest for the non-blocking P2P call. This might be happening because the overhead with creating a progress thread and keeping it running hampers the overall performance.