COL380 ASSIGNMENT 2

Shayan Aslam Saifi
Raval Vedant Sanjay

1 Introduction

The results are obtained corresponding to a system with $i5 - 7^{th}$ generation processor, having 4 cores. The code can be compiled and Run by the command by the command

```
mpicc serial.c -o code
mpirun -np <No of PROCESSORS> ./code <N> <32> <N>
```

2 Algorithm Description

For the Serial part of the code, we used the $O(n^3)$ algorithm as given in the problem statement. We also implemented the **isEqual()** function by taking the differences of the float values and checking if it's lesser than a threshold or not (of the order 10^{-4}). We generated A and B as random Matrices having the values within 0 to 1000. This implementation of the Serial code is the same among all the other procedures developed

2.1 Blocking Point to Point

For this, the communication strategy is as follows:

- Process 0 generates the random A and B matrices
- The rows of matrix A are divided into **size-1** parts, Which will be uniform only if $N|(size-1)$
- The matrices A and B are sent from the master process (P0) to the rest, with A being sent to each processor in a divided form while B is sent as the entire matrix
- The different processes receive the matrices A and B, compute the serial matrix multiplication of them (Note that the size of A for any process is not $N \times 32$ now) and store the result in the corresponding sub-parts of the matrix C

- They then send the different parts of the result obtained to the Master Process
- The master process receives all the different parts of the matrix C, by carefully assigning the appropriate memory offset for each processor it is receiving from

Note that a processor P1 will start executing it's code once it receives the matrix A and B, and it won't wait for some other processor P2 to receive them. That is, all the computations involved above are happening in a parallel for the different processes.

2.1.1 Some Optimizations that failed or worked

- It seemed very wasteful to be having to send the entire matrix B to all the processors every time and so I thought of randomizing the matrix B in the main function while creating the Matrix A in the processor 0. Then, we would be saved from the overhead of sending the entire matrix B all the time, though it would incur the cost of creating B for each of the process, but it should also not be ignored that memory operations are slower than the ALU operations. And this start to give a significant improvement on the parallel performance. So, this optimization **worked** well
- To access the Matrix A in the different processes, I was passing an offset from the 0^{th} location and an integer specifying the number of rows for each process. But this seemed very wasteful since we are just passing and retrieving constant values. So I thought of having an array initialized beforehand for the offset and the size of the rows, rather than having them passed from the Process 0 to the rest and vice versa. Though, this optimization did **Not** work good, and it was observed that the performance was slightly degraded. This would be because of the fact that storing and retrieving these values from an array is indeed a memory operation and it might get really slow.
- If we have an assumption that the number of rows going to each parallel thread are equal, then rather than passing the number of rows between the processors, we can directly compute it and that would save time. Thus, this method will **work** well under an **assumption**

2.2 Collective Communication using Scatter and Gather

For this, the communication strategy is as follows (Note that for this algorithm, it has been assumed that $N|(size)$:

- Process 0 generates the random A and B matrices
- The Matrix A is scattered among all the processes by dividing the rows and the Matrix B is broadcasted

- The Matrix product of the Scattered A and the broadcasted B is computed for all the processes and then the result obtained is gathered to process 0

Note that here, we are using communication barriers so that the Process 0 does not go ahead with printing the partially computed C as the final product and waits for everyone to finish computing the Matrix multiplication

2.2.1 Some Optimizations that failed or worked

- Broadcasting the matrix B everywhere seemed somewhat wasteful and so I thought of letting all the processes get the matrix B directly from the random number generation, but doing so clearly increases the time of execution of the parallel program. It may be because of the reason that the MPI libraries are highly optimized for their operations and so though it may seem kind of trivial to not broadcast the matrix B but actually doing that helps our performance. So, this method did **Not** work

- It's very interesting to see that doing the same thing for the Point to Point Blocking case seemed to improve the running time clearly, and here it is clearly getting worse. So from this, we can even see that the broadcast function is much efficient as compared to the ordinary Send & Recv. Thus, the highly **optimized** implementation for the Collective communication modules makes it very fast and efficient

- Another issue in my communication model was that that it was calling MPI_Finalize() after all the threads have done their work. But in this method, it is very easy to see that after doing the parallel task, all the work is done by only the Process 0. And also, since the parallel task is done by all the processors in a uniform manner, so we can separate the sequential work done by the Process 0 and the work done by all the processors in parallel (Note that such kind of separation is not possible for the point to point models). So, I restructured my program in this manner and then called the MPI_Finalise() function right after the parallel segment. By doing this, there was not much change in the parallel execution (Obviously!), but the sequential code got optimized to a great extent. The reason behind this is that after the parallel work, the MPI environment is effectively running all the processes and looking for the messages by sending signals etc. But this kind of communication is not needed at all, since now only one thread is executing and so it would be performing better outside of the MPI environment and so is the case as well! Thus, this optimization performed **much better** in terms of the total time taken

2.3 Non Blocking Point to Point

The Basic structure for the communication protocol for this method is the same as the one considered for the Point to Point blocking case. Though, there are some different implementation ways to consider for this protocol (Along with the Blocking one). So, in general they can have the following implementations possible:

- Non Blocking Send and Non Blocking Receive
- Non Blocking Send and Blocking Receive
- Blocking Send and Non Blocking Receive

So, there can be three different alternatives to implement the Communication between the Master process and the Worker processes for any single variable to be passed. Though, while implementing this method, we used a minimum amount of communication, as opposed to the case of the Blocking protocol where we passed many parameters between the different processes to generalize for any condition. But here, to have the number of rows of A matrix distributed uniformly among all the other processes, we take the condition that $N|(size)$ Here, we basically implemented the protocol as a combination of the second and third point as discussed above after a series of careful experimentation

2.3.1 Some Optimizations that failed or worked

- As seen in the collective communication, broadcasting the matrix B performs better than having that matrix created for all the processors. So, I tried to broadcast the matrix B among all the processors rather than having any send-recv communication among them and that improved the performance somewhat. So, Broadcasting the matrix B rather than having point to point communication for that **worked** pretty well. Though, to keep things very distinct for all the methods, I did not use it in my final code

- In my original implementation, I was doing a blocking send of the Matrix A from the Master process and a Non blocking receive for that in the processors (Along with the corresponding MPI_Wait to ensure that it has acceptable values before it starts doing the computations) and then a blocking send of the resultant matrices from the processor to the processor 0, having a non blocking receive for that. Thus, the entire model was using the protocol of Blocking Send and Non Blocking Receive everywhere. Though, for that, when I tried to optimize the system by having a non blocking send from the workers and a blocking receive for the master, and it was observed that the time of the computations improved a little bit on an average (there was not a huge difference and the ranges of their values were kind of overlapping). So, this improvement **worked** slightly better for the average case

- Initially I was passing a large number of parameters among the processors to take care for the general scenario. But that seemed to be having a large communication overhead as compared to just two integers being passed and their use would be just to handle the issue of uneven distribution of the rows. And so, to take care of this, I added the assumption for the rows to be evenly distributed among the processes and thus reduced the number of variables getting exchanged and that resulted in improving the performance. So, this method **worked** somewhat good

- I even tried of sending one row of A matrix at a time to the different processes rather than sending the entire chunk at once, to see if the memory overhead is more than the communication overhead or not and if I can get any improvements by this assumption or not. But I found instead that the communication overhead gets indeed worse than the memory overhead and so doing this doesn't work out. So, this approach did **Not** work

2.4 Overall Summary

The overall summary of the brief implementational choices for all the different methods is tabulated below

P2P Blocking	Collective	P2P Non-Blocking
The communicational calls are blocking i.e the execution of the process is suspended till the data being received/sent is safe to use	The communicational calls are collective i.e all the processes within a communicator are involved	The communicational calls are non-blocking i.e the process is not suspended by the environment and it becomes the duty of the programmer to verify that the data being transferred is safe to use
P-1 processors are actually being used in parallel	**P** processors are actually being used in parallel	**P-1** processors are actually being used in parallel
Process 0 is the Master and the rest are the workers	All the processes are equivalent	Process 0 is the master and the rest are the workers
Matrix A is generated by the Master and sent to the workers	Matrix A is scattered among the different processes	Matrix A is generated by the Master and sent to the workers
Matrix B is generated by the Master and sent to the workers	Matrix B is broadcasted among the different processes	Matrix B is generated by the Master and sent to the workers
Matrix C is generated by the Workers and sent to the Master	Matrix C obtained by the different processes is gathered	Matrix C is generated by the Workers and sent to the Master
MPI_Send() and MPI_Recv() are the main functions used	MPI_Scatter(), MPI_Gather(), MPI_Bcast(), MPI_Barrier() are the main functions used	MPI_Isend(), MPI_Irecv(), MPI_Wait() are the main functions used
No Assumptions are there	Assumed that $N\|P$ for the distribution of the rows of A to be uniform among all the processes	Assumed that $N\|(P-1)$ for the distribution of the rows of A to be uniform among all the workers

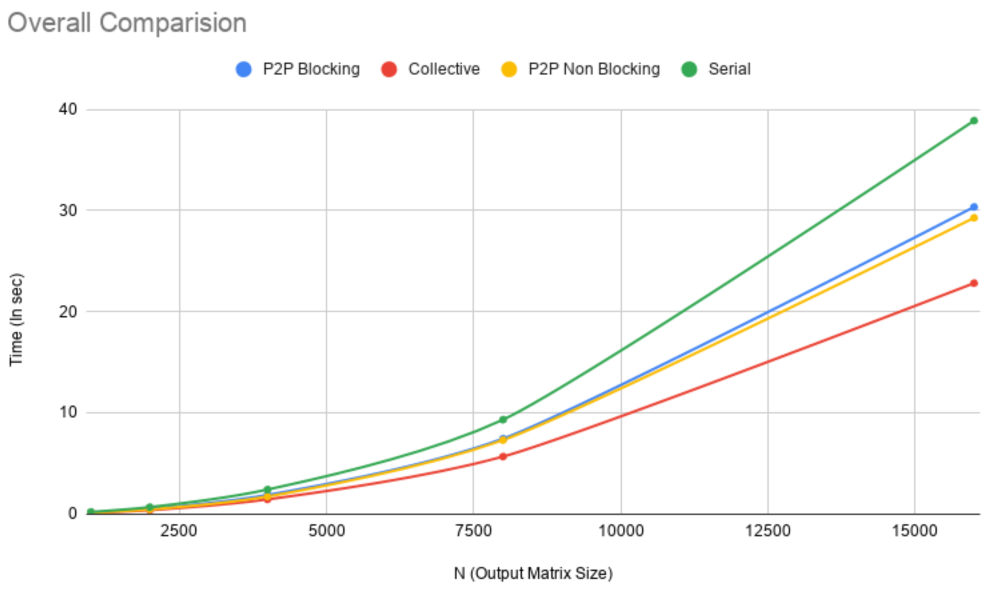
3 Results

All the results obtained by me were corresponding to taking four threads. The table and the collective plot are given below:

3.1 Data Table

Matrix Size (N)	Time for P2P Blocking	Time for Collective	Time for P2P Non-Blocking	Time for Serial
1000	0.109934 sec	0.084594 sec	0.105412 sec	0.166381 sec
2000	0.470378 sec	0.346454 sec	0.420738 sec	0.646632 sec
4000	1.852006 sec	1.413231 sec	1.700941 sec	2.386484 sec
8000	7.418002 sec	5.645778 sec	7.277276 sec	9.298531 sec
16000	30.346025 sec	22.805662 sec	29.269631 sec	38.886002 sec

3.2 Comparision plot



3.3 Observations

If we look at the ratios of the consecutive running times in the table (for any method), then we can see that this value is nearly equals to 4. And it is the same as the square of the ratio of the corresponding matrix sizes. Thus from this, we can see that the theoretical complexity of $O(n^2)$ is indeed satisfied!

From the data as well as the plot, we can also see that the running time of the serial algorithm is much lower than the running times of all these three parallel algorithms. So, we can infer from this that parallelism has indeed achieved! Though, the extent of improvement for all these different methods is somewhat different with their own sets of reasons

We observe that the time for the collective algorithm is the lowest among all the different algorithms because of the nature of the communication involved and the optimized way of doing the operations like broadcast, scatter etc for the data which are supposed to be seen by all the processors, rather than having many naive point to point communications among the different processes

The time for both the implementations of the Point to Point communication methods are nearly the same and their plots are very closely lying to each other. Since, the only big difference between these methods is that the Blocking method halts the execution while in the Non Blocking method, we added Wait after every send and receive operations to ensure that the data being sent or received is safe to use. Or in other words, we can say that the Blocking program kind of automates things for us by halting the execution in any case, while in the Non Blocking case though we get the benefits of setting the Wait ourselves, but for this program it is not possible to exploit the stronger features of this method because of the nature of the communications involved. Though, Non Blocking performs slightly better than the Blocking case because of the Not-So-Naive Waiting scheme