Lab2 – MPI

- **Please briefly explain how the data and computation are partitioned among the processors. Also, briefly explain how the communication among processors is done.**

  In my implementation, I partitioned the data by rows. The master process would share kI/Size rows of matrix a with all (Size) processes. The 'Size' term refers to the number of processes used in the MPI. So, the master process scatters the matrix a by rows sharing (kI= total number of rows in a)/Size to all the processes, and then broadcast all the b matrix with all processes. Then, after all the processors finished their calculations, the master processor would gather all the calculated results and collect them in the matrix c.

- **Please analyze (theoretically or experimentally) the impact of differen communication APIs (e.g. blocking: MPI_Send , MPI_Recv , buffered blocking: MPI_Bsend , non-blocking: MPI_Isend , MPI_Irecv , etc). Attach code snippets to the report if you verified experimentally. Pleasechoose the APIs that provide the best performance for your final version.**

  Blocking communication generally do not return until the communication is finished as they do block while the non- blocking communication return immediately even if the communication is not finished yet as they do noy block which allow the work to be overlapped. On the other hand, the buffered mode does not rely on system buffers. The developer must supply a buffer that must be large enough for the messages.

  Generally, buffered blocking has significantly better performance than buffered non-blocking because of the existence of the buffer. The master processor doesn't have to wait for other processes to start sending data. If the messages are sent in non-blocking buffered way in a space managed by the OS, then a process may fill this space by flooding the system with a large number of messages. As for the non-buffered non-blocking communication, it is much better, and faster as it allows more parallelism, and work overlapping. The non-buffered blocking communication is the safest, and more portable as it does not depend upon the order in which the send and receive are executed or the amount of buffer space, but it can incur substantial synchronization overhead.

- **Please report the performance on three different problem sizes ($1024^3$, $2048^3$, and $4096^3$). If you get significantly different throughput numbers for the different sizes, please explain why.**

**For $1024^3$ size:**
Run parallel GEMM with MPI
Time: 0.0379193 s
Perf: 56.633 GFlops

**For $2048^3$size:**
Run parallel GEMM with MPI
Time: 0.258378 s
Perf: 66.4913 GFlops

**For $4096^3$ size:**
Run parallel GEMM with MPI
Time: 1.86489 s
Perf: 73.6982 GFlops

**Sequencial GEMM:**

**For size 4096:**
Time: 734.154 s
Perf: 0.187207 GFlops
**For size=2048:**
Time: 59.5108 s
Perf: 0.288685 GFlops
**For size=1024 :**
Time: 3.47097 s
Perf: 0.618699 GFlops

**For the size = $4096^3$:** speedup = 393.67x, 73.6982 GFlops
**For the size= $2048^3$:** Speedup = 230.3x, 66.4913 GFlops
**For the size= $1024^3$:** Speedup = 91.54x, 56.633 GFlops

- **Please report the scalability of your program and discuss any significant non-linear part of your result. Note that you can, for example, make np=8 to change the number of processors. Please perform the experiment np=1, 2, 4, 8, 16, 32.**

**For np=1**
Problem size: 4096 x 4096 x 4096
Initialize matrices a and b

Run parallel GEMM with MPI
Time: 6.82199 s
Perf: 20.1465 GFlops

**For np=2**
Problem size: 4096 x 4096 x 4096
Initialize matrices a and b

Run parallel GEMM with MPI
Time: 3.49439 s
Perf: 39.3313 GFlops

**For np=4**
mpiexec -np 4 ./gemm
Problem size: 4096 x 4096 x 4096
Initialize matrices a and b

Run parallel GEMM with MPI
Time: 1.86489 s
Perf: 73.6982 GFlops

**For np=8**
Problem size: 4096 x 4096 x 4096
Initialize matrices a and b

Run parallel GEMM with MPI
Time: 1.7425 s
Perf: 78.8747 GFlops

**For np=16**
Problem size: 4096 x 4096 x 4096
Initialize matrices a and b

Run parallel GEMM with MPI
Time: 7.05612 s
Perf: 19.478 GFlops

**For np =32**
Problem size: 4096 x 4096 x 4096
Initialize matrices a and b

Run parallel GEMM with MPI
Time: 28.9866 s
Perf: 4.74146 GFlops

**For np=1:** Speedup= 107.6x, 20.1465 GFlops
**For np=2:** Speedup= 210.1x, 39.3313 GFlops
**For np=4:** Speedup= 393.67x, 73.6982 GFlops
**For np=8:** Speedup= 421.32x, 78.8747 GFlops
**For np=16:** Speedup= 104x, 19.478 GFlops
**For np=32:** Speedup= 25.33x, 4.74146 GFlops

From these results, we can see that the performance increases linearly until it reaches np=8, and then it starts decreasing (for np=16). So, my results are not in a linear speedup as the number of processors increases. This is due to the parallel overhead during the communication between the processes since it will take noticeable time copying and transferring data between processes which can affect negatively the performance as we increase the number of processes.

- **Please discuss how your MPI implementation compares with your OpenMP implementation in Lab 1 in terms of the programming effort and the performance. Explain why you have observed such a difference in performance *(Bonus +5)***

In term of programming effort, I believe that OpenMP was easier than MPI since it just consists of adding pragma lines (directives) in the correct places and everything else is done for us to parallelize the program. While with MPI we need to understand how the communication between processes occurs and all communication modes. We also need to allocate the buffers and make them available to other processes, and we need to be careful with deadlocks. MPI is more complicated than OpenMP in term of programming effort. In term of performance, OpenMP in lab1 gave us a better performance for the calculation of the matrix than MPI in lab2. This is due to the fact that we are not actually using multiple nodes. We are actually running MPI in a single node and moving data in the DRAM between processes (MPI in shared memory set up). This operation comes with overhead of forking the processes, copying data, and allocating memories in one single node which affect negatively the performance. I believe that if we are parallelizing our program in a single node, it is better to use OpenMP since it will give us better performance than MPI as the parallelization occurs between the cores, and OpenMP handles it perfectly. MPI is more suitable for parallelization between multiple nodes in which the overhead of copying data, memory allocation, and calculations can be split between the nodes.