CPSC 524 Assignment 3: Parallel Matrix Multiplication

Rami Pellumbi*

November 13, 2023

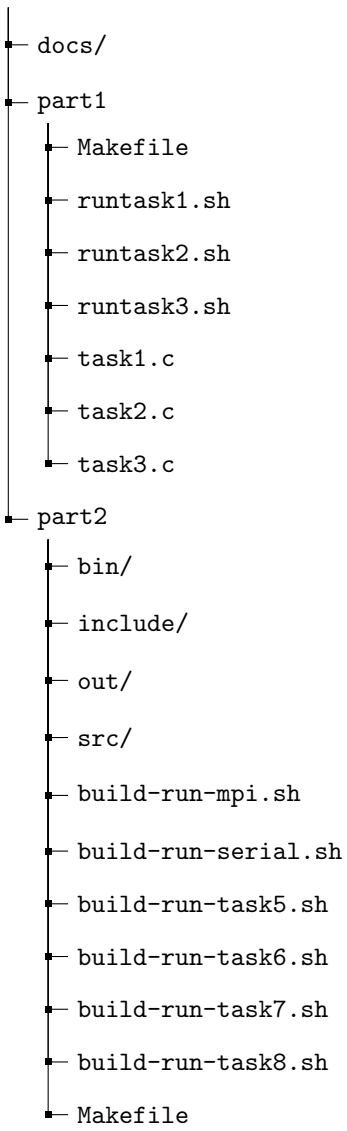
*M.S., Statistics & Data Science

1 Introduction

This assignment investigates the implementation and performance of an MPI- based program tasked with parallel computation of large double precision matrices. The program's objective is to effectively utilize multiple processors to perform the multiplication, optimizing the use of system resources and minimizing computation time. The implementation required careful consideration of data distribution and process synchronization, ensuring accuracy while maximizing computational efficiency. Performance analysis focuses on the scalability of the solution across various matrix sizes and the number of processors used. This report outlines the design considerations, describes the implementation strategy, and presents an evaluation of the program's computational performance.

2 Project Organization

The project is laid out as follows:



- **docs/**: This folder contains LaTeX files and other documentation materials that pertain to the report.
- **part1/**: Code for MPI basics and part 1 tasks.
- **part2/**: Code for parallel matrix multiplication.
  - **bin/**: The `bin` folder holds compiled objects and executable files, centralizing the output of the compilation process.
  - **include/**: Here, all the header files (`.h`) are stored.
  - **out/**: The `out` folder stores the outputs from each task. It also houses the csv file containing data generated by the programs.
  - **src/**: This directory houses the source files (`.c`) that make up the benchmarks.
  - **Shell Scripts**: The shell scripts are used to submit the job for the relevant task to slurm via `sbatch`.

3 Code Explanation, Compilation, and Execution

This section outlines the steps required to build and execute the code. The provided Bash scripts automate the entire process, making it straightforward to compile and run the code. All the below steps assume you are in the root of the project directory.

3.1 Automated Building and Execution

- **Part 1: Understanding MPI Basics** All part 1 related code is in the `part1/` directory. There are three scripts for part 1, one for each task The commands `sbatch runtask1.sh`, `sbatch runtask2.sh`, and `sbatch runtask3.sh` will compile and run the files `task1.c`, `task2.c`, and `task3.c`, respectively. The output of the experiments will be in `out/serial.csv`.
- **Part 2: Parallel Matrix Multiplication** All part 2 related code is in the `part2/` directory. There are multiple programs run in Part 2:
  - `serial.c` Simple, serial, matrix multiplication.
  - `task5.c`: Parallel matrix multiplication of square matrices with blocking collectives. Assumes that the number of rows is evenly divisible by the number of processes.
  - `task6.c`: Parallel matrix multiplication of square matrices with non blocking collectives. Assumes that the number of rows is evenly divisible by the number of processes.
  - `task7.c`: Parallel matrix multiplication of square matrices with non blocking collectives, with a focus on interleaving communication and computation. Assumes that the number of rows is evenly divisible by the number of processes.
  - `taks8.c` Parallel matrix multiplication of square matrices with blocking collectives. Appropriately handles when the number of rows is not evenly divisible by the number of processes.

  To run all the MPI experiments, execute `sbatch build-run-mpi.sh`. The results are stored in `out/omp.csv`.

3.2 Post-Build Objects and Executables

For part 2, upon successful compilation and linking, an `obj/` subdirectory will be generated within the directory. This directory will contain the compiled output files. Additionally, the executable files for running each part will be situated in the `bin/` subdirectory.

3.3 Output Files From `sbatch`

For part 1, the output files generated from running the code by submitting the relevant Bash script via `sbatch` will be in the `part1` directory. For part 2, the output files generated from running the code by submitting the relevant Bash script via `sbatch` will be stored in the `out` directory.

4 MPI Basics

5 Parallel Matrix Multiplication

At a high level, we have input $N \times N$ matrices A and B, and an output $N \times N$ matrix C.. The input matrices, A and B, are divided into blocks of rows and columns, respectively. Each block row of matrix A is then multiplied with the corresponding block column of matrix B to compute a segment of the resultant matrix C. This block-wise computation enables the distribution of the workload among the available processors in the MPI environment. The distribution and collection of data blocks utilize MPI's collective communication operations, ensuring that each processor has the necessary data to perform its assigned calculations. The final step involves aggregating the computed blocks from all processors to form the complete resultant matrix.

The implementation of the parallel matrix multiplication algorithm follows the ring-pass strategy, which is outlined as follows for the case where N is divisible by p (the number of processes):

1. The manager initializes memory of A, B, and C and initializes A and B with the data to be multiplied.
2. The manager partitions A and C into p blocks rows each. It partitions B into p block columns.
   - A and C are of dimension $(N/p) \times N$.
   - B is of dimension $N \times (N/p)$.
3. The manager permanently assigns to each MPI process one block row each of A and C, and it assigns each MPI process one block column of B as its initial assignment.
4. The computation proceeds iteratively until completed:
   (a) Each MPI process does all the computation it can with the data at hand.
   (b) MPI processes then pass their block columns of B to the next higher-ranked MPI process.
5. The manager uses MPI collective operations to assemble the full C matrix by collecting all block rows of C from the other MPI processes.

Task 5: Blocking Collectives

The first attempt at this algorithm was utilizing MPI's blocking collectives. Namely,

1. The manager process initializes matrices A and B with random double precision values and creates an empty matrix C to store the result.
2. `MPI_Scatter` is used to distribute the blockes of A, B, and C across all processes.
3. Each process computes a partial result by multiplying its block of A with the corresponding block of B and stores it in `multiply_result`.
4. The partial results are stored in the local block of matrix C, `blockC`.
5. Processes then pass their block of B to the next process using `MPI_Send` and `MPI_Recv`.
   - To avoid deadlock, even-ranked processes send before they receive and odd-ranked processes receive before they send.
   - Each process makes use of a secondary `blockB` buffer, named `tempB`, else a process may overwrite its initial segment of B before it is able to use it in a matrix multiplication. This allows one buffer to send while the other is being used in an operation.
6. After all computations complete, the processes gather their computed blocks into C using `MPI_Gather`.

The results of this approach are in Table 1.

Table 1: Blocking Collectives - Average Performance (s)

p/N	1000	2000	4000	8000
1	0.313	2.806	39.271	327.932
2	0.210	1.697	21.191	169.081
4	0.141	0.836	10.058	85.685
8	0.093	0.517	4.083	43.770

Comparing MPI blocking collectives to serial performance as process count increases.

Graphically:

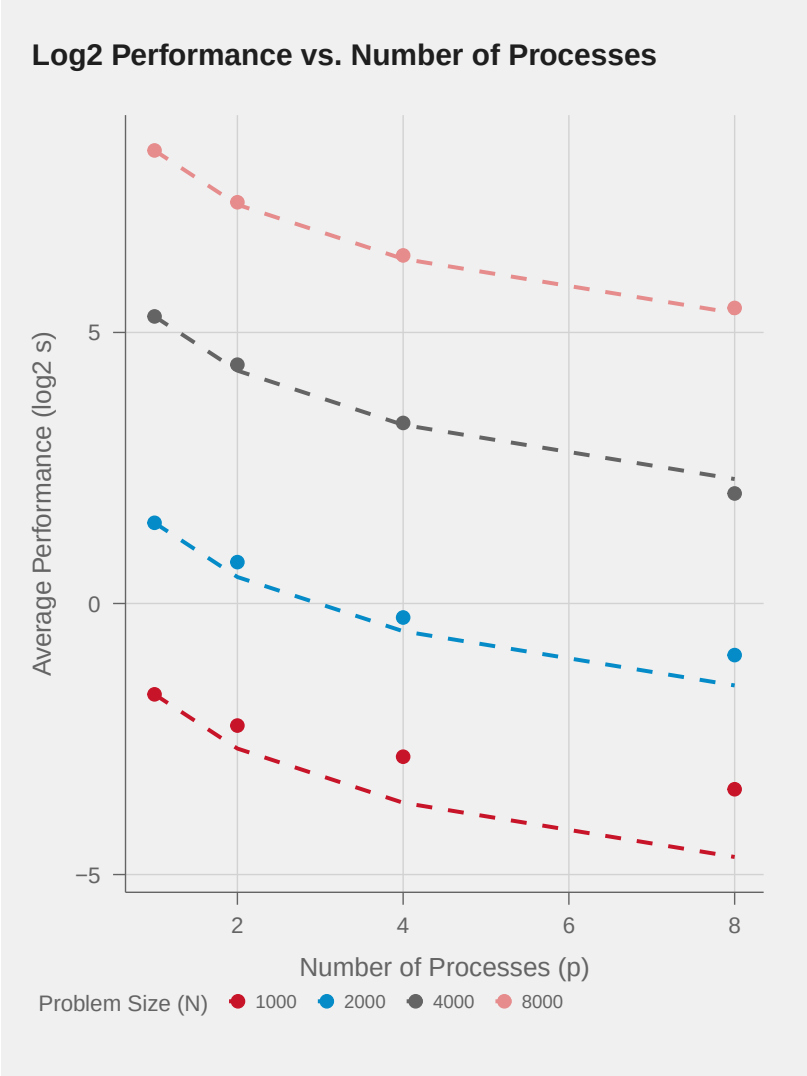


Figure 1: Blocking Communication: Log2 Performance vs. Number of Processes

The dashed line represents a doubling in performance (halving the runtime) as the number of processes doubles relative to the serial runtime. We see that for increasing N, we achieve closer to the desired throughput via scaling. For $N = 1000$, increasing the number of prcesses increases performance but at a smaller magnitude relative to larger N. It is likely that the overhead in communication here is to blame. As N increases, doubling the process count just about doubles performance. It seems as though the computational load is heavy enough that the communication overhead becomes a smaller fraction of the total runtime, allowing for better scalability. That is, the serial portion of a task (in this case, the communication overhead) becomes negligible as the problem size increases.

Task 6: Non Blocking Collectives

Next, we simply replace our blocking collectives with non blocking collectives. The algorithm and ring passing stayed entirely the same. What changes was:

- `MPI_Irecv` and `MPI_Isend` instead of `MPI_Recv` and `MPI_Send`, respectively.
- `MPI_Iscatter` and `MPI_Igather` instead of `MPI_Scatter` and `MPI_Gather`, respectively.

- Adding `MPI_Waitall` and `MPI_Wait` as necessary, e.g., to ensure the blocks have all been scattered.

The performance is seen in Table 2 and Figure 2.

Table 2: Non Blocking Collectives - Average Performance (s)

p/N	1000	2000	4000	8000
1	0.313	2.806	39.271	327.932
2	0.206	1.653	21.036	168.987
4	0.128	0.809	9.908	85.196
8	0.079	0.480	4.050	43.294

Comparing MPI non blocking collectives to serial performance as process count increases.

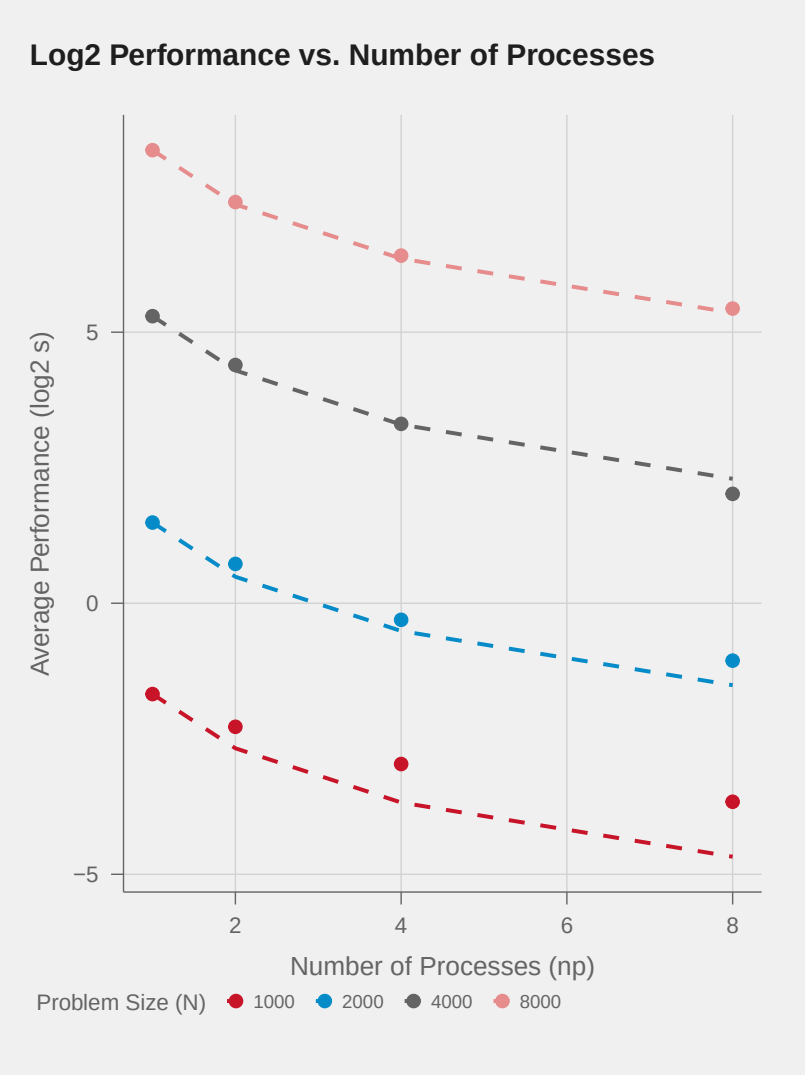


Figure 2: Non Blocking Communication: Log2 Performance vs. Number of Processes

The performance assessment here is precisely that of the blocking collectives. Since very little was changed in terms of computation and communication, we expect performance to be about similar, as observed.

Task 7: Overlapping Computation and Communication

Task 8: Dealing with Non Uniform Buffer Sizes

Lastly, we modify the blocking collectives code to handle the case where N is not divisible by p. The following had to be considered (in continuation from task 5):

- N/p will have a remainder, i.e., $N\%p$. This remainder will be less than the total number of processes and so all ranks less than the remainder will get $\mathsf{floor}(N/p)+1$ rows (or columns) and the rest of the ranks get $\mathsf{floor}(N/p)$ rows (or columns).
  - Now, every `blockA`, `blockB`, or `blockC` has array size $(\texttt{floor}(N/p)+1)\times N$ or $\texttt{floor}(N/p)\times N$.
- For the initial assignments, `MPI_Scatterv` is used with the appropriately passed in displacements array to distribute the blocks.
  - Each process still gets a permanant allocation of `blockA` and `blockC` and starts with an initial assignment of `blockB`.
  - Each process allocates a size for `blockB` and `tempB` that is the largest possible size. In doing so, the differently sized buffers can be passed around without having to reallocate memory.
  - At each time step on a process, a multiply might be between:
    1. An $(\texttt{floor}(N/p)+1)\times N$ `blockA` and an $N\times(\texttt{floor}(N/p)+1)$ `blockB`.
    2. An $(\texttt{floor}(N/p)+1)\times N$ `blockA` and an $N\times\texttt{floor}(N/p)$ `blockB`.
    3. An $\texttt{floor}(N/p)\times N$ `blockA` and an $N\times(\texttt{floor}(N/p)+1)$ `blockB`.
    4. An $\texttt{floor}(N/p)\times N$ `blockA` and an $N\times\texttt{floor}(N/p)$ `blockB`.

    To handle this multiply, each process has an array `multiply_result` of the maximum possible multiply result size that it allocates, i.e., $(\texttt{floor}(N/p)+1)\times(\texttt{floor}(N/p)+1)$.
- At step i in the ring swap, process p is working on the `blockB` initially allocated to rank `(rank - i + p) % p`. The size of each ranks initial allocation is kept track of to appropriately index the `blockB` array for multiplication (since it is allocated to the maximum possible size it may have data in it we do not want). The computed `multiply_result` is appropriately placed in `blockC`, offset by the appropriate index relative to the initial allocations.

The resulting code is an algorithm that produces the following performance and F-norm on an $N = 7633$, $p = 7$ program:

Table 3: Non Uniform Size Allocation with Blocking Collectives

p	N	time	F-norm
7	7633	43.22639	1.535e-09

6 Conclusion