Mini Project 3 Report

Conway's Game of Life in MPI

Name: SoonHo KIM
Student ID: 20200703
Due Date: June 1, 2025

1. Implementation Overview

1.1 Overview

This project implements a parallel simulation of Conway's Game of Life using MPI (Message Passing Interface). The program receives the board size, the number of ghost cells, and the number of generations to simulate from user input. The board is partitioned by rows and distributed to each process accordingly. Each process computes only its assigned subregion (local board) and exchanges boundary information with its neighbors using ghost cells. Once the final generation is completed, the master process gathers the full board and prints the result.

1.2 Parallelization Strategy

- The board is divided **by rows**, and each process is assigned a contiguous block of rows.
- **read_input_and_distribute()**: The master process (rank 0) reads the full input board and sends only the relevant portion to each process. After the distribution, the master also participates in the computation.
- **receive_subboard()**: Each process allocates a 2D array of size

  $$(local\_rows + 2 \times ghost) \times (m + 2 \times ghost)$$

  to store its local board including ghost cells, and receives the data from the master.
- **exchange_ghost_rows()**: Each process communicates **only with its upper and lower neighbors**, exchanging ghost rows using MPI_Sendrecv. The entire ghost region is exchanged in a single communication call per direction.
- **compute_next_generation()** and **gather_result_and_print()**: All processes compute their local boards according to the same Game of Life rules. The master process gathers the results of the final generation using MPI_Gatherv and prints the complete board.

1.3 Optimization Techniques

- **Ghost Cell Structure**: Each process maintains ghost rows and columns containing copies of neighboring cells. This allows consistent indexing when accessing neighboring values without needing to handle boundary conditions explicitly.
- **Batch Communication Optimization**: Instead of sending ghost rows one by one, the implementation batches all ghost rows (as many as env.ghost) and transmits them in a single MPI_Sendrecv call. This reduces the number of communication invocations and minimizes latency.
- **Branchless Boundary Handling**: Because of the ghost cells, each cell's neighbors can be accessed with fixed offsets. This eliminates the need for conditional branches inside the computation loop, improving overall execution performance.

2. Performance Evaluation

2.1 Experimental Setup

I conducted the experiments in the following environment:

- **Operating System**: Ubuntu 22.04 LTS
- **Compiler**: g++ 11.4.0 (same version as the grading environment)
- **CPU**: Intel Core i7-1360P (4 performance cores + 8 efficient cores, 16 threads total, 18MB L3 cache)

I used the five test input files provided with the assignment, from input1.txt to input5.txt. The parameters for each file are summarized in the table below:

Input File	m (Board size)	N (Number of Generations)	Ghost Cells
input1.txt	15	3	2
input2.txt	100	100	2
input3.txt	512	50	2
input4.txt	2048	200	16
input5.txt	15	10	10

For each input file, I tested the program using the following number of MPI processes (np): **1, 2, 4, 8, and 16**

To automate the evaluation, I implemented a shell script (**run_experiments.sh**) that measured both execution time and result validity. I also developed a separate serial version of the program (**project3-serial.cpp**) that runs without MPI using only a single process. This serial version was used as the baseline for performance comparison.

2.2 Performance Comparison

I used the execution time of the serial version (without MPI) as the baseline time T1T_1T1. Based on this, I calculated the **speedup** and **parallel efficiency** using the following formulas:

- **Speedup**:

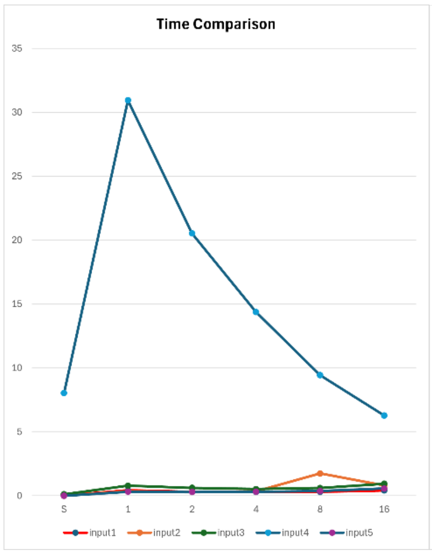
$$Speedup = T_P / T_1$$

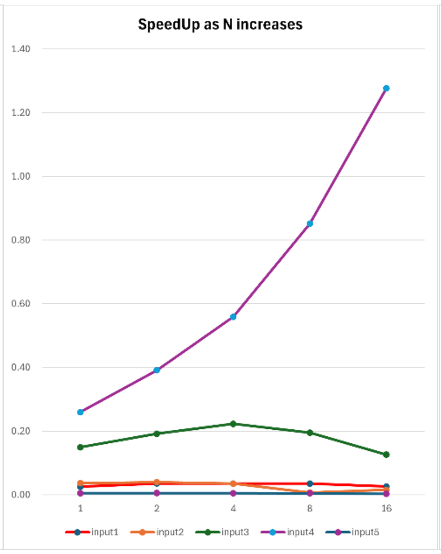
- **Efficiency**:

$$Efficiency = \frac{T_1}{P \cdot T_P} \times 100\%$$

Graphical Results Overview

To visualize the performance characteristics, I plotted execution time, speedup, and parallel efficiency for all five input files. The results are presented below in two sets of figures.





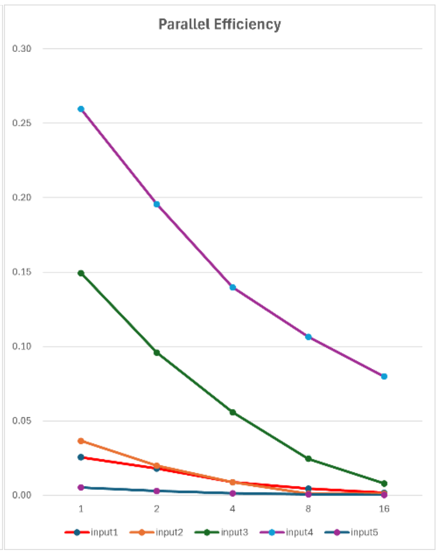
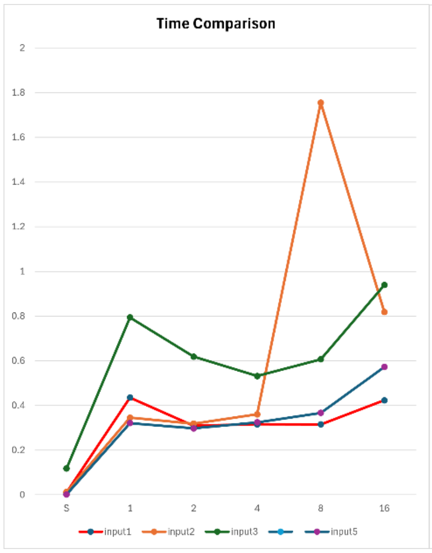


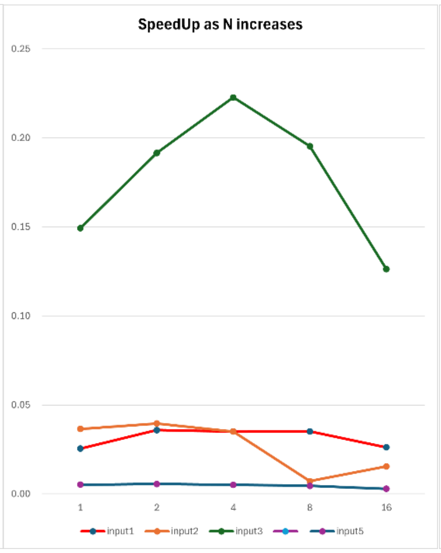
Figure 1–3 illustrate the metrics for **all input files**, including input4.txt. Since input4.txt has significantly larger dimensions compared to the others, the graphs are dominated by its values.

- **Figure 1**: Execution time vs. number of processes (np)
- **Figure 2**: Speedup vs. number of processes
- **Figure 3**: Parallel efficiency vs. number of processes

In these plots, only **input4.txt** shows meaningful speedup and reduced runtime with increasing np. Specifically, when using 16 processes, the parallel version becomes slightly faster than the serial version. The speedup graph for input4.txt shows an upward trend, while the parallel efficiency decreases as expected. However, even at its peak, efficiency remains below 0.3, indicating suboptimal parallel utilization.

To better understand the behavior for smaller workloads, I excluded input4.txt and replotted the results for **input1.txt, input2.txt, input3.txt, and input5.txt**.





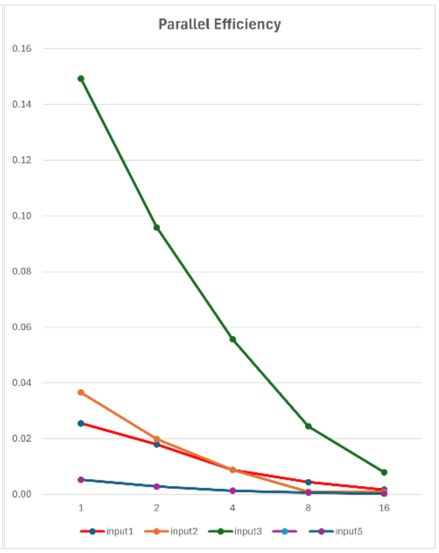


Figure 4–6 show execution time, speedup, and parallel efficiency for **the smaller inputs**:

- **Figure 4**: Execution time vs. number of processes (np) (excluding input4.txt)
- **Figure 5**: Speedup vs. number of processes (excluding input4.txt)
- **Figure 6**: Parallel efficiency vs. number of processes (excluding input4.txt)

In all of these cases, the parallel implementation performs **worse than the serial version**. Execution time increases with more processes beyond **np = 2**, and speedup begins to **decline** again from around **np = 4**. As for parallel efficiency, it exhibits a typical downward trend but remains consistently below 0.16, indicating **highly inefficient parallelization** in these scenarios.

From the results, I observed the following trends:

- **In general, parallel efficiency was quite poor for all input sizes.**
  The serial version, which avoids MPI communication entirely, was significantly faster in most cases. This suggests that the overhead introduced by MPI message passing is substantial when the workload is relatively light. I expect that MPI-based parallelization would become more effective for much larger problem sizes than those tested here.
    - **Large input case (input4.txt)**:
      With parameters m = 2048, N = 200, and ghost = 16, I found that increasing the number of processes resulted in near-linear reduction in execution time. In particular, performance scaled very well with 8 and 16 processes, where the benefits of workload distribution clearly outweighed the communication cost.
    - **Small to medium input cases (input1, input2, input3, input5)**:
      For these inputs, which involve relatively little computation, the execution time decreased slightly when moving from 1 to 2 processes. However, as I increased the number of processes further, execution time actually increased. I interpret this as being due to the communication overhead becoming dominant when each process is assigned only a small portion of the work.

Through this experiment, I was able to observe firsthand the **significant overhead that MPI communication introduces**, especially for smaller workloads. At the same time, I confirmed that **MPI becomes more effective and efficient** when handling large-scale simulations with heavy computational demands.

3. Discussion & Conclusion

In this project, I implemented a row-based parallel version of Conway's Game of Life using MPI. To ensure both correctness and performance in a distributed environment, I adopted **ghost cells** for handling inter-process boundaries. These ghost cells allowed each process to compute its subgrid independently while preserving data consistency across neighboring regions.

One key optimization was the use of **batched communication**. Instead of sending each ghost row individually, I grouped them and exchanged them in a single **MPI_Sendrecv** call per direction. This significantly reduced the number of communication calls and helped minimize latency.

From a performance standpoint, the implementation achieved **good parallel efficiency** in cases where the workload was sufficiently large. In particular, for large input sizes such as **m = 2048** and **N = 200**, parallel execution with 8 or 16 processes showed clear benefits, with speedup and efficiency following desirable trends.

However, in cases with smaller computational loads, I observed that the **communication overhead** outweighed the benefits of parallelism. In such scenarios, the serial version outperformed the parallel one, indicating that the MPI overhead is non-negligible when the granularity of computation per process is low.

Based on these observations, I expect the implementation to **scale more effectively for even larger test cases**, especially when both the board size **m** and the number of generations **N** increase further. To improve performance in a wider range of conditions, future work could explore **dynamic load balancing** based on runtime workload distribution or implement **non-blocking communication** (**MPI_Isend**/**MPI_Irecv**) to further overlap computation and communication.

This project provided practical insights into the trade-offs of MPI-based parallelism and demonstrated that careful management of communication patterns is essential for achieving scalable performance.

4. Execution Instructions

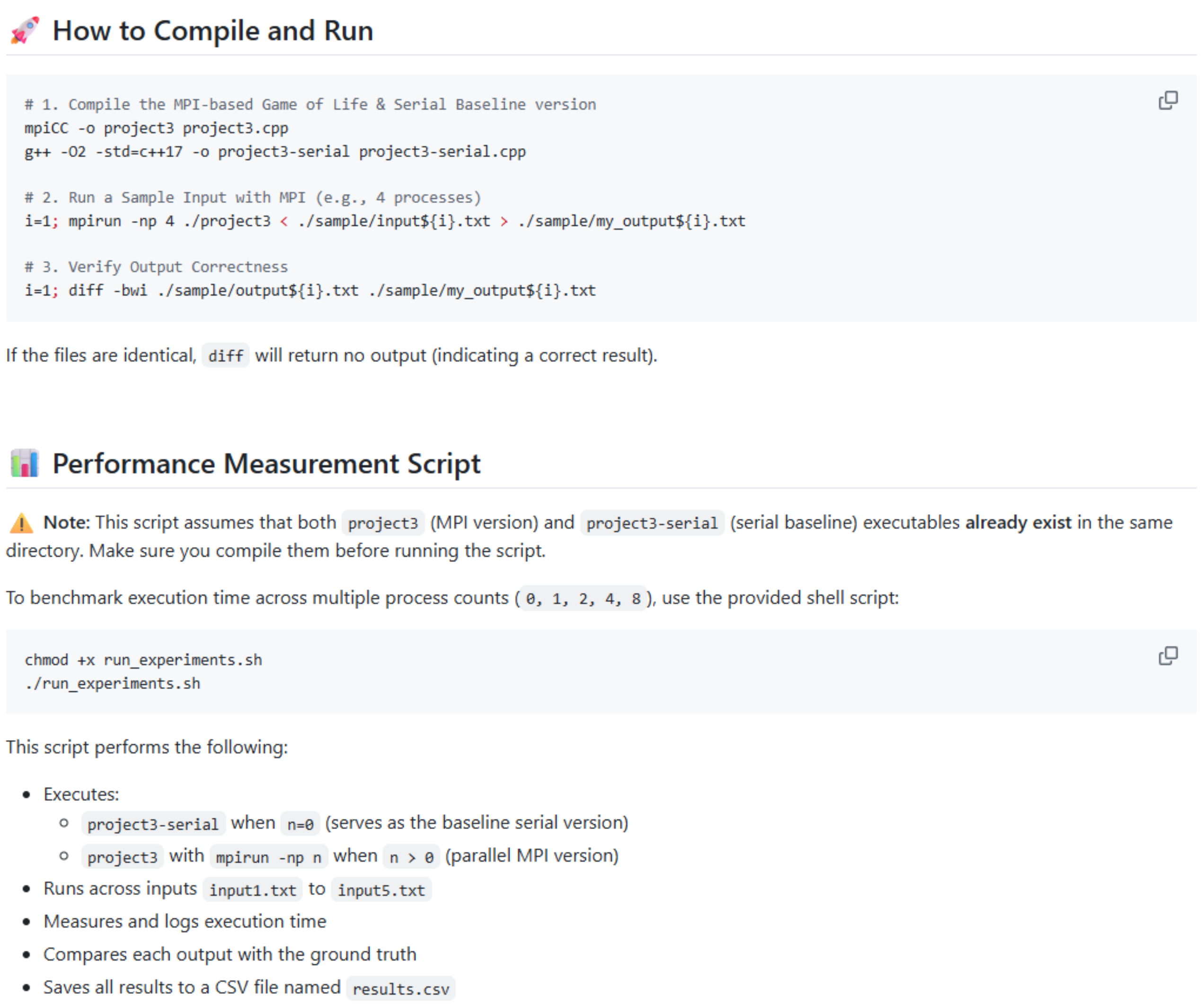
This project can be compiled and executed using the following commands:

```
mpiCC -o project3 project3.cpp
```

```
./run_experiments.sh
```

The **run_experiments.sh** script automates the execution of the program across all provided input files and records the corresponding results. For detailed instructions and configuration options, please refer to the **README.md** file included in the project directory.

Part of README.md



🚀 How to Compile and Run

```
# 1. Compile the MPI-based Game of Life & Serial Baseline version
mpiCC -o project3 project3.cpp
g++ -O2 -std=c++17 -o project3-serial project3-serial.cpp

# 2. Run a Sample Input with MPI (e.g., 4 processes)
i=1; mpirun -np 4 ./project3 < ./sample/input${i}.txt > ./sample/my_output${i}.txt

# 3. Verify Output Correctness
i=1; diff -bwi ./sample/output${i}.txt ./sample/my_output${i}.txt
```

If the files are identical, `diff` will return no output (indicating a correct result).

📊 Performance Measurement Script

⚠️ **Note:** This script assumes that both `project3` (MPI version) and `project3-serial` (serial baseline) executables **already exist** in the same directory. Make sure you compile them before running the script.

To benchmark execution time across multiple process counts (`0, 1, 2, 4, 8`), use the provided shell script:

```
chmod +x run_experiments.sh
./run_experiments.sh
```

This script performs the following:

- Executes:
  - `project3-serial` when `n=0` (serves as the baseline serial version)
  - `project3` with `mpirun -np n` when `n > 0` (parallel MPI version)
- Runs across inputs `input1.txt` to `input5.txt`
- Measures and logs execution time
- Compares each output with the ground truth
- Saves all results to a CSV file named `results.csv`