CMSC416: Assignment 2 Report

Krisnajit Rajeshkhanna

October 2024

1 Overview of Implementation

In this implementation, the input data representing live cell coordinates is initially read by the root process (rank 0) from an input file. Rank 0 is responsible for parsing these coordinates and determining how to distribute the rows among the available processes using a 1D row-wise decomposition. Each row is assigned to a process, ensuring an even distribution while accounting for any remainder. Once assigned, the coordinates of live cells are stored in buffers for the respective processes. Rank 0 then sends the live cell data to each process using `MPI_Send`, while each non-root process receives its allocated cells using `MPI_Recv`. This 1D decomposition allows each process to manage a distinct set of rows, reducing communication to exchanges with neighboring processes and optimizing workload balance.

The implementation uses MPI non-blocking communication to efficiently exchange boundary rows between neighboring processes. Non-blocking operations like `MPI_Isend` and `MPI_Irecv` allow processes to initiate data transfer while continuing with computation, overlapping communication with local processing to reduce waiting times. This approach ensures that the boundaries between adjacent processes are updated asynchronously, which is especially useful for maintaining efficient parallel execution. By overlapping communication and computation, the non-blocking method helps mitigate communication overhead, allowing for better scalability and improved parallel performance.

2 Performance Analysis

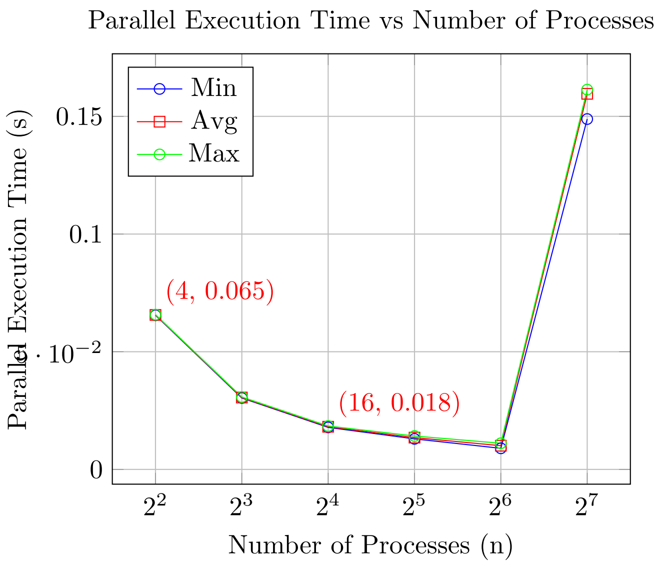

Figure 1: Parallel Times vs # Processes for 500 Generations over 512x512 Grid

The graph illustrates the parallel execution time versus the number of processes used, and the following are some key observations can be made from the plot:

1. **Reduction in Execution Time:** As the number of processes increases from 4 to 16, there is a noticeable reduction in the execution time from around 0.065 seconds to 0.018 seconds, with the minimum time occurring at 64 processes, around 0.01 seconds. This is expected, as increasing the number of processes allows for a better distribution of the workload across different nodes, thereby reducing the execution time. The 1D decomposition used to divide the rows effectively reduces the workload for each process and leads to faster processing, especially when there is a high enough number of rows to evenly split across all processes.

2. **Execution Time Spike At n = 128:** When the number of processes increases to 128, there is a significant jump in execution time. This behavior is due to communication overhead, as adding more processes, each process manages a smaller portion of the grid, which results in more frequent communication between neighboring processes, especially at the boundaries. The communication cost can outweigh the benefits of dividing the computation further, leading to increased execution time as more processes are involved. Additionally, at such high levels of parallelism, load imbalance or idle processes can also contribute to the rise in execution time.

3. **Min, Avg, and Max Execution Times:** The difference between the minimum, average, and maximum times is minimal throughout, indicating that the workload is fairly balanced. However, as the number of processes increases, the communication overhead—particularly the cost of boundary data exchanges—becomes more significant. This is particularly evident at 128 processes, where communication dominates the overall execution time, leading to inefficiency compared to the optimal 64-process scenario.

3 Conlusion

In summary, the findings highlight that there is an optimal number of processes for parallel efficiency, beyond which the gains diminish and the overheads start to dominate. The best performance was observed at 64 processes with time of 0.01 seconds, where workload distribution and communication overhead are well-balanced. With a further increase in the number of processes, the communication costs grow, leading to diminished performance gains or even increased execution times. This behavior is typical for distributed memory parallelization, where a balance between computation and communication is critical to achieving optimal performance.