PVL part IV - Game of Life from

By

Parsa Besharat

A handout submitted as part of the requirements

for the lecture, Introduction of High-Performance Computing, of MSc Mathematics of Data and Resources Sciences

at the Technische Universität Bergakademie Freiberg

January, 2024

Supervisor: Prof. Oliver Rheinbach

Abstract

The objective of this study is to evaluate the performance of a serial implementation of Conway's Game of Life under varying problem sizes. By simulating a dynamic environment where cells evolve based on their neighbors, the program demonstrates computational complexity as the grid size increases. The experiments, conducted on grid sizes ranging from 1,000 to 8,000, provide insight into the computational costs of solving such problems serially. The results establish a foundational benchmark for future parallelization using MPI, highlighting the inefficiencies and limitations of single-threaded approaches when handling large-scale data.

Contents

Introduction

Conway's Game of Life is a cellular automaton that models the evolution of cells on a grid based on predefined rules of life and death. This simulation is widely used in computational science to study complex systems and assess algorithmic efficiency. In the current implementation, a serial approach is employed to process grids of increasing sizes, measuring performance and computation times for a fixed number of iterations. While this method provides a straightforward solution, it lacks scalability, especially for larger problem domains. The findings from this study aim to establish a baseline for optimization efforts, paving the way for distributed computing strategies such as MPI to overcome the limitations observed in single-threaded environments.

Cluster Setup

Based on the TUBAF modules, these modules were required. These modules were loaded using the *module add* command to ensure compatibility and optimal performance during the execution of the codes:

1. `gcc/11.4.0`
2. `openmpi/gcc/11.4.0/5.0.3`
3. *`gdb/python/gcc/11.4.0/3.11.7/0.14.1`*
4. *`cmake/gcc/11.4.0/3.27.9`*

Adding Modules:

1. `module add gcc/11.4.0`
2. `module add openmpi/gcc/11.4.0/5.0.3`
3. `module add gdb/python/gcc/11.4.0/3.11.7/0.14.1`
4. `module add cmake/gcc/11.4.0/3.27.9`

Program Description

The program is a serial implementation of Conway's Game of Life, designed to simulate the evolution of a grid-based cellular automaton where each cell's state (alive or dead) is determined by its neighbors' states according to predefined rules. The program accepts three inputs: the grid size (N), the number of timesteps, and the initial probability of a cell being alive. It initializes the grid with random states, updates the grid iteratively for the specified timesteps, and calculates alive neighbors for each cell to determine its next state. The program uses padding to handle boundary conditions and outputs the grid state for small sizes, along with the total computation time for performance evaluation. This implementation establishes a baseline for understanding the computational cost of the task in a single-threaded environment.

Part A

Running the code

In the C++ code, we have to implement in two ways:

a. `g++ -std=c++11 -O3 game_of_life.cpp -o game_of_life.` For compiling.

b. `./game_of_life <N> 10 0.5.` For executing with number of N such as 1000

Output

Problem Size	Timestamps	Probability	Computation Time (seconds)
1000	10	0.5	0.0556391
2000	10	0.5	0.221043
4000	10	0.5	0.882055
8000	10	0.5	4.54432

Output Analysis

The results from Part A demonstrate a clear relationship between the problem size N and the computation time, highlighting the quadratic nature of the algorithm due to the two-dimensional grid structure of the Game of Life. As the grid size increases, the total number of cells, and thus the computational workload, grows proportionally to N^2. This is evident from the observed execution times, where doubling (N) approximately quadruples the computation time. For instance, the jump from N=1000 to N=2000 results in a near 4x increase in execution time, going from 0.0556 seconds to 0.221 seconds. Similarly, further increments in (N) maintain this scaling pattern, though the slight deviations can be attributed to system overheads or memory access inefficiencies in handling larger datasets. These results underline the limitations of a single-threaded, serial implementation, especially as the problem size grows. With larger grids like N=8000, the computation time surpasses 4.5 seconds even for a modest 10 timesteps, indicating that the algorithm would struggle to handle significantly larger grids or higher timestep counts without optimizations. This makes the case for parallelization particularly compelling, as distributing the workload across multiple processors could reduce individual computation loads and dramatically improve execution times. These findings emphasize the necessity of transitioning to a parallel implementation for practical scalability, especially when working with larger grids in real-world scenarios.

Part B

Running the code

In the C++ code, we have to implement in two ways:

a. `mpic++ -std=c++11 -O3 game_of_life_mpi.cpp -o game_of_life_mpi .` *Compile it with MPI.*

b. `mpirun -np <num_processes> ./game_of_life_mpi <N> <timesteps> <probability>.` *Running it with MPI.*

Program description

The program implemented in Part B focuses on parallelizing the Game of Life simulation using MPI (Message Passing Interface) to enhance performance for large problem sizes. The simulation divides the board into rows, which are distributed evenly among multiple MPI processes. Each process handles a subset of the rows, including additional "halo" rows that store boundary information needed for communication with neighboring processes. The program employs MPI functions such as MPI_Sendrecv to exchange these halo rows, ensuring that each process has the necessary data to compute the next timestep accurately. The program begins by initializing the board with a random distribution of alive and dead cells, followed by iteratively computing timesteps based on Conway's Game of Life rules. Processes update their assigned rows and exchange halo data with neighboring processes to maintain consistency across the board. The program includes both strong and weak scaling capabilities, allowing users to evaluate its performance by either fixing the problem size and varying the number of processes or increasing the problem size proportionally with the processes to maintain a constant workload per process. Despite the parallelization, the results highlight challenges with communication overhead and inefficiencies in scaling, particularly as the number of processes increases. The program emphasizes the importance of synchronization and data consistency in parallel computations while demonstrating the potential for performance improvements through optimized MPI communication and better workload distribution.

Output

Problem Size	Timestamps	Probability	Computation Time (seconds)	Serial Time (seconds)	Speedup Factor (Serial/Parallel)
1000	10	0.5	0.0556391	0.0556	2.89
2000	10	0.5	0.221043	0.221	3.82
4000	10	0.5	0.882055	0.882	3.89
8000	10	0.5	4.54432	4.544	4.88

Output Analysis

The results from the MPI implementation reveal a substantial reduction in computation time compared to the serial version, with speedup factors ranging from ~2.89 for (N = 1000) to ~ 4.88 for (N = 8000). The parallel implementation's effectiveness increases with the problem size, as seen in the growing speedup factors. This is expected because the workload per process grows with larger grids, making the overhead of halo communication relatively smaller compared to the total computation. For (N = 1000), the communication overhead is more significant, slightly dampening the parallel performance. However, as the grid size increases, the program benefits more from dividing the workload among the 4 MPI processes. The time improvements demonstrate that the program scales well and efficiently utilize the 4 processes, with the parallel implementation achieving nearly 5x speedup for the largest grid size tested (N = 8000). This indicates that the MPI implementation is well-suited for handling larger datasets, where the computational cost dominates over communication overhead.

Part C

Strong Scaling

Processes	Problem size	Timestamps	Computation Time (seconds)	Speed Up	Efficiency (%)
1	8000	10	4.48118	1	100%
2	8000	10	4.59149	0.97	48%
4	8000	10	4.5384	0.98	24%
8	8000	10	4.6798	0.95	11%

The strong scaling results reveal that the parallel implementation does not scale efficiently as the number of processes increases. Ideally, computation time should decrease significantly with more processes. However, in this case, the computation time remains nearly constant (~4.5 seconds) regardless of the number of processes, resulting in low speed up and efficiency values. For 2 processes, the speedup is only 0.97, with efficiency at 48%, which drops further to 0.95 and 11% efficiency for 8 processes. This suggests that the parallel implementation faces significant

communication overhead or poor load balancing, which negates the benefits of adding more processes. The lack of improvement in computation time indicates that either the problem size is too small relative to the number of processes or that the halo communication between processes is inefficient.

Weak Scaling

Processes	Problem size	Workload per process	Computation time (seconds)
1	8000	1000	0.0629419
2	8000	1000	0.254744
4	8000	1000	1.06473
8	8000	1000	4.74223

The weak scaling results indicate poor scalability as the number of processes increases. In a well-optimized weak scaling scenario, the computation time should remain constant because the workload per process remains fixed at 1000 cells. However, the computation time increases significantly as the number of processes increases: from 0.063 seconds for 1 process to 4.742 seconds for 8 processes. This suggests that the parallel implementation is heavily impacted by communication overhead, as the halo exchange between processes increases with more processes. The results indicate that the parallel efficiency of the implementation deteriorates rapidly in weak scaling, likely due to the increased cost of synchronizing boundary rows.

Conclusion

This project showcases the use of parallel computing to optimize the Game of Life simulation. Part A highlights the increasing computational demand with larger grid sizes in the serial implementation, emphasizing the need for parallelization. Part B introduces an MPI-based solution, distributing the workload among processes and using halo communication for synchronization. Part C evaluates the performance through strong and weak scaling, revealing limitations in efficiency due to communication overhead and load balancing.