Distributed System for AI
Parallel version of Game of life (OpenMP)
Assignment 2

Mohammed Sraj [GSR/5161/16]

November 23, 2024

1 Introduction

This report explores the parallelization of the Game of Life, a popular cellular automaton. The assignment begins with a provided serial implementation, and the primary objective is to transform this code into a parallel version. The focus is on enhancing both graphical user interface (GUI) management and data processing using parallel programming techniques such as **OpenMP** or POSIX threads **Pthreads**.

2 Experiment Setup

Hardware Specifications

System Specification	**System 1: Dell**	**System 2: Precision 7920 Tower**
Operating System	Ubuntu 22.04	Ubuntu 22.04
Computer Model	Dell	Precision 7920 Tower
Processor	Intel Core i5	Intel Xeon
Physical Cores	2	26
Logical CPUs (Threads)	4 (2 threads per core)	52 (2 thread per core)
RAM	16GB	64GB
GPU	None	NVIDIA RTX 4000 (approx. 6000 cores)

Table 1: System Specifications

Software Specifications

The program was compiled using **GCC version 11.4.0**,and uses **gnuplot version 5.4** ,**matplotlib** and **pandas** for visualization

3 Results

In order to measure the performance of both the serial and parallel implementations of the Game of Life, I wrote a shell script that runs the Game of Life program using different parameters for both the serial and parallel versions, with varying numbers of threads. To achieve parallelism, I used OpenMP. The shell script runs the Game of Life for each iteration five times and calculates the average execution time of the output.

The reason for choosing a shell script was to separate the main logic of the Game of Life from the performance analysis part. This separation ensures that the performance evaluation does not interfere with the core functionality of the program and allows for greater flexibility in testing and profiling. The script automates the execution of the Game of Life with various configurations, enabling efficient performance measurement without manual intervention.

The shell script allows for running both serial and parallel versions of the Game of Life under different conditions and provides a consistent method of collecting performance metrics for each configuration.

Task Parallelism

In the task parallelism part, the section responsible for displaying the output using Gnuplot is parallelized. Both the calculation of each cell's state and the display part run serially for each iteration. However, the display part takes significantly more time compared to the calculations. This discrepancy becomes evident when the program is run without the GUI, as shown in the results below.

The program is divided into two sections using OpenMP. One section handles the GUI, while the other handles the data calculation for the state of each cell, determining whether it is alive or dead based on the rules of the Game of Life. The section responsible for calculating the state of each cell uses data parallelism via OpenMP, as shown in the following code:

```
#pragma omp sections
{
    #pragma omp section
    {
        // GUI handling code here (e.g., display results)
    }

    #pragma omp section
    {
        // Data parallelism: Calculate the state of each cell
        // based on the Game of Life rules
        #pragma omp parallel for
        for (int i = 0; i < num_rows; i++) {
            for (int j = 0; j < num_columns; j++) {
                // Compute next state of cell (alive or dead)
                // based on surrounding cells' states
                }
            }
        }

}
```

Here, the #pragma omp section directive divides the task into two parts: one part handles the GUI, and the other performs the data calculation. Inside the data calculation section, the #pragma omp parallel for directive is used to parallelize the loop that calculates the state of each cell in the Game of Life grid. This division of work allows for improved performance by handling the computationally intensive part in parallel while leaving the GUI section to run serially.

Data Parallelism

In the data parallelization part, the portion of the code responsible for calculating and updating the state of each cell can be parallelized. This is because the two "worlds" are independent of each other, meaning that the current world is different from the next world. Therefore, the iterations can be parallelized using OpenMP's `#pragma omp parallel for collapse(2)` construct.

By collapsing the two nested loops into one big loop (using `collapse(2)`), each thread can handle a specific portion of the entire array. To avoid race conditions during updates, we use `reduction(+ :  population[w_update])`. This ensures that each thread has its own private copy of the `population[w_update]` variable and aggregates the results at the end of the computation.

This approach prevents data races between threads, thereby enabling efficient parallelization of the data processing.

```
#pragma omp parallel for collapse(2)
            for (int i = 0; i < num_rows; i++) {
                for (int j = 0; j < num_columns; j++) {
                    // Compute next state of cell (alive or dead)
                    // based on surrounding cells' states
                }
            }
```

Testing

The testing mechanism I used involved running both the serial and parallel programs with OpenMP and tracking their performance. To facilitate this, I created a separate Python script for visualization. As described above, my shell script executes the serial and parallel programs using different iterations and thread configurations, and it saves the execution time in a CSV file. This process was repeated on both System 1 and System 2, whose specifications are outlined above. The resulting data is as follows:

The testing process focused on evaluating performance based on the number of iterations for the serial program and the number of iterations and threads for the parallel part. The results demonstrate that, in the serial part of the code, execution time increases as the number of iterations increases. However, in the parallelized version, as depicted in the graph, the execution time decreases with an increase in the number of threads.

To ensure realistic and accurate results, each configuration of iterations and threads was executed 5 times, and the average execution time was calculated. This repeated testing helps mitigate anomalies and provides a more reliable measure of performance.

The experiment was conducted using a `-n 500` (or a 500×500 grid), `-p 0.2` (probability of 0.2) for initialization, and the `-d` flag (disabling GUI display). In other words, the results reflect the performance of the data parallelization approach and using system 2.

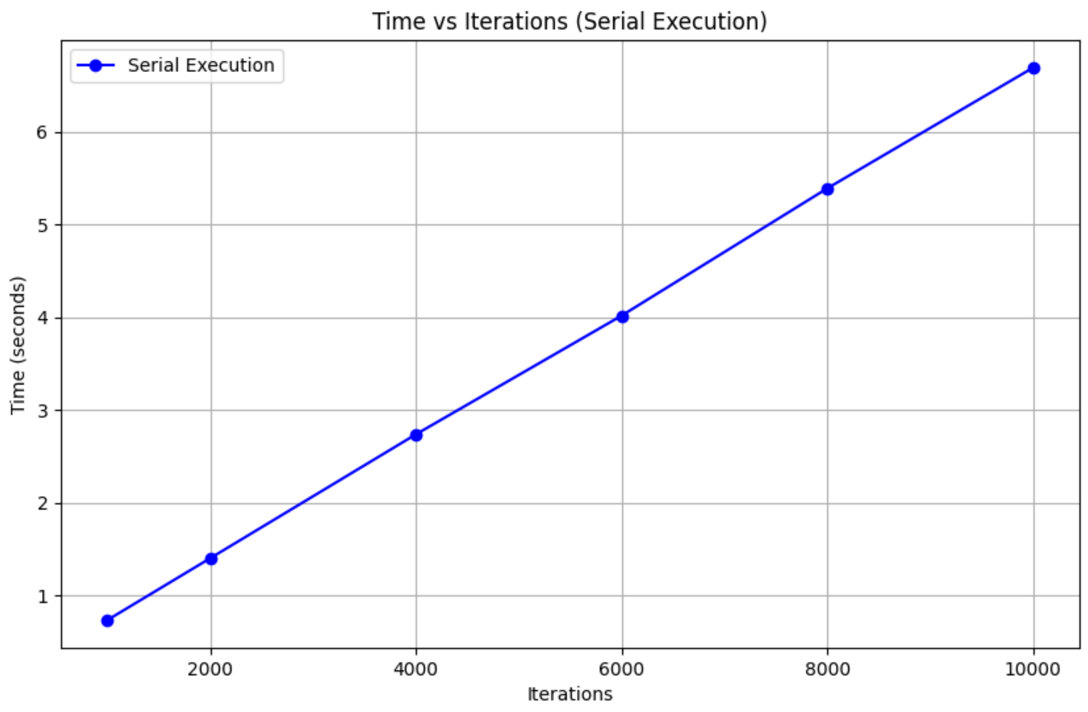

Figure 1: Serial code.

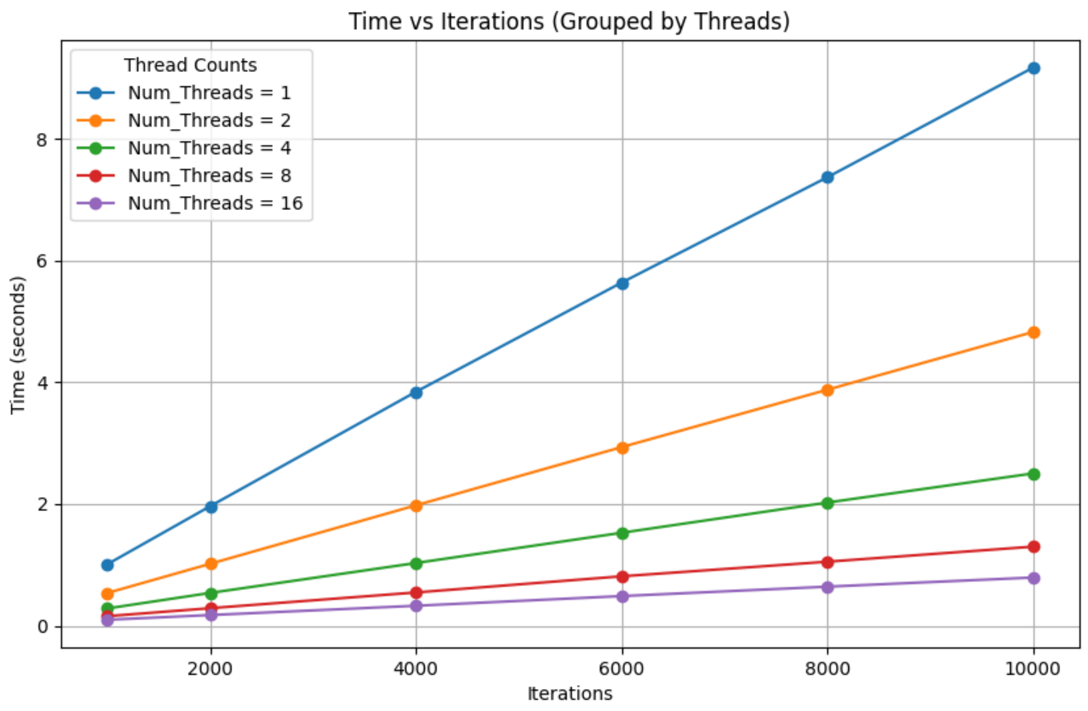

Figure 2: parallel code

4 Conclusions

The conclusion of this assignment and experiment emphasizes the importance of parallelization techniques and the corresponding libraries, such as `pthread` and `OpenMP`. One of the challenges I faced was analyzing the Gnuplot C library for visualization. My goal was to use it in this assignment rather than adding another dependency (such as Python) to the codebase, as I wanted to keep the project entirely C-based. Unfortunately, since I am not well-acquainted with this library, I was unable to produce an elegant graph. This is an area that needs improvement in future work.

Another important takeaway is the need for a deep understanding of these parallelization libraries, which is crucial for AI practitioners. A comprehensive knowledge of these libraries is essential for optimizing code performance, as I believe parallelization is a key factor in improving computational efficiency.

Additionally, I learned the importance of C and C++ build systems. Understanding and mastering these build systems will be an area I focus on in the future to further improve my development workflow.