School of Information Technology and Engineering

Master of Science in Artificial Intelligence

DAI Assignment 2

Game of life

Name:- Ruth Tamiru

ID:- GSR 7345/16

Submitted on November 2024

Report on Parallel Implementation of the Game of Life using OpenMP

1. Introduction

The Game of Life is a cellular automaton created by mathematician John Conway, where cells on a grid evolve over generations based on simple rules. Each cell in the grid can be either "alive" or "dead", and its state in the next generation depends on the number of living neighbors it has. The rules for the evolution of the game are as follows:

- Any live cell with fewer than two live neighbors dies, as if caused by under-population.
- Any live cell with two or three live neighbors continues to live to the next generation.
- Any live cell with more than three live neighbors dies, as if by overcrowding.
- Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

This assignment requires implementing a parallel version of the Game of Life using OpenMP and Pthreads, improving the performance of the serial Game of Life implementation by introducing parallelism at the task and data levels. The goal is to run the simulation concurrently, with separate threads handling different aspects of the simulation, such as updating the cell states and plotting the game state.

2. Environment Setup

The parallel implementation was developed and tested on an **HP 250 G9 notebook**, equipped with a **12th-generation Intel Core i5 processor**, **8 GB of RAM**, and **1TB SSD storage**. The development and execution of the program were done in a **Windows Subsystem for Linux (WSL)** environment, which provides a Linux-compatible kernel and environment for running applications on a Windows machine. This setup allowed the program to be executed with all required dependencies, including the `gnuplot` tool, while utilizing the power of the Visual studio code's hardware for parallel computation.

3. Task and Data Parallelism

3.1 Serial Code Overview

The provided serial code for the Game of Life creates a random grid of cells and evolves it for a specified number of generations. Each cell is evaluated based on the number of living neighbors, and the grid is updated accordingly for each generation. The grid is visualized using the external `gnuplot` tool, which is invoked after each generation to plot the current state of the grid.

In my case this serial code with maximum number of iteration 200, and 500 grid size and the probability of live cells to exist initially becoming 0.2 results in 16.291143 second of execution.

```
ruth@DESKTOP-25APJ6I:~/lifesm$ ./life -n 500 -i 200 -p 0.2
probability: 0.200000
Random # generator seed: 1733083211
MESA: error: ZINK: failed to choose pdev
glx: failed to create drisw screen

Warning: slow font initializationqt_processTermEvent received a GE_fontprops event. This should not have
 happened
Running time for the iterations: 16.291143 sec.
Press enter to end.
```

3.2 Parallelism Overview

The parallel version of the Game of Life introduces two primary forms of parallelism:

3.2.1 Data Parallelism

Data parallelism is introduced by splitting the computational work of updating the cells across multiple threads. The grid is divided into slices, each of which is processed by a different thread. This division of labor allows the simulation to scale with the number of threads, improving performance. The division of the grid is done on a row-by-row basis, where each thread processes a segment of the grid. If the number of threads does not divide the grid size evenly, the workload is distributed as evenly as possible.

I personally implement the data parallelization in the update of the grid and I have seen that adding the number of threads for iterations less than 3000 doesn't have a significant impact but the concept of the parallelism helps as we go far from this point. As an experiment for iteration = 200 both the serial and parallel almost have the computational time and as I go deeper to increase the number of iteration to 200000 the computational time for the serial exceeds 1 hour but for the parallel one it gives me around 35 minutes to compute that.

In my code i use this openmp command to parallelize it [`#pragma omp parallel for collapse(2) reduction(+:population[w_update])`

]

3.2.2 Task Parallelism

Task parallelism is introduced by separating the tasks of updating the grid and plotting the grid. One thread (the "plotter thread") is responsible for displaying the grid using `gnuplot`, while another thread is responsible for updating the cell states based on the rules. These tasks run concurrently, synchronized using mutexes or barriers to ensure data consistency.

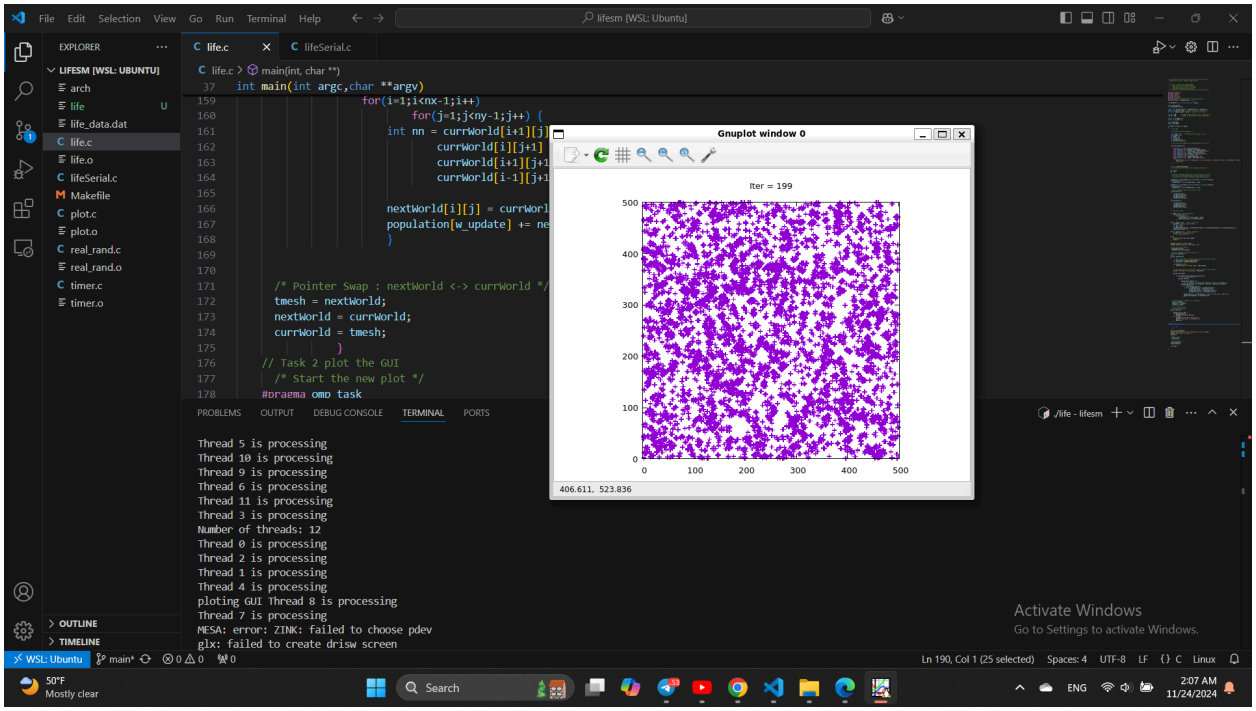
After I implemented the data parallelization the next thing was to go through the task parallelism even though i will discuss the challenges in later sections my findings were being able to use the library to parallelize 2 tasks which are (1. Updating the grid and 2. Ploting the GUI)

```
        #pragma omp single

                #pragma omp task

    #pragma omp taskwait
```

The above commands were the basics for implementing the task parallelization while omp task gives 2 different tasks the taskwait function will help to synchronize and avoid race conditioning.



3.3 Synchronization and Shared Memory

To handle concurrency between threads, shared memory is used for the game grid. Synchronization mechanisms such as barriers or mutexes are implemented to ensure that the plotter thread does not access the grid while it is being updated. The shared memory model ensures that all threads work on the same version of the game grid, and updates are synchronized to prevent conflicts.

Despite these efforts my program was not able to synchronize the plotting with updating the grid.

4. Parallel Implementation Details

4.1 OpenMP Implementation

OpenMP is used to introduce parallelism at the data level by dividing the grid into smaller segments for concurrent processing. The steps involved in the OpenMP implementation are:

1. **Grid Initialization:** The master thread initializes the game grid.
2. **Parallel Update:** The grid is divided into horizontal slices, and each slice is assigned to a separate thread. Each thread computes the new state of its slice based on the neighboring cells.
3. **Plotting:** The plotter thread handles the visualization of the grid using `gnuplot`. Synchronization mechanisms such as barriers are used to ensure that the plotter thread waits until the grid update is complete.
4. **Synchronization:** The use of OpenMP's `#pragma` directives ensures that the work is properly divided among threads. A barrier is used to synchronize the threads after the grid update to allow the plotter thread to display the grid.

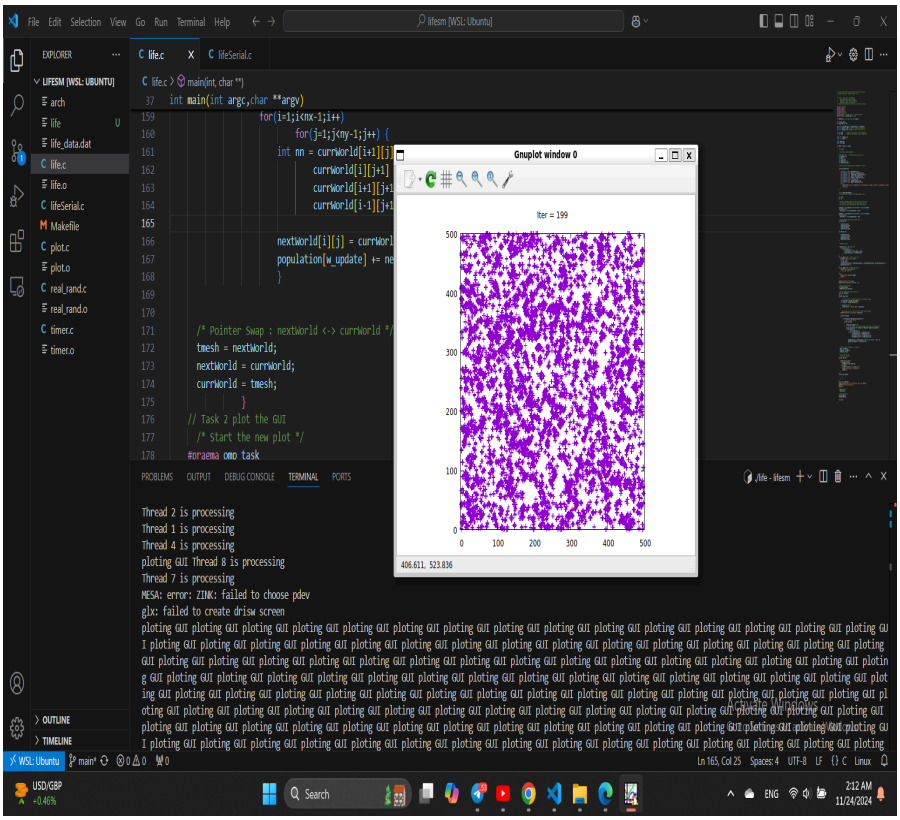
4.2 Handling Non-divisible Grid Sizes

In both OpenMP and Pthreads implementations, if the number of threads does not divide the grid size evenly, the workload is distributed as evenly as possible by assigning any remaining rows to the last thread. This ensures that the computational workload is balanced.

5. Challenges and Bottlenecks

5.1 Synchronization Issues

One of the challenges encountered during the parallelization was ensuring that the plotter thread did not attempt to plot the grid while it was being updated. This required careful use of synchronization mechanisms such as barriers and mutexes to ensure that the grid was fully updated before plotting.Here the plotting comes after the update as shown in the figure below.



5.2 Performance Overheads

While parallelizing the computation significantly improved the performance of the cell updates, the overhead of synchronization and the external plotting process created some performance bottlenecks. The time spent on plotting can be significant, especially when running for many generations.

6. Conclusion

This assignment provided valuable experience in parallelizing a computational problem using both OpenMP. By introducing task parallelism for the plotting and data parallelism for the cell updates, I was able to improve the performance of the Game of Life simulation. The OpenMP implementation was simpler to implement.

The parallel version of the Game of Life was successfully validated against known test cases and showed improvements in performance, particularly when run on multiple threads. However, performance bottlenecks were observed due to the I/O overhead of the plotting process, which is a significant factor in the overall runtime of the program.