**Benjamin Farias Dela Mora**

**Washington State University Tri-Cities**

**October 21, 2024**

Homework 3

CPT\_S 411

**Table of Contents:**

**1 Introduction:**

The goal of this project is to implement a parallel solution using OpenMP to simulate Conway’s Game of Life, demonstrating an understanding of parallel programming concepts.

1. Purpose:

The purpose of this document is to provide a method to solving Conway’s Game of Life using parallel programming.

1.1 Constraints:

Now that I have some experience working in parallel, I can say that this shouldn’t be too difficult given the rules for Conway’s Game of Life seem relatively simple, however separating the data into multiple processes and having them deal with the calculation of neighbors and updating the cells could potentially be tricky especially since I do not have as much experience with OpenMP in comparison to MPI.

**2 Design:**

2.0 Design overview:

2.1 Design process:

I reviewed some of the slides and chapters for how the OpenMP functions work. I also brainstormed how the functions for calculating live neighbors and initializing the grids would work separate from the main functionality of the program.

**3**

**3.1 Analysis Formulas:**

i) A living cell that has less than 3 living neighbors dies (underpopulation).  
ii) A living cell with more than 5 living neighbors also dies (overpopulation).  
iii) A living cell with between 3 and 5 living neighbors live on to the next generation;  
iv) A dead cell will come (back) to life if it has between 3 and 5 living neighbors. (reproduction).

**3.2 Tables:**

**4 Analysis of learning:**

In this programming assignment we got the opportunity to learn how to apply parallel programming concepts and techniques to solve Conway’s Game of Life.

**5 Conclusion:**

Creating the code to solve Conway’s Game of Life in a parallel manner was relatively simple until it wasn’t. The helper functions I created such as initialize, printGrid and calculateLiveNeighbors I was able to think of in a serial manner, whereas the main functionality for updating the state in each process and then returning the updated state was more challenging. This was more interesting of a homework to do in comparison to HW2 and I would like to see more often.

Appendix:

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <time.h>**

**#include <omp.h>**

**#define MAX\_N 1000 // Define a reasonable maximum size for n**

**// Initialize the grid with randomized 0 or 1's**

**void initialize(int grid[MAX\_N][MAX\_N], int n)**

**{**

**for (int i = 0; i < n; i++)**

**{**

**for (int j = 0; j < n; j++)**

**{**

**grid[i][j] = rand() % 2;**

**}**

**}**

**}**

**// Print the grid**

**void printGrid(int grid[MAX\_N][MAX\_N], int n)**

**{**

**for (int i = 0; i < n; i++)**

**{**

**for (int j = 0; j < n; j++)**

**{**

**printf("%d ", grid[i][j]);**

**}**

**printf("\n");**

**}**

**printf("\n");**

**}**

**// Calculate live neighbors of a cell**

**int calculateLiveNeighbors(int grid[MAX\_N][MAX\_N], int x, int y, int n)**

**{**

**int liveNeighbors = 0;**

**// Check the 8 neighbors of the cell (up, down, left, right, and diagonals)**

**for (int i = -1; i <= 1; i++)**

**{**

**for (int j = -1; j <= 1; j++)**

**{**

**int newX = x + i;**

**int newY = y + j;**

**if (newX >= 0 && newX < n && newY >= 0 && newY < n)**

**{**

**liveNeighbors += grid[newX][newY];**

**}**

**}**

**}**

**// Subtract cell itself (don't include the cell itself in the amount of live neighbors)**

**return liveNeighbors - grid[x][y];**

**}**

**int main(int argc, char \*argv[])**

**{**

**if (argc != 4) {**

**printf("Usage: %s <n> <timesteps> <thread\_num>\n", argv[0]);**

**return 1;**

**}**

**int n = atoi(argv[1]);**

**int t = atoi(argv[2]);**

**int num\_threads = atoi(argv[3]);**

**if (n <= num\_threads || n % num\_threads != 0) {**

**printf("Error: n must be greater than the number of threads and divisible by the number of threads.\n");**

**printf("Example: srun ./golOMP 8 4 4.\n");**

**return 1;**

**}**

**int grid[MAX\_N][MAX\_N];**

**int nextGrid[MAX\_N][MAX\_N];**

**// Initialize the grid**

**srand(time(NULL));**

**initialize(grid, n);**

**omp\_set\_num\_threads(num\_threads);**

**double startTime = omp\_get\_wtime();**

**for (int step = 0; step < t; step++)**

**{**

**#pragma omp parallel**

**{**

**// Parallelize the computation of the next state**

**#pragma omp for**

**for (int i = 0; i < n; i++)**

**{**

**for (int j = 0; j < n; j++)**

**{**

**int liveNeighbors = calculateLiveNeighbors(grid, i, j, n);**

**if (grid[i][j] == 1)**

**{**

**// If cell is alive check if it has less than 2 or more than 3 live neighbors and update its state accordingly**

**nextGrid[i][j] = (liveNeighbors < 3 || liveNeighbors > 5) ? 0 : 1;**

**}**

**else**

**{**

**// If cell is dead check if it has 3 neighbors and update its state accordingly**

**nextGrid[i][j] = (liveNeighbors >= 3 && liveNeighbors <= 5) ? 1 : 0;**

**}**

**}**

**}**

**// Copy the nextGrid to grid**

**#pragma omp for**

**for (int i = 0; i < n; i++)**

**{**

**for (int j = 0; j < n; j++)**

**{**

**grid[i][j] = nextGrid[i][j];**

**}**

**}**

**}**

**}**

**double endTime = omp\_get\_wtime();**

**printf("Final Grid:\n");**

**printGrid(grid, n);**

**printf("total time taken: %f \n", endTime - startTime);**

**return 0;**

**}**