Conway's Game of Life in 3D Space

Gourav Pullela
ENGR-E 517 Project Report

Introduction:

A cellular automaton is a group of "colored" cells in a grid of a specified shape that evolves through a discrete number of time steps according to a set of rules based on the states of the neighboring cells. Stanislaw Ulam and John von Neumann invented the first cellular automata when they devised a method to calculate liquid motion in the 1950s. Cellular automata were initially used to study and model artificial life, but as technology progressed and newer methods such as machine learning were introduced, they became less popular. However, despite advancements, due to the ease of scalability and parallelization, they are still used to model and study artificial life.

Conway's Game of Life, or simply Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970. The Game of Life is a zero-player game and is determined by its initial state, which evolves based on the rules of the game. The game is Turing complete and can simulate a universal conductor or any other Turing machine due to its virtue of self-replication. In this game, cells have two states - dead or alive and depending on their own state and the states of surrounding neighbor cells, the state of a cell is determined, for the next cycle.

The rules of Conway's Game of Life are fairly simple and can be condensed into the following:

1. Any live cell with two or three live neighbors lives in the next cycle.
2. Any dead cell with 3 live neighbors, becomes a live cell in the next cycle.
3. All other live cells die in the next cycle.

These simple rules are what govern the state of the game of Life, and can be used and modified to produce subtle or drastic changes in patterns and states that can be observed in the automata.

Aim:

The aim of this project was to implement a simulation of Conway's Game of Life in three-dimensional space, as opposed to the traditional two-dimensional form that is widely known. The reasoning behind this was to study how the behavior of the environment would change with the introduction of depth, which increases the combination of patterns and also fundamentally changes the way the game is simulated.

Adaptations:

The game and its rules had to be modified so that the essential nature of the game would remain constant, barring the addition of an extra dimension. This came in the form of rule modifications; specifically a change in the conditions for a cell to stay alive or die. As another dimension was added, the number of neighboring cells for each cell increased from a maximum of 8, in two-dimensions, to 26 cells, in three-dimensions. This allowed for a larger number of simulations and unlike most scenarios, where one would run the simulation of the game for a certain number of time steps, or cycles, I decided to run simulations until the entire population of the environment reached zero, or essentially, became extinct.

Some other modifications included the population of the three-dimensional environment, which is an 8000 cell environment, with an edge of 20 cells, that is populated with cells at random so that the initial population starts at approximately 33% of the maximum population.

The modified rules for the three-dimensional adapted game of Life were based on the original rules but were slightly modified while maintaining the same ratios as the original game:

1. Any live cell with more than 6 and less than 11 live neighboring cells lives in the next cycle.
2. Any dead cell with more than 7 and less than 12 live neighboring cells, becomes a live cell in the next cycle.
3. Each cell can have a maximum of 26 neighboring cells, which can be horizontally, vertically, or diagonally adjacent to that cell.

Including these rules, the last rule of the original game was also used, where all other live cells that do not fit these criteria die in the next cycle.

Implementation:

The OpenMP API was used to implement the game and the program was written in C. The reasoning behind this was that OpenMP is an API that I am familiar with and comfortable with, which gave me the room to implement the game in two dimensions and then expand on it while adapting to new conditions, including the seeding of the population, and introduction of the modified rules.

Testing Environment:

The Big Red 3 cluster at Indiana University Bloomington was used. The simulations ranged from 8 core parallel to 24 core parallel and were run on a single node.

Outlining the test case:

The test case had a fixed problem of 20^3 (8000 cells total). The initial population of the 8000 cell environment is done at random, using the srand() function to seed the rand() function using the system's local time. The simulation is then run until the total population in the environment reaches zero. Some data measured during the simulation include the peak population, the number of cycles till extinction, the runtime of the simulation, and the average runtime per cycle.

Understanding the program:

The code for the three-dimensional implementation can be found in Appendix A, but we will unpack some important parts of the program.

```
//Returns number of living neighbors
int numAlive(struct Cell game[N][N][N], struct Cell point, struct Vector* n) {

   int i,alive = 0;
   #pragma omp parallel for
      for(i=0; i < 26; i++) {
         if((-1 < n[i].x && n[i].x < N) && (-1 < n[i].y && n[i].y < N) && (-1 < n[i].z && n[i].z < N)) {
            if(game[n[i].x][n[i].y][n[i].z].doa == 1) {
               alive += 1;
            }
         }
      }

   return alive;
}
```

The function depicted above is used to find the number of live neighboring cells to a given cell, with its input parameters being the three-dimensional array or the cell environment, the cell itself, and a Vector pointer, pointing to an array of Vector coordinates for any neighboring cells. The program would then use OpenMP parallelization to check the validity of coordinates, and as to whether they existed within the bounds of the environment, and then computed a sum of the total number of live neighboring cells to the input cell.

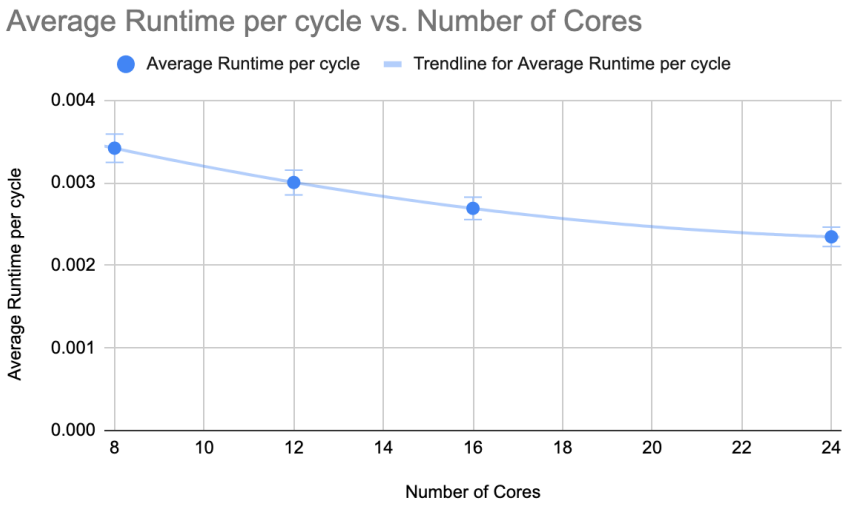
The following code snippet is the most crucial part of the game and the program and contains the code that is responsible for the simulation of the environment.

```
//Simulation
while(count > 0) {
   #pragma omp parallel for collapse(3)
      for(i=0; i < N; i++) {
         for(j=0; j < N; j++) {
            for(k=0; k < N; k++) {
               input = neighbors(game[i][j][k]);
               alive = numAlive(game, game[i][j][k], input);
               if(game[i][j][k].doa == 1 && (alive < 6 || alive > 11)) {
                  game[i][j][k].doa = 0;
                  count -= 1;
               }
               if(game[i][j][k].doa == 0 && (alive > 7 && alive < 12)) {
                  game[i][j][k].doa = 1;
                  count += 1;
               }
               if(count > maxcount) {
                  maxcount = count;
               }
            }
         }
      }
   steps += 1;
}
```

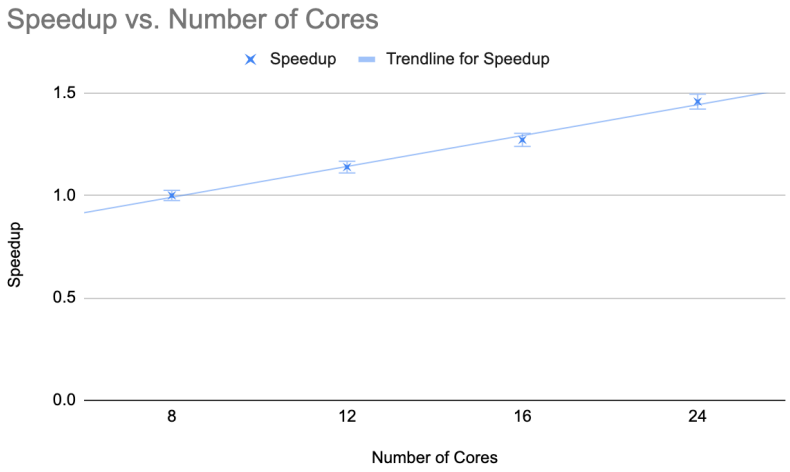
The while loop surrounding the collapsed parallel for loops is what is responsible for the simulation's continuation until the total population of the environment, or the total number of cells, becomes zero. The modified rules are applied within the parallel block, and the previous function, as well as another one, are used in tandem to compute the number of live neighboring cells so that the rules can be applied to each cell in parallel and the cell's state can be modified for the next cycle or timestep. The total number of steps is measured here and the maximum population is also recorded and changed if there is an increase. After the simulation is complete, the time taken to run the simulation is recorded, and the average run time per cycle/step is calculated, which is used to evaluate the performance of the program when run with strong scaling.

Data, Analysis, and Findings:

The following section includes data and analysis of findings from the simulation of the three-dimensional model of Conway's Game of Life in parallel, run on 8, 12, 16, and 24 cores, as a strong scaling problem, due to the fixed problem set.



The graph depicted above is a strong scaling plot of the average run time per cycle, against the number of cores used in parallel. There is a very marginal increase in performance when the number of cores is increased, which shows that the program is not entirely optimized, and also that it to some extent is connected to Amdahl's law, and thus there is not a linear relationship between the increasing number of processing cores and the time taken to run each cycle.



The graph depicted on the previous page is a plot of the relative speedup against the number of cores used to run simulations. The maximum speedup is 1.5x when the number of

cores is increased by a factor of 3, which indicates that the speedup, in this case, was only half of the factor used to increase the number of cores. This also shows that Amdahl's law is in effect and that there is an amount of latency that affects the performance of the program, and effectively causing a scalable amount of performance degradation.

The initial population of the environment hovered between 32 and 34%, with an average of 2641 cells at initial seeding. The peak population that the simulations reached was approximately 50%, not going past 4100 cells, which is approximately 51.25% of the total population of the environment. The number of cycles till extinction varied each time the simulation was run, due to the randomness of the initialization and population of the environment, which in turn played their respective roles in randomizing the time it took for the population to go extinct.

One issue that I faced when setting up and running simulations with my test case was that the system did not have enough memory for the test case when using less than 8 cores, even when the problem size was reduced to 10^3 or 1000 cells. The reason for this, as I later realized was the absence of memory allocation or the malloc() function in the initialization of the three-dimensional array, which in turn led to memory over-usage when the number of cores reduced, which eventually led to memory errors. This is something that caused me lots of problems when working on the program and is attributed to my inexperience in parallel programming and also with this specific field, cellular automata.

Future Implementation:

Some aspects of the problem that I was not able to study, included population density, the average life span of cells, and the rates of growth and decay in the environment. I also realize that my implementation of the model was not exceptionally great and there are a lot of simple aspects that can be changed to add more flexibility in terms of parallelism and testing, including but not limited to changing the way the three-dimensional environment is initialized, and its data type, which could allow for more optimized memory usage and less latency, which could, in turn, improve performance.

Some aspirations I have for my project also include 3D visualization of the model in close to real-time, and implementation of the model in CUDA C, which is an API that I have little to no experience with.

As for this project - I learned a lot from my work and will continue to work and improve on what I have learned.

Bibliography:

Artificial life. (2020, November 20). Retrieved from https://en.wikipedia.org/wiki/Artificial_life#:~:text=Artificial life (often abbreviated ALife,models, robotics, and biochemistry.

Cellular Automaton. (n.d.). Retrieved from https://mathworld.wolfram.com/CellularAutomaton.html#:~:text=A cellular automaton is a,many time steps as desired.

Cellular automaton. (2020, December 05). Retrieved from https://en.wikipedia.org/wiki/Cellular_automaton#:~:text=Ulam and von Neumann created,first system of cellular automata.

Conway's Game of Life. (2020, December 18). Retrieved from https://en.wikipedia.org/wiki/Conway's_Game_of_Life

Izhikevich, E. M., Conway, J. H., & Seth, A. (n.d.). Game of Life. Retrieved from http://www.scholarpedia.org/article/Game_of_Life

Wonders of Math - The Game of Life. (n.d.). Retrieved from http://www.math.com/students/wonders/life/life.html

Appendix A (Project Code):

https://github.iu.edu/gpullela/HPC_FALL_2020/blob/master/asg3/two.c

```
/***
        Conway's Game of Life 3D
        Written by Gourav Pullela
        ENGR-E 517
***/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#include <unistd.h>
#include <time.h>
#include <sys/time.h>

#define N 20

struct Cell
{
        int x;
        int y;
        int z;
        int doa;
};

struct Vector
{
        int x;
        int y;
        int z;
};

//returns array of Vector struct pointers that are point's neighbors
struct Vector* neighbors(struct Cell point) {

        struct Vector* out;
        out = malloc(26 * sizeof(struct Vector));
```

```
	out[0].x = point.x - 1; out[0].y = point.y - 1; out[0].z = point.z - 1;
	out[1].x = point.x - 1; out[1].y = point.y - 1; out[0].z = point.z;
	out[2].x = point.x - 1; out[2].y = point.y - 1; out[2].z = point.z + 1;
	out[3].x = point.x - 1; out[3].y = point.y; out[3].z = point.z - 1;
	out[4].x = point.x - 1; out[4].y = point.y; out[4].z = point.z;
	out[5].x = point.x - 1; out[5].y = point.y; out[5].z = point.z + 1;
	out[6].x = point.x - 1; out[6].y = point.y + 1; out[6].z = point.z - 1;
	out[7].x = point.x - 1; out[7].y = point.y + 1; out[7].z = point.z;
	out[8].x = point.x - 1; out[8].y = point.y + 1; out[8].z = point.z + 1;
	out[9].x = point.x; out[9].y = point.y - 1; out[9].z = point.z - 1;
	out[10].x = point.x; out[10].y = point.y - 1; out[10].z = point.z;
	out[11].x = point.x; out[11].y = point.y - 1; out[11].z = point.z + 1;
	out[12].x = point.x; out[12].y = point.y; out[12].z = point.z - 1;
	out[13].x = point.x; out[13].y = point.y; out[13].z = point.z + 1;
	out[14].x = point.x; out[14].y = point.y + 1; out[14].z = point.z - 1;
	out[15].x = point.x; out[15].y = point.y + 1; out[15].z = point.z;
	out[16].x = point.x; out[16].y = point.y + 1; out[16].z = point.z + 1;
	out[17].x = point.x + 1; out[17].y = point.y - 1; out[17].z = point.z - 1;
	out[18].x = point.x + 1; out[18].y = point.y - 1; out[18].z = point.z;
	out[19].x = point.x + 1; out[19].y = point.y - 1; out[19].z = point.z + 1;
	out[20].x = point.x + 1; out[20].y = point.y; out[20].z = point.z - 1;
	out[21].x = point.x + 1; out[21].y = point.y; out[21].z = point.z;
	out[22].x = point.x + 1; out[22].y = point.y; out[22].z = point.z + 1;
	out[23].x = point.x + 1; out[23].y = point.y + 1; out[23].z = point.z - 1;
	out[24].x = point.x + 1; out[24].y = point.y + 1; out[24].z = point.z;
	out[25].x = point.x + 1; out[25].y = point.y + 1; out[25].z = point.z + 1;
	return out;
}

//Returns number of living neighbors
int numAlive(struct Cell game[N][N][N], struct Cell point, struct Vector* n) {

	int i,alive = 0;
	#pragma omp parallel for
		for(i=0; i < 26; i++) {
			if((-1 < n[i].x && n[i].x < N) && (-1 < n[i].y && n[i].y < N) && (-1 <
n[i].z && n[i].z < N)) {
				if(game[n[i].x][n[i].y][n[i].z].doa == 1) {
					alive += 1;
				}
```

```
                        }
                }

        return alive;
}

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

        //Initialization of variables
        int i,j,k;
        int steps = 0;
        int alive;
        int nthreads;
        int count = 0;
        int maxcount = 0;
        struct timeval bsim, endsim;
        struct timeval bpop, endpop;
        struct Vector* input;
        struct Cell game[N][N][N];

        //Seeding rand() with time at runtime
        srand(time(NULL));

        //Pre-population time
        gettimeofday(&bpop, 0);

        //Populating the 3D Array of Cells
        #pragma omp parallel for collapse(3)
                for(i=0; i < N; i++) {
                        for(j=0; j < N; j++) {
                                for(k=0; k < N; k++) {
                                        game[i][j][k].x = i;
                                        game[i][j][k].y = j;
                                        game[i][j][k].z = k;
                                        if(rand() % 3 == 0) {
                                                game[i][j][k].doa = 1;
                                                count += 1;
                                        } else {
                                                game[i][j][k].doa = 0;
                                        }
```

```
                        }
                    }
                }

        maxcount = count;

        //Post-population time
        gettimeofday(&endpop, 0);

        #pragma omp parallel
        {
                nthreads = omp_get_num_threads();
        }

        printf("\n-----------------------------------------\n");
        printf("Number of Threads: %d\n", nthreads);
        printf("Initial population: %d\n", count);

        //Pre-simulation time
        gettimeofday(&bsim, 0);

        //Simulation
        while(count > 0) {
                #pragma omp parallel for collapse(3)
                        for(i=0; i < N; i++) {
                                for(j=0; j < N; j++) {
                                        for(k=0; k < N; k++) {
                                                input = neighbors(game[i][j][k]);
                                                alive = numAlive(game, game[i][j][k], input);
                                                if(game[i][j][k].doa == 1 && (alive < 6 || alive >
11)) {
                                                        game[i][j][k].doa = 0;
                                                        count -= 1;
                                                }
                                                if(game[i][j][k].doa == 0 && (alive > 7 && alive <
12)) {
                                                        game[i][j][k].doa = 1;
                                                        count += 1;
                                                }
                                                if(count > maxcount) {
```

```
                                                maxcount = count;
                                            }
                                        }
                                    }
                                }
                        steps += 1;
        }

        //Post-simulation time
        gettimeofday(&endsim, 0);

        free(input);

        //Time computation
        double simelapsed = (endsim.tv_sec - bsim.tv_sec) + ((endsim.tv_usec - bsim.tv_usec) /
1000000.0);
        double popelapsed = (endpop.tv_sec - bpop.tv_sec) + ((endpop.tv_usec - bpop.tv_usec) /
1000000.0);
        double avgpercycle = simelapsed / steps;

        //Data output
        printf("Maximum population: %d\n", maxcount);
        printf("Time take to populate the 3D environment: %lg\n", popelapsed);
        printf("Time taken to run simulation of %d cycles: %lg\n", steps, simelapsed);
        printf("Average runtime per cycle: %lg\n", avgpercycle);
        printf("----------------------------------------------\n\n");

        return(0);
}
```