UPPSALA
UNIVERSITET

High Performance Programming

Conway's Game of Life

Individual Assignment

Author:

Matilda Tiberg

Uppsala

August 20, 2021

Contents

1 Introduction

1.1 The Game of Life

The Game of Life was invented in 1970 by John Conway, and is a two-dimensional grid containing cells that are either alive or dead. This is a simulation of cellular automaton, where a grid of cells are seeded (initialized), and then evolves according to the given rules. Since the cells are autonomous, no input from a user is required except for the initial seeding. The interesting part of the Game of Life is that there is no algorithm to decide, from an initial configuration, whether a pattern will appear or not (a chaotic system). This implies that the only way to examine whether a pattern will appear or not, is to simulate the problem. Since the grid could be of an enormous size, and a lot of generations required, the need of code optimization is required to get a decent memory usage as well as time consumption.

1.1.1 The Rules

The cells in the two dimensional grid have two possible states: *alive* and *dead*. The status of the cell is depending on the eight adjacent cells: *the neighbors*. For each time step, the cell will either lives or dies depending on the amount of living neighbors around it. If the cell has too few neighbors, it dies due to underpopulation. If it has too many, it dies of overpopulation. A dead cell can also become a living cell. A more elaborate description of the rules is given in the list below.

1. If a **living cell** has fewer than **two** living neighbors: the cell dies.
2. If a **living cell** has more than **three** living neighbors: the cell dies.
3. If a **living cell** has exactly **two or three** living neighbors: the cell survives.
4. If a **dead cell** has exactly **three** living neighbors: the cell becomes a living cell.

1.2 Project Description

The goal of this individual project was to write a code in C, and then optimize it with regard to both time and memory consumption. In addition, the complexity of the code was determined.

Conway's Game of Life was chosen, and this kind of algorithm could be expanded to contain a lot of different things depending on the programmers creativity: e.g. initialize the world in different ways to make interesting patterns occur, enable it to detect steady states etc. However, this project only focuses on implementing it in the simplest form: initialize a random pattern, and then run it for a given amount of generations.

2 Code Description

The main idea of the program is that the world consists of a 2D-array with a total of $N \times N$ amount of cells. Each cell is represented by a struct which contains the status of the cell (alive or dead) and the amount of living neighbors that the cell has:

```
typedef struct cell {
    int neighbors;
    int status;
} cell_t;
```

The living status of the cell is represented by status = 1, and dead cells have status = 0. To calculate the amount of living neighbors, the status of the adjacent cells are being added together. This works, since the living status = 1, thus adding 1 neighbor to the total, and the dead neighbors have status = 0, thus adding zero to the total of neighbors. This kind of structure greatly simplified the process of calculating the next generation of cells in the function `updatePattern(cell_t** cells)`, since the calculations of the neighbors could be done without being affected by the updated cells, thus eliminating the need of another 2D-array. The function `updatePattern`, as stated above, updates the status of the cell depending on the amount of living neighbors, according to the rules given in Section 1.1.1.

The function `printGrid(cell_t** cells)` does what it sounds like: it prints white squares if the cell is alive, and nothing if the cell is dead. This function was disabled for all the timings, but is simply enabled by uncommenting the function in main().

`intialPattern(cell_t** cells, int p)` initializes a random pattern by using the `rand()`-command, seeded by using `time(0)`.

2.1 Running the Code

A Makefile compiles the code with the optimal compiler flags (determined in Section "Use of Compiler Flags"). This creates the file ”GameOfLife”, which is run in the following way:

```
./GameOfLife [size of world] [#generations] [probability of living cell] [amount of threads]
```

So, if the following is wanted: 1000×1000 grid, 500 generations, 30 % probability of the cell initially being alive, and 5 threads running, the following is stated: `./GameOfLife 1000 500 3 5`.

3 Optimizations

Firstly, an original code, seen in Appendix B, was written as a foundation to start the optimization from. This code was created without thinking about any optimizations of speed or memory usage, which lead to a badly written code. From this code, improvements were made and the final and optimized code can be seen in Appendix A.

The first optimizations were made to the structure of the code, e.g. avoiding unnecessary loops and unnecessary memory allocations. When this was done, different compiler flags were tested to determine which one that was optimal. Lastly, the program was parallelized using OpenMP.

All the measurements was done on a computer with the CPU Intel® Core™ i5-8250U CPU @ 1.60GHz, running on Linux as a subsystem to Windows 10. Compiler version *gcc (Ubuntu 9.3.0-17ubuntu1 20.04) 9.3.0* was used.

3.1 Alteration of Code Structure

Using the profiling tool `gprof`, it was seen that 45 % of the total time was spent on the `countNeighbors`-function and 55 % was spent at the`updatePattern`-function. This was expected, since these are the only functions that are being called multiple times (directly or indircetly) in the main()-function. These numbers are later used to show the difference from the code alterations.

Every compilation of the code in this part was made with the flags *-o -Wall -g -lm*, and the timing/memory measurements were run for a 1000×1000 world size, 1000 generations and a 50 % probability of a living cell. For all timings, all `printf()`-statements were commented, as well as `usleep()` (this function is only used to be able to see a visual representation of the cells).

3.1.1 Added Padding

In the original code, an $N \times N$-matrix was created, but this lead to a rather complicated `countNeighbors`-function, as seen in Appendix B (line 128-167), since the corners and the edges of the matrix needed to be treated as special cases. This resulted in many different for-loops. In order to avoid this, an $(N+2) \times (N+2)$-matrix was created instead, which is called *padding*. This means that the matrix has 2 extra rows: one up top, and one at the bottom, as well as 2 extra columns: one at each side. A visual example for a 4×4 world can be seen in Figure 1, where the padding is colored red. This significantly reduced the complication of the `countNeighbors`-function since no special cases to treat the edges/corners were needed. This was possible, since the statuses of the padding-cells always were set to zero, and thus not affecting the updating of the real cells.

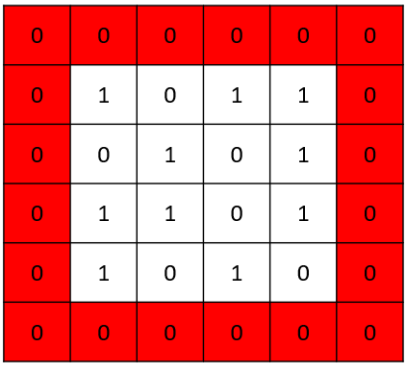

Figure 1: *A visual representation of the padding for a 4×4 world. The red cells are the padding, and will always have the status 0 (dead).*

When using the `gprof`-tool after this alteration, it could now be seen that `countNeighbors`-function only consumed 39 % of the total time, while `updatePattern` used 61 % of the total runtime. This shows that the alterations to the `countNeighbors`-function, enabled by the added padding, reduced the runtime for said function. Since this `countNeighbors` then was reduced to only 7 lines of code, it was instead directly implemented in the `updatePattern`-function to avoid unnecessary function-calls.

3.1.2 Removing extra 2D-array

In the original code, a new 2D-array was created for every iteration of the `updatePattern`-function, and was used to store the status of the next generation of cells. This was unnecessary, since the whole idea behind the cell-struct was to store each cell's amount of living neighbors and thus being able to update the cell's status directly without any copying of the arrays. The memory allocation for the updatedCells-pointer was therefore removed, as well as the freeing of the original cell-array. To verify the improvement of the code alteration, `valgrind` was used to check the memory allocation, and the results are shown in Table 1. Here, it is clearly shown that there was a significant decrease of total byte allocated.

Table 1: *Differnce in memory allocations for unoptimized and optimized code.*

	Allocations	**Total bytes allocated**
Unoptimized	1,002,002	8 016 009 024
Optimized	1004	8 041 072

3.1.3 Initializing Cells

Instead of having one extra nested for-loops that initializes all values to zero, the initialization was merged with the malloc-loop, as shown below. This is just a tiny improvement, since this part only appears once in the code, so no measurement of time was made.

```
cell_t** cells = (cell_t**)malloc((sizeWorld+2)*sizeof(cell_t*));
for(int i = 0; i < sizeWorld+2; i++){
    cells[i]=(cell_t*)malloc((sizeWorld+2)*sizeof(cell_t));
    //Sets all initial values to zero
     for (int j = 0; j < sizeWorld+2;j++){
        cells[i][j].status = 0;
        cells[i][j].neighbors = 0;
    }
}
```

3.1.4 Order of If-statments

The part where the cell's status is being updated was written in three different ways and the time consumption was measured. The ideas of the three different ways are shown in the list below. The code for the different versions is shown in Appendix C.

1. Check status of cells, then check the amount of neighbors.
2. Check amount of neighbors, and ignore status of cells
3. Check amount of neighbors, and then check status of cell

The idea of version 2 was that the method was supposed to overwrite the status of the cell, even if it was not needed (e.g. if a cell has status 0, as well as 2 living neighbors, the updated status is still 0). This was to reduce the amount of if-statements in the loop. As shown in Table 2, this was not quicker, which probably was due to that it takes more time to overwrite the status compared to the time that it takes to check the if-statements.

Continuing from the second version, a third version was developed, where the status of the cell was checked after checking the amount of living neighbors. This was to make sure that only the cells that actually needed to have their status updated were the only ones that got updated. At first, this seemed like the superior method, but it was then shown this was due to an error in the code and it was in fact the slowest method, as shown in Table 2.

Table 2: *The method corresponds to the same number in the list above. The best timing out of 5 timings is listed here.*

Method	**Timing** [s]
1	4.590
2	4.799
3	5.511

From the results stated above, it was clear that the first version was the quickest, and therefore this version was kept.

3.1.5 Different Keywords

The struct was declared together with `__attribute__((__packed__))` in order to reduce the size of the structs, but this resulted in a slower time: 4.640 s as compared to 4.590 s above. Due to this, the alteration was not kept.

The keyword `__restrict` was added in front of the pointers in the function calls, which significantly decreased the runtime: 3.395 s (compared to 4.590 s). This keyword is to make sure that no pointers overlap in the memory. Since this alteration made the code run significantly faster, this was kept.

3.2 Use of Compiler Flags

After the alterations to the code structure was done, different compiler flags were tested. The results are shown in Table 3.

Table 3: *Performance difference by using different compiler flags in the Makefile. Measured in wall seconds. The quickest run time is marked in bold. All versions were being run 5 times each, and the quickest time was chosen.*

Flags	**Time** [s]
-O0	13.109
-O1	13.192
-O2	3.402
-O3	3.019
-Ofast	2.928
-Os	3.525
-O2 -march=native -ffast-math	3.776
-O3 -ffast-math	2.924
-O3 -march=native	2.918
-O3 -march=native -ffast-math	2.934
-Ofast -march=native	**2.903**

From this, it is shown that "-Ofast -march=native" was the quickest one. The *-Ofast* flag is equivalent to *-O3 -ffast-math*. This means that *-Ofast* turns on the same compiler flags as *-O3*, e.g. *-ftree-loop-vectorize*, which autovectorizes the code, and *-floop-unroll-and-jam* which unrolls loops where it is possible.[1] Therefore, flags like *-funroll-loops* and *-ftree-vectorize* were not tested.

When using *-ffast-math* (or *-Ofast*), one has to be careful, because the calculations can lose some accuracy if floats are being used. However, since the only numbers used in the calculations are the integers 1 and 0, the accuracy is not a problem. Therefore, the code was compiled with the *-Ofast* flag together with the *-march=native* flag, where the latter one compiles the code with regards to the CPU of the compilation machine.

3.3 Parallelization with OpenMP

After the the compiler flags had been chosen, the code was parallelized with OpenMP, which was fairly simple. The header `include <omp.h>` was included in the .c-file, and the compiler flag *-fopenmp* was included in the Makefile. Then the loops in the `updatePattern`-function was parallelized by adding `pragma omp parallel for private(row, col)` over the two paragraphs with for-loops. The results are shown in Table 4, and visualized in Figure 2. Here, it can be seen that the time consumption has some kind of exponential decrease with the increased amount of threads. The time for 3 threads looks a bit off compared to the other timings, but this could be due to some background process in the computer, among other things.

Table 4: *The timings for different amount of threads.*

Number of threads	**Time** [s]
1	3.894
2	2.130
3	1.869
4	1.349
5	1.275
6	1.098
7	0.965
8	0.888

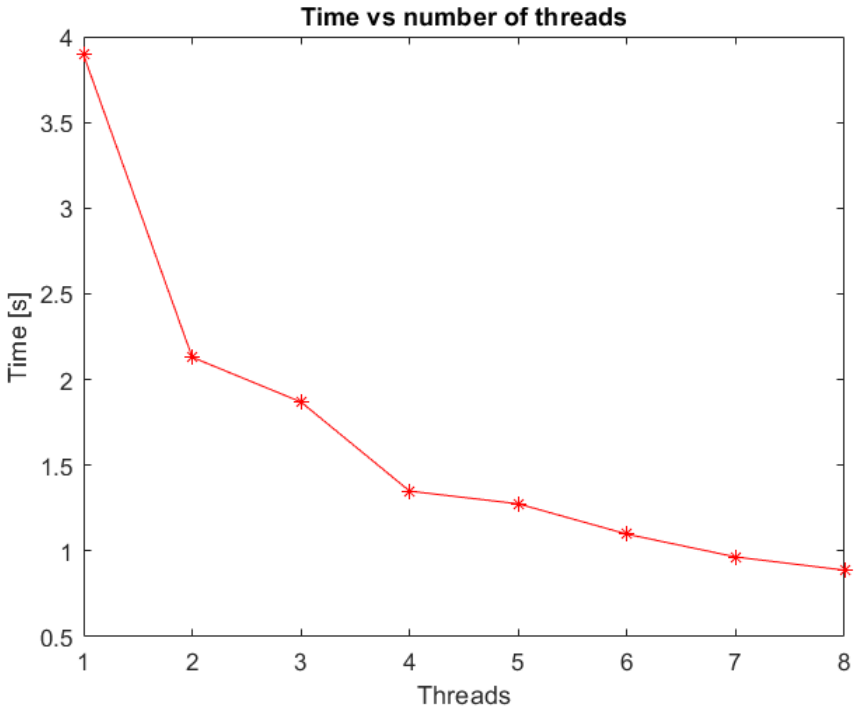

Figure 2: *This shows the time consumption as a function of the number of threads used. Simulated for a* 1000×1000 *grid, for 1000 generations.*

4 Results and Discussion

4.1 Timing

Stated in the Optimization-section above, the code was improved in three ways: alteration of the code structure, varying the compiler flags and parallelization with OpenMP. The best timings for the finalized versions of the different updates are being shown in Table 5. The difference in percentage was calculated by $\Delta t = \frac{t_{i-1}-t_i}{t_{i-1}}$. From this, it can be seen that the compiler flags contributed to the most significant difference (77 % increase in speed), but the parallelization using 8 threads (maximum on the computer being used) was close behind (69 % increase in speed).

Table 5: *The best runtimes for each improvement of the code. Each code was run 5 times, and the best time was noted. Time for 1000 × 1000 grid, 1000 steps and 50 % chance of living cell. Original code and "Code structure" run with -O0 flag.*

Improvement	**Time** [s]	Difference in time [%]
Original	15.564	-
Code structure	13.103	15.81
Compiler flags	2.903	77.84
Parallelization (8 threads)	0.888	69.41

4.2 Time Complexity

When examining the complexity of the algorithm, the serial code was used (the parallelization was disabled) together with the optimized compiler flags. The complexity was examined by taking the time for different grid sizes $N = 10, 50, 100, 500, 1000, 2500, 5000$ (here an $N \times N$ grid is written as just N). 1000 generations were simulated. The results are shown in Figure 3. Here, it can be seen that the time complexity has some kind of exponential growth.

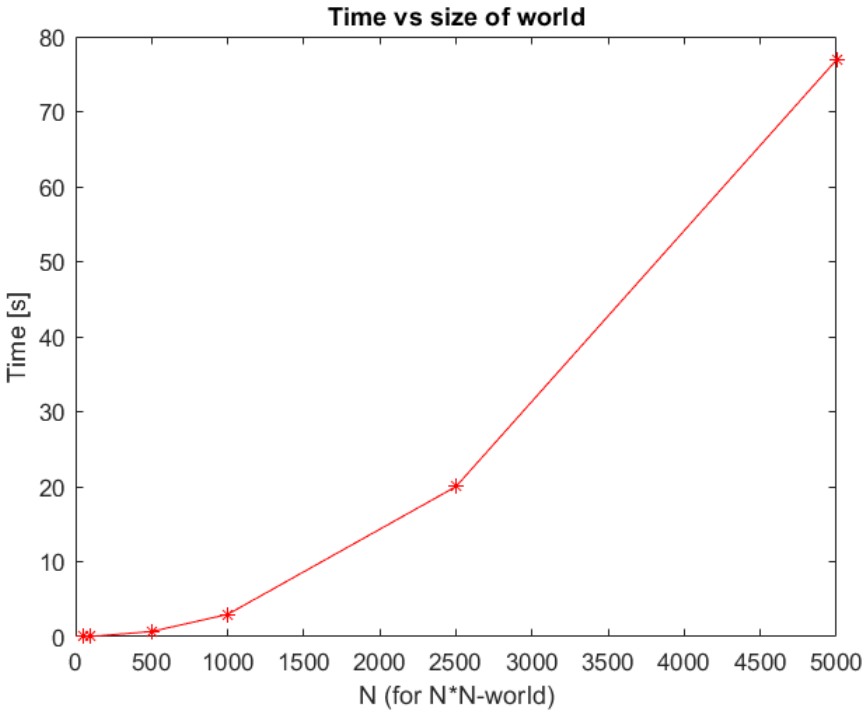

Figure 3: *The time consumption for a 1000 generations, given an $N \times N$-grid size.*

To determine the time complexity, the time consumption and the grid sizes were logarithmically plotted against each other. Then the functions N^2, and $Nlog(N)$ were plotted logarithmically in the same figure to enable visual comparison of the slopes. The results are shown in Figure 4. Here, it can be seen that the measured times has the same slope as the logarithmically plotted N^2. This means that the Game of Life-simulation has time complexity $\mathcal{O}(N^2)$.

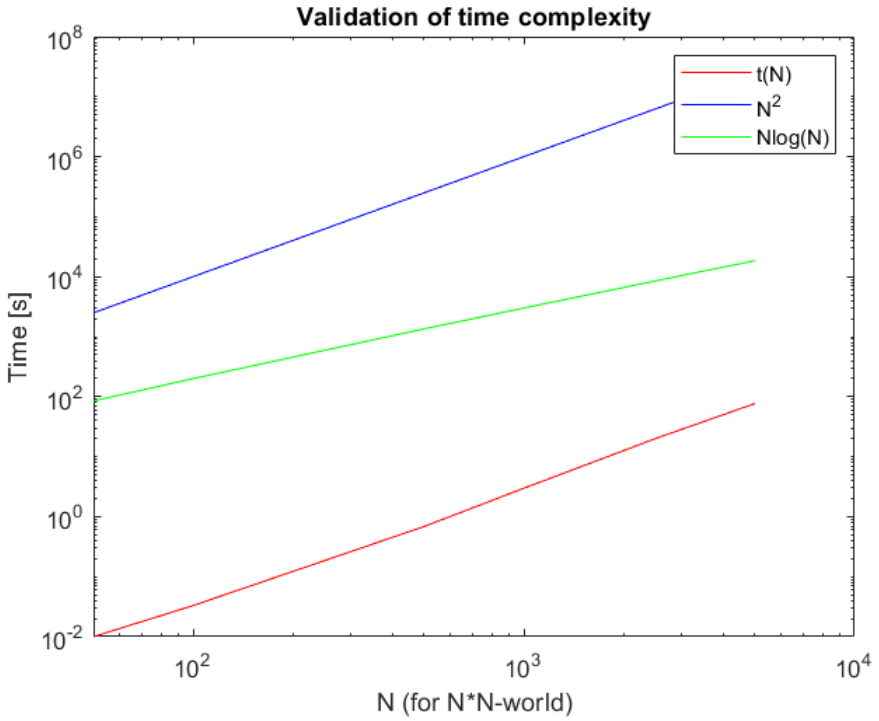

Figure 4: *The time consumption for 1000 generations simulated for different N (given as t(N) in figure. Plotted against two other functions to determine complexity.*

4.3 Further Improvements

In hindsight, there are lot of other ways to maybe optimize the code even further. One method could be to use a temporary array in the `updatePattern`-function, to enable merging of the two different sections containing for-loops. The code could be structured something like this (just giving the idea of the structure):

```
void updatePattern(cell_t** cells, cell_t** temp){
    for (rows){
        for (cols){
            cell[row][col].neighbors = count of neighbors;
            temp[row][col].status = update depending on result from above neighborcount
        }
    }
    swap pointers between temp and cells with help of dummy pointer.
}
```

By doing this, the code may run quicker since $N \times N$ iterations are being avoided from a nested for-loop. However, by doing it this way, we introduce the need for two more 2D-arrays, thus requiring more memory allocation, which is the reason why this was not tested in the optimization-process.

A better way of representing the grid could be by using a 1D-representation of the 2D-array instead:

```
cell_t* cells = (cell_t *)malloc((sizeWorld+2)*(sizeWorld+2)*sizeof(cell_t *));
```

(where the +2 comes for padding of the array). However, when this was tried, either the counting of the neighbors or the updating of the pattern was wrongly executed. Due to lack of time, this was not explored further. This method of storing the matrices is better for the cache performance, and is thus likely to

increase the speed.

Lastly, a minor thing could to be avoid the need for initialization of the cells, by using `calloc` instead of `malloc`. This form of memory allocation sets all uninitialized values to zero.

5 Conclusion

The code was optimized successfully, which a great improvement of time consumption: the unoptimized code took 15.564 seconds, while the final optimized code took 0.888 seconds to run a 1000×1000 grid for 1000 generations. Significant improvement could be seen from all alterations: the alterations to the structure of the code increased the speed by 15 %, the compiler flags by 78 % and the paralleliaztion by 69 % (increased speed as compared to the alteration before said method). From this, it could be concluded that the usage of compiler flags as well as parallelization is really important when developing a code. Nevertheless, the code itself is shown to be very important.

The results were satisfying, but more alterations to the code could be done in order to improve e.g. the cahce performance.

References

[1] *Options That Control Optimization*, https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html [Updated: ??, Read: August 20, 2021]

Appendix A - Optimized Code

```
1
2  /*  Conway's Game of Life - OPTIMIZED, PARALLEL VERSION
3
4      The cells live in a NxN matrix, and are initilized
5      to 0 (dead) or alive (1) randomly.
6      The cells statuses are updated accoring to the following rules:
7          alive #neighbors < 2 -> dies
8          alive #neighbors == 2 || 3 -> lives
9          alive #neighbors > 3 -> dies
10         dead #neighbors == 3 -> lives
11
12     The code is run by entering the following in the terminal:
13
14     ./GameOfLife [N] [steps] [probability] [numberOfThreads]
15 */
16 #include <stdio.h>
17 #include <stdlib.h>
18 #include <sys/time.h>
19 #include <omp.h>
20
21 //Global variables
22 int sizeWorld;
23
24 /*Keep tracks of each cell in world, and its neighbors.
25     Status = 1 --> alive,
26     Status = 0 --> dead
27 */
28 typedef struct cell {
29     int neighbors;
30     int status;
31 } cell_t;
32
33 void printGrid(cell_t** __restrict cells);
34 void initialPattern(cell_t** __restrict cells, int p);
35 void updatePattern(cell_t** __restrict cells);
36
37 static double get_wall_seconds() {
38     struct timeval tv;
39     gettimeofday(&tv, NULL);
40     double seconds = tv.tv_sec + (double)tv.tv_usec / 1000000;
41     return seconds;
42 }
43
44
45 //Main program
46 int main(int argc, char* argv[]){
47     double time_start = get_wall_seconds();
48     if (argc != 5){
49         printf("Wrong input! \nFollowing arguents required: \n");
50         printf("    N: size of world\n");
51         printf("    s: number of steps in the simulation. Integer.\n");
52         printf("    p: chance of living cell. Given as integer [1,10]\n");
53         printf("    t: number of threads for parallelization\n");
54         printf("\nEnding program.\n");
55         return -1;
56     }
57
58     sizeWorld = atoi(argv[1]);
59     int steps = atoi(argv[2]);
60     int probability = atoi(argv[3]);
61     int numThreads = atoi(argv[4]);
```

```
62      omp_set_num_threads(numThreads);
63
64      cell_t** cells = (cell_t**)malloc((sizeWorld+2)*sizeof(cell_t*));
65      for(int i = 0; i < sizeWorld+2; i++){
66          cells[i]=(cell_t*)malloc((sizeWorld+2)*sizeof(cell_t));
67          //Sets all initial values to zero
68           for (int j = 0; j < sizeWorld+2;j++){
69              cells[i][j].status = 0;
70              cells[i][j].neighbors = 0;
71          }
72      }
73
74      initialPattern(cells,probability);
75
76      int i = 0;
77      /*  The commented lines in this section should be
78          uncommented for a visual representation.*/
79      while (i < steps){
80          //system("clear");
81          //printf("\n--------------NEW GRID----------------\n");
82          //printGrid(cells);
83          updatePattern(cells);
84          //usleep(600000);
85          i = i+1;
86      }
87
88      for (int i=0; i<sizeWorld+2; i++){
89          free(cells[i]);
90      }
91      free(cells);
92
93      printf("\nProgram ended. Have a Good Life! \n");
94      printf("GameOfLife main took %7.3f wall seconds.\n", get_wall_seconds()-time_start);
95      return 0;
96  }
97
98
99  // ==================== FUNCTIONS ===================================
100 void printGrid(cell_t** __restrict cells){
101     for (int row=1; row<sizeWorld+1; row++){
102         for(int col=1; col < sizeWorld+1; col++){
103             if(cells[row][col].status == 1){
104                 printf("\u2B1C ");
105             }
106             else{
107                 printf("   ");
108             }
109         }
110         printf("\n");
111     }
112 }
113
114 // Creates a random pattern with a set size
115 void initialPattern(cell_t** __restrict cells, int p){
116     int prob = 10-p;
117     int chance;
118     srand(time(0));
119     for(int row = 1; row < sizeWorld+1; row++){
120         for (int col = 1; col < sizeWorld+1; col++){
121             chance = rand()%10;
122             if (chance ≥ prob){
123                 cells[row][col].status = 1;
124             }
125         }
126     }
```

```
127 }
128
129
130 void updatePattern(cell_t** __restrict cells){
131     //Counts neighbors
132     int row, col;
133
134     #pragma omp parallel for private(row, col)
135     for(row=1; row<sizeWorld+1; row++){
136         for (col = 1; col < sizeWorld+1; col++){
137             cells[row][col].neighbors =
138                     cells[row][col-1].status + cells[row][col+1].status +
139                     cells[row-1][col-1].status + cells[row-1][col].status + ...
                        cells[row-1][col+1].status+
140                     cells[row+1][col-1].status + cells[row+1][col].status + ...
                        cells[row+1][col+1].status;
141         }
142     }
143
144     //Updates statuses
145     #pragma omp parallel for private(row, col)
146     for(row=1; row<sizeWorld+1; row++){
147         for (col = 1; col < sizeWorld+1; col++){
148             if (cells[row][col].status == 0){
149                 if (cells[row][col].neighbors == 3){
150                     cells[row][col].status = 1;
151                 }
152             }
153             else{
154                 if (cells[row][col].neighbors > 3){
155                     cells[row][col].status = 0;
156                 }
157                 else if (cells[row][col].neighbors < 2){
158                     cells[row][col].status = 0;
159                 }
160             }
161         }
162     }
163 }
```

Appendix B - Unoptimized Code

```
1
2  /*  Conway's Game of Life - UNOPTIMIZED VERSION
3
4      The cells live in a NxN matrix, and are initilized
5      to 0 (dead) or alive (1) randomly
6      The cells are updated accoring to the following rules:
7          alive #neighbors < 2 -> dies
8          alive #neighbors == 2 || 3 -> lives
9          alive #neighbors > 3 -> dies
10         dead #neighbors == 3 -> lives
11
12     The code is run by entering the following in the terminal:
13
14     ./GameOfLife [sizeWorld]
15 */
16
17 #include <stdio.h>
18 #include <stdlib.h>
19 #include <sys/time.h>
20
21 //Global variables
22 int sizeWorld;
23
24 /*Keep tracks of each cell in world.
25     status = 1 --> alive,
26     status = 0 --> dead
27 */
28 typedef struct cell {
29     int neighbors;
30     int status;
31 } cell_t;
32
33
34 //Functions
35 void printGrid(cell_t** cells);
36 void initialPattern(cell_t** cells);
37 void countNeighbors(cell_t** cells);
38 cell_t** updatePattern(cell_t** cells);
39
40 //Function for time measurement
41 static double get_wall_seconds() {
42     struct timeval tv;
43     gettimeofday(&tv, NULL);
44     double seconds = tv.tv_sec + (double)tv.tv_usec / 1000000;
45     return seconds;
46 }
47
48 //Main program
49 int main(int argc, char* argv[]){
50     double time_start = get_wall_seconds();
51
52     if (argc != 2){
53         printf("Wrong input! \nFollowing arguents required: \n");
54         printf("    N: size of world\n");
55         printf("\nEnding program.\n");
56         return -1;
57     }
58
59     sizeWorld = atoi(argv[1]);
60
61     cell_t** cells = (cell_t**)malloc(sizeWorld*sizeof(cell_t*));
```

```
62      for(int i = 0; i < sizeWorld; i++){
63          cells[i]=(cell_t*)malloc(sizeWorld*sizeof(cell_t));
64      }
65
66      for(int i = 0; i<sizeWorld; i++){
67          for(int j=0; j<sizeWorld; j++){
68              cells[i][j].status = 0;
69              cells[i][j].neighbors = 0;
70          }
71      }
72
73      initialPattern(cells);
74
75      int i = 0;
76      /*  The commented lines in this section should be
77          uncommented for a visual representation.*/
78      while(i < 1000){
79          //system("clear");
80          //printf("\n--------------NEW GRID----------------\n");
81          //printGrid(cells);
82          cells = updatePattern(cells);
83          //usleep(600000);
84          i=i+1;
85      }
86      for (int i=0; i<sizeWorld; i++){
87          free(cells[i]);
88      }
89      free(cells);
90
91      printf("\nProgram ended. Have a Good Life! \n");
92      printf("GameOfLife main took %7.3f wall seconds.\n", get_wall_seconds()-time_start);
93      return 0;
94  }
95
96  // ==================== FUNCTIONS ===================================
97  void printGrid(cell_t** cells){
98      for (int row=0; row<sizeWorld; row++){
99          for(int col=0; col < sizeWorld; col++){
100             if(cells[row][col].status == 1){
101                 printf("\u2B1C ");
102             }
103             else{
104                 printf("   ");
105             }
106         }
107         printf("\n");
108     }
109 }
110
111 /*Sets a randomized initial configuration of the grid.*/
112 void initialPattern(cell_t** cells){
113     srand(time(0));
114     int chance;
115     for(int row = 0; row < sizeWorld; row++){
116         for (int col = 0; col < sizeWorld; col++){
117             chance = rand()%10;
118             if (chance > 4){    // 50 % chance to live
119                 cells[row][col].status = 1;
120             }
121             else{
122                 cells[row][col].status = 0;
123             }
124         }
125     }
126 }
```

```
127
128 void countNeighbors(cell_t** cells){
129     //Handles first and last row, except corners
130     for(int col=1; col<sizeWorld-1; col++){
131         cells[0][col].neighbors =
132                 cells[0][col-1].status + cells[0][col+1].status +
133                 cells[1][col-1].status + cells[1][col].status + cells[1][col+1].status;
134
135         cells[sizeWorld-1][col].neighbors =
136                 cells[sizeWorld-1][col-1].status + cells[sizeWorld-1][col+1].status +
137                 cells[sizeWorld-2][col-1].status + cells[sizeWorld-2][col].status + ...
                        cells[sizeWorld-2][col+1].status;
138     }
139
140     //Handles first and last column, except corners
141     for(int row=1; row<sizeWorld-1; row++){
142         cells[row][0].neighbors =
143                 cells[row-1][0].status + cells[row+1][0].status+
144                 cells[row-1][1].status + cells[row][1].status + cells[row+1][1].status;
145
146         cells[row][sizeWorld-1].neighbors =
147                 cells[row-1][sizeWorld-1].status + cells[row+1][sizeWorld-1].status+
148                 cells[row-1][sizeWorld-2].status + cells[row][sizeWorld-2].status + ...
                        cells[row+1][sizeWorld-2].status;
149     }
150
151     //Handles corners
152     cells[0][0].neighbors = cells[0][1].status + cells[1][0].status + cells[1][1].status;
153     cells[0][sizeWorld-1].neighbors = cells[0][sizeWorld-2].status + ...
            cells[1][sizeWorld-1].status + cells[1][sizeWorld-2].status;
154     cells[sizeWorld-1][0].neighbors = cells[sizeWorld-1][1].status + ...
            cells[sizeWorld-2][0].status + cells[sizeWorld-2][1].status;
155     cells[sizeWorld-1][sizeWorld-1].neighbors = cells[sizeWorld-1][sizeWorld-2].status + ...
            cells[sizeWorld-2][sizeWorld-1].status +
156                                                 cells[sizeWorld-2][sizeWorld-2].status;
157
158     //Handles the rest
159     for (int row=1; row<sizeWorld-1; row++){
160         for (int col = 1; col<sizeWorld-1; col++){
161             cells[row][col].neighbors =
162                     cells[row][col-1].status + cells[row][col+1].status +
163                     cells[row-1][col-1].status + cells[row-1][col].status + ...
                            cells[row-1][col+1].status+
164                     cells[row+1][col-1].status + cells[row+1][col].status + ...
                            cells[row+1][col+1].status;
165         }
166     }
167 }
168
169
170 //Updates cellpattern
171 cell_t** updatePattern(cell_t** cell){
172     cell_t** updatedCells = (cell_t**)malloc(sizeWorld*sizeof(cell_t*));
173     for(int i = 0; i < sizeWorld; i++){
174         updatedCells[i]=(cell_t*)malloc(sizeWorld*sizeof(cell_t));
175     }
176
177     for(int i = 0; i<sizeWorld; i++){
178         for(int j=0; j<sizeWorld; j++){
179             updatedCells[i][j].status = 0;
180             updatedCells[i][j].neighbors = 0;
181         }
182     }
183
184     countNeighbors(cell);
```

```
185
186     for(int row=0; row<sizeWorld; row++){
187         for (int col = 0; col < sizeWorld; col++){
188             if (cell[row][col].status == 0){
189                 if (cell[row][col].neighbors == 3){
190                     updatedCells[row][col].status = 1;
191                 }
192                 else{
193                     updatedCells[row][col].status = 0;
194                 }
195             }
196             else{
197                 if (cell[row][col].neighbors == 2 || cell[row][col].neighbors == 3){
198                     updatedCells[row][col].status = 1;
199                 }
200                 else if (cell[row][col].neighbors > 3){
201                     updatedCells[row][col].status = 0;
202                 }
203                 else{
204                     updatedCells[row][col].status = 0;
205                 }
206             }
207         }
208     }
209
210     for (int i=0; i<sizeWorld; i++){
211         free(cell[i]);
212     }
213     free(cell);
214
215     return updatedCells;
216 }
```

Appendix C - updatePattern versions

```
1
2  //FIRST VERSION:
3  //============================================================
4  for(int row=1; row<sizeWorld+1; row++){
5     for (int col = 1; col < sizeWorld+1; col++){
6         if (cells[row][col].status == 0){
7             if (cells[row][col].neighbors == 3){
8                 cells[row][col].status = 1;
9             }
10        }
11        else{
12            if (cells[row][col].neighbors > 3){
13                cells[row][col].status = 0;
14            }
15            else if (cells[row][col].neighbors < 2){
16                cells[row][col].status = 0;
17            }
18        }
19    }
20 }
21
22 //SECOND VERSION:
23 //============================================================
24  for(int row=1; row<sizeWorld+1; row++){
25        for (int col = 1; col < sizeWorld+1; col++){
26            if(cells[row][col].neighbors == 3){
27                cells[row][col].status = 1;
28            }
29            else if(cells[row][col].neighbors < 2){
30                    cells[row][col].status = 0;
31            }
32            else if(cells[row][col].neighbors >3){
33                cells[row][col].status = 0;
34            }
35        }
36    }
37 }
38
39 //THIRD VERSION:
40 //============================================================
41 for(int row=1; row<sizeWorld+1; row++){
42     for (int col = 1; col < sizeWorld+1; col++){
43         if(cells[row][col].neighbors == 3){
44             if (cells[row][col].status == 0){
45                 cells[row][col].status = 1;
46             }
47         }
48         else if(cells[row][col].neighbors < 2){
49             if (cells[row][col].status == 1){
50                 cells[row][col].status = 0;
51             }
52         }
53         else if(cells[row][col].neighbors >3){
54             if (cells[row][col].status == 1){
55             cells[row][col].status = 0;
56             }
57         }
58     }
59 }
```