Individual Project

High Performance Programming

Author:

Moa Li Lekander

Uppsala

March 8, 2021

Contents

1 Introduction

Code optimizations is a matter of great importance in many areas of science today. Although the performance and computing capacity of the modern computers is rapidly improving, there still exist many scenarios where it is highly auspicious to minimize the run time of programs to the greatest extent. By learning techniques of optimizing and parallelizing code, heavy simulations and computations that originally could takes days or even weeks to run, can instead be written so that its run time can be reduced significantly.

The aim of this project is to implement an efficient code using parallelization, for an application or algorithm of own choosing. In this report, a Game of Life simulation was implemented, and the code was firstly being serial optimized and thereafter parallelized using Pthreads. Also, the time complexity of the final algorithm was determined and the effect of parallellization with different numbers of threads was analysed.

2 Problem Description

A Game of Life simulation visualizes the evolution of a specific initial state in a 2D binary world. Hence, the simulation is played out on an quadratic grid of cells, where each cell is defined as either dead or alive. Depending on the amount of alive cells among the eight neighbours of each cell (see Figure 1), the next state of each cell is determined by a set of rules, that are as follows:

- If a cell is *alive* and has two or three neighbours it stays alive in the next time step, otherwise it will be *dead* in the next time step.
- If a cell is *dead* and has exactly three neighbours it will become alive in the next time step, otherwise it will stay dead in the next time step.

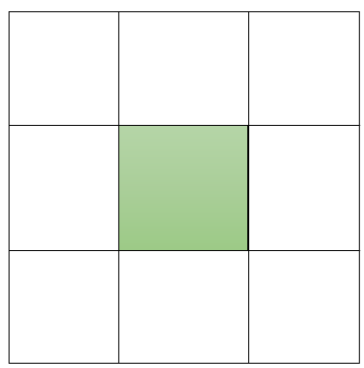

Figure 1: The current cell displayed as green and its eight surrounding neighbours

The goal of this project is to implement a Game of Life simulation, optimize the serial code and finally parallelize the code using Pthreads. Thereafter, the time complexity of the program will be analysed and the effects of the parallelization will be evaluated.

3 Solution Method

3.1 Original Implementation

The original code for the Game of Life simulation can be found in Appendix A. The script consists of a main function that asks the user for an input of the size of the grid, the number of time steps and a variable that determines if the grid state should be printed on the screen at each time step (i.e. if a visual simulation should be enabled or not). The last mentioned variable should take the value 1 if the simulation is desired or 0 if it is not. If the wrong input format is entered, the user will receive an error message describing the expected input and the program will terminate.

The size of the grid is declared and stored in the variable `grid_size`. The grid is then initiated and filled with a randomly generated initial state by the function `void initGrid(int grid_size, int** grid)`. The grid is represented by a 2D array of the datatype `int`, and required memory for this is allocated using `malloc`. The `initGrid` function uses two nested for-loops to fill the grid with a randomized initial state. The cells of the grid will randomly obtain either the number 0, indicating that the cell is *dead*, or the number 1 meaning that the cell is *alive*.

After the grid has been initialized, the goal is to update the grid at every time step, which is done by the function `void updateGrid(int grid_size, int** grid, int** temp_grid)`, that updates the grid according to the rules described in section 2. Before this function is called however, a larger grid `temp_grid` is initialized. The purpose of this grid is to make sure that the edges of the grid are handled correctly, and therefore at the beginning of every time step a deep copy of the main grid is made and stored in this larger grid. This grid is also given a padding of zeros on its edges that will ensure that the edge cells of the main grid will be handled correctly in the computations. Then, both the main grid and the padded grid is sent into the function `updateGrid`.

Inside the `updateGrid` function, two nested for loops are used to iterate over the cells, and based on the padded array `temp_grid` the number of alive neighbours are computed by the function `int countNeighbours (int grid_size, int** grid, int i, int j)`. This function takes the temporary padded grid as input together with the position of the current cell within the padded grid, and returns the sum of all the eight surrounding cells which corresponds to the number of alive neighbours (since alive is defined as 1 and dead as 0). The new state of each cell is then updated in the original (unpadded) array according to the rules described in section 2. This process for updating the grid will be repeated for the given number of time steps by using a for loop in the main function. When this process is finished, all the allocated memory is freed and the program terminates.

If the simulation option is enabled (i.e. the variable `simulation` is set to 1), the main grid will be printed at every time step by the function `void printGrid(int grid_size, int** grid)`. This function takes the grid as input and displays the cells as '`.`' if the cell is dead and '`*`' if the cell is alive. The function `nanosleep` from the library `time.h` is called after every print of the grid in order to make the simulation show the states more smoothly.

3.2 Serial Optimizations

The code for when the serial optimizations had been made is presented in Appendix B. When the code was being optimized, several different methods for optimizing the code were tested and systematically evaluated by measuring the time it took for the program to run. Those optimizations that resulted in an improved performance were kept. Even though some alterations might have made the readability of the code worse, they were kept anyways since the main focus of this project was to minimize the run time of the program. In this section a summary of all the adjustments that were made to the code is presented. A more thorough description of the optimization techniques that were tested can be found in section 4.1.

Firstly, it was found that the -O3 compiler flag generated the fastest code, and therefore that compiler flag was used when compiling the program. Secondly, several if-statements within loops were removed and replaced by additional for-loops instead. After this change, the remaining if-statements that existed in the code were optimized by merging two else-if branches. Another optimization that was made was to remove the `countNeighbours` function that was called inside a nested for-loop. Instead, the computations that were made by the function were computed directly inside the loop instead. Also, the variable `int neighbours` was initiated in the main function instead of inside the nested for-loop in order to minimize the memory usage of the program. Loop unrolling was introduced with the `-funroll-loops` compiler flag and the datatype for the arrays used in the program was changed to `short int` instead of `int`. Also, both the `constant` keyword and the `restrict` keyword were added to multiple places in the code. Additionally, the number of `malloc` calls were reduced significantly by pointing arrays into larger buffer arrays instead. Lastly, it was confirmed that the -O3 compiler flag successfully managed to vectorize 6 loops in the code.

3.3 Parallelization Using Pthreads

The code for the parallelized implementation is presented in Appendix C. In this implementation, the two arrays used in the program (`grid` and `temp_grid`) are made global so that all threads easily can have access to them. Also, another input argument is required by the user to enter, which states the number of threads to use when running the program. This value is stored in the variable `num_of_threads`.

In order to be able to send data to the different threads, a struct `thread_data_t` is created. This enables every thread to get access to specific values that are needed for the computations. An array of the struct `thread_data_t` is initialized for the given number of threads, and the amount of rows in the grid for every thread to work on is stored in the variable `work_size`. If the size of the grid is not evenly divisible by the number of threads entered, this work size amount will not cover the last rows in the grid. This problem is solved by computing the rest of the division and storing this in a variable called `rest`. From this, the last rows will be assigned to the last thread that is being created.

The grids are initialized and filled with values in the same manner before. Then, a global mutex and a global condition variable is initiated using the functions `pthread_mutex_init` and `pthread_cond_init`, that later are used in the synchronisation of the threads. After this step, the threads are given their needed values for the computations and their start and stop indexes that specify which rows they should do their work on. All threads are pointed to the same thread function `void* updateGrid(void* arg)`. This function updates the grid for every time step (in the same manner as in previous implementation), and uses the barrier function `void barrier(const int num_of_threads, const int grid_size, const int simulation)` to make sure that every thread has updated the current grid before they all move on to the next time step.

The `barrier` function keeps count of how many threads that have finished their computations and are ready to move on to the next time step. This value is kept in the global variable `waiting` that is increased every time the barrier function is called (which is done by every thread in the beginning of every time step). As long as all threads are not finished with their work for the current time step, the thread will be put to sleep by the function `pthread_cond_wait`. When the last thread is done with its computations and calls for the `barrier` function (i.e. when all other threads are waiting for the last thread to finish), the variable `waiting` is set to zero and a call for the function `pthread_cond_broadcast` is made in order to wake up all waiting threads that have been set to sleep. If the simulation is desired (i.e. the `simulation` variable has been set to 1), a call for the `printGrid` function is made, which displays the current grid on the screen. When this has been done, the newly awaken threads will continue to do their work for the following time steps according to the same course of action that has just been described.

When the grid has been updated for all time steps, the main function will call the function `pthread_join` for every thread. The allocated memory is freed and the mutex and condition variables are destroyed by the functions `pthread_mutex_destroy` and `pthread_cond_destroy` respectively. Finally, the program terminates by returning 0.

4 Experiments

When the code was being serial optimized, time measurements were made in order to evaluate which optimization techniques that were effective. When these experiments have been performed, the executable file has been run with the following command: `./gameOfLife 1000 1000 0`, and the time was measured in wall seconds (by using the function `static double get_wall_seconds()`). This command indicates that the simulation was run for 1000 time steps with a grid size of 1000x1000. Also, no visual simulation was made when optimizing, i.e. the `simulation` variable was set to 0 (as can be seen in the input command). Whenever the code was changed and a time measurement was made, the code was run five times and the measurement with the best performance was considered as the final result. This approach was done for the purpose of minimizing the effects that other programs running in the background of the computer could have had on the measurements. The code was optimized on a *Macbook Pro Intel(R) Core(TM) i5-5257U CPU @ 2.70GHz* with the compiler version *clang-1001.0.46.4*.

4.1 Serial Optimizations

The first step when optimizing the serial code was to test different compiler flags and examine their effect on the time performance of the program. In Table 1 the measured times is presented for the different compiler flags. It can be noted that the flag `-O3` gave the best performance.

Table 1: Time measurements for different compiler flags, where the compiler flag with the best performance is marked in bold

Flags	**Time** [s]
-O0 (default)	20.085
-O1	6.290
-O2	4.790
-O3	**4.403**
-Ofast	4.452
-Os	7.577
-O3 -ffast-math	4.407
-O3 -march=native	9.245
-O3 -march=native -ffast-math	4.427
-O2 -march=native -ffast-math	4.712
-Ofast -march=native -ffast-math	4.485

The next step in the optimization process was to evaluate all loops used in the program and examine if they could be rewritten in any advantageous way. The first goal was to try to avoid having if-statements within loops, since that often can worsen the performance of a program. The if-statements in the nested for-loops that made the deep copy of the grid and added padding to the temporary grid was removed (see lines 54-58 and 75-79 in Appendix A). Instead, two more for-loops were added in order to make the code function correctly (see lines 55-67 and 80-92 in Appendix B). When implementing this change, the time required for the code to run was reduced to 4.185 seconds and thus this alteration was kept.

It was not possible to remove the if-statement that checked the rules of the game (lines 116-124 in Appendix A) in the `updateGrid` function. However, the two else-if statements on lines 118-120 in Appendix A could be merged together (see result in lines 125-126 in Appendix B) which resulted in an improvement of the performance since the time reduced to 3.726 seconds. The order of the boolean expressions within the first if-statement (on line 116 in Appendix A) was changed as well, for the purpose of implementing boolean short circuits. Since there is a greater probability that the number of neighbours is not correct, compared to the 50% chance that the cell is either dead or alive, it seemed more beneficial to have the boolean expression checking the number of neighbours first. Despite this reasoning, that change made the performance slightly worse, as the time increased to 4.150 seconds, and therefore the change was not kept.

It was also tested to put the computation of the number of neighbours directly in the loop instead of calling for the function `countNeighbours` (line 114 in Appendix A). This alteration resulted in a small improvement since the time reduced to 3.650, and so the change was kept. Additionally, for the purpose of reducing the memory footprint of the program, the initiation of the variable `int neighbours` (on line 114 in Appendix A) was moved to the main function and the variable was then used as an input to the function `updateGrid` instead. From this, the same memory location for that variable could be used through the whole simulation instead of declaring a new variable every time the function `countNeighbours` was called.

Loop unrolling is also an approach that sometimes can improve the performance of a code. The compilation flag `-funroll-loops` was added in the compilation in order to test its effect. It did not give any improvement in the performance however, nor did it make the performance worse, so the change was kept.

For the purpose of minimizing the memory usage of the program, the arrays for the grids were changed to be of the datatype `short int` instead of `int`. Because the array elements only take on the values 0 or 1, it seemed unnecessary to allocate more memory for the arrays than was needed, and therefore less memory could be used by the program which often can improve the performance. When testing this change, it could be noted that the performance did improve as the time reduced to 3.590 seconds. Therefore, this alteration was kept.

The `const` keyword was added to the unchanging variables that existed in the code, i.e. the variables `double time1`, `int time_steps`, `int simulation`, `int grid_size` (on lines 20, 26, 27 and 25 in Appendix A respectively) and all `int grid_size` input parameters to the functions. Because of this keyword, the compiler would not have to assume that the variables can change during the execution of the program, which sometimes can improve the performance. In this case, a small improvement of the performance could be noted as the time reduced to 3.535 seconds. Thus, the keywords were kept.

In all functions the keyword `__restrict` was added in the function input parameters for the arrays used in the program. This keyword could be used since it was known that none of the pointers used as input in the respective functions pointed to the same memory address. However, this alteration did not have any significant improvement of the performance, nor did it make it any worse, and so the keywords were kept.

Another thing that can improve the performance of a program is to minimize the number of `malloc` and `free` calls. In order to do this, two buffer arrays for the rows and columns in the two grids used in the program were created. From this, only two `malloc` call to the buffer arrays were needed since they allocated enough memory for both arrays. Then, the two arrays for the grids were pointed into the larger buffer arrays instead (see implementation in Appendix B lines 33-46 and 99-100). This change reduced the number of `malloc` and `free` calls significantly, which could also be noticed in the performance of the program since the run time reduced to 3.249 seconds. Thus, this change was kept.

Vectorization of the code is also an approach that can improve the performance of a code. Since the `-O3` compiler flag was used, which by default includes auto-vectorization of the code, it was assumed that auto vectorization already was enabled. For the purpose of examining if the code could be rewritten in any way so that more auto vectorizations could be made, the flag `-ftree-vectorizer-verbose=2` was included when compiling. From this, it was concluded that 6 loops in the program had successfully been vectorized, namely the loops starting on lines 38, 44, 56, 60, 81 and 85 in Appendix B. The nested for-loops inside the `updateGrid` function and inside the `printGrid` function were two of the loops that were not vectorized, which probably was due to the if-statements that existed within the loops. For the purpose of vectorizing the rest of the unvectorized loops, it was tested to add the `-march=native`, `-ffast-math` and `-mavx` flags when compiling. However, these flags made no difference, and unfortunately, since the rest of the loops did not have any distinct loop invariants, it was difficult to figure out why these loops were not being vectorized. Hence, no more loops in the code were successfully vectorized.

4.2 Parallelization Using Pthreads

In order to determine what parts of the code that consumed most of the time, the tool `gprof` was used. From this it was clear that the function `updateGrid` occupied a the majority of the time required for the code to run (more precisely 86% of the time). Thus, the main focus of the parallelization process was laid on optimizing this function.

The final code, presented in Appendix C, was run for different grid sizes on one thread for the purpose of analysing the time complexity of the algorithm. The code was run with the following command: `./gameOfLife grid_size 1000 0 1`, meaning that the variable `grid_size` was varied and that the code was run for 1000 time steps, without the simulation enabled and with only one thread. In Figure 2 the result of these measurements is displayed. It can be observed from the shape of the curve that the time complexity of the program most probably is $\mathcal{O}(N^2)$. The time for when the command `./gameOfLife 1000 1000 0 1` was run (the same command used when the serial optimized code was being developed) was measured to be 3.207 seconds which is slightly better than its previous version of the code (the one presented in Appendix B).

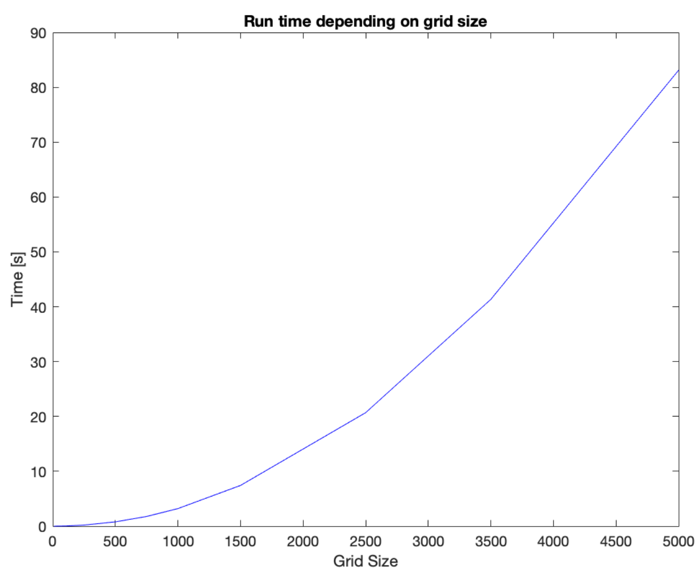

Figure 2: Run time of the program for different sizes of the grid

An analysis of the run time of the program for different number of threads was also done. The code was run with the command `./gameOfLife 1000 1000 0 num_of_threads` where the `num_of_threads` variable was varied. This experiment was executed on one of the University's Linux computers, due to the large amount of cores available on it (16 cores). More precisely, the used computer was an Intel® Xeon® E5520 @ 2.27GHz, with a total of 16 CPU:s (4 cores $\times$ 2 sockets, 2 threads per core), and compiler *gcc version*

7.5.0 (Ubuntu 7.5.0-3ubuntu1 18.04). The result of these measurements is presented in Figure 3. It can be noted that the time reduces in an $1/x$ like fashion and that when the number of threads exceeds the number of cores available, the time does not continue to decrease anymore. The lowest time measurement was 0.670 seconds which was when 16 threads were used.

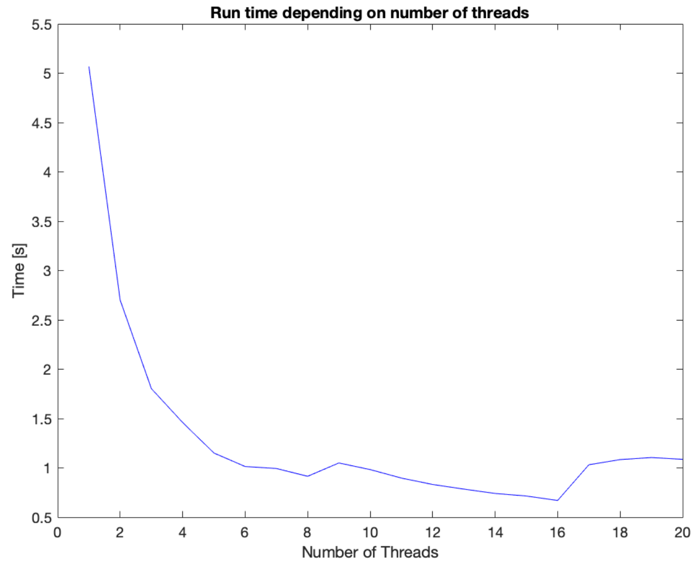

Figure 3: Run time of the program for different number of threads

5 Conclusions

In summary, the performance of the implemented Game of Life simulation has increased significantly after serial optimizations and parallelization of the code was made. Also, the time measurements when different numbers of threads were used is behaving as expected. However, the time complexity of the program was determined to be $\mathcal{O}(N^2)$ which means that the run time of the program will increase very fast as the grid size increases, which is not a preferred aspect of an algorithm. This time complexity is due to the fact that iterations over every cell in the grid is made in order to find the next state.

An approach that could be considered to improve the complexity of the algorithm could be to only count the number of neighbours of the alive cells in the grid. In order for this approach to work properly, all 3x3 neighbourhoods with exactly 3 alive cells must also be detected, so that the dead cells within that neighbourhood will be updated correctly. Since these situations only can occur in the presence of at least one alive cell however, only these potential areas would have to be searched instead of the whole grid. This algorithm might be effective for states with few alive cells in the grid (for example when specific seeds with few alive cells ought to be tried). However, for a randomized initial state as used in this implementation, this algorithm will most likely not result in any significant improvement of the time. Another thing that could be considered though, is to somehow keep track of which cells that were changed in the previous time step, so that only these areas of these cells of the grid must be examined. This approach might in some cases reduce the computations needed by the program, but depending on the initial state, it can not be ensured. In summary, it was hard to come up with an algorithm with a better time complexity than $\mathcal{O}(N^2)$, when also considering the worst case of the simulation. Therefore, the current implementation can be considered as an optimal solution of the problem.

The parallelization of the code was made using the `barrier` function, for the purpose of keeping the threads alive through all time steps instead of creating and terminating threads on every time step. This approach is highly preferable since creating and joining threads can take up a significant amount of the run time of a program if it has to be done often. When running the program for many time steps, the barrier implementation is therefore more efficient than it would have been if the other approach would have been used. Additionally, the work amount is evenly distributed to all the threads which is favorable for parallelized work since it minimizes the time spent on waiting for the last thread to finish its work (on every time step). Thus, the parallelized implementation made in this project can be considered as an optimal solution to the problem.

Regarding the precision of the experiments in the project, it might have been more beneficial to start every simulation with the same initial seed. This would have made sure that exactly the same computations were made during all runs of the program and it would also have been easier to determine the correctness of the simulation after changes were made to the code (since the grid would always look the same after a specific number of time steps). Now, a different evolution of the states were generated every time the code was run which could have had a small impact on the time. Perhaps, in this project it would have been better to take an average value of the 5 runs that were made for every measurement, instead of considering the fastest time as the final result.

When evaluating the serial optimizations that were made, another approach could have been to allocate the arrays on stack instead of calling `malloc`. This approach can sometimes give an improved performance, however it was not chosen in this project because it would set a limit on the size of the grid. If a too large grid size would be entered by the user, the code would generate a `segmentation fault` error due to the fact that not enough memory on stack would be available for the desired grid size. Although, in similar implementations if large grid sizes are not needed for the purpose of the program, it might be beneficial to allocate the arrays on stack instead.

Another serial optimization that was not tested, was to introduce loop unrolling manually (instead of using the compiler flag `-funroll-loops`). This was not done because it seemed relatively complicated since nearly all loops in the program were dependent on the input parameter `grid_size`. In order to handle the cases where the grid size was not divisible with the unroll factor, an additional loop would be necessary to compute the remaining values. For small unroll factors, this would most likely not have made a big difference in the performance because the loop condition for the remaining values would have to be checked anyways. However, if larger unroll factors were to be used, it might have given an improved performance if it was implemented.

Appendix A

Listing 1: *Original implementation*

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

void initGrid(int grid_size, int** grid);
void updateGrid(int grid_size, int** grid, int** temp_grid);
int countNeighbours(int grid_size, int** grid, int i, int j);
void printGrid(int grid_size, int** grid);

static double get_wall_seconds(){
  struct timeval tv;
  gettimeofday(&tv, NULL);
  double seconds = tv.tv_sec + (double)tv.tv_usec / 1000000;
  return seconds;
}

int main(int argc, char* argv[]){
    double time1 = get_wall_seconds(); // start time
    if (argc != 4){
        printf("Expected input: ./gameOfLife grid_size time_steps simulation\n");
        return -1;
    }
    int grid_size = atoi(argv[1]);
    int time_steps = atoi(argv[2]);
    int simulation = atoi(argv[3]); // 1 = simulation on, 0 = simulation off
    if(simulation != 0 && simulation != 1){
        printf("Error: Last input argument must be 1 or 0. \n");
        return -1;
    }

    // Initiate grids
    int** grid = (int**)malloc(grid_size*sizeof(int*)); // main grid
    for(int i = 0; i < grid_size; i++){
        grid[i] = (int*)malloc(grid_size*sizeof(int*));
    }
    initGrid(grid_size, grid); // fill grid with start seed

    int** temp_grid = (int**)malloc((grid_size+2)*sizeof(int*)); // "old" (padded) grid
    for(int i = 0; i < grid_size+2; i++){
        temp_grid[i] = (int*)malloc((grid_size+2)*sizeof(int*));
    }

    if(simulation == 1){ // Simulation ON
        printf("Start grid:\n");
        printGrid(grid_size, grid);

        // Update grid for every timestep
        for(int k = 0; k < time_steps; k++){
            // Make a deep copy of current grid and add padding to temp_grid
            for(int i = 0; i < grid_size+2; i++){
                for(int j = 0; j < grid_size+2; j++){
                    if(i < 1 || i > grid_size || j < 1 || j > grid_size){
                        temp_grid[i][j] = 0;
                    } else {
                        temp_grid[i][j] = grid[i-1][j-1];
                    }
                }
```

```c
            }

            updateGrid(grid_size, grid, temp_grid);

            printf("\nUpdated grid:\n");
            printGrid(grid_size, grid);
            nanosleep((const struct timespec[]){{0, 100000000L}}, NULL);
        }
    } else { // Simulation OFF

        // Update grid for every timestep
        for(int k = 0; k < time_steps; k++){
            // Make a deep copy of current grid and add padding to temp_grid
            for(int i = 0; i < grid_size+2; i++){
                for(int j = 0; j < grid_size+2; j++){
                    if(i < 1 || i > grid_size || j < 1 || j > grid_size){
                        temp_grid[i][j] = 0;
                    } else {
                        temp_grid[i][j] = grid[i-1][j-1];
                    }
                }
            }

            updateGrid(grid_size, grid, temp_grid);
        }
    }

    // Free allocated memory
    for(int i = 0; i < grid_size; i++){
        free(grid[i]);
    }
    free(grid);
    for(int i = 0; i < grid_size+2; i++){
        free(temp_grid[i]);
    }
    free(temp_grid);

    printf("gameOfLife took %7.3f wall seconds.\n", get_wall_seconds()-time1);
    return 0;
}

void initGrid(int grid_size, int** grid){
    srand(1);
    for(int i = 0; i < grid_size; i++){
        for(int j = 0; j < grid_size; j++){
            grid[i][j] = rand()%2; // 0 = dead, 1 = alive
        }
    }
}

void updateGrid(int grid_size, int** grid, int** temp_grid){
    // Compute new grid
    for(int i = 1; i ≤ grid_size; i++){
        for(int j = 1; j ≤ grid_size; j++){
            int neighbours = countNeighbours(grid_size+2, temp_grid, i, j);

            if(temp_grid[i][j] == 0 && neighbours == 3){
                grid[i-1][j-1] = 1;
            } else if(temp_grid[i][j] == 1 && neighbours == 2){
                grid[i-1][j-1] = 1;
            } else if(temp_grid[i][j] == 1 && neighbours == 3){
                grid[i-1][j-1] = 1;
            } else {
                grid[i-1][j-1] = 0;
            }
```

```
        }
    }
}

int countNeighbours(int grid_size, int** grid, int i, int j){
    return grid[i-1][j-1]+grid[i-1][j]+grid[i-1][j+1]+grid[i][j-1]+grid[i][j+1]+
    grid[i+1][j-1]+grid[i+1][j]+grid[i+1][j+1];
}

void printGrid(int grid_size, int** grid){
    char c;
    for(int i = 0; i < grid_size; i++){
        for(int j = 0; j < grid_size; j++){
            if(grid[i][j] == 1){ // alive
                c = '*';
            } else { // dead
                c = '.';
            }
            printf("%c ", c);
        }
        printf("\n");
    }
}
```

Appendix B

Listing 2: *Serial optimized implementation*

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

void initGrid(const int grid_size, short int** __restrict grid);
void updateGrid(const int grid_size, short int** __restrict grid, short int** __restrict ...
    temp_grid, int neighbours);
void printGrid(const int grid_size, short int** __restrict grid);

static double get_wall_seconds(){
  struct timeval tv;
  gettimeofday(&tv, NULL);
  double seconds = tv.tv_sec + (double)tv.tv_usec / 1000000;
  return seconds;
}

int main(int argc, char* argv[]){
    const double time1 = get_wall_seconds(); // start time
    if (argc != 4){
        printf("Expected input: ./gameOfLife grid_size time_steps simulation\n");
        return -1;
    }
    const int grid_size = atoi(argv[1]);
    const int time_steps = atoi(argv[2]);
    const int simulation = atoi(argv[3]); // 1 = simulation on, 0 = simulation off
    if(simulation != 0 && simulation != 1){
        printf("Error: Last input argument must be 1 or 0. \n");
        return -1;
    }
    int neighbours;

    // Initiate grids
    short int** buffer_rows = (short int**)malloc((grid_size+grid_size+2)*sizeof(short int*));
    short int* buffer_col = (short ...
        int*)malloc((grid_size*grid_size+(grid_size+2)*(grid_size+2))*sizeof(short int*));

    short int** grid = buffer_rows;
    for(int i = 0; i < grid_size; i++){
        grid[i] = (buffer_col + i*grid_size);
    }
    initGrid(grid_size, grid); // fill grid with start seed

    short int** temp_grid = (buffer_rows+grid_size); // "old" (padded) grid
    for(int i = 0; i < grid_size+2; i++){
        temp_grid[i] = (buffer_col + grid_size*grid_size + i*(grid_size+2));
    }

    if(simulation == 1){ // Simulation ON
        printf("Start grid:\n");
        printGrid(grid_size, grid);

        // Update grid for every timestep
        for(int k = 0; k < time_steps; k++){
            // Make a deep copy of current grid and add padding
            for(int i = 1; i ≤ grid_size; i++){ // fill middle of grid
                for(int j = 1; j ≤ grid_size; j++){
                    temp_grid[i][j] = grid[i-1][j-1];
```

```
                }
            }
            for(int j = 0; j < grid_size+2; j++){ // fill horizontal padding
                temp_grid[0][j] = 0;
                temp_grid[grid_size+1][j] = 0;
            }
            for(int i = 1; i ≤ grid_size; i++){ // fill vertical padding
                temp_grid[i][0] = 0;
                temp_grid[i][grid_size+1] = 0;
            }

            updateGrid(grid_size, grid, temp_grid, neighbours);

            printf("\nUpdated grid:\n");
            printGrid(grid_size, grid);
            nanosleep((const struct timespec[]){{0, 100000000L}}, NULL);
        }
    } else { // Simulation OFF

        // Update grid for every timestep
        for(int k = 0; k < time_steps; k++){
            // Make a deep copy of current grid and add padding
            for(int i = 1; i ≤ grid_size; i++){ // fill middle of grid
                for(int j = 1; j ≤ grid_size; j++){
                    temp_grid[i][j] = grid[i-1][j-1];
                }
            }
            for(int j = 0; j < grid_size+2; j++){ // fill horizontal padding
                temp_grid[0][j] = 0;
                temp_grid[grid_size+1][j] = 0;
            }
            for(int i = 1; i ≤ grid_size; i++){ // fill vertical padding
                temp_grid[i][0] = 0;
                temp_grid[i][grid_size+1] = 0;
            }

            updateGrid(grid_size, grid, temp_grid, neighbours);
        }
    }

    // Free allocated memory
    free(buffer_col);
    free(buffer_rows);

    printf("gameOfLife took %7.3f wall seconds.\n", get_wall_seconds()-time1);
    return 0;
}

void initGrid(const int grid_size, short int** __restrict grid){
    srand(1);
    for(int i = 0; i < grid_size; i++){
        for(int j = 0; j < grid_size; j++){
            grid[i][j] = rand()%2; // 0 = dead, 1 = alive
        }
    }
}

void updateGrid(const int grid_size, short int** __restrict grid, short int** __restrict ...
    temp_grid, int neighbours){
    // Compute new grid
    for(int i = 1; i ≤ grid_size; i++){
        for(int j = 1; j ≤ grid_size; j++){
            neighbours = temp_grid[i-1][j-1]+temp_grid[i-1][j]+temp_grid[i-1][j+1]+
            temp_grid[i][j-1]+temp_grid[i][j+1]+temp_grid[i+1][j-1]+temp_grid[i+1][j]+
            temp_grid[i+1][j+1];
```

```

            if(temp_grid[i][j] == 0 && neighbours == 3){
                grid[i-1][j-1] = 1;
            } else if(temp_grid[i][j] == 1 && (neighbours == 2 || neighbours == 3)){
                grid[i-1][j-1] = 1;
            } else {
                grid[i-1][j-1] = 0;
            }
        }
    }
}

void printGrid(const int grid_size, short int** __restrict grid){
    char c;
    for(int i = 0; i < grid_size; i++){
        for(int j = 0; j < grid_size; j++){
            if(grid[i][j] == 1){ // alive
                c = '*';
            } else { // dead
                c = '.';
            }
            printf("%c ", c);
        }
        printf("\n");
    }
}
```

Appendix C

Listing 3: *Parallelized implementation*

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#include <time.h>
#include <pthread.h>

// Structs
typedef struct thread_data {
  int start;
  int stop;

  int grid_size;
  int time_steps;
  int num_of_threads;
  int simulation;

  pthread_t thread;
} thread_data_t;

// Global variables
short int** grid;
short int** temp_grid;

pthread_mutex_t lock;
pthread_cond_t cv;
int state = 0;
int waiting = 0;

// Functions
void initGrid(const int grid_size, short int** __restrict grid);
void* updateGrid(void* arg);
void printGrid(const int grid_size, short int** __restrict grid);
void barrier(const int num_of_threads, const int grid_size, const int simulation);

static double get_wall_seconds(){
  struct timeval tv;
  gettimeofday(&tv, NULL);
  double seconds = tv.tv_sec + (double)tv.tv_usec / 1000000;
  return seconds;
}

int main(int argc, char* argv[]){
    const double time1 = get_wall_seconds(); // start time
    if (argc != 5){
        printf("Expected input: ./gameOfLife grid_size time_steps simulation ...
            num_of_threads\n");
        return -1;
    }
    const int grid_size = atoi(argv[1]);
    const int time_steps = atoi(argv[2]);
    const int simulation = atoi(argv[3]); // 1 = simulation on, 0 = simulation off
    if(simulation != 0 && simulation != 1){
        printf("Error: simulation input argument must be 1 or 0. \n");
        return -1;
    }
    const int num_of_threads = atoi(argv[4]);

    thread_data_t threads[num_of_threads];
```

```
    const int work_size = grid_size/num_of_threads;
    const int rest = grid_size%num_of_threads; // extra work amount for last thread

    // Initiate grids
    short int** buffer_rows = (short int**)malloc((grid_size+grid_size+2)*sizeof(short int*));
    short int* buffer_col = (short ...
        int*)malloc((grid_size*grid_size+(grid_size+2)*(grid_size+2))*sizeof(short int*));

    grid = buffer_rows;
    for(int i = 0; i < grid_size; i++){
        grid[i] = (buffer_col + i*grid_size);
    }
    initGrid(grid_size, grid); // fill grid with start seed

    temp_grid = (buffer_rows+grid_size); // "old" (padded) grid
    for(int i = 0; i < grid_size+2; i++){
        temp_grid[i] = (buffer_col + grid_size*grid_size + i*(grid_size+2));
    }

    // Make deep copy of grid to temp_grid and add padding on edges
    for(int i = 1; i <= grid_size; i++){ // fill middle of grid
        for(int j = 1; j <= grid_size; j++){
            temp_grid[i][j] = grid[i-1][j-1];
        }
    }
    for(int j = 0; j < grid_size+2; j++){ // fill horizontal padding
        temp_grid[0][j] = 0;
        temp_grid[grid_size+1][j] = 0;
    }
    for(int i = 1; i <= grid_size; i++){ // fill vertical padding
        temp_grid[i][0] = 0;
        temp_grid[i][grid_size+1] = 0;
    }

    // Initiate mutex and condition variables
    pthread_mutex_init(&lock, NULL);
    pthread_cond_init(&cv, NULL);

    // Create threads
    for(int k = 0; k < num_of_threads-1; k++){
        threads[k].start = (k+1) * work_size - work_size;
        threads[k].stop = threads[k].start + work_size;

        threads[k].grid_size = grid_size;
        threads[k].time_steps = time_steps;
        threads[k].num_of_threads = num_of_threads;
        threads[k].simulation = simulation;

        pthread_create(&(threads[k].thread), NULL, updateGrid, &threads[k]);
    }
    threads[num_of_threads-1].start = num_of_threads * work_size - work_size;
    threads[num_of_threads-1].stop = threads[num_of_threads-1].start + work_size + rest;
    threads[num_of_threads-1].grid_size = grid_size;
    threads[num_of_threads-1].time_steps = time_steps;
    threads[num_of_threads-1].num_of_threads = num_of_threads;
    threads[num_of_threads-1].simulation = simulation;
    pthread_create(&(threads[num_of_threads-1].thread), NULL, updateGrid, ...
        &threads[num_of_threads-1]);

    // Join threads
    for(int k = 0; k < num_of_threads; k++){
        pthread_join(threads[k].thread, NULL);
    }

    // Free allocated memory
```

```c
    free(buffer_col);
    free(buffer_rows);

    // Destroy mutex and condition variables
    pthread_mutex_destroy(&lock);
    pthread_cond_destroy(&cv);

    printf("gameOfLife took %7.3f wall seconds.\n", get_wall_seconds()-time1);
    return 0;
}

void initGrid(const int grid_size, short int** __restrict grid){
    srand(1);
    for(int i = 0; i < grid_size; i++){
        for(int j = 0; j < grid_size; j++){
            grid[i][j] = rand()%2; // 0 = dead, 1 = alive
        }
    }
}

void* updateGrid(void* arg){
    thread_data_t* info = (thread_data_t *) arg;
    int neighbours;

    for(int k = 0; k < info->time_steps; k++){ // For every time step
        barrier(info->num_of_threads, info->grid_size, info->simulation);

        // Update temp_grid with most recent values
        for(int i = info->start+1; i ≤ info->stop; i++){ // fill middle of grid
            for(int j = 1; j ≤ info->grid_size; j++){
                temp_grid[i][j] = grid[i-1][j-1];
            }
        }

        // Compute new grid
        for(int i = info->start+1; i ≤ info->stop; i++){ // specific rows
            for(int j = 1; j ≤ info->grid_size; j++){ // all columns
                neighbours = temp_grid[i-1][j-1]+temp_grid[i-1][j]+temp_grid[i-1][j+1]+
                temp_grid[i][j-1]+temp_grid[i][j+1]+temp_grid[i+1][j-1]+temp_grid[i+1][j]+
                temp_grid[i+1][j+1];

                if(temp_grid[i][j] == 0 && neighbours == 3){
                    grid[i-1][j-1] = 1;
                } else if(temp_grid[i][j] == 1 && (neighbours == 2 || neighbours == 3)){
                    grid[i-1][j-1] = 1;
                } else {
                    grid[i-1][j-1] = 0;
                }
            }
        }
    }
    return NULL;
}

void printGrid(const int grid_size, short int** __restrict grid){
    char c;
    for(int i = 0; i < grid_size; i++){
        for(int j = 0; j < grid_size; j++){
            if(grid[i][j] == 1){ // alive
                c = '*';
            } else { // dead
                c = '.';
            }
            printf("%c ", c);
        }
```

```c
        printf("\n");
    }
}

void barrier(const int num_of_threads, const int grid_size, const int simulation){
    int mystate;
    pthread_mutex_lock(&lock);
    mystate = state;
    waiting++; // Number of threads ready for next time step

    if(waiting == num_of_threads){ // All threads ready for next timestep
        waiting = 0;
        state = 1-mystate;

        if(simulation == 1){ // simulation enabled
            printf("Current state:\n");
            printGrid(grid_size, grid); // Print grid
            nanosleep((const struct timespec[]){{0, 100000000L}}, NULL);
        }

        pthread_cond_broadcast(&cv); // Wake up sleeping threads
    }
    while(mystate == state){ // Not all threads are done
        pthread_cond_wait(&cv, &lock); // Let thread sleep
    }
    pthread_mutex_unlock(&lock);
}
```