UPPSALA
UNIVERSITET

Individual Project

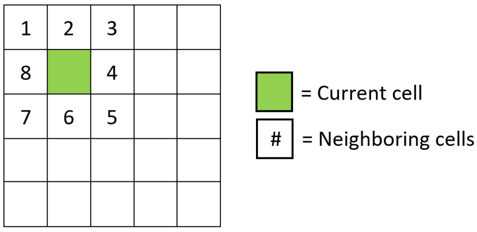
Aikaterini Manousidou
High Performance Programming
Uppsala University
VT 2021

12 mars 2021

Table of contents

1 Introduction

This report focuses on the implementation of the *Game of Life*. The *Game of Life* was created by John Horton Conway in 1970. This game does not require any players and it describes the evolution of a world of cells. The cells can either be live or dead and the evolution of the game depends on its initial state.[1]



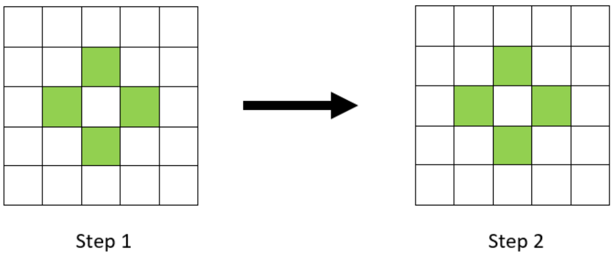
Figur 1: Schematic figure over a live cell and its neighboring cells.

Each cell has eight neighbors, as seen in figure 1. The evolution of the world could be divided into steps. The number of live neighbors along with the state of the cell, in other words if the cell is live or dead, determine the status the cell for the next step. For each step and each cell the following rules apply:

- If a live cell has fewer than 2 neighbors it dies by under-population.
- If a live cell has 2 or 3 neighbors it remains live.
- If a live cell has more than 3 neighbors it dies by over-population.
- If a dead cell has exactly 3 neighbors it becomes live by reproduction.[1]

These rules can then be simplified to the following set of rules:

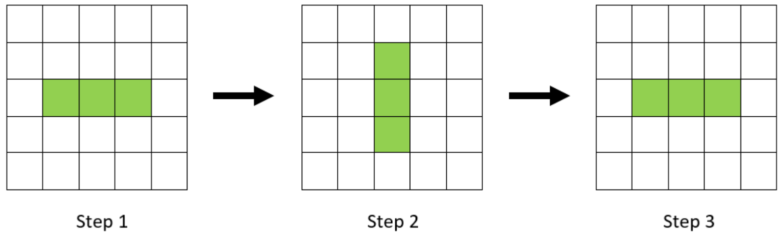
- Any live cell with 2 or 3 neighbors remains live.
- Any dead cell with 3 neighbors become live.
- All other live cells die and all other dead cells remain dead.[1]



Figur 2: Schematic figure over a stable group of live cells.

Each initial state can result in one of the following final states:

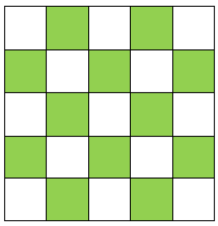
- **Stable state**: A group of cells has reached a stable pattern, see figure 2, where no matter how many steps are performed the specific group of live cells will not evolve further.
- **Alternating state:** A group of cells has reached an alternating pattern, see figure 3, where for each step the group of cells will alternate between two patterns.
- **Zero state**: All the cells are dead.



Figur 3: Schematic figure over an alternating group of live cells.

2 Problem Description

The purpose of this project is to implement a program which plays the *Game of Life*. The program should creates a world of cells and calculate the evolution of the world with each step by following the rules of the *Game of Life*. The game should continue until the world stabilizes either by reaching a stable, alternating or zero state. The program should take three integers as inputs, the size of the square world, N, the type of initial state and the type of output the user would like to have. Here are the various options for the initial state and the output:



Figur 4: Schematic figure over the chessboard pattern.

- **Initial state** = **0**: Random pattern.
- **Initial state** = **1**: Chessboard pattern, see figure 4.
- **Output** = **0**: Prints only the number of steps needed for the world to stabilize.
- **Output** = **1**: Prints the initial and the final states and the number of steps needed for the world to stabilize.
- **Output** = **2**: Prints the states of all steps, including the initial and the final steps, and the number of steps needed for the world to stabilize.

3 Solution Method

3.1 The Un-optimized code

3.1.1 The sub-functions

In order for the code to work some necessary sub-functions were introduced and are listed bellow:

- `print_world`: This function prints the world by looping over all the rows and then all the columns. The function takes two inputs, the size of the world and a pointer to the world's location.
- `count_neighbors`: This function counts the amount of live neighboring cells for each cell in the world. It goes through the corners, the sides and then the interior of the world by looping through the rows and the columns. This function takes three inputs, the size of the world, a pointer to the world's location and a pointer to the location of the matrix of the neighbors.
- `next_step`: This function predicts the next state of the world by checking the fulfilment of the rules of the *Game of Life* as mentioned in section 1. It loops through the rows and the columns of the world controls each cell. It takes four inputs, the size of the world and pointers to the matrices of the world, the neighbors and the world's next step.
- `update`: This function updates the world by looping through the rows and the columns and setting each element of the current state equal to the element of the next state. It takes three inputs, the size of the world and pointers to the matrices of the world's current and next states.
- `compare`: This function compares the current state with the next state, the current state with zero and the previous state with the state after the next state. This way, it can be determined if the world has reached one of the three types of stability mentioned in section 1.

    1. If the current state is exactly the same with the next state, the world has reached a pattern of live cells that cannot evolve any further.
    2. If the current state is filled with zeros, then the world is empty and therefore cannot evolve any further.
    3. If the previous state is exactly the same as the state after the next state (two states forward), the world has reached a pattern that alternates between two phases and therefore will not evolve into a new state.

    The function then counts through all the elements that are the same for each of the three cases. If the counter for one of the cases is equal to the amount of elements then the status of the world turns to zero.

3.1.2 Summary of the Un-optimized code

The un-optimized code can be seen in section 5.1. The algorithm starts by controlling that enough input arguments are given when running the program. These are the size of the world, the initial state pattern and the type of output. The program moves on to allocating enough memory for each of all the necessary matrices for the program to work. Then, the initial state pattern is set into the matrix. The matrix that describes the previous state of the world is then updated.

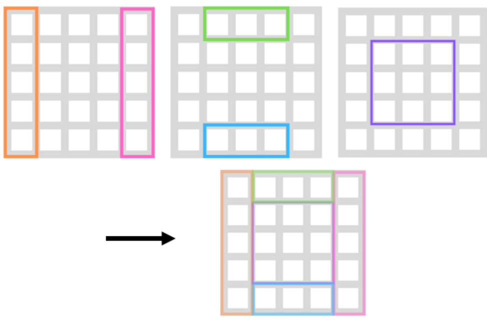
Thereafter, the main while-loop is initiated where with the help of the `count_neighbors` and `next_step` functions two steps of the world are predicted. The matrices for the previous, the current and the two new steps are all compared with the help of the `compare` function and the status is updated. The world is then updated with one step with the help of the `update` function. After that, the matrices describing the amount of neighbors for each cell in both steps are set to zero. Following, the main function prints the current state of the world, if it is required, and the previous state of the world is also updated. The while-loop continues as long as the status is set to one.

When the while-loop is broken the total amount of steps needed to stabilize the world is printed. The final state of the world is also printed if it is required. At last, all the memory is freed and the program ends.

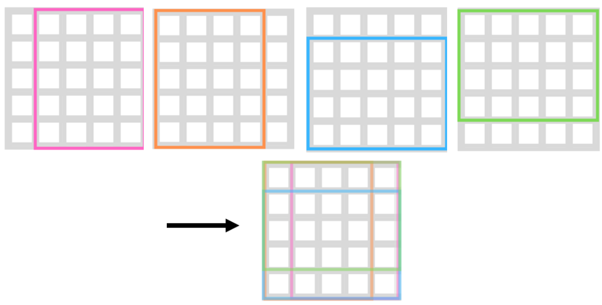
3.2 The Optimized code

3.2.1 The sub-functions

Most of the sub-functions remained the same. However, the `count_neighbors` function was simplified significantly. The function still looped over all the rows and columns, but the if-statements were generalised, see figures 5 and 6. For example, instead of checking if the index were equal to the minimum or the maximum of the interval the condition was altered to check if the index was smaller than the maximum or larger than the minimum. That way, a larger portion of the matrix was covered with less if statements



Figur 5: Schematic figure over the indexing in the `count_neighbors` function before optimization.



Figur 6: Schematic figure over the indexing in the `count_neighbors` function after optimization.

3.2.2 Summary of the Optimized code

The optimized code, as seen in section 5.2, has the same structure as the un-optimized code, however some thing could be improved. One of the improvements was that instead of allocating memory for each individual matrix, memory was allocated to a buffer. Pointers were then used to point to the different parts of the buffer which described the different matrices. Another improvement was that all integer parameters which did not vary during the course of the program were changed from `int` to `const int`. Also loop-unrolling was introduced when resetting the matrices for the neighbors to zero.

A major improvement was that since the world's next two states were calculated in each loop, then the current world could be updated with two steps at a time. This meant that the execution time for the while-loop was expected to be half as long as before.

3.3 The Parallel Code

The parallel code can be seen in section 5.3. The code is divided into three functions, the main function, the thread function and a print function. All operations on the world are performed in parallel, thus the main function has been simplified significantly.

The code starts once again by checking the input arguments. It is worth mentioning that the option to see all steps of the algorithm had to be discarded because it over-complicated the algorithm. A number of variables had to become global variables in order to be reachable by all the threads, these include the size of the world, the number of threads and the pointers to all the different matrices. Two additional global variables are introduced, an integer for the amount of threads that were and a barrier variable, the function of whom will be explained later. The necessary information for each thread is passed with the help of a structure called `t_info`. The content of the structures will be discussed in the next section.

The program moves on to allocating enough memory for the necessary matrices and setting the initial state pattern. The threads, the structures and the barrier are then initialized. Thereafter, the structures are filled with the necessary information. The threads are created in a for-loop and in a separate for-loop the threads are joined. The number of steps needed for the world to stabilize is printed and the allocated memory is freed.

3.3.1 The thread function

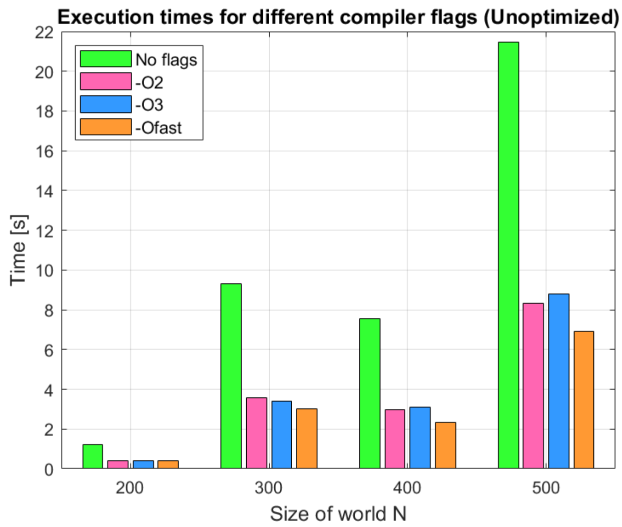
To be able to parallelize the optimized code a thread function had to be introduced. In the thread function, each thread performs the same type of work as in the serial code, that meant to count the neighbors, calculate one step forward, count the neighbors again, calculate two steps forward, compare the matrices, update the world and so on. However, in this case each thread is responsible for a specific amount of rows. In other words, the world's rows are divided among the threads and each thread performs operations only on those specific rows.

The information of which rows each thread is responsible for is passed through a structure called `t_info`, this means the lower end and the higher end of the interval. The structure also includes information about the thread's index and the amount of for-loops or steps each thread performed. In the serial code, the `compare` function counted the amount of cell that were the same in each of the cases and broke the while-loop once all that number reached the total number of elements of the world. In order for that to be performed in parallel, each thread counts the equal number of elements in the rows it is responsible for. When that number reaches the total amount of element of these rows the thread is added into the threads that are done. Thereafter, when the number of done threads is equal to the total number of threads the for-loop is broken and the program terminates as per usual.

For the algorithm to function correctly, the threads had to be synchronised. Thus three barriers had to be introduced. The barriers are placed before each thread starts counting neighbors and before the if-statement that checks if all threads are done or not. This means that the threads would not be able to continue past the barrier unless all the threads had reached the barrier, thus the threads are synchronized.

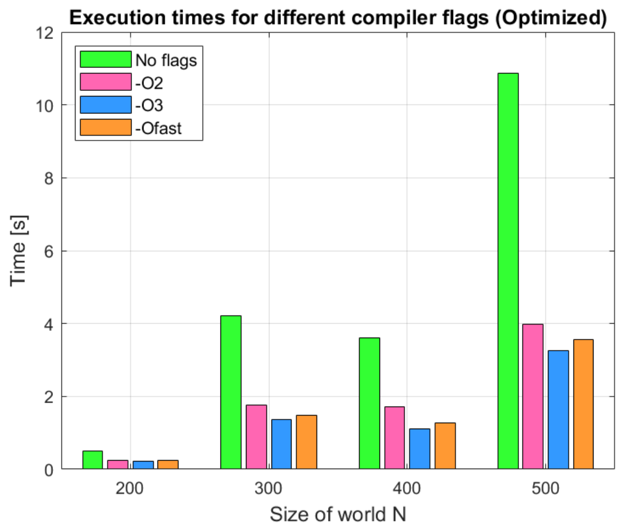
4 Experiments

In order to be able to compare the results of the different implementations a chessboard pattern was introduced, as previously mentioned. Thus, the number of steps for the world to stabilize were consistent for a specific size. This allowed for the execution time to be measured for each run. Each of the codes mentioned are tested with the compiler flags `-O2`, `-O3`, `-Ofast` but also with no compiler flags. These tests were performed on a Intel(R) Core(TM) i5-8265U processor, which has four cores. The Matlab code which generates the figures listed bellow can be found in section 5.4.



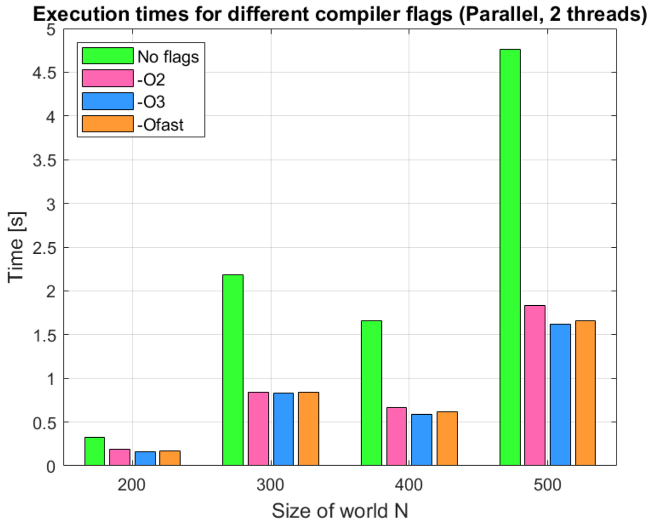
Figur 7: Bar plot over the execution time of the un-optimized code with the different compiler flags for different sizes.

In figure 7, the execution times for different compiler flags and world sizes for the unoptimized code can be seen. From that figure, it is obvious that the unoptimized code is reduced by half when using a compiler flag, however the code performs best when using the `-Ofast` flag.



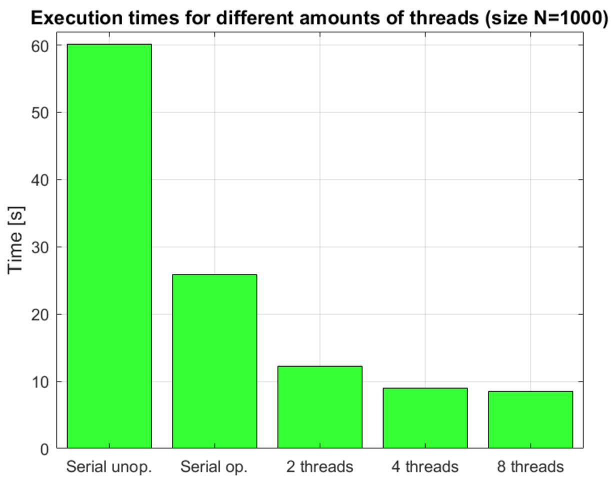
Figur 8: Bar plot over the execution time of the optimized code with the different compiler flags for different sizes.

When testing the optimized code, figure 8 was produced. From that figure it can be concluded that the execution time was once again reduced by almost a half when using a compiler flag. However, the best performance was reached when using the `-O2` flag this time. By comparing figures 7 and 8, it is apparent that even when no compiler flag is used, the optimized code performs double as fast as the unoptimized one. This can only be explained by the world being updated with two steps in each loop and the `count_neighbors` function being simplified, resulting in the execution time being cut in half.



Figur 9: Bar plot over the execution time of the parallel code with two threads with the different compiler flags for different sizes.

In order to determine which compiler flag suited the parallel code the best, the code was run with two threads for different sizes and compiler flags. The results can be seen in figure 9. Once again, the compiler flags were able to optimize the parallel code and resulted in half as long execution times. Similarly to the serially optimized code, the compiler flag that resulted in the shortest execution times was `-O2`. Nonetheless, the difference in execution times between the compiler flags was not as critical. By comparing figures 8 and 9, it is determined that the execution time is reduced by half when running the parallel code. This was expected since the workload was divided into two threads performing operations in parallel, thus working double as fast.



Figur 10: Bar plot over the execution time of the parallel code with different amounts of threads with the `-O3` compiler flag and world size $N = 1000$.

Finally, a study comparing all three produced algorithms was performed. The case chosen for this study was for a size of $N = 1000$ and with the `-O3` compiler flag. The parallel code was tested with two, four and eight threads in order to determine the amount of threads which results into the highest performance. The results of this study can be found in figure 10. The CPU usage reached 189% for two threads, 322% for four threads and 586% for eight threads. However, by analyzing the figure it is obvious that the execution time for eight threads is not significantly when compared to the execution time for four threads. This concludes that for the specific processor using four threads results in the best performance possible without unnecessarily over-working the processor. This was expected since the specific processor has four cores in total. A total 85% reduction of the execution time was achieved through optimizing and paralllelizing the original algorithm.

5 Conclusions

In this report three algorithms implementing the Conway's *Game of Life* were developed, an un-optimized one, an optimized one and a parallel one. The three algorithms were tested with different problem sizes and compiler flags.

The main difference between the un-optimized and the optimized algorithms was that the amount of total time loops was reduced by updating the world with two steps at a time and by simplifying the if-statements needed to count the neighboring live cells for each element. The main difference between the optimized and the parallel algorithms was that the total workload was divided into a number of threads which performed operations independently. However, the parallel code needed to be synchronized in order for the results to be accurate. This meant introducing the use of barriers in several positions in the code.

The algorithms were tested and compared by introducing a chessboard pattern as an initial state, however the algorithms can also be run with a randomized initial pattern. Overall, it was determined that the compiler flags were able to further optimize the algorithms and in each case reduce the execution times by approximately a half compared to not using compiler flags. The best performance for a world size of 1000 was achieved when running the parallel code with four threads and using the `-O3` compiler flag. This was expected because the algorithm was run on a quad-core processor. With these specifications, the parallel code accomplished an 85% reduction of the execution time in comparison to the unoptimized code. It is worth mentioning that the parallel code was not optimized further due to lack of time. Thus, it is unclear if this is the most optimal algorithm for this specific problem.

5.1 Un-optimized code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

void print_world(int N, int **world);
void count_neighbors(int N, int **world, int **neighbors);
void next_step(int N, int **world, int **one_step, int **neighbors);
void update(int N, int **world, int **one_step);
int compare(int N, int **world, int **one_step, int **two_steps, int **old, int status);


int main(int argc, const char *argv[])
{
    if (argc != 4)
    {
        printf("Give the following input arguments:\n");
        printf("N: Size of the NxN world (integer)\n");
        printf("Initial state: random (0), chessboard (1)\n");
        printf("Output: Number of steps until final state (0) \n");
        printf("        Number of steps until final state, initial and final states (1) \n");
        printf("        Number of steps until final state and all states states (2) \n");
        exit(0);
    }

    int N = atoi(argv[1]);
    int pattern = atoi(argv[2]);
    int output = atoi(argv[3]);

    // Create necessary matrices
    int **world = (int **)malloc(N * sizeof(int *));
    int **neighbors = (int **)malloc(N * sizeof(int *));
    int **neighbors_2 = (int **)malloc(N * sizeof(int *));
    int **one_step = (int **)malloc(N * sizeof(int *));
    int **old = (int **)malloc(N * sizeof(int *));
    int **two_steps = (int **)malloc(N * sizeof(int *));
    for (int i = 0; i < N; i++)
    {
        world[i] = (int *)malloc(N * sizeof(int));
        neighbors[i] = (int *)malloc(N * sizeof(int));
        neighbors_2[i] = (int *)malloc(N * sizeof(int));
        one_step[i] = (int *)malloc(N * sizeof(int));
        old[i] = (int *)malloc(N * sizeof(int));
        two_steps[i] = (int *)malloc(N * sizeof(int));
    }

    // Setting a random initial pattern
    if(pattern == 0){
        srand(time(0));
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < N; j++)
            {
                int r = rand() % 10;
                if (r > 5)
                    world[i][j] = 1;
                else
                    world[i][j] = 0;
            }
        }
    }
    // Setting a chessboard initial state
    else if(pattern == 1){
        for (int i = 0; i < N; i++)
        {
```

```
            for (int j = 0; j < N; j++)
            {
                if(i%2 == 0){
                    if(j%2 == 0)
                        world[i][j] = 0;
                    else
                        world[i][j] = 1;
                }
                else{
                    if(j%2 == 0)
                        world[i][j] = 1;
                    else
                        world[i][j] = 0;
                }
            }
        }
    }

    if(output==1 || output==2){
        printf("Initial state:\n");
        print_world(N, world);
    }

    int status = 1;
    int t = 1;
    update(N, old, world);

    while(status == 1)
    {
        // Predict one step forward
        count_neighbors(N, world, neighbors);
        next_step(N, world, one_step, neighbors);

        // Predict two steps forward
        count_neighbors(N, one_step, neighbors_2);
        next_step(N, one_step, two_steps, neighbors_2);

        // Compare all predicted steps
        status = compare(N, world, one_step, two_steps, old, status);

        // Update world with one step
        update(N, world, one_step);


        for(int i = 0; i < N; i++)
        {
            for(int j = 0; j < N; j++)
            {
                neighbors[i][j] = 0;
                neighbors_2[i][j] = 0;
            }
        }

        if((output == 2) && (status == 1)){
            printf("Step %d:\n", t);
            print_world(N, world);
        }

        // Save previous step
        update(N, old, world);
        t +=1;
    }

    printf("It took %d steps to reach the final state\n", t-2);
    if(output==1 || output ==2){
        printf("Final state:\n");
        print_world(N, world);
    }

    for (int i = 0; i < N; i++)
```

```
    {
        free(world[i]);
        free(neighbors[i]);
        free(neighbors_2[i]);
        free(one_step[i]);
        free(two_steps[i]);
        free(old[i]);
    }
    free(world);
    free(neighbors);
    free(neighbors_2);
    free(one_step);
    free(two_steps);
    free(old);
}

void print_world(int N, int **world)
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            printf("%d ", world[i][j]);
        printf("\n");
    }
    printf("\n");
}

void count_neighbors(int N, int **world, int **neighbors)
{
    int i; //rows
    int j; //col
    for (i = 0; i <= N-1; i++){
        for (j = 0; j <= N-1; j++){
            if(i == 0){
                // Point (0,0)
                if(j == 0){
                    if (world[i][j+1] == 1)
                        neighbors[i][j] +=1;
                    if(world[i+1][j+1] == 1)
                        neighbors[i][j] +=1;
                    if(world[i+1][j] == 1)
                        neighbors[i][j] +=1;
                }
                // Point (0,N-1)
                else if (j == N-1){
                    if (world[i][j-1] == 1)
                        neighbors[i][j] +=1;
                    if(world[i+1][j-1] == 1)
                        neighbors[i][j] +=1;
                    if(world[i+1][j] == 1)
                        neighbors[i][j] +=1;
                }
                // Points between (0,0)-(0,N-1)
                else if ((j > 0) && (j < N-1)){
                    if (world[i][j-1] == 1)
                        neighbors[i][j] +=1;
                    if(world[i+1][j-1] == 1)
                        neighbors[i][j] +=1;
                    if(world[i+1][j] == 1)
                        neighbors[i][j] +=1;
                    if(world[i+1][j+1] == 1)
                        neighbors[i][j] +=1;
                    if(world[i][j+1] == 1)
                        neighbors[i][j] +=1;
                }
            }
            if(i == N-1){
                // Point (N-1,0)
                if(j == 0){
                    if (world[i-1][j] == 1)
```

```
                        neighbors[i][j] +=1;
                    if(world[i-1][j+1] == 1)
                        neighbors[i][j] +=1;
                    if(world[i][j+1] == 1)
                        neighbors[i][j] +=1;
                }
                // Point (N-1,N-1)
                else if (j == N-1){
                    if (world[i-1][j-1] == 1)
                        neighbors[i][j] +=1;
                    if(world[i-1][j] == 1)
                        neighbors[i][j] +=1;
                    if(world[i][j-1] == 1)
                        neighbors[i][j] +=1;
                }
                // Points between (N-1,0)-(N-1,N-1)
                else if ((j > 0) && (j < N-1)){
                    if (world[i][j-1] == 1)
                        neighbors[i][j] +=1;
                    if(world[i-1][j-1] == 1)
                        neighbors[i][j] +=1;
                    if(world[i-1][j] == 1)
                        neighbors[i][j] +=1;
                    if(world[i-1][j+1] == 1)
                        neighbors[i][j] +=1;
                    if(world[i][j+1] == 1)
                        neighbors[i][j] +=1;
                }
            }
            // Left side
            if(j == 0){
                if ((i != 0) && (i!=N-1)){
                    if (world[i-1][j] == 1)
                        neighbors[i][j] +=1;
                    if(world[i-1][j+1] == 1)
                        neighbors[i][j] +=1;
                    if(world[i][j+1] == 1)
                        neighbors[i][j] +=1;
                    if(world[i+1][j+1] == 1)
                        neighbors[i][j] +=1;
                    if(world[i+1][j] == 1)
                        neighbors[i][j] +=1;
                }
            }
            // Right side
            if(j == N-1){
                if((i > 0) && (i < N-1)){
                    if (world[i-1][j] == 1)
                        neighbors[i][j] +=1;
                    if(world[i-1][j-1] == 1)
                        neighbors[i][j] +=1;
                    if(world[i][j-1] == 1)
                        neighbors[i][j] +=1;
                    if(world[i+1][j-1] == 1)
                        neighbors[i][j] +=1;
                    if(world[i+1][j] == 1)
                        neighbors[i][j] +=1;
                }
            }
            // Interior points
            if((i > 0) && (i < N-1) && (j > 0) && (j < N-1)){
                if (world[i-1][j-1] == 1)
                    neighbors[i][j] +=1;
                if(world[i-1][j] == 1)
                    neighbors[i][j] +=1;
                if(world[i-1][j+1] == 1)
                    neighbors[i][j] +=1;
                if(world[i][j+1] == 1)
                    neighbors[i][j] +=1;
                if(world[i+1][j+1] == 1)
```

```
                    neighbors[i][j] +=1;
                if(world[i+1][j] == 1)
                    neighbors[i][j] +=1;
                if(world[i+1][j-1] == 1)
                    neighbors[i][j] +=1;
                if(world[i][j-1] == 1)
                    neighbors[i][j] +=1;
            }
        }
    }
}

void next_step(int N, int **world, int **one_step, int **neighbors)
{
    int i, j;
    for (i = 0; i < N; i++){
        for (j = 0; j < N; j++){
            if (world[i][j] == 1)
            {
                if (neighbors[i][j] == 2 || neighbors[i][j] == 3)
                    one_step[i][j] = 1;
                else
                    one_step[i][j] = 0;
            }
            else if (world[i][j] == 0)
            {
                if (neighbors[i][j] == 3)
                    one_step[i][j] = 1;
                else
                    one_step[i][j] = 0;
            }
        }
    }
}

void update(int N, int **world, int **one_step)
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            world[i][j] = one_step[i][j];
    }
}

int compare(int N, int **world, int **one_step, int **two_steps, int **old, int
    status)
{
    int counter1=0, counter2=0, counter3=0;
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            if(world[i][j] == one_step[i][j])
                counter1++;
            if(world[i][j] == 0)
                counter2++;
            if(old[i][j] == two_steps[i][j])
                counter3++;
        }
    }
    if (counter1 == (N*N))
        status = 0;
    else if(counter2 == (N*N))
        status = 0;
    else if(counter3 == (N*N))
        status = 0;
    return status;
}
```

5.2 Optimized code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

void print_world(int N, int **world);
void count_neighbors(int N, int **world, int **neighbors);
void next_step(int N, int **world, int **one_step, int **neighbors);
void update(int N, int **world, int **one_step);
int compare(int N, int **world, int **one_step, int **two_steps, int **old, int
    status);

/*  Optimization techniques:
    In main function:
    Buffer for all memory allocation
    Changed int to const int (where possible)
    Loop unrolling when resetting the neighbor matrices
    Update the world with two steps in one iteration

    Simplified count_neighbors function
*/

int main(int argc, const char *argv[])
{
    if (argc != 4)
    {
        printf("Give the following input arguments:\n");
        printf("N: Size of the NxN world (integer)\n");
        printf("Initial state: random (0), chessboard (1)\n");
        printf("Output: Number of steps until final state (0) \n");
        printf("        Number of steps until final state, initial and final states
 (1) \n");
        printf("        Number of steps until final state and all states states (2)
 \n");
        exit(0);
    }

    const int N = atoi(argv[1]);
    const int pattern = atoi(argv[2]);
    const int output = atoi(argv[3]);

    // Create necessary matrices
    const int n = N+1;
    int **buffer = (int **)malloc(6 * n * sizeof(int *));
    for(int i = 0; i < (6*n); i++)
    {
        buffer[i] = (int *)malloc(n*sizeof(int));
    }

    int **world = &buffer[0];
    int **neighbors = &buffer[n];
    int **neighbors_2 = &buffer[2*n];
    int **one_step = &buffer[3*n];
    int **two_steps = &buffer[4*n];
    int **old = &buffer[5*n];

// Setting a random initial pattern
    if(pattern == 0){
        srand(time(0));
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < N; j++)
            {
                int r = rand() % 10;
                if (r > 5)
                    world[i][j] = 1;
                else
                    world[i][j] = 0;
```

```
            }
        }
    }
    // Setting a chessboard initial state
    else if(pattern == 1){
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < N; j++)
            {
                if(i%2 == 0){
                    if(j%2 == 0)
                        world[i][j] = 0;
                    else
                        world[i][j] = 1;
                }
                else{
                    if(j%2 == 0)
                        world[i][j] = 1;
                    else
                        world[i][j] = 0;
                }
            }
        }
    }

    if(output==1 || output==2){
        printf("Initial state:\n");
        print_world(N, world);
    }

    int status = 1;
    int t = 1;
    update(N, old, world);

    while(status == 1)
    {
        // Predict one step forward
        count_neighbors(N, world, neighbors);
        next_step(N, world, one_step, neighbors);

        // Predict two steps forward
        count_neighbors(N, one_step, neighbors_2);
        next_step(N, one_step, two_steps, neighbors_2);

        // Compare all predicted steps
        status = compare(N, world, one_step, two_steps, old, status);

        // Update world with two steps
        update(N, world, two_steps);

        for(int i = 0; i < N; i++)
        {
            for(int j = 0; j < N; j+=2)
            {
                neighbors[i][j] = 0;
                neighbors[i][j+1] = 0;
                neighbors_2[i][j] = 0;
                neighbors_2[i][j+1] = 0;
            }
        }

        if((output == 2) && (status == 1)){
            printf("Step %d:\n", t);
            print_world(N, one_step);
            printf("Step %d:\n", t+1);
            print_world(N, two_steps);
        }

        // Save previous step
        update(N, old, world);
```

```
        t+=2;
    }

    printf("It took %d steps to reach the final state\n", (t-3));
    if(output==1 || output ==2){
        printf("Final state:\n");
        print_world(N, world);
    }

    for (int i = 0; i < (6*n); i++)
    {
        free(buffer[i]);
    }
    free(buffer);
}

void print_world(int N, int **world)
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            printf("%d ", world[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}

void count_neighbors(int N, int **world, int **neighbors)
{
    int i; //rows
    int j; //col
    for (i = 0; i <= N-1; i++){
        for (j = 0; j <= N-1; j++){
            if (i > 0){
                if (j > 0){
                    if (world[i-1][j-1] == 1)
                        neighbors[i][j] +=1;
                }
                if (j < N-1){
                    if (world[i-1][j+1] == 1)
                        neighbors[i][j] +=1;
                }
                if (world[i-1][j] == 1)
                        neighbors[i][j] +=1;
            }
            if (i < N-1){
                if (j > 0){
                    if (world[i+1][j-1] == 1)
                        neighbors[i][j] +=1;
                }
                if (j < N-1){
                    if (world[i+1][j+1] == 1)
                        neighbors[i][j] +=1;
                }
                if (world[i+1][j] == 1)
                        neighbors[i][j] +=1;
            }
            if (j > 0){
                if (world[i][j-1] == 1)
                        neighbors[i][j] +=1;
            }
            if(j < N-1){
                if (world[i][j+1] == 1)
                        neighbors[i][j] +=1;
            }
        }
    }
}
```

```
void next_step(int N, int **world, int **one_step, int **neighbors)
{
    int i, j;
    for (i = 0; i < N; i++){
        for (j = 0; j < N; j++){
            if (world[i][j] == 1)
            {
                if (neighbors[i][j] == 2 || neighbors[i][j] == 3)
                    one_step[i][j] = 1;
                else
                    one_step[i][j] = 0;
            }
            else if (world[i][j] == 0)
            {
                if (neighbors[i][j] == 3)
                    one_step[i][j] = 1;
                else
                    one_step[i][j] = 0;
            }
        }
    }
}

void update(int N, int **world, int **one_step)
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            world[i][j] = one_step[i][j];
        }

    }
}

int compare(int N, int **world, int **one_step, int **two_steps, int **old, int
    status)
{
    int counter1=0, counter2=0, counter3=0;
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            if(world[i][j] == one_step[i][j])
                counter1++;
            if(world[i][j] == 0)
                counter2++;
            if(old[i][j] == two_steps[i][j])
                counter3++;
        }
    }
    if (counter1 == (N*N))
        status = 0;
    else if(counter2 == (N*N))
        status = 0;
    else if(counter3 == (N*N))
        status = 0;
    return status;
}
```

5.3 Parallel code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <pthread.h>

/*  Parallel code with PTHREADS */

//Global variables
int N;                          // Size of the world
int nthreads;                   // Number of threads
int **world;                    // pointer to the world
int **neighbors;                // pointer to the neighbors matrix
int **neighbors_2;              // pointer to the neighbors matrix
int **old;                      // pointer to the old matrix
int **one_step;                 // pointer to the old matrix
int **two_steps;                // pointer to the old matrix
int done;                       // nr of threads that are done
pthread_barrier_t barrier;      // pthread barrier

typedef struct info_thread
{
    int threadID;           // threadID
    int low;                // lower limit of interval
    int high;               // higher limit of interval
    int status;             // status for when the for loop should stop
    int timesteps;          // maximum amount of time-steps
    int steps;              // nr of performed time-steps
}t_info;

void * thread_func(void *arg);
void print_world(int **world);

int main(int argc, const char *argv[])
{
    if (argc != 5)
    {
        printf("Give the following input arguments:\n");
        printf("N: Size of the NxN world (integer)\n");
        printf("Initial state: random (0), chessboard (1)\n");
        printf("Output: Number of steps until final state (0) \n");
        printf("        Number of steps until final state, initial and final states
     (1) \n");
        printf("Threads: Number of threads (integer)\n");
        exit(0);
    }

    N = atoi(argv[1]);
    const int pattern = atoi(argv[2]);
    const int output = atoi(argv[3]);
    nthreads = atoi(argv[4]);

    // Create necessary matrices
    const int n = N+1;
    int **buffer = (int **)malloc(6 * n * sizeof(int *));
    for(int i = 0; i < (6*n); i++)
    {
        buffer[i] = (int *)malloc(n*sizeof(int));
    }

    world = &buffer[0];
    neighbors = &buffer[n];
    neighbors_2 = &buffer[2*n];
    one_step = &buffer[3*n];
    two_steps = &buffer[4*n];
    old = &buffer[5*n];

    // Setting a random initial pattern
```

```c
    if(pattern == 0){
        srand(time(0));
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < N; j++)
            {
                int r = rand() % 10;
                if (r > 5)
                    world[i][j] = 1;
                else
                    world[i][j] = 0;
            }
        }
    }
    // Setting a chessboard initial state
    else if(pattern == 1){
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < N; j++)
            {
                if(i%2 == 0){
                    if(j%2 == 0)
                        world[i][j] = 0;
                    else
                        world[i][j] = 1;
                }
                else{
                    if(j%2 == 0)
                        world[i][j] = 1;
                    else
                        world[i][j] = 0;
                }
            }
        }
    }

    if(output==1){
        printf("Initial state:\n");
        print_world(world);
    }

    // Create threads and thread info
    pthread_t thread[nthreads];
    t_info threadinfo[nthreads];
    pthread_barrier_init (&barrier, NULL, nthreads);
    const int interval = N/nthreads;

    for (int k=0; k<nthreads; k++)
    {
        threadinfo[k].threadID = k;
        threadinfo[k].low = k*interval;
        threadinfo[k].high = (k+1)*interval-1;
        threadinfo[k].status = 1;
        threadinfo[k].timesteps = 10000;
        threadinfo[k].steps = 0;
    }
    threadinfo[nthreads-1].high = N-1;
    done = 0;

    // Run threads
    for (int k=0; k<nthreads; k++)
        pthread_create(&thread[k], NULL, thread_func, (void *)&threadinfo[k]);
    for (int k=0; k<nthreads; k++)
        pthread_join(thread[k],NULL);

    if(threadinfo[nthreads-1].steps != threadinfo[nthreads-1].timesteps)
    {
        printf("It took %d steps to reach the final state\n", threadinfo[nthreads
    -1].steps);
        if(output==1){
```

```c
            printf("Final state:\n");
            print_world(world);
        }
    }
    else
    {
        printf("The maximum amount of iterations has been reach.\n");
        printf("The problem couldn't be solved.\n");
    }

    // Free allocated memory
    for (int i = 0; i < (6*n); i++)
    {
        free(buffer[i]);
    }
    free(buffer);
}

void * thread_func(void *arg)
{
    t_info *threadinfo = arg;
    int low = threadinfo->low;
    int high = threadinfo->high;

    for (int t=1; t < threadinfo->timesteps; t++)
    {
        pthread_barrier_wait(&barrier);

        for (int i = low; i <= high; i++){
            for (int j = 0; j <= N-1; j++){
                if (i > 0){
                    if (j > 0){
                        if (world[i-1][j-1] == 1)
                            neighbors[i][j] +=1;
                    }
                    if (j < N-1){
                        if (world[i-1][j+1] == 1)
                            neighbors[i][j] +=1;
                    }
                    if (world[i-1][j] == 1)
                        neighbors[i][j] +=1;
                }
                if (i < N-1){
                    if (j > 0){
                        if (world[i+1][j-1] == 1)
                            neighbors[i][j] +=1;
                    }
                    if (j < N-1){
                        if (world[i+1][j+1] == 1)
                            neighbors[i][j] +=1;
                    }
                    if (world[i+1][j] == 1)
                        neighbors[i][j] +=1;
                }
                if (j > 0){
                    if (world[i][j-1] == 1)
                        neighbors[i][j] +=1;
                }
                if(j < N-1){
                    if (world[i][j+1] == 1)
                        neighbors[i][j] +=1;
                }
            }
        }

        for (int i = low; i <= high; i++){
            for (int j = 0; j <= N-1; j++){
                if (world[i][j] == 1)
                {
                    if (neighbors[i][j] == 2 || neighbors[i][j] == 3)
```

```
                        one_step[i][j] = 1;
                    else
                        one_step[i][j] = 0;
                }
                else if (world[i][j] == 0)
                {
                    if (neighbors[i][j] == 3)
                        one_step[i][j] = 1;
                    else
                        one_step[i][j] = 0;
                }
            }
        }

        pthread_barrier_wait(&barrier);

        for (int i = low; i <= high; i++){
            for (int j = 0; j <= N-1; j++){
                if (i > 0){
                    if (j > 0){
                        if (one_step[i-1][j-1] == 1)
                            neighbors_2[i][j] +=1;
                    }
                    if (j < N-1){
                        if (one_step[i-1][j+1] == 1)
                            neighbors_2[i][j] +=1;
                    }
                    if (one_step[i-1][j] == 1)
                        neighbors_2[i][j] +=1;
                }
                if (i < N-1){
                    if (j > 0){
                        if (one_step[i+1][j-1] == 1)
                            neighbors_2[i][j] +=1;
                    }
                    if (j < N-1){
                        if (one_step[i+1][j+1] == 1)
                            neighbors_2[i][j] +=1;
                    }
                    if (one_step[i+1][j] == 1)
                        neighbors_2[i][j] +=1;
                }
                if (j > 0){
                    if (one_step[i][j-1] == 1)
                        neighbors_2[i][j] +=1;
                }
                if(j < N-1){
                    if (one_step[i][j+1] == 1)
                        neighbors_2[i][j] +=1;
                }
            }
        }

        for (int i = low; i <= high; i++){
            for (int j = 0; j <= N-1; j++){
                if (one_step[i][j] == 1)
                {
                    if (neighbors_2[i][j] == 2 || neighbors_2[i][j] == 3)
                        two_steps[i][j] = 1;
                    else
                        two_steps[i][j] = 0;
                }
                else if (one_step[i][j] == 0)
                {
                    if (neighbors_2[i][j] == 3)
                        two_steps[i][j] = 1;
                    else
                        two_steps[i][j] = 0;
                }
            }
```

```
        }

        int counter1=0, counter2=0, counter3=0;
        int diff = high - low +1;
        for (int i = low; i <= high; i++)
        {
            for (int j = 0; j <= N-1; j++)
            {
                if(world[i][j] == one_step[i][j])
                    counter1++;
                if(world[i][j] == 0)
                    counter2++;
                if(old[i][j] == two_steps[i][j])
                    counter3++;
            }
        }

        if (counter1 == (diff*N))
            threadinfo->status = 0;
        else if(counter2 == (diff*N))
            threadinfo->status = 0;
        else if(counter3 == (diff*N))
            threadinfo->status = 0;


        if(threadinfo->status == 0)
            done++;
        pthread_barrier_wait(&barrier);

        if(done == nthreads)
            break;

        for (int i = low; i <= high; i++)
        {
            for (int j = 0; j <= N-1 ; j++)
                world[i][j] = two_steps[i][j];
        }

        for (int i = low; i <= high; i++)
        {
            for (int j = 0; j <= N-1 ; j++)
                old[i][j] = world[i][j];
        }

        for(int i = low; i <= high; i++)
        {
            for(int j = 0; j <= N-1; j++)
            {
                neighbors[i][j] = 0;
                neighbors_2[i][j] = 0;
            }
        }

        done = 0;
        threadinfo->steps+=2;
    }

    pthread_exit(NULL);
}

void print_world(int **world)
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N ; j+=1)
            printf("%d ", world[i][j]);
        printf("\n");
    }
    printf("\n");
}
```

5.4 Matlab code

```
close all;
clear all;

%------------- Unoptimized code test -------------

% gameoflife_unoptimized.c
% sizes = [200 300 400 500]
% for each method [-O,-O2,-O3,-Ofast]
% steps needed [456 508 1878 870 1670]
size200_unop = [1.212, 0.421, 0.424, 0.407];
size300_unop = [9.317, 3.568, 3.423, 3.003];
size400_unop = [7.546, 2.987, 3.117, 2.345];
size500_unop = [21.446, 8.307, 8.816, 6.925];

times1 = [size200_unop; size300_unop; size400_unop; size500_unop];
X1 = categorical({'200','300','400','500'});
X1 = reordercats(X1,{'200','300','400','500'});
figure(1)
b1 = bar(X1,times1);
b1(1).FaceColor = [51/255 255/255 51/255];
b1(2).FaceColor = [255/255 102/255 178/255];
b1(3).FaceColor = [51/255 153/255 255/255];
b1(4).FaceColor = [255/255 153/255 51/255];
ylim([0 22])
title('\fontsize{12}Execution times for different compiler flags (Unoptimized)')
ylabel('\fontsize{12}Time [s]')
xlabel('\fontsize{12}Size of world N')
legend('\fontsize{10}No flags','\fontsize{10}-O2','\fontsize{10}-O3','\fontsize{10}-Ofast','Location','northwest')
grid on

%------------- Optimized code test -------------

% gameoflife_optimized.c
% sizes = [200 300 400 500]
% for each method [-O,-O2,-O3,-Ofast]
% steps needed [456 508 1878 870 1670]
size200_op = [0.496, 0.249, 0.226, 0.242];
size300_op = [4.209, 1.774, 1.363, 1.486];
size400_op = [3.597, 1.724, 1.116, 1.265];
size500_op = [10.878, 3.975, 3.256, 3.551];

times2 = [size200_op; size300_op; size400_op; size500_op];
X2 = categorical({'200','300','400','500'});
X2 = reordercats(X2,{'200','300','400','500'});
figure(2)
b2 = bar(X2,times2);
b2(1).FaceColor = [51/255 255/255 51/255];
b2(2).FaceColor = [255/255 102/255 178/255];
b2(3).FaceColor = [51/255 153/255 255/255];
b2(4).FaceColor = [255/255 153/255 51/255];
ylim([0 12])
title('\fontsize{12}Execution times for different compiler flags (Optimized)')
ylabel('\fontsize{12}Time [s]')
xlabel('\fontsize{12}Size of world N')
legend('\fontsize{10}No flags','\fontsize{10}-O2','\fontsize{10}-O3','\fontsize{10}-Ofast','Location','northwest')
grid on

%------------- Parallel code test -------------

% sizes = [200 300 400 500]
% for each method [-O,-O2,-O3,-Ofast]
% steps needed [456 508 1878 870 1670]
size200_p = [0.328, 0.190, 0.163, 0.169];
size300_p = [2.184, 0.843, 0.831, 0.838];
size400_p = [1.663, 0.664, 0.593, 0.615];
size500_p = [4.758, 1.838, 1.624, 1.662];
```

```
%for 2 threads
times3 = [size200_p; size300_p; size400_p; size500_p];
X3 = categorical({'200','300','400','500'});
X3 = reordercats(X3,{'200','300','400','500'});
figure(3)
b3 = bar(X3,times3);
b3(1).FaceColor = [51/255 255/255 51/255];
b3(2).FaceColor = [255/255 102/255 178/255];
b3(3).FaceColor = [51/255 153/255 255/255];
b3(4).FaceColor = [255/255 153/255 51/255];
ylim([0 5])
title('\fontsize{12}Execution times for different compiler flags (Parallel, 2 threads)')
ylabel('\fontsize{12}Time [s]')
xlabel('\fontsize{12}Size of world N')
legend('\fontsize{10}No flags','\fontsize{10}-O2','\fontsize{10}-O3','\fontsize{10}-Ofast','Location','northwest')
grid on

%------------ Thread code test ------------

% sizes = [1000] --> 2692 steps
% for each method [-O3]
unoptimized = 60.091;
optimized = 25.927;
thread2 = 12.233;   % 189% CPU
thread4 = 8.953;    % 322% CPU
thread8 = 8.557;    % 586% CPU

times4 = [unoptimized; optimized; thread2; thread4; thread8];
X4 = categorical({'Serial unop.', 'Serial op.','2 threads','4 threads','8 threads'});
X4 = reordercats(X4,{'Serial unop.', 'Serial op.','2 threads','4 threads','8 threads'});
figure(4)
b4 = bar(X4,times4);
b4.FaceColor = [51/255 255/255 51/255];
ylim([0 62])
title('\fontsize{12}Execution times for different amounts of threads (size N=1000)')
ylabel('\fontsize{12}Time [s]')
grid on
```

Referenser

[1] Andrew Adamatzky. *Game of Life Cellular Automata*. Springer London, 2010. ISBN: 9781849962162.