
UPPSALA
UNIVERSITET

# Final Project: Game of Life
# High Performance Programming

**Written by:** Jakob Gölén

March 20, 2022

# 1 Introduction

As computers get more and more powerful and more cores can be used, the advantage of running programs over multiple threads increases also. The main goal of this report is to choose a problem, write a program to study the problem, optimize it serially and then parallelize it so it can run on multiple threads. The programming language used is C and OpenMP was used to parallelise the code.

# 2 Problem description

The problem chosen for this report is Game of life. Game of life is a so called cellular automaton which is a system of cells that changes over time according to a specific set of rules. Game of life was created in 1970 by the mathematician John Conway and is a 2 dimensional system of cells where the following rules apply:

- A cell is either alive or dead
- A dead cell becomes alive if it has 3 neighbors
- A living cell only survives if it has either 2 or 3 neighbors, otherwise it dies.

The cells are represented as tiles in an infinite two dimensional grid. The neighbor of a cell in this case refers to the 8 tiles directly surrounding the current cell that is being considered. [1] [2]. In order to implement the algorithm, an initial condition is chosen consisting of a few living cells. The state of the cells in the next time instance is calculated according to the rules and then the system advances in time.

The task was to create a program in C which implements the Game of Life algorithm, advances the system a number of steps defined by the user, and saves a file with the positions of the living cells at the final time step.

# 3 Solution method

The first step was to figure out how to store an infinite two dimensional grid. This was done by allocating two matrices, with the size of the matrices being defined by the user, and then letting the matrices wrap around the edges. This was done by the following equation:

$$M[i][j] = M[(i + x_{length}) \ mod \ x_{length}][(i + y_{length}) \ mod \ y_{length}] \tag{1}$$

More concretely, the i-th row is added by length of the row, then the result is divided by the length of the row and the remainder is taken to represent the i-th row. Similarly the j-th column is added by the length of the column and then the result is divided by the length of the column and the remainder represents the j-th column. This allows the system to move a great distance, for example if the evolution of the system causes the system to drift to one side it will eventually loop around. The drawback is that cells on one side of the system can interact with cells on the other side of the system if the

system is large enough. A large matrix size should be chosen to avoid this.

Two matrices were allocated in order to save the state of cells in the current time step and to save states the cells will have in the next time step, with the states meaning if a cell is dead (represented by 0) or if it is alive (represented by 1).

When the code was being optimized, the two matrices could be combined by saving the current and next states in a struct and allocating a matrix of structs instead (the cell states for the next step is called the cellbuffer in the program). This reduced the number of malloc and free calls that were needed. With the matrices created, the current and next state of all cells was initialized to zero.

With the matrices created, a function was written to handle the starting configuration. There were two cases. Either the user provided a filename of a file which contained information about which elements in the matrix should be set to alive, or the user did not provide a filename in which case a manual configuration was done.

If a filename was provided, the file was opened with fread. The first number indicated how many cells were alive and then the x and y position was read and the cell in the corresponding row and column in the matrix was set to alive.

If a filename was not provided, the user would type in a filename and then choose how many alive cells they wanted in the starting configuration and this was written into the new file. Then the user typed in the x and y positions of the cells which should be alive. The cells in the corresponding row and column in the matrix was set to alive and the coordinates were written into the new file, which could then be used for further tests.

When the inital conditions had been programmed, a simple graphics function was written in order to see the starting configuration and would later be used to see how the system evolved. The function looped through the row and columns of the matrix and checked if the cell was alive or not. If the cell was alive it printed a filled box and if the cell was dead it printed a hollow box. When the end of a row was reached, it printed a new line. The user chose how often the graphics would be displayed with 1 representing every step, 2 representing every other step and so on. 0 printed only the starting and final configuration and -1 turned off graphics completely. The output of the graphics function is shown in figure 1. The function was simple so the inline keyword was used.

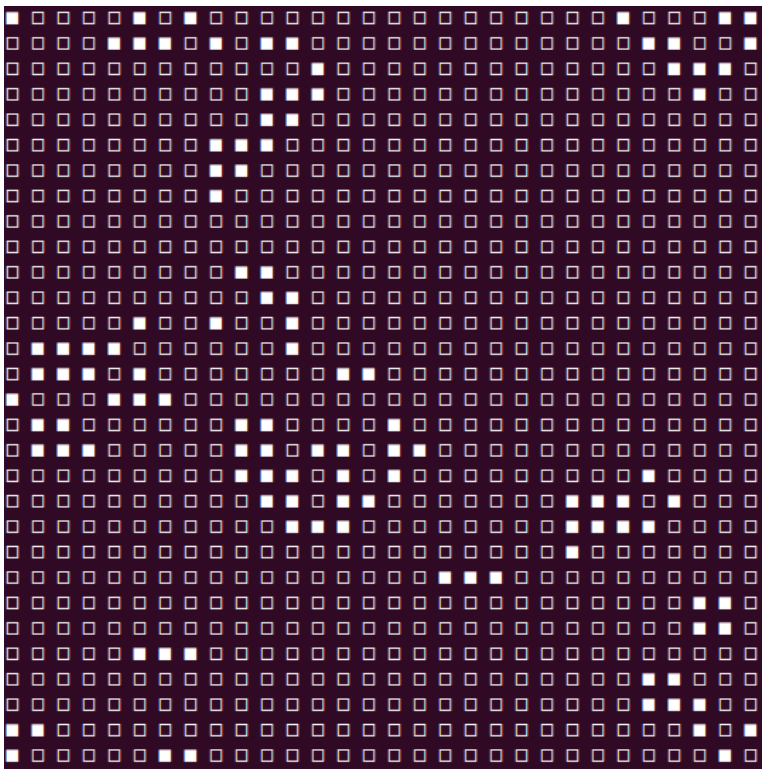

Figure 1: The graphics function showing cells on a gridsize of 30x30, with some alive and some dead cells

With a graphics function created, the next step was to implement the actual Game of Life algorithm. A for loop was created which represented the steps. Inside this for loop, two loops were created which looped over the rows and columns of the matrix containing the information of the cells. For every cell, the state of the eight neighbors were checked and the number of alive neighbors was saved. Next, the rules were applied using two if statements. If the current cell was alive, it was checked if the number of neighbors were fewer than two or more than three. During tests it was noted that it was more usual for a cell to have fewer than two neighbors than it was for a cell to have more than three neighbors, so the check to see if there were fewer than two neighbors was set first in the or statement, thus reducing the number of times the second or statement had to be checked. This is called short circuit programming [3]. If the number of neighbors were fewer than two or more than three, the cellbuffer was set to zero, otherwise the cellbuffer was set to one. If the current cell was dead however, it was checked if the number of neighbors were exactly three. If this was the case, the cellbuffer was set to one, otherwise it was set to zero.

When the state for the next time step had been checked for all cells, two new loops were created, looping over the rows and columns of the matrix and copying the value from the cellbuffer to the current cellvalue, thus advancing the system in time. The reason the values first had to be written into the cellbuffer and then be copied over to the actual cell state was that every cell in the system had to be checked for what the state in the next time step would be before any of them were advanced. Otherwise, a cell which had a neighbor that was already updated would get the wrong result if the neighboring cell had been changed.

When all cell states had been updated, the outer loop was restarted. When the number of timesteps defined by the user had been reached, the loop ended and an output file was created by first checking how many cells are alive in the system, writing that to the file and then writing the coordinates of the living cells into the file. Lastly the memory for the matrix representing the grid was freed.

The main optimization done on the code in serial was to reduce the number of elements in the matrix that needed to be checked. The key component is that if a cell is dead and is not adjacent to a living cell, that cell will remain dead in the next timestep and therefore do not have to be checked. Therefore, the function handling the initial configuration was changed. When a cell was initialized, the position was compared against four variables: $x_{max}$, $x_{min}$, $y_{max}$ and $y_{min}$. If the x-position of the current cell was larger then $x_{max}$, that x-coordinate replaced $x_{max}$, in the same way, if the position had a smaller value than $x_{min}$, that x-coordinate replaced $x_{min}$. The same thing was done for the y-position and $y_{max}$ and $y_{min}$. This meant that all the alive cells were within the region $x_{min} \leq x \leq x_{max}, y_{min} \leq y \leq y_{max}$. Only cells in this region and cells directly adjacent to the region had to be checked, because every other cell was guaranteed to remain dead in the next time step. Therefore, instead of looping over every row and column, the two loops were from $x_{min} - 1$ to $x_{max} + 1$, including $x_{max} + 1$ and from $y_{min} - 1$ to $y_{max} + 1$, including $y_{max} + 1$. The loops where the next state of the cells were copied from the cellbuffer to the cellstate was modified to check for active cells and then update $x_{max}$, $x_{min}$, $y_{max}$ and $y_{min}$ according to the minimum and maximum position of the active cells in the next time step. These updated variables then became the new limits of the for loops.

For optimization flags, the -O3 optimization flag was chosen which instantly made the program an order of magnitude faster. Wall was added to catch any errors or warnings. the -ffast-math and -faggressive-loop-optimizations flags could be used since no float calculations occured [4] and the loop limits were well behaved [5]. Finally, -march=native was used to make the compiler optimize the code for the processor architecture the code is compiled on [6]. No difference in running time was noticed with the optimization flags apart from the -O3 optimization flag, but the -march=native might lead to a speedup on a different machine since it differs between architecture. When the code had been optimized, it was checked for memory leaks and errors with valgrind, and none were detected.

The next step was to parallelize the code. Open MP was chosen due to its ease of implementation. First, an input parameter was implemented so the user could choose how many threads should be created. This was then set with the omp_set_num_threads function. In the function setting up the initial configuration of the system, the loops reading from file were parallized using the for clause and two reduction clauses. The for clause implemented the parallization on the outer for loop and the reduction clauses were needed in order to parallelize the search for the minimum and maximum position of the particles. One reduction clause handled the search for $x_{max}$ and $y_{max}$ using the max operator, and one reduction clause handled the search for $x_{min}$ and $y_{min}$ using the min operator. A #pragma omp critical was added around the block where the file is read, to ensure that two values (the x and y coordinates) are read in sequence.

Next the loops inside the timestep loop were parallelized. The timestep loop could not be parallelized since the steps are taken in sequence and a timestep is dependant on the previous timestep. The loops where the neighbors are calculated and the cells next state was written to the cellbuffer could be parallelized since looking up neighbors to a cell is an independant task. The loop was parallelized with the for clause. The loop where the values are written from the cellbuffer to the actual cell state could also be parallelized, since only the data in the current cell is read. Two reduction clauses were needed here as well since the new minimum and maximum coordinates were determined in this loop. This was implemented in the same way as in the initialization function, with one reduction handling the minimum coordinates and one reduction clause handling the maximum coordinates [7].

The matrix was allocated and freed in parallel using #pragma omp parallel for and the final configuration could also be written in parallel with a #pragma omp critical around the block where coordinates are written to the file, in order to ensure that two fwrite calls (for the x and y coordinates) were called in sequence.

The manual configuration block in the function for the initial configuration was not parallelized since in the manual configuration, the user wrote the coordinates of the active cells manually. The loops inside the graphics function were not parallelized either, since the printf calls had to happen in sequence in order to correctly print out the cells in the correct order.

# 4 Experiments

All experiments were performed on the ”vitsippa” scientific linux machine hosted by Uppsala University, which runs a AMD Opteron (Bulldozer) 6282SE@2.6Ghz on Scientific Linux 6 [8]. Three main tests were performed to check how the runtime changed depending on number of steps, grid size and number of cores. The tests ran with a starting configuration that spread out across the system as well as a starting configuration that stayed a constant size but moved around in the system. The starting configurations can be seen in figures 2 and 3.

Figure 2: This starting configuration will spread out across the system

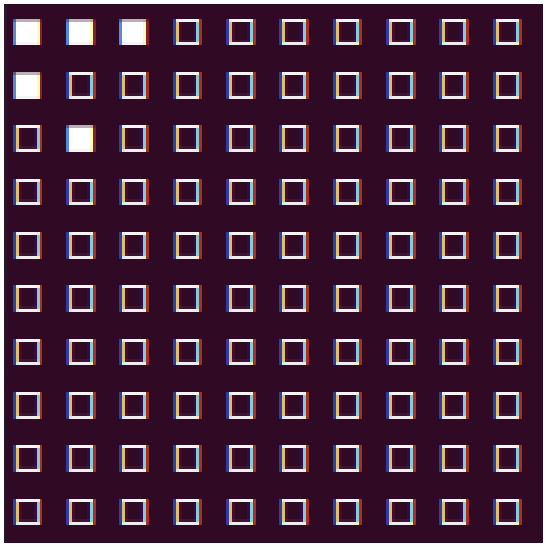

Figure 3: This starting configuration will stay the same size

Table 1: Testing of serial optimization (size 50x50, 20 000 steps)

| Serial Code | Time with no flags [s] | Time with flags [s] |
|---|---|---|
| Unoptimized(fig. 2) | 1.270 | 0.174 |
| Optimized(fig. 2) | 0.991 | 0.131 |
| Unoptimized(fig. 3) | 1.288 | 0.175 |
| Optimized(fig. 3) | 0.645 | 0.092 |

From table 1 it can be seen that the serial optimization gives a better performance boost for systems that stay small. A large performance boost can be seen when the optimization flags are set, decreasing the run time by an order of magnitude.

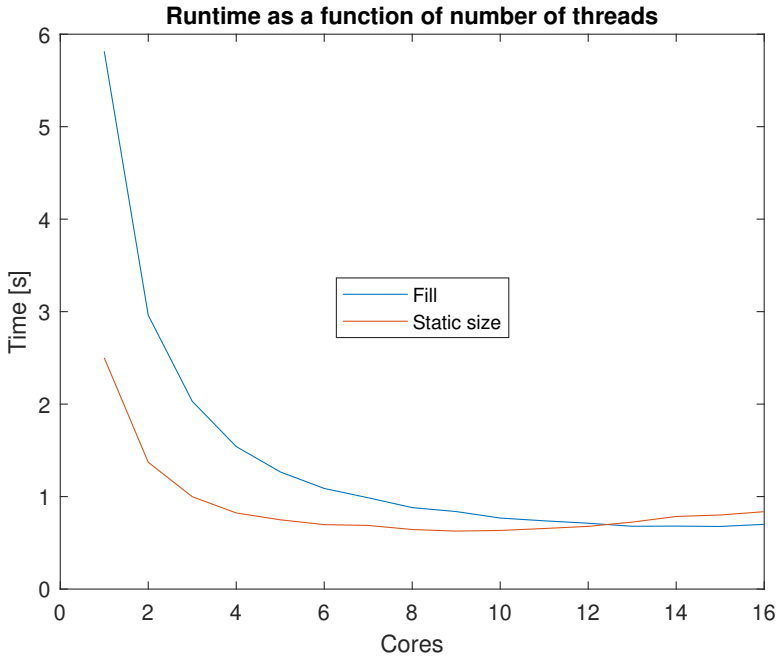


Figure 4: Runtime plotted against number of threads

Figure 4 shows the runtime plotted against the number of threads used. A gridsize of 100x100 was used with 10 000 steps. The configuration which fills the system scales reasonable with the number of threads, with the time decreasing for every added thread but the decrease in time becomes smaller the more threads that are added. The configuration which stay at a constant size behaves interestingly, with the runtime being at the lowest around 9 threads, then the runtime increases slightly for every added thread.

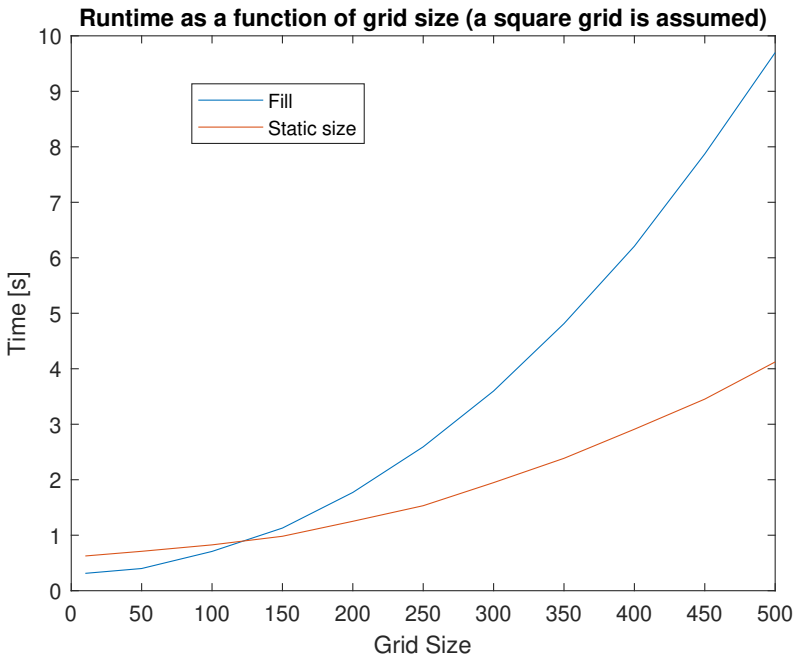


Figure 5: Runtime plotted against the size of the system

Figure 5 shows the runtime plotted against the grid size. The program ran 10 000 steps with 16 threads. Both plots show an expexted behavour, with the runtime increasing quadratically which is reasonable for a square system, with the static configuration increasing more slowly.

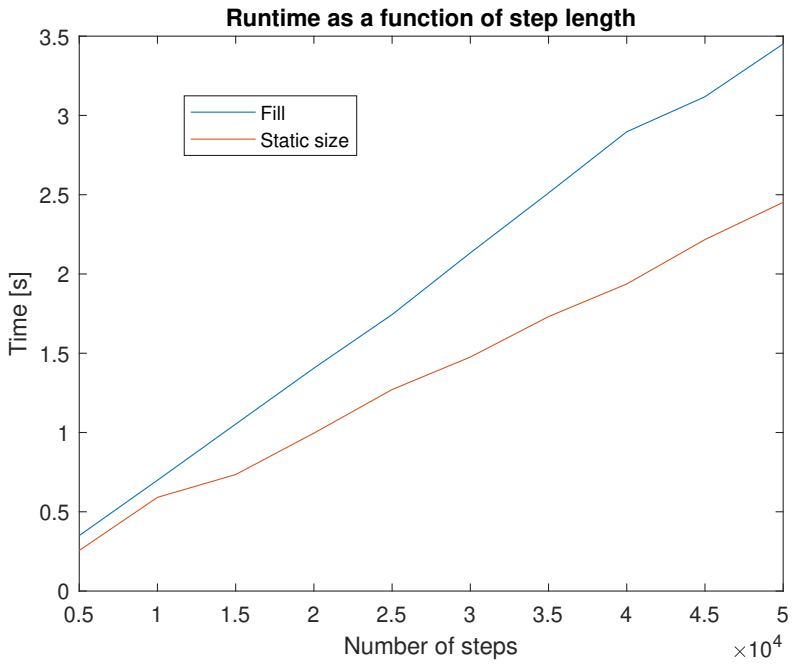


Figure 6: Runtime plotted against number of steps taken

Figure 6 shows the runtime plotted against the number of steps taken. The grid size was 100x100 and the program ran on 16 threads. From the plots, the runtime appears to increase linearly, with the configuration filling the system having a steeper slope.

# 5 Conclusion

Table 1 showing the difference between the unoptimized and optimized serial code appears to show a reasonable behavour. Fewer cells has to be compared to the min and max values for the static sized system thus making the difference between the optimized and unoptimized code large. For the configuration that spreads out across the system (figure 2) a large number of cells means many computations since every living cell has to be compared to the min and max x and y variables, thus making the serial optimization less useful, which can be seen in table 1.

Figure 4 showing the runtime plotted against the number of threads has an expected behavour for the starting configuration that spreads out across the system, but the configuration with a static size behaves somewhat strangely, with the runtime decreasing up to around 9 threads and then increasing when more threads are added. This is probably due to the reduction clause. Four variables are created per thread and are then compared with the variables in the other threads to return the largest or the smallest, depending on the operator defined in the reduction clause [9]. For a small system with constant size, the comparisons between the threads dominate and thus increases the run time when more threads are added. This then explains why the problem is not visible for a configuration which fills up the system, since the comparisons between the threads due to the reduction clauses doesn't dominate.

The reason the configuration with a static size has a slower increased runtime in figure 5 is because it only calculates an entire row or column when the cells wrap around, i.e when some cells are on one side of the matrix and some cells are on the other side of the matrix. Otherwise, only a few cells are calculated regardless of grid size. The cells rarely wrap around, so the increase in runtime is slower, so the figure is reasonable.

Figure 6 also behaves as expected, with both the static and filling configurations increasing linearly with step size. The only difference is that fewer calculations have to be done on the static configuration, thus making the runtime increase slower than the configuration that fills the system.

The optimization of the serial code and the parallelization appears to work as intended by looking at the plots and the table, but there are a few improvements that can be made. One method that was tested was to check if fewer cells had to be calculated if cells were checked from the middle of the matrix, i.e. from $x_{length}/2$ to $3 * x_{length}/2$ mod $x_{length}$ and the same way in the y direction. This was then compared to the check from 0 to $x_{length}$ and in the same way in the y direction to see which check gave the smallest region, i.e. smallest $x_{max} - x_{min}$ and $y_{max} - y_{min}$. The idea behind this is to remove the problem where the whole matrix has to be checked if a few cells are on one side of the matrix and a few cells are on the other side of the matrix. When this was tested however, it gave no decrease in runtime, so it was not implemented.

Another idea for decreasing the number of cells would be to split the matrix into small regions and only calculating the regions which contains active cells. This would solve the problem of living cells forming clusters far away from each other, since in its current form, the algorithm has to calculate the empty cells between the clusters.

Overall, the program appears to be working well. No memory leaks or errors were detected and the graphics function makes it easy to see how the system evolves over time.

# References


[1] Siobhan Roberts. The Lasting Lessons of John Conway's Game of Life. *The New York Times*, 2020. https://www.nytimes.com/2020/12/28/science/math-conway-game-of-life.html (retrieved 2022-03-07).

[2] Carter Bays. *Introduction to Cellular Automata and Conway's Game of Life*. Springer London, 2010.

[3] Siddiqi. Short Circuit Evaluation in Programming. *Coders Legacy*, 2022. https://coderslegacy.com/short-circuit-evaluation-in-programming/ (retrieved 2022-03-09).

[4] Krister Walfridsson. Optimizations enabled by -ffast-math. *Krister Walfridsson's blog*, 2021. https://kristerw.github.io/2021/10/19/fast-math/ (retrieved 2022-03-13).

[5] The GCC Team. Options That Control Optimization. *GCC, the GNU Compiler Collection*, 2022. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html (retrieved 2022-03-13).

[6] The GCC Team. x86 Options. *GCC, the GNU Compiler Collection*, 2022. https://gcc.gnu.org/onlinedocs/gcc/x86-Options.html (retrieved 2022-03-13).

[7] OpenMP Architecture Review Board. Openmp 5.2 API Syntax Reference Guide. *Reference Guides*, 2021. https://www.openmp.org/resources/refguides/ (retrieved 2022-03-13).

[8] Uppsala University. Linux hosts. https://www.it.uu.se/datordrift/maskinpark/linux (retrieved 2022-03-17).

[9] OpenMP Architecture Review Board. OPENMP API Specification: Version 5.0 November 2018. https://www.openmp.org/spec-html/5.0/openmpsu107.html (retrieved 2022-03-18).