Uppsala University

High Performance Programming

1TD062

Project Report:

Game of Life

Student:
Julián Ramón Marrades Furquet

Teacher:
Jarmo Rantakokko

May 6, 2021

Contents

1 Introduction **2**

2 Problem description **2**

3 Solution method **3**
3.1 Base code 5
3.2 Optimized code 5
3.3 Parallel code 7

4 Experiments **8**
4.1 Algorithm time complexity 9
4.2 Base vs. optimized code 9
4.3 Parallel performance 10

5 Conclusions **10**

References **13**

1 Introduction

As the final task for the High Performance Programming course, implementing, optimizing, and parallelizing an interesting algorithm is requested. If one is working with small input, e.g. a short list, a reduced computational domain, etc. the running time of the solving algorithm will most likely be affordable regardless of its time complexity. However, as problems scale to larger sizes, so do the execution times. Therefore, it is vital to optimize algorithm performance as much as possible. One can distinguish several options for doing so:

- Reducing the time complexity of the algorithm. E.g. from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$, where n is the input size.
- Reducing the space complexity. E.g. solving the problem with an $n \times 1$ vector as a data structure instead of an $n \times n$ matrix.
- Code optimization. E.g. avoiding expensive computer operations, preventing branching within loops, exploiting cache lines, employing vector registers, etc.
- Parallelization: finding parts of the algorithm which can be completed simultaneously by different cores, exploiting modern multi-core multi-thread CPUs.

2 Problem description

For this project, the Game of Life [1] is the selected problem. It consists of an infinite orthogonal grid of square cells, which can be either alive or dead. Each cell interacts with its neighborhood (see Fig. 1) to decide its state at the next time step via the following rules:

1. If a cell is alive and has fewer than 2 alive neighbors, it will die.
2. If a cell is alive and has 2 or 3 alive neighbors, it will live.
3. If a cell is alive and has more than 3 alive neighbors, it will die.
4. If a cell is dead and has 3 alive neighbors, it will live.
5. If a cell is dead and has fewer or more than 3 alive neighbors, it will die.

It is likely that patterns of alive cells which may be created when declaring the initial state, remain in the grid. Such patterns can be classified into:

- still lifes, which do not change with time,
- oscillators, which return to their initial pattern after a finite number of time steps, and
- spaceships, which propagate through the grid.

Since the grid is infinite, the only way for a pattern to be destroyed is to collide with another pattern.

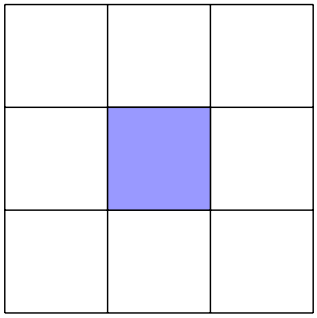
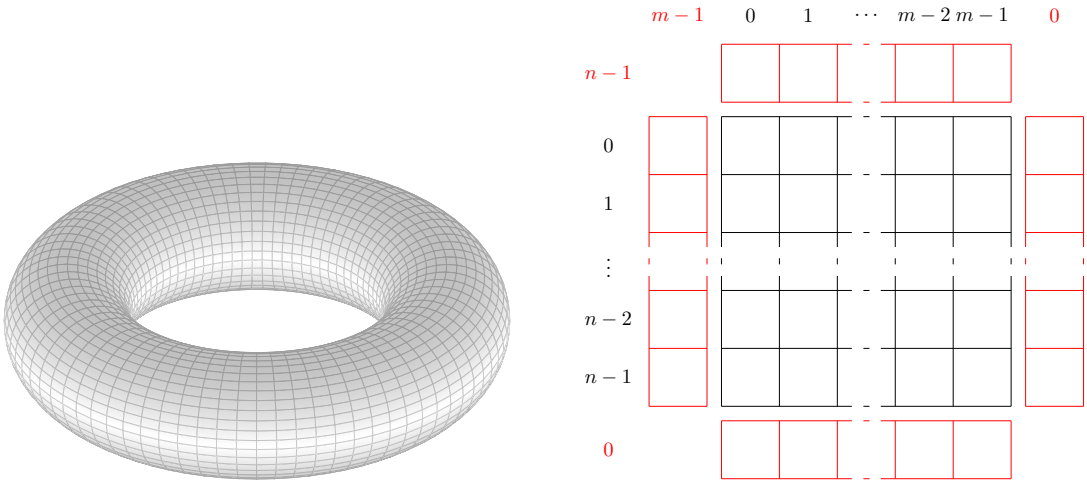


Figure 1: White cells indicate the neighborhood of the blue cell.



(a) 3D representation, where each rectangle corresponds to a cell.

(b) 2D representation as a matrix.

Figure 2: Toroidal computational domain.

3 Solution method

Since computers do not have infinite memory, it is common to establish a bounded computational domain. The easiest solution might be to define an $n \times m$ grid and state that any cell outside the domain is dead. However, such approach is highly problematic, since patterns that come into contact with the domain boundary will most likely die. Therefore, a torus is selected as the computational domain, where patterns will only disappear if they touch another pattern. Fig. 2 displays both a 3D and a 2D representation of a toroidal matrix. From Fig. 2b it becomes clear that patterns leaving from the top will enter from the bottom, patterns leaving from the left will enter from the right, and so on.

The command line interface of the C implementation takes the form:

```
./gol n m prob nSteps seed (nThreads) debug,
```

where:

- `n` and `m` denote the number of rows and columns in the grid, respectively.

- `prob` denotes the probability of a cell being alive. For details on grid initialization see Algorithm 1.
- `nSteps` represents the number of time steps.
- `seed` indicates the seed for random number generation. If a negative integer is passed, a seed will be set based on POSIX time. Seeding makes results reproducible and allows to test performance under the same *arbitrary* grids.
- `nThreads` denotes the number of threads. Note that this parameter is only accepted by the parallel implementation.
- `debug` runs the program in debug mode, printing the initial and final grids so that correctness can be verified.

Algorithm 1 Grid initialization

Input: $grid$: a matrix representing the grid
n, m: number of rows and columns of the grid, respectively
$prob \in [0, 1]$: probability of a cell being alive

```
1: for i = 0, 1, ..., n − 1 do
2:   for j = 0, 1, ..., m − 1 do
3:     // Draw a random number from a uniform distribution in [0, 1]
4:     arb ← U[0, 1]
5:     if arb ≤ prob then
6:       // Set cell as alive
7:       grid[i][j] ← 1
8:     else
9:       // Set cell as dead
10:      grid[i][j] ← 0
11:    end if
12:  end for
13: end for
```

The code is compiled by GCC 7.5.0 on Ubuntu 18.04.5 LTS. The system uses 16 Intel® Xeon® CPU at 2.27 GHz. There are 2 sockets with 4 cores each, and each core provides support for 2 threads. In total, that accounts for 8 *physical* threads per socket. In addition, the system contains 2 NUMA nodes.

For all implemented solutions, the following compiler optimization flags are employed:

- `-O3`: allows the compiler to perform heavy optimizations when creating assembly files.
- `-march=native`: allows the compiler to optimize the code based on the CPU it compiles on.
- `-ffast-math`: speeds up floating point operations, but may severely impact accuracy if values get close to 0. Since alive probabilities tend to be around 0.5 and the rest of arithmetic operations are performed on integers, the flag is safe to use.

However, the usage of `-funroll-loops` is intentionally avoided. This flag can be dangerous when loops contain a considerable workload, which is the case for the implemented solutions. Therefore, loop unrolling is performed by hand for the optimized and parallel versions.

3.1 Base code

The most straightforward way to implement Game of Life is by storing cell states as integers (4 bytes) in a matrix which represents the grid. Then, one can iterate over time as described in Algorithm 2, using Algorithm 3 for cell state transitions (lines 6 and 9). One should note that, code-wise, line 13 in Algorithm 2 requires an extra pointer.

Algorithm 2 Time iteration

Input: *grid*: a matrix representing the grid
other: a matrix to store grid updates (same dimensions as *grid*)
n, m: number of rows and columns of the grid, respectively
nSteps: number of time steps to perform

```
1:  for k = 0, 1, ..., nSteps − 1 do
2:    for i = 0, 1, ..., n − 1 do
3:      for j = 0, 1, ..., m − 1 do
4:        if (i, j) lays on the boundary of the grid then
5:          Count neighbors taking into account the special torus wrap-around case
6:          Decide next state of the cell and put it in other
7:        else
8:          Count neighbors as usual
9:          Decide next state of the cell and put it in other
10:       end if
11:     end for
12:   end for
13:   Swap grid and other, so that grid now stores the updated cell states
14: end for
```

3.2 Optimized code

In order to improve performance, the first thing that is changed is Algorithm 3. The transition rules depicted in Section 2 can be simplified by looking at the 3×3 field spanned by a cell **and** its neighborhood:

1. If the number of alive cells in the field is 3, the cell will live.
2. If the number of alive cells in the field is 4, the cell will keep its current state.
3. Otherwise, the cell dies.

Such rules lead to Algorithm 4.
Next, since the integers which the algorithm deals with are always between 0 and 9, there is no need to employ 4 bytes for each number. Instead, one can use the `char` datatype, which only occupies 1 byte. This way, cache lines will be able to store more information, leading to performance improvements.
Then, some keywords are added to the code:

Algorithm 3 Cell state transition

Input: $alive \in \{0, 1\}$: current state of the cell
$future$: matrix representing the grid at the next time step
i, j: row and column indexes of the cell, respectively
$neighbors$: number of alive neighbors

1: **if** $alive \wedge neighbors < 2$ **then**
2: $\quad future[i][j] \leftarrow 0$
3: **else if** $alive \wedge neighbors \in \{2, 3\}$ **then**
4: $\quad future[i][j] \leftarrow 1$
5: **else if** $alive \wedge neighbors > 3$ **then**
6: $\quad future[i][j] \leftarrow 0$
7: **else if** $\neg alive \wedge neighbors = 3$ **then**
8: $\quad future[i][j] \leftarrow 1$
9: **else**
10: $\quad future[i][j] \leftarrow 0$
11: **end if**

Algorithm 4 Optimized cell state transition

Input: $alive \in \{0, 1\}$: current state of the cell
$future$: matrix representing the grid at the next time step
i, j: row and column indexes of the cell, respectively
$field$: number of alive cells in the neighborhood + the cell itself

1: **if** $field = 3$ **then**
2: $\quad future[i][j] \leftarrow 1$
3: **else if** $field = 4$ **then**
4: $\quad future[i][j] \leftarrow alive$
5: **else**
6: $\quad future[i][j] \leftarrow 0$
7: **end if**

- `restrict`: added to a pointer to indicate that it is the only way to access the memory it points to. It allows the compiler to avoid extra checks.
- `const`: indicates that a variable will keep its value for the rest of its lifespan.
- `inline`: tells the compiler that a function definition can be substituted at the place where the function call is. Whether to inline a function or not is a choice made by the compiler. Usually, small functions where the output is fully determined by the input are inlined (e.g. Algorithm 4). To ensure inlining, the flag `-Winline` is employed so that the compiler emits a warning if a requested inline has not been performed. In the end, all requested inlines in the implementation are carried out by GCC.

Finally, loop unrolling is performed such that the grid advances 2 time steps at once. This way, one can have two matrices, one storing the grid at even times and one storing it at odd times, avoiding the extra pointer to perform the matrix swap in line 13 of Algorithm 2. Moreover, the in-loop branching in lines 4 and 7 is substituted by several non-conditional sections operating on different regions of the grid. Such adjustments yield Algorithm 5.

Algorithm 5 Optimized time iteration

Input: *grid*: a matrix representing the grid
other: a matrix to store grid updates (same dimensions as *grid*)
n, m: number of rows and columns of the grid, respectively
nSteps: number of time steps to perform (assumed to be even)

1: **for** $k = 0, 1, ..., \frac{nSteps}{2} - 1$ **do**
2: // First iteration: read from *grid* and update *other*
3: Handle the top-left corner $(i, j) = (0, 0)$
4: Handle the top-right corner $(i, j) = (0, m - 1)$
5: Handle the rest of the first row $(i, j) = (0, [1, m - 2])$
6: **for** $i = 1, 2, ..., n - 2$ **do**
7: Handle the first column $(i, j) = (i, 0)$
8: Handle the last column $(i, j) = (i, m - 1)$
9: Handle the rest of the row $(i, j) = (i, [1, m - 2])$
10: **end for**
11: Handle the bottom-left corner $(i, j) = (n - 1, 0)$
12: Handle the bottom-right corner $(i, j) = (n - 1, m - 1)$
13: Handle the rest of the last row $(i, j) = (n - 1, [1, m - 2])$
14:
15: // Second iteration: read from *other* and update *grid*
16: Same procedure as lines 3-13
17: **end for**

3.3 Parallel code

When designing parallel code, like it is done with OpenMP in this case, one must tackle several performance obstacles [5]:

- Thread creation/deletion and serial sections. While adding a `#pragma omp parallel for` clause to the loop iterating over the rows of the grid may be the first idea which comes

to mind, this will give OpenMP the task to create and destroy threads every 2 time steps. Therefore, to create less overhead, work will be split at the highest level. Threads will be created outside the time stepping loop and will be deleted once the final cell grid has been reached.

- Synchronization. Waiting for all threads to reach a certain execution point may take some time. Therefore, it is important to avoid synchronization as much as possible. The parallel implementation in this report only uses 1 global barrier per time-step.

- Load imbalance. If some threads execute for much longer than others, synchronization becomes really expensive. Ideally, one would want all threads to have a balanced amount of work. Since computing the number of alive neighbors is equally expensive for all cells, then each row in the grid requires the same amount of work. Hence, given that the first and last row are being handled independently (see Algorithm 5), and that the grid dimensions that will be employed are in the order of thousands, the first and last row can be assigned to the first and last thread, respectively. After, the rest of the rows can be distributed evenly among all threads. E.g. if the rest of the rows is 4 and there are 2 threads, the first 2 rows will be given to the first thread and the other 2 to the second.

- Cache misses. Such events happen when a thread writes in the cache line that another thread is working with. In principle, this should only happen if the number of threads is large. This is because if we have few threads, they will be writing different rows of the grid and using their own cache lines.

- Non-optimal data placement on NUMA. When placing memory, it will be situated close to the thread in charge of doing such operation. Therefore, one would in principle like to have the grid rows which thread t operates on with to be close to that thread, so that memory access is cheaper. This can be done by creating the initial grid in a parallel fashion with the same work distribution as in the time stepping phase.
  However, there is a major drawback. Except for the first and last thread, cells on the top and bottom row of the block handled by a thread have neighbors on a row which belongs to a different thread. Hence, as the number of threads increases, neighbors become spread, and access turns out to be more expensive. This issue would be even worse if the rows were distributed cyclically, since each cell in a given row would have 6 neighbors in rows belonging to other threads.
  In order to later test the performance of parallel memory initialization, two versions of the parallel code are implemented: one with parallel initialization and one without.

Based on the ideas above, Algorithm 6 is implemented.
It is important to note that, when parallel memory initialization is used, the same random number seed can generate multiple initial grids. Since the execution order of threads cannot be predicted, even though the number of alive and dead cells will be the same, their distribution on the grid will most likely be different in each run.

4 Experiments

To evaluate the performance of the different versions of the code, execution time will be measured for the entirety of the program run. For each combination of parameters, the code will be run 10 times and the results will be averaged. For run $R \in \{1, 2, ..., 10\}$, seed R will be given to the random

Algorithm 6 Parallel time iteration

Input: *grid*: a matrix representing the grid
other: a matrix to store grid updates (same dimensions as *grid*)
n, m: number of rows and columns of the grid, respectively
nSteps: number of time steps to perform (assumed to be even)

1: $threadID \leftarrow omp\_get\_thread\_num()$
2: **for** $k = 0, 1, ..., \frac{nSteps}{2} - 1$ **do**
3: First iteration: read from *grid* and update *other* for the rows assigned to this $threadID$
4: #pragma omp barrier
5: Second iteration: read from *other* and update *grid* for the rows assigned to this $threadID$
6: #pragma omp barrier
7: **end for**

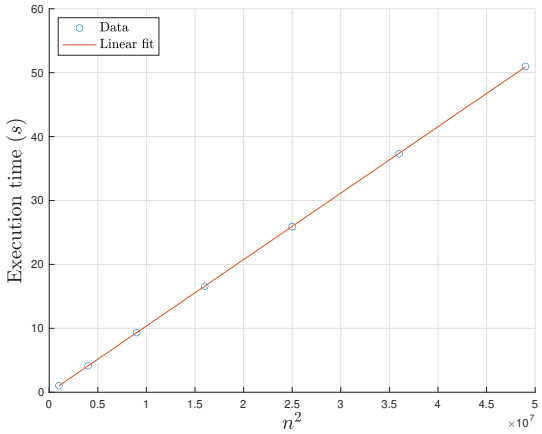


Figure 3: Pearson correlation coefficient (r) test. Resulting coefficient is $r = 0.9999999...$

number generator.
Hereinafter, square grids $n = m$ will be employed. Thus, only n will be mentioned. Other parameters are fixed as follows: `prob = 0.5`, `nSteps = 100`, and `debug = 0`.

4.1 Algorithm time complexity

In theory, the algorithm has a time complexity of $\mathcal{O}(n^2)$, since it iterates over n^2 cells, and then aggregates neighbors and computes the next state of the cell in constant time. In order to verify such hypothesis, execution times of the optimized code are scattered against n^2, a linear fit is plotted, and the Pearson correlation coefficient (r) is computed (see Fig. 3). Obtaining $r = 0.9999999...$ indicates that the algorithm is indeed $\mathcal{O}(n^2)$, since there is a nearly perfect positive linear correlation between execution time and n^2.

4.2 Base vs. optimized code

The base and optimized serial code are compared by plotting their execution time against $n \in \{1000, 2000, 3000, 4000, 5000, 6000, 7000\}$. Fig. 4 shows the running times for both implementations. It can be observed that the time difference grows as n increases, reaching a 10 seconds margin at

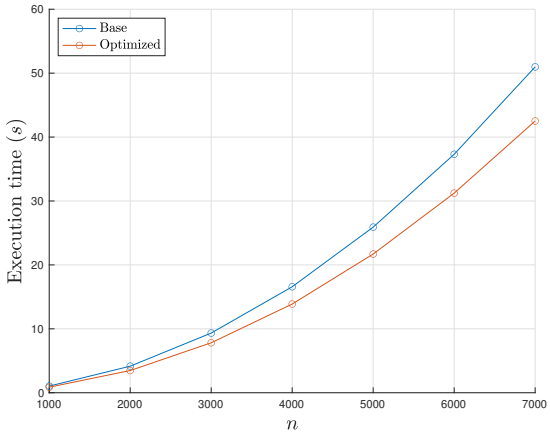


Figure 4: Execution time of serial code for different n.

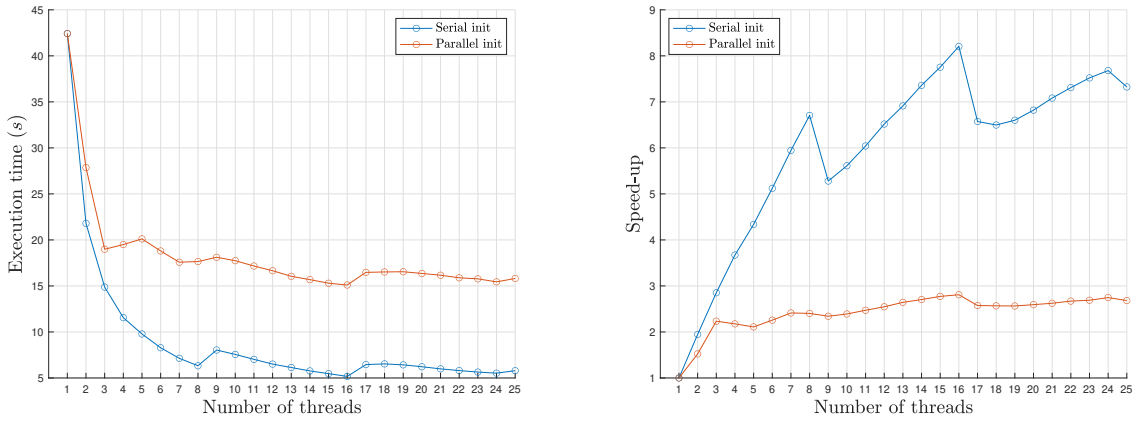
$n = 7000$. It is not a great improvement, especially when compared to the available upgrades for the N-body problem. In that assignment, apart from preventing branching, vectorizing operations, etc. since the distance from planet i to planet j is the same as the distance from planet j to planet i, half of the computations could be saved by storing distances in a matrix, almost halving the execution time. Similar tricks can be employed for Game of Life, but are much more complicated and will be discussed in the end of Section 5.

4.3 Parallel performance

In order to test how effective parallel memory initialization is, execution times for `nThreads` $\in [1, 25]$ are plotted for both parallel versions. In Section 3.3, it was reasoned that parallel memory initialization might be counterproductive. Such hypothesis is confirmed by Fig. 5a, which shows that parallel initialization performs much worse than serial initialization, having longer execution time. This difference is not notorious when there are just a couple of threads. However, for `nThreads` ≥ 4, the time difference rapidly grows up to 10 seconds.
Focusing on the serial initialization curve in Fig. 5a, one can observe a little fall-back in execution time when a new socket starts to receive threads (`nThreads` $\in \{9, 17, 25\}$). Threading overhead and cache misses start to manifest at 9 threads, but become obvious at 17 threads, where the curve turns out to be almost flat.
Focusing on speed-up (see Fig. 5b), one may observe that the parallel initialization does not even reach a value of 3 for any number of threads. On the other hand, the serial initialization seems to scale decently until 8 threads. Then, it falls back but keeps increasing to reach a peak speed-up of 8 at 16 threads. However, the speed-up curve begins to lose its steepness from 17 threads onwards.

5 Conclusions

In this report, code optimization has been motivated and demonstrated. As an example, the Game of Life was introduced and a basic algorithm for time stepping was explained. Then, many serial optimizations such as reformulating the rules, preventing in-loop branching, using special keywords for variables and functions, loop unrolling, etc. were presented. Afterwards, parallelism with OpenMP was discussed, and an implementation attempting to avoid parallel performance obstacles was proposed.



(a) (b)

Figure 5: Performance of parallel code for different number of threads. Cell grid is set for $n = 7000$.

Later, the performance of the algorithms was put to the test, finding out that serial optimizations were effective in fair measure. What is more, it was demonstrated that parallel memory initialization is not effective when a thread has to read data which has been initialized by many different threads and is thus spread across the physical memory.

The question that remains is: **can it be faster?**
Further adjustments can be classified in minor and major changes. Here are a couple of simple suggestions:

- One may point out that row/column-wise storage of the grid matrix is not optimal. To fully exploit cache lines, one would want the neighbors of a cell to be close in memory, so that when reading the cell, its neighborhood is also loaded.

- Regarding SSE3 vector registers (128 bit), one could load the neighbors of 2 contiguous cells into a `__m128i` vector using `_mm_setr_epi8()`. However, SSE3 does not provide an `hadd()` function for 8-bit integers (`char`) to add up the states of the neighbors. Therefore, one would have to use `short` (16 bit), which allows to load the neighbors of a **single** cell with `_mm_setr_epi16()` and accumulate them with `_mm_hadd_epi16()`.

Finally, there exist more complicated approaches to Game of Life, including:

- Gosper's Algorithm (Hashlife) [2], a recursive method which is designed to perform cheap long-term time stepping by identifying regularities on the grid. If two regions are identical, their non-boundary cells will also be identical on the next time step, since they are only affected by cells within the region (see Fig. 6). Hence, one only needs to iterate over the non-boundary cells of 1 region, getting all the other identical regions for free [4].

- Identifying inactive regions on the grid [3]. If a cell nor its neighbors have changed state at time τ, the cell is guaranteed to keep its state at time $\tau + 1$. This reasoning can be scaled for regions. If all cells in a 4×4 region have not changed at time τ, nor have its neighboring 4×4 regions, all cells in the central region are guaranteed to keep their state at $\tau + 1$.

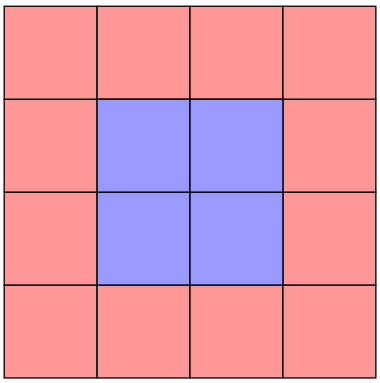


Figure 6: 4×4 region where red squares are boundary cells and blue squares are non-boundary cells.

If one were to parallelize these complicated adjustments, one would have to pay close attention to load balancing, since each cell/region is not guaranteed to require the same amount of work anymore. It would be beneficial to employ dynamic load balancing, so that one thread creates tasks and other available threads pick them up, finish them, and come back for more work if needed. On top of that, one could attempt to get rid of the global barrier at the end of each time step. For instance, if rows 1, 2 and 3 are at time τ, there is no need to wait for all the other rows to be updated before computing row 2 at $\tau + 1$. Hence, using `pthreads`, the global barrier could be substituted by `mutex` and `cond` (condition) variables.

References

[1] J. Conway. The game of life. *Scientific American*, 223(4):4, 1970.

[2] R. W. Gosper. Exploiting regularities in large cellular spaces. *Physica D: Nonlinear Phenomena*, 10(1-2):75–80, 1984.

[3] A. Hensel. About my Conway's Game of Life Applet, jan 2001.

[4] J. Owen. Complexity Measures on the Game of Life. Master's thesis, University of York, 2008.

[5] J. Rantakokko. Lecture notes in High Performance Programming, January 2021.