An Optimized Conway's Game of life version with Pthreads and OpenMP

Xabier García Andrade

UPPSALA
UNIVERSITET

22th of March, 2019

Contents

1 Theoretical Introduction

Game of life is a cellular automaton devised by the British mathematician John Horton Conway in 1970. First I should introduce the concept of cellular automaton. It is a class of mathematical object that consists of:

- Grid of cells
- Set of allowed states for each cell. Each assignment of an state to every cell is known as a configuration.
- A neighbourhood which defines the range of the interaction among cells
- A set of rules that specify how new states are produced.

The state of the cellular automaton evolves in discrete time, and the state in the next step is determined by the previous generation according to the transition rule previously specified.

As stated in the first sentence, Conway's Game of Life is an example of a cellular automaton. It is a zero-player game because it does not require any interaction from a human player. The outcome of the game will be determined by the initial configuration of states alone. The player only interacts by means of setting an initial condition and then observing how the system develops different patterns.

The transition rules that specify which cell should live or die in the next generation are:

- Any live cell with fewer than two live neighbours dies, as if by underpopulation.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overpopulation.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

Every generation update is a pure function of the preceding one. Then these rules are continuously applied to create further generations.

When creating rules for cellular automatons, Conway's target was to build an unpredictable behaviour. With his final set of rules, he prevented explosive growth but admitted small initial patterns with chaotic behaviour and the potential for von Neumann universal constructors (self-replicating machines).

These characteristics made Game of Life attractive for scientist in various fields because it provides an example of emergence and self-organization. It can also serve as an analogy to explain that design and organization can occur spontaneously without an external designer.

Game of Life rules could be stated more formally mathematically speaking by resorting to logic. Nevertheless, for practical reasons in this report the previous formal explanation would suffice. The next step is discussing our code that simulates this cellular automaton.

2 Selection of the World

Depending on the selection of the boundary conditions, two different topologies can be obtained for the universe where the simulation happens. Under the assumption that all cells outside the universe are non-existing, a plane-like topology would be obtained. This would mean that the neighbours of a cell at the border, are treated as if they were dead. The other option is to treat cell in the border as if they were linked to the opposing one (using periodic boundary conditions). This strategy would lead to a toroidal topology.

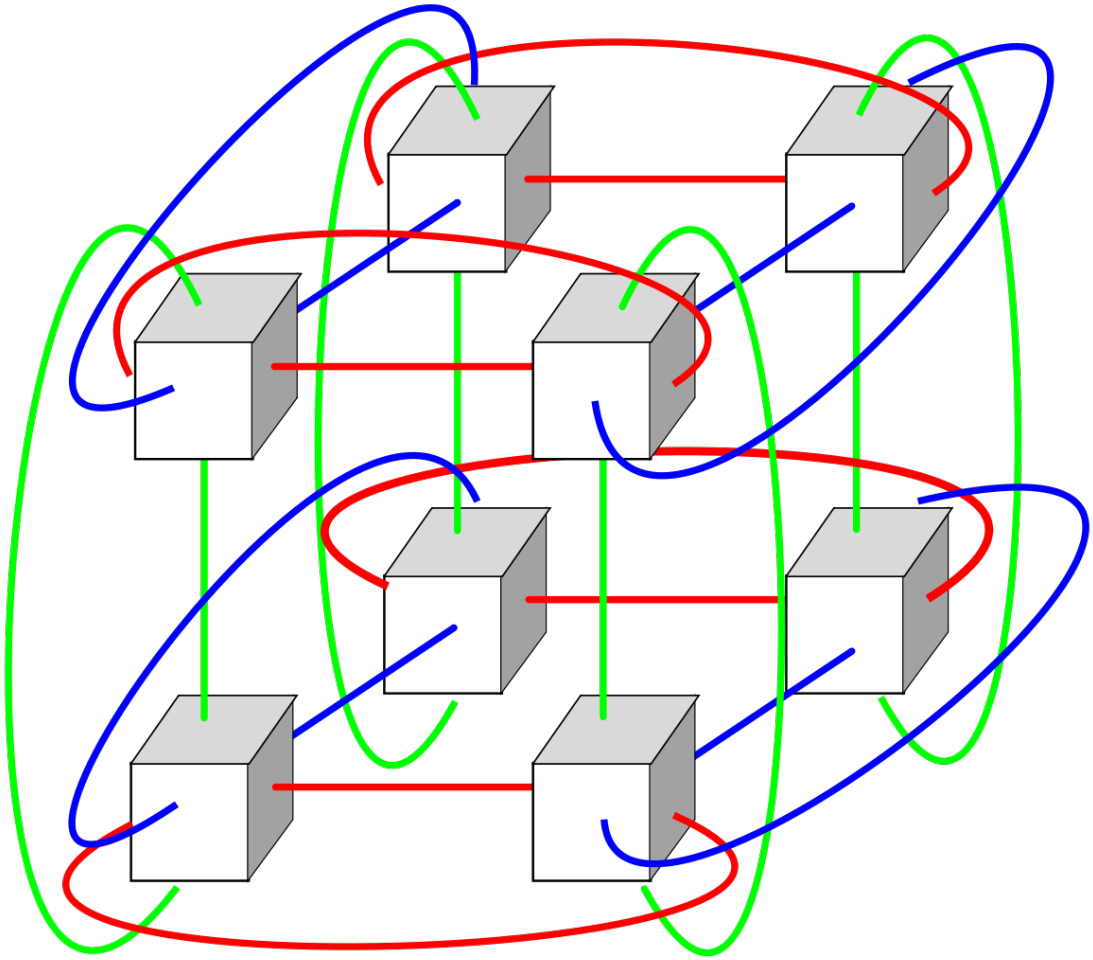

Figure 1: Toroidal Topology

For this project, I opted for the former because periodic boundary conditions are unstable with the size of the problem that I tackle in this project.

3 Test Instructions and Error Treatment

I may mention that in order to measure the performance of every modification, I executed the script 100 times and calculated the mean and the standard deviation of the execution times. Then, each time is expressed as the mean value and a confidence interval spanned by

its standard deviation up to two significant figures (so that it is consistent with common error theory conventions).

Now I proceed to discuss how the error of the physical quantities that we measured (time) was treated. The obtained execution time measurements are used to calculate other values, namely speedup, whose error has to be propagated according to the following formula:

$$\sigma_{(y)} = \sqrt{\sum_{i}^{n} \left[\left(\frac{\partial y}{\partial x_i} \right)^2 \cdot \sigma_i^2 \right]} \tag{1}$$

Where $\sigma_i$stands for the error of the ith variable. In all of our cases, it would be equal to the standard deviation of the time samples. In this assignment, we only needed to calculate a single magnitude that required error propagation. That was the speedup, which is defined as:

$$S = \frac{T_{old}}{T_{optimized}} \tag{2}$$

Then equation (1) would turn to:

$$\sigma_{(S)} = \sqrt{\left(\frac{1}{T_{optimized}} \right)^2 \sigma^2_{(T_{old})} + \left(\frac{-T_{old}}{T^2_{optimized}} \right)^2 \sigma^2_{(T_{optimized})}} \tag{3}$$

Unless stated otherwise, the timings were carried out with $N = 800$, $M = 800$ and $n_{steps} = 500$ using the following computer:

Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz - Ubuntu 16.04 - x86 architecture

3.1 Optimization Flags

In every change or micro-optimization that I performed on the program, I tested it and timed it with different optimization flags, just to be sure that I was capitalizing on compiler perks as much as possible. Apart from the most fundamental ones **(-O1 -O2 -O3)**, I decided to add as well **-ffast-math** and checked that it did not lead to wrong results. Then, since all tests were carried out on my computer, I could not use every optimization flag related to auto-vectorization. I did not include autovectorization flags such as **-ftree-vectorize** because this is already automatically enabled with the -O3 flag.

Regarding vectorization flags, I found that only SSE extensions were available in my computer. After issuing the command "cat /proc/cpuinfo", I found that the latest SSE extension available was **-msse4.2** , which is the one that was tried in all of our tests , as well as the -march=native. With this enabled, the goal was to make code as appropriate for autovectorization as possible. Then I also tried the **-funroll-loops** but with the **-O3** flag it did not have any effect. After checking gcc manual, I understood that **-funroll-loops** was automatically enabled every time **-O3** flag was enabled, so I decided that it would not be necessary and it would just jeopardize readability.

4 Straight Forward Implementation

First of all, we needed to think about an structure to store our world. After thinking about what data structure should be used, the final resolution was to create a two dimensional matrix $N \times M$, where the dimensions are read as arguments to our main subroutine. Each element inside this matrix corresponds to the following struct:

```
typedef struct _cell{
  struct _cell** neighbours;
  int current;
  int next;
} cell;
```

Listing 1: "Choice of Struct"

That is, each element would contain a cell which inside contains an array of other cells for a reason that will become clear in the following step. Apart from that, this cell should have two extra members, its current status (dead or alive) and the status in the next generation. In every iteration, we must store the value of the next generation so that it does not affect the outcome of the other cells.

After our world is created, we must specify the initial conditions in every cell. I used the rand() function inside C standard library to generate random numbers between 0 and 1 and then select either 1 or 0.

```
void seed_cells(cell** cells_grid , int N , int M){

  for (int i = 0 ; i<N ; i++){
    for (int j = 0 ; j<M ; j++){
      int cur = rand()%2;
      cells_grid[i][j].current = cur;
      cells_grid[i][j].next = cur;
    }
  }
}
```

Listing 2: "Initial Conditions"

Then the next conceptual step that I needed to revise was how we managed each cell's neighbours. Using the cell struct, I could first think that I could seek the cell's neighbours in each generation. Nevertheless, this implementation would not be very efficient because it would not not preserve **data locality**, which is key to the performance of the program. Instead of doing this, I thought of storing the pointer to each cell in the array containing the neighbours, rather than the content of each cell. This way, I would only have to run the function find neighbours once and then we would just need to iterate over the array of neighbours (size 8) so that we improve **data locality** by fitting data into cache lines in a sequential fashion.

In order to illustrate our method of finding neighbours, I include a snippet of the overhead of the function as well as part of the function. In particular, it represents how neighbours were obtained for cells that were not in any of the borders. For cells that are in the borders and in the corners we should have a special treatment. For readability reasons, The whole body of the function is included in the Appendix.

```
void obtain_neighbours(cell** cells_grid , int N , int M , cell* b ){
```

```
  for (int i = 1 ; i < (N-1) ; i++){
    for (int j = 1 ; j < (M-1) ; j++){
      cells_grid[i][j].neighbours = (cell**)malloc(8*sizeof(cell*));

      cells_grid[i][j].neighbours[0] = &cells_grid[i-1][j-1];
      cells_grid[i][j].neighbours[1] = &cells_grid[i-1][j];
      cells_grid[i][j].neighbours[2] = &cells_grid[i-1][j+1];
      cells_grid[i][j].neighbours[3] = &cells_grid[i][j-1];
      cells_grid[i][j].neighbours[4] = &cells_grid[i][j+1];
      cells_grid[i][j].neighbours[5] = &cells_grid[i+1][j-1];
      cells_grid[i][j].neighbours[6] = &cells_grid[i+1][j];
      cells_grid[i][j].neighbours[7] = &cells_grid[i+1][j+1];

        }
    }
}
```

Listing 3: "Find Neighbours"

Now that our world is initialized and the connections among neighbours established, we can start to simulate the evolution of the system. The simulation should follow the rules explained in the theoretical introduction, as it is shown in the next snippet:

```
void evolution(cell* cell){
  int counter = 0;
  for (int i = 0 ; i < 8 ; i++){
    counter+=(*cell).neighbours[i]->current;
  }
  if ((*cell).current == 1){
    if (counter<2){
      (*cell).next = 0;
    }
    else if(counter == 2 || counter == 3){
      (*cell).next = 1;
    }
    else{
      (*cell).next = 0;
    }
  }
  else{
    if (counter ==3){
      (*cell).next = 1;
      }
    }
  }
```

Listing 4: "Evolution of the system"

A counter is initialized that counts the number of living cells among the neighbours. Then, depending on the result of the counter and the current status of the cell itself, a decision on whether this cell should live of die in the next generation will be made.

After obtaining the status of every cell for the next generation, an update is carried out by the generation update function:

```
void generation_update(cell* cell){
  cell->current = cell->next;
```

```
}
```

Listing 5: "Generation Update"

This is done for every cell in each generation inside main subroutine:

```
for (int k = 0 ; k < n_steps ; k++){
  for (int i = 0 ; i < N ; i++){
    for (int j = 0 ; j < M ; j++){
        cell* pointer = &cells_grid[i][j];
        evolution(pointer);
    }
  }
  for (int i = 0 ; i < N ; i++){
    for (int j = 0 ; j < M ; j++){
        generation_update(&cells_grid[i][j]);
    }
  }
}
```

Listing 6: "Main Function"

In every generation, the content of the matrix is printed to the screen or to an external windows by means of Xl1 library (this will be discussed in the following section).

Using this implementation with the input specified in previous sections we obtained the following results:

Time (s)	Time (s)	Time (s)	Time (s)	Time (s)	Time (s)
-O1	-O2	-O3	-O3 -ffast-math	-O3 -ffast-math -march=native	-O3 -ffast-math -msse4.2
3.16 ± 0.20	3.14 ± 0.24	2.80 ± 0.26	2.678 ± 0.035	2.664 ± 0.016	2.714 ± 0.060

5 Debugging

5.1 Valgrind

In order to find memory leaks and solve segmentation faults, I resorted to Valgrind. There were many opportunities were this tool was useful, but we will show an example. When I was first creating the serial version, the output from Valgrind memcheck was the following:

```
Memcheck, a memory error detector
Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
Command: ./gameoflife 5 4 10

Use of uninitialised value of size 8
   at 0x400805: seed_cells (gameoflife.c:55)
   by 0x40074E: main (gameoflife.c:36)
 Uninitialised value was created by a heap allocation
   at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
   by 0x4006F8: main (gameoflife.c:30)

Invalid write of size 4
   at 0x400805: seed_cells (gameoflife.c:55)
   by 0x40074E: main (gameoflife.c:36)
 Address 0x8 is not stack'd, malloc'd or (recently) free'd


Process terminating with default action of signal 11 (SIGSEGV)
 Access not within mapped region at address 0x8
   at 0x400805: seed_cells (gameoflife.c:55)
   by 0x40074E: main (gameoflife.c:36)
 If you believe this happened as a result of a stack
 overflow in your program's main thread (unlikely but
 possible), you can try to increase the size of the
 main thread stack using the --main-stacksize= flag.
 The main thread stack size used in this run was 8388608.

HEAP SUMMARY:
    in use at exit: 296 bytes in 5 blocks
  total heap usage: 5 allocs, 0 frees, 296 bytes allocated

LEAK SUMMARY:
   definitely lost: 0 bytes in 0 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 296 bytes in 5 blocks
        suppressed: 0 bytes in 0 blocks
Rerun with --leak-check=full to see details of leaked memory

For counts of detected and suppressed errors, rerun with: -v
ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Figure 2: Valgrind Memcheck

After analyzing this output, it showed that not all memory allocated for our matrix was being freed properly. The reason behind this was that the boundaries of the loops were reversed. Apart from checking memory leaks, Valgrind can also be used to check errors:

```
HEAP SUMMARY:
    in use at exit: 0 bytes in 0 blocks
  total heap usage: 13 allocs, 13 frees, 1,768 bytes allocated

All heap blocks were freed -- no leaks are possible

ERROR SUMMARY: 14 errors from 1 contexts (suppressed: 0 from 0)

14 errors in context 1 of 1:
Conditional jump or move depends on uninitialised value(s)
   at 0x4C2EDA1: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
   by 0x400C61: free_neighbours (gameoflife.c:194)
   by 0x40078A: main (gameoflife.c:42)
 Uninitialised value was created by a heap allocation
   at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
   by 0x40072B: main (gameoflife.c:32)

ERROR SUMMARY: 14 errors from 1 contexts (suppressed: 0 from 0)
```

Figure 3: Valgrind Error Check

In this case, the errors happened because of uninitialized variables. The reason behind this was that the struct that I was using contained an array of the same struct and its content was not set to zero upon initialization.

6 Profiling

In order to find which functions and parts of the program are the most time consuming, I used gprof tool for profiling. It was used with optimization flags disabled (**-O0**). First I will discuss the output from the flat profile, which shows how much time was spent in each function:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 81.48      4.86     4.86 320000000     0.00     0.00  evolution
 14.11      5.70     0.84                             main
  4.20      5.95     0.25 320000000     0.00     0.00  generation_update
  0.34      5.97     0.02        1    20.03    20.03  obtain_neighbours
  0.00      5.97     0.00        1     0.00     0.00  free_neighbours
  0.00      5.97     0.00        1     0.00     0.00  seed_cells
```

Figure 4: Flat Profile

From the flat profiler I can observe immediately that the most time consuming function is "evolution", which I could have expected because of the number of iterations that must be performed inside. All my optimization plans from this point will be targeting this function and how it is used inside main subroutine. The following step is revising the call graph , which shows the relation among functions, as well as how many times each function was called.

```
                     Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.17% of 5.97 seconds

index % time    self  children    called     name
                                                <spontaneous>
[1]    100.0    0.84    5.13                 main [1]
                4.86    0.00 320000000/320000000     evolution [2]
                0.25    0.00 320000000/320000000     generation_update [3]
                0.02    0.00       1/1           obtain_neighbours [4]
                0.00    0.00       1/1           seed_cells [6]
                0.00    0.00       1/1           free_neighbours [5]
-----------------------------------------------
                4.86    0.00 320000000/320000000     main [1]
[2]     81.4    4.86    0.00 320000000         evolution [2]
-----------------------------------------------
                0.25    0.00 320000000/320000000     main [1]
[3]      4.2    0.25    0.00 320000000         generation_update [3]
-----------------------------------------------
                0.02    0.00       1/1           main [1]
[4]      0.3    0.02    0.00       1         obtain_neighbours [4]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[5]      0.0    0.00    0.00       1         free_neighbours [5]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[6]      0.0    0.00    0.00       1         seed_cells [6]
-----------------------------------------------
```

Figure 5: Call Graph

This information reinforces what I already knew. In every iteration within the main subroutine, evolution and generation-update functions are called, without producing any children functions. The majority of the time is spent in evolution. I can conclude that improving this function would lead to major enhancements in timings.

6.1 Cachegrind

Valgrind is an extremely powerful tool that can be used to extract even more information on the performance of our code. First I use it as a branch-prediction profiler. Branch mispredictions force the pipeline to be flushed, rolled back and restarted. Valgrind provides the number of conditional branches executed and the number of conditional branches mispredicted. The result was the following:

```
I   refs:      3,137,064,444

Branches:        265,110,569  (264,989,808 cond + 120,761 ind)
Mispredicts:      23,844,788  ( 23,844,661 cond +     127 ind)
Mispred rate:            9.0% (       9.0%     +     0.1%   )
```

Figure 6: Branch-Profiler

The misprediction rate is relatively high, meanining that it is a problem that I could solve to

improve performance. The next step is finding where most of the branches are mispredicted:

```
--------------------------------------------------------------------------------
          Ir          Bc        Bcm      Bi Bim
--------------------------------------------------------------------------------
3,137,064,444 264,989,808 23,844,661 120,761 127  PROGRAM TOTALS

--------------------------------------------------------------------------------
          Ir          Bc        Bcm      Bi Bim  file:function
--------------------------------------------------------------------------------
2,269,800,211 222,060,276 23,636,834       0   0  /home/xabierga/Game_of_life/gameoflife_timings.c:evolution
  641,812,532  40,401,905    201,113       0   0  /home/xabierga/Game_of_life/gameoflife_timings.c:main
  200,000,000           0          0       0   0  /home/xabierga/Game_of_life/gameoflife_timings.c:generation_update
    8,475,684      39,800        206       0   0  /home/xabierga/Game_of_life/gameoflife_timings.c:obtain_neighbours
    5,987,612     683,063        117       0   0  /build/glibc-LK5gWL/glibc-2.23/malloc/malloc.c:_int_malloc
```

Figure 7: CG annotate output

As expected, the function that calculates the state of each cell in the next generation is the one with the worst result. This could be foreseen because I already detected that it was the most time consuming function.

Other feature that can condition performance are cache misses. Cache works more efficiently if the data is stored in nearby locations. A cache miss in L1 level can cost around 10 cycles, but if it happens at LL level, it can cost as much as 200 cycles. Using valgrind cachegrind the output generated is:

```
I   refs:      606,584,945
I1  misses:          1,190
LLi misses:          1,168
I1  miss rate:        0.00%
LLi miss rate:        0.00%

D   refs:      409,171,502  (384,545,351 rd   + 24,626,151 wr)
D1  misses:     35,361,086  ( 35,280,098 rd   +     80,988 wr)
LLd misses:         62,571  (      1,980 rd   +     60,591 wr)
D1  miss rate:         8.6% (        9.2%     +        0.3%  )
LLd miss rate:         0.0% (        0.0%     +        0.2%  )

LL refs:        35,362,276  ( 35,281,288 rd   +     80,988 wr)
LL misses:          63,739  (      3,148 rd   +     60,591 wr)
LL miss rate:          0.0% (        0.0%     +        0.2%  )
```

Figure 8: Cache misses Profiler

The miss rate at LL level is close to 0, which is a feature of my code that I will try to preserve. At D1 level the percentage of misses is 8.6%. Whereas it is not the most time consuming aspect in the code, I will amend it. Separating the misses by function:

```
--------------------------------------------------------------------------------
         Ir  I1mr  ILmr          Dr       D1mr  DLmr         Dw   D1mw   DLmw
--------------------------------------------------------------------------------
606,584,945 1,190 1,168 384,545,351 35,280,098 1,980 24,626,151 80,988 60,591  PROGRAM TOTALS

--------------------------------------------------------------------------------
         Ir I1mr ILmr          Dr       D1mr DLmr         Dw   D1mw   DLmw  file:function
--------------------------------------------------------------------------------
589,173,660   12   12 380,201,120 35,176,530    1 22,077,886     24     24  /home/xabierga/Game_of_life/gameoflife_timings.c:main
  5,987,612   36   36     763,526         11    1  1,005,065 40,181 40,173  /build/glibc-LK5gWL/glibc-2.23/malloc/malloc.c:_int_malloc
  2,988,309   26   25     728,407     40,016    0    323,828 10,001      0  /build/glibc-LK5gWL/glibc-2.23/malloc/malloc.c:_int_free
  1,875,647   25   25     958,608         28    0    400,609 20,003 10,000  /home/xabierga/Game_of_life/gameoflife_timings.c:obtain_neighbours
  1,849,350    6    6     522,644          7    0    120,611      0      0  /build/glibc-LK5gWL/glibc-2.23/malloc/malloc.c:malloc
  1,045,304    3    3     321,632        604    0    201,020      0      0  /build/glibc-LK5gWL/glibc-2.23/malloc/malloc.c:free
  1,038,710    2    2     320,000          4    2    160,000      0      0  /build/glibc-LK5gWL/glibc-2.23/stdlib/random_r.c:random_r
    732,096   12   12     164,485     49,991    0     82,677     36     34  /build/glibc-LK5gWL/glibc-2.23/malloc/malloc.c:malloc_consolidate
    680,000    3    3     240,000          1    1     40,000      0      0  /build/glibc-LK5gWL/glibc-2.23/stdlib/random.c:random
```

Figure 9: CG annotate output

Most of the misses happen when allocating or freeing memory dynamically. This means that the error would be hard to detect and fix.

Using less space in memory can also impact performance. Valgrind again provides a tool that can be used as a heap profiler, which measures how much heap memory a program uses.

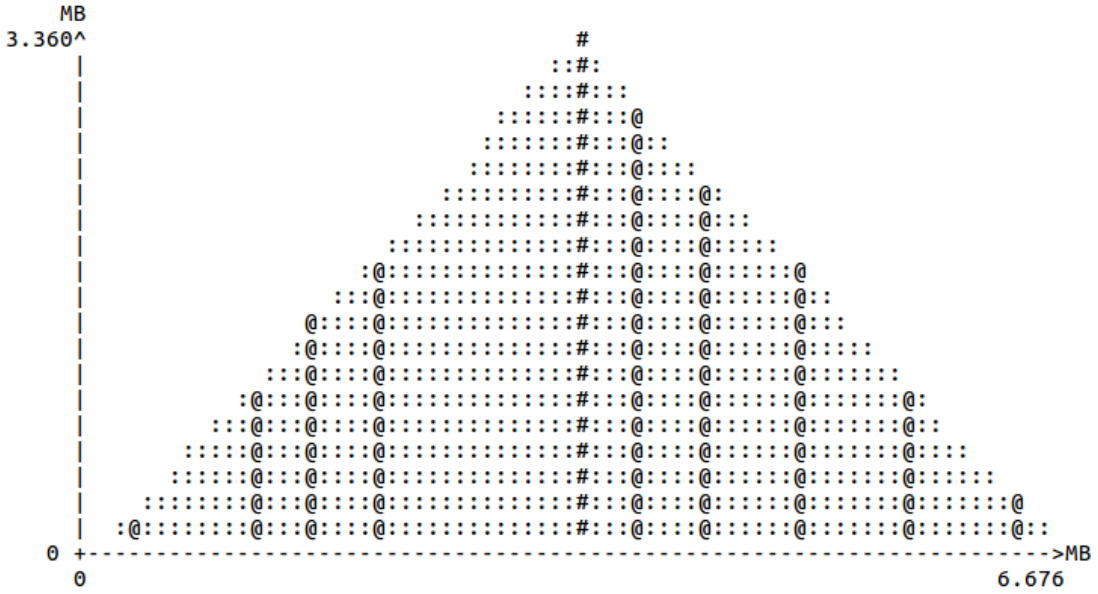

Figure 10: Massif Heap Profiler

The # symbol represents the peak of heap memory usage. It corresponds to the call of the function "obtain neighbours" (found in the appendix). It is reasonable since arrays are allocated in each cell of the world to store its neighbours. The output has this pyramidal shape because it has to account to a linear allocation and posterior freeing of the arrays.

Now that I understood the fundamental functioning of my code, I should construct and develop all the possible optimizations keeping this information in mind.

7 Graphics

Using graphics library with support for C, I made another program with output to a visual interface which helps to understand how the game works. The following pictures show screenshots from the game:

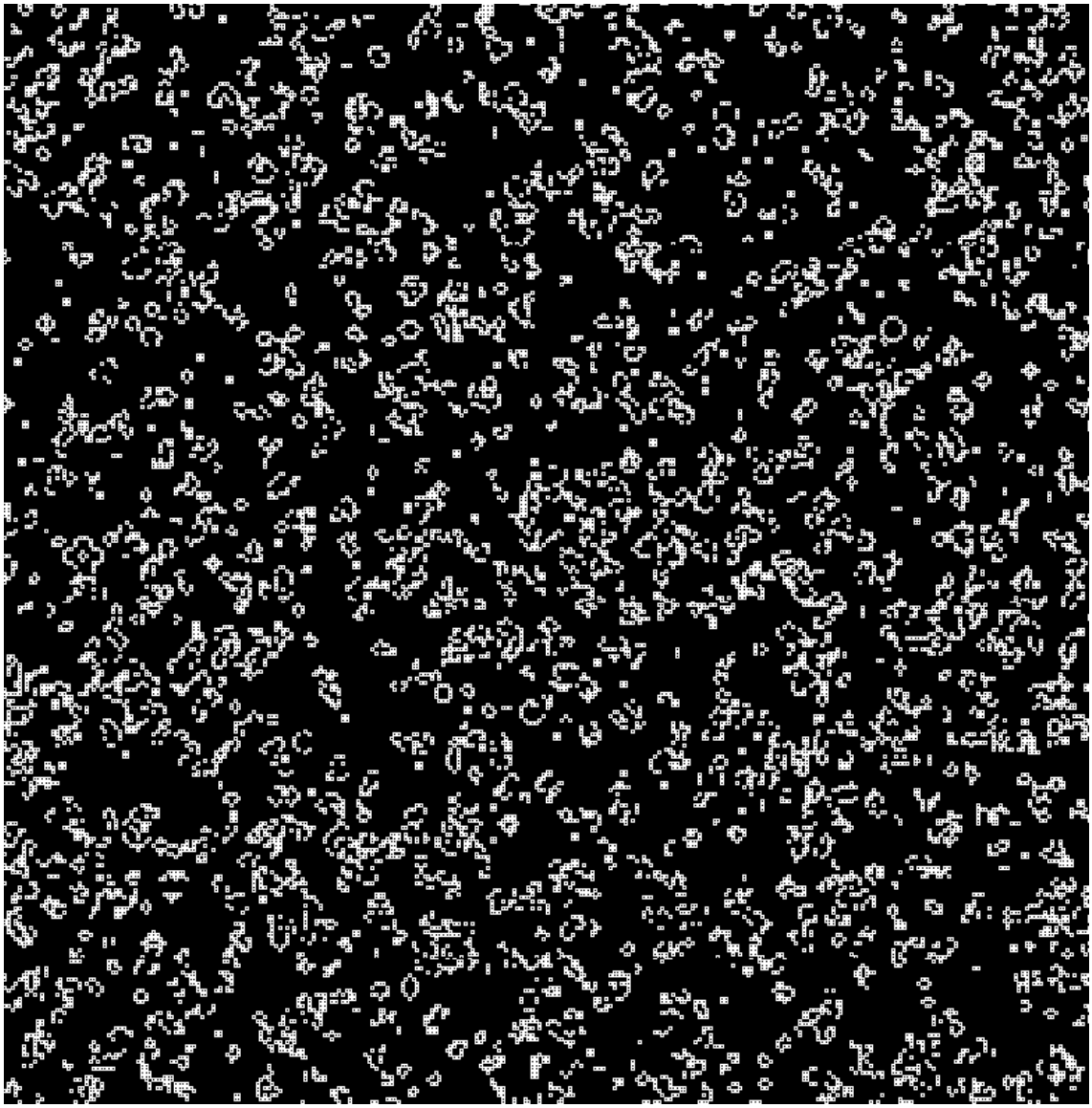

Figure 11: Game of Life

Another screenshot using a different number of rows and columns:

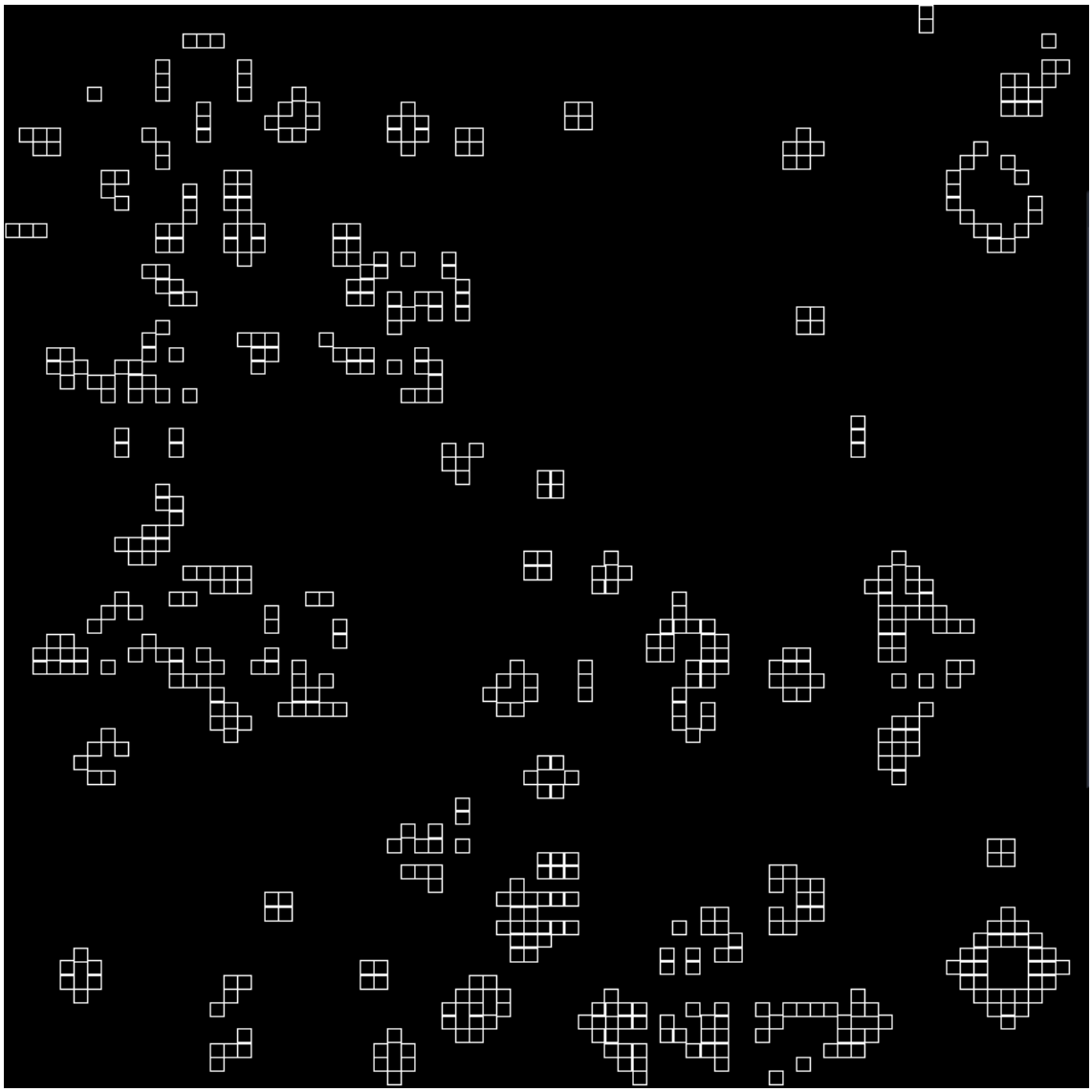

Figure 12: Game of Life

These screenshots were taken after 100 generations initializing the cells at random. It is possible to obtain special structures by picking explicitly the initial conditions. Nevertheless, this is not the scope of this project. Now I should begin with the possible micro-optimizations.

8 Micro-optimizations

8.1 Declaring variable with const

If a variable is not changed during the execution of the program, using the const keyword can improve performance.

By explicitly declaring a variable as const, compiler can optimize it away by not providing storage to this variable and rather add it in a symbol table. Subsequent read just need to refer

to this symbol table instead of fetch the value from memory. In our case, declaring the variables obtained as arguments from the main function could lead to performance differences:

```
  const int N = atoi(argv[1]);
const int M = atoi(argv[2]);
  const int n_steps = atoi(argv[3]);
```

Listing 7: "Const Keyword"

Time (s)	Time (s)	Time (s)	Time (s)	Time (s)	Time (s)
-O1	-O2	-O3	-O3 -ffast-math	-O3 -ffast-math -march=native	-O3 -ffast-math -msse4.2
3.07 ± 0.22	3.00 ± 0.44	2.924± 0.052	2.863 ± 0.035	2.809 ± 0.048	2.810 ± 0.027

Since the time was larger in this case, we reverted the changes.

8.2 Declaring pointer with __restrict

restrict keyword can be used in pointer declarations. It means that for the lifetime of the pointer, only the pointer itself or a value directly derived from it will be used to access the object to which it points. This prevents pointer aliasing (memory location accessed using different names) from happening and it helps compiler to optimize the code.

We will measure timings after using restrict on the following functions (only the overhead will be changed):

```
void evolution(cell *restrict cell){
//body of the function
  }

void generation_update(cell *restrict cell){
//body of the function
}
```

Listing 8: "Restrict Keyword"

Time (s)	Time (s)	Time (s)	Time (s)	Time (s)	Time (s)
-O1	-O2	-O3	-O3 -ffast-math	-O3 -ffast-math -march=native	-O3 -ffast-math -msse4.2
3.25 ± 0.27	3.04 ± 0.55	2.975± 0.041	2.89 ± 0.35	2.818 ± 0.048	3.210 ± 0.012

Since this change did not lead to any performance improvement, we checked the assembly code output to understand the compiler interpretation. After examining the assembly code for these functions, we noticed that including the restrict keyword did not affect the assembly code at all. As an example, let's show the output for the second function (using **-O3 -ffast-math** flags):

```
generation_update:
```

```
        mov     eax, DWORD PTR [rdi+12]
        mov     DWORD PTR [rdi+8], eax
        ret
```

Listing 9: "Assembly code output without restrict keyword"

```
generation_update:
        mov     eax, DWORD PTR [rdi+12]
        mov     DWORD PTR [rdi+8], eax
        ret
```

Listing 10: "Assembly code output with restrict keyword"

I showed the output of this function for simplicity reasons, but it can be extrapolated to the other one. In this functions, pointer aliasing is not an issue and then using restrict does not change the output at all.

After thoroughly examining the assembly code for each of my functions, I found a more interesting case where restrict could affect performance:

```
void obtain_neighbours(cell**restrict cells_grid , int N , int M , cell* b ){
//body of the function
```

Listing 11: "Restrict keyword Obtain_Neighbours"

Now examining closely assembly code (only parts that changed are shown in the following snippet):

```
.L5:
        mov     edi, 64
        call    malloc
        lea     rcx, [r14+32]
        mov     QWORD PTR [r14+16], rax
        lea     rsi, [rbx+16]
        mov     QWORD PTR [rax], rbx
        mov     QWORD PTR [rax+32], rcx
        add     rbx, 32
        lea     rcx, [r15+16]
        mov     QWORD PTR [rax+40], r15
        add     r15, 32
        mov     QWORD PTR [rax+16], rbx
        mov     QWORD PTR [rax+24], r14
        mov     QWORD PTR [rax+56], r15
        mov     QWORD PTR [rax+8], rsi
```

Listing 12: "Assembly code without restrict keyword"

In this case, mov instruction is executed right after lea instruction (load effective address) in both r14 and rbx registers. This means that data locality (time) is preserved, since we are accesing the same register sequentially.

```
.L5:
        mov     edi, 64
        call    malloc
        lea     rcx, [r14+32]
        lea     rsi, [rbx+16]
        mov     QWORD PTR [rax], rbx
        mov     QWORD PTR [rax+32], rcx
        add     rbx, 32
        lea     rcx, [r15+16]
        mov     QWORD PTR [rax+40], r15
        add     r15, 32
        mov     QWORD PTR [r14+16], rax
        mov     QWORD PTR [rax+16], rbx
        mov     QWORD PTR [rax+24], r14
        mov     QWORD PTR [rax+56], r15
        mov     QWORD PTR [rax+8], rsi
```

Listing 13: "Assembly code with restrict keyword"

When using the restrict keyword, both addresses are loaded at the same time and then mov instruction is called. This would mean that we would be losing time locality, leading to a decrease in performance. In order to test this hypothesis, we can show a table with the timings using different flags:

Time (s)	Time (s)	Time (s)	Time (s)	Time (s)	Time (s)
-O1	-O2	-O3	-O3 -ffast-math	-O3 -ffast-math -march=native	-O3 -ffast-math -msse4.2
3.25 ± 0.27	3.21 ± 0.55	3.194± 0.023	3.10 ± 0.32	3.098 ± 0.092	3.22 ± 0.17

Our hypothesis was satifisfied, the timings were worse with this change and then we reverted it.

8.3 Packed attribute for Struct

Minimizing the size of structs sometimes can lead to an improvement in performance. Variables of fundamental type are given addresses that are a multiple of the size of the type. The compiler aligns them using some padding so that hardware can work more efficiently. The main drawback is that part of the memory is wasted with the added padding.

Members of a struct are stored consecutively in the order they are declared. The compiler again inserts padding to ensure natural alignment that can be avoided by declaring members hierarchically with decreasing size. The struct itself is aligned according to the size of its largest member.

Using gcc compiler, I can try the packed attribute to remove natural alignment that can lead to a performance gain because of optimization of cache usage. The results were the following:

Time (s)	Time (s)	Time (s)	Time (s)	Time (s)	Time (s)
-O1	-O2	-O3	-O3 -ffast-math	-O3 -ffast-math -march=native	-O3 -ffast-math -msse4.2
3.10 ± 0.98	3.01 ± 0.45	3.032 ± 0.032	2.98 ± 0.75	2.891 ± 0.037	3.21 ± 0.22

It did not grant any performance improvement.

8.4 Loop Fusion and Loop Unrolling

Loop unrolling means rewriting a loop so that more work is being done simultaneously in each iteration. The unroll factor depends on computer architecture. In this implementation, I used loop unrolling since the beginning in my obtain neighbours function, which I will not include in this page for readability reasons but can be found in the Appendix.

Loop fusion means combining two loops that have the same bounds so that they can be parallelized in an easier way. This was done in the same function mentioned above.

8.5 Inlining Functions

Using the inline keyword serves as a compiler directive that suggests that the compiler replaces a function call site with the body of the called function. This optimization would save the overhead of the function call. According to this explanation, we should try to inline the functions that are called more times in our code:

```
1 inline void evolution(cell * cell){
```

```
//body of the function
  }

inline void generation_update(cell * cell){
//body of the function
}
```

Listing 14: "Inlining Functions"

Time (s) -O1	Time (s) -O2	Time (s) -O3	Time (s) -O3 -ffast-math	Time (s) -O3 -ffast-math -march=native	Time (s) -O3 -ffast-math -msse4.2
3.050± 0.047	2.91 ± 0.19	2.732± 0.052	2.735 ± 0.052	2.84 ± 0.16	2.781 ± 0.052

We only obtain a speedup with the **-O1** flag optimization. Then we conclude that the compiler already takes care of inlining functions by itself with the other optimization flags.

8.6 Improving Cache Usage

Increasing the probability that the next data accessed is already loaded on a cache line can lead to major performance gains. If the data is already on cache, spatial data locality is improved. I tried to reorder the inner loop in the obtain neighbours function to see how this affected performance. Instead of ij I tried ji:

Time (s) -O1	Time (s) -O2	Time (s) -O3	Time (s) -O3 -ffast-math	Time (s) -O3 -ffast-math -march=native	Time (s) -O3 -ffast-math -msse4.2
5.071± 0.075	5.38 ± 0.35	5.96 ± 0.11	5.55 ± 0.20	5.79 ± 0.16	5.65 ± 0.20

I found out that ij was the best performing loop order.

8.7 Modifying Bounds Checking and Fast Boolean Evaluations

After reflecting on the conditional branches inside our evolution function, I found out that the conditions could be merged into a single branch. The result was the following:

```
inline void evolution(cell * cell){
  int counter = 0;
  for (int i = 0 ; i < 8 ; i++){
    counter+=(*cell).neighbours[i] >current;
  }
  if (((counter ==2 || counter == 3) && (*cell).current == 1)  || (counter ==3
   && (*cell).current == 0 )) {
    (*cell).next = 0;
    }
```

```
  else{
    (*cell).next = 1;
    }
  }

```

Listing 15: "Bounds Checking Modified"

Here I was looking for Boolean short circuits. The scope of this technique is terminating the evaluation of a Boolean expression in the shortest possible time possible. When I am using logical AND, the most condition most likely to be false is the one that should be placed first. On the other hand, when evaluating conditions with logical OR , the condition most likely to be true should be the first one to be examined.

In this case, (*cell).current can either be 1 or 0, so there is a $\frac{1}{2}$ chance that it will be true. On the other hand, counter can take 8 different values, meaning that there is a $\frac{1}{8}$ chances that it is in each state. Keeping this in mind, I measured the times with this implementation:

Time (s)	Time (s)	Time (s)	Time (s)	Time (s)	Time (s)
-O1	-O2	-O3	-O3 -ffast-math	-O3 -ffast-math -march=native	-O3 -ffast-math -msse4.2
2.854± 0.013	2.761± 0.095	2.758± 0.081	2.581 ± 0.010	2.575 ± 0.094	2.673 ± 0.089

The computed speedup was:

$$S = 1.034 \pm 0.011$$

Which is statistically significant even though there is not a huge impact. It is worth noting that with **-O1** optimization flag, there is a larger speedup and then I can conclude that the compiler takes care of this by itself.

Checking the differences in the Assembly code between **-O1** and **-O2** from our first implemenation:

```
evolution:
        mov     rax, QWORD PTR [rdi]
        lea     rsi, [rax+64]
        mov     edx, 0
.L2:
        mov     rcx, QWORD PTR [rax]
        add     edx, DWORD PTR [rcx+8]
        add     rax, 8
        cmp     rax, rsi
        jne     .L2
        cmp     DWORD PTR [rdi+8], 1
        je      .L8
        cmp     edx, 3
        je      .L9
.L1:
        ret
.L8:
```

```
        cmp     edx, 1
        jle     .L10
        sub     edx, 2
        cmp     edx, 2
        setb    al
        movzx   eax, al
        mov     DWORD PTR [rdi+12], eax
        ret
.L10:
        mov     DWORD PTR [rdi+12], 0
        ret
.L9:
        mov     DWORD PTR [rdi+12], 1
        jmp     .L1
```

Listing 16: "Assembly code -O1"

```
evolution:
        mov     rax, QWORD PTR [rdi]
        xor     edx, edx
        lea     rsi, [rax+64]
.L2:
        mov     rcx, QWORD PTR [rax]
        add     rax, 8
        add     edx, DWORD PTR [rcx+8]
        cmp     rax, rsi
        jne     .L2
        cmp     DWORD PTR [rdi+8], 1
        je      .L11
        cmp     edx, 3
        je      .L4
        ret
.L11:
        sub     edx, 2
        cmp     edx, 1
        ja      .L12
.L4:
        mov     DWORD PTR [rdi+12], 1
        ret
.L12:
        mov     DWORD PTR [rdi+12], 0
        ret
```

Listing 17: "Assembly code -O2"

The main difference is that with **-O2** flag, a bitwise exclusive is carried out at the beginning. This procedure then leads to less "cmp" instructions (comparisons between operands) being called. Now we can examine the assembly code after modifying boundaries (**-O1**):

```
evolution:
        mov     rax, QWORD PTR [rdi]
        lea     rsi, [rax+64]
        mov     edx, 0
.L2:
        mov     rcx, QWORD PTR [rax]
        add     edx, DWORD PTR [rcx+8]
        add     rax, 8
        cmp     rax, rsi
```

```
        jne     .L2
        mov     ecx, DWORD PTR [rdi+8]
        cmp     ecx, 1
        je      .L7
.L3:
        cmp     edx, 3
        sete    al
        test    ecx, ecx
        sete    dl
        and     eax, edx
        xor     eax, 1
        movzx   eax, al
.L4:
        mov     DWORD PTR [rdi+12], eax
        ret
.L7:
        lea     esi, [rdx 2]
        mov     eax, 0
        cmp     esi, 1
        ja      .L3
        jmp     .L4
```

Listing 18: "Modified Boundaries -O1"

The number of "cmp" instructions carried out is the same as in the previous implementation with **-O2** and bitwise exclusive XOR is also carried out in L8. Then, I conclude that with **-O2** or better optimization flags, the compiler already takes care of terminating evaluations in an efficient way.

In an attempt to further reduce the number of evaluations, I tried the following change:

```
inline void evolution(cell * cell){
  int counter = 0;
  for (int i = 0 ; i < 8 ; i++){
    counter+=(*cell).neighbours[i] >current;
  }
  if ((counter ==3) || (counter == 2 && (*cell).current ==1)){
    (*cell).next = 1;
    }
  else{
    (*cell).next = 0;
    }
  }
```

Listing 19: "Modified Boundaries Second Attempt "

Time (s)	Time (s)	Time (s)	Time (s)	Time (s)	Time (s)
-O1	-O2	-O3	-O3 -ffast-math	-O3 -ffast-math -march=native	-O3 -ffast-math -msse4.2
3.320± 0.013	3.21 ± 0.18	3.044± 0.013	3.048 ± 0.022	3.022 ± 0.019	3.044 ± 0.018

Since it lead to a decrease in performance, this modification had to be reverted.

8.8 Strength Reduction

The computational cost of arithmetic operations varies a lot depending on our choice. The cheapest operations are bitwise operations. I decided to use bitwise or in evolution function to check if it affected performance:

```
void evolution(cell * cell){
  int counter = 0;
  for (int i = 0 ; i < 8 ; i++){
    counter+=(*cell).neighbours[i]>current;
  }
  if (!(*cell).current){
    if (counter == 3) {
      (*cell).next = 1;
    }
  }
  else{
    if (!(counter | 1)){
      (*cell).next = 0;
    }
  }
}
```

Listing 20: "Strength Reduction"

Time (s)	Time (s)	Time (s)	Time (s)	Time (s)	Time (s)
-O1	-O2	-O3	-O3 -ffast-math	-O3 -ffast-math -march=native	-O3 -ffast-math -msse4.2
3.93 ± 0.10	3.52 ± 0.11	2.88 ± 0.14	2.919 ± 0.074	2.879 ± 0.025	2.732 ± 0.052

The main difference lies in the least aggressive optimization flags. It is harder for the compiler to interpret conditional branches with **-O1** and **-O2** and this leads to a worse performance.

8.9 Pure Function

A pure function is any function that meets this requirements:

- Its return value is the same for the same arguments.
- Its evaluation does not have side effects.

Theoretically, in Conway's game of life each generation is a pure function of the preceding one. Declaring a function as a pure function can help the compiler perform further optimizations. Declaring my evolution and generation update functions are pure:

```
void generation_update(cell*) __attribute__((pure)) ;
void evolution(cell*) __attribute__((pure)) ;
```

Listing 21: "Pure functions"

With this optimization I obtained the following results:

Time (s)	Time (s)	Time (s)	Time (s)	Time (s)	Time (s)
-O1	-O2	-O3	-O3 -ffast-math	-O3 -ffast-math -march=native	-O3 -ffast-math -msse4.2
3.16 ± 0.18	2.97 ± 0.12	2.901 ± 0.087	2.87 ± 0.18	2.757 ± 0.043	2.78 ± 0.012

Where the speedup $S = 1.013 \pm 0.025$ is not statistically significant. After reflecting on this, I found that gcc enables **-fipa-pure-const** optimization flag already since **-O1**. This optimization flag forces the compiler to search for pure and constant functions, thus explaining my results (**https://embarc.org/man-pages/gcc/Optimize-Options.htmlOptimize-Options**).

9 Parallelization

Traditional processor designs are limited and in order to continue enhancing performance, exploring parallelism could solve many of the previous limitations.

Usually software was written to be run on a single computer with single CPU. Each task was broken into a series of instructions that were executed sequentially.

Parallel computing is a type of computation in which instruction are carried out simultaneously. A problem is divided into tasks that can be tackled concurrently which are then further broken down to a series of instructions.

For this problem, I will be using the global name space model, where each thread can access and change global variables.

After the conclusions derived from the profiler, parallelization should be focused on evolution function.

The first approach will be dividing our world into different regions (rows our columns), so that every thread iterates over a given region in parallel to the other computations. Then, after all the computations in this function are finished, the master thread should wait for synchronization of all the threads before starting evolution function. In order to achieve this, a "Peers" model will be the optimal strategy because the world will be divided into equally spaced regions, so that the work done by each thread will have a similar load balance. The whole process would be a static homogeneous task.

9.1 Pthreads

In order to implement a parallelized version using pthreads, I needed to perform some modifications in my serial version as well. From the profiler output, it was clear that the part that would lead to a better improvement in performance was updating the state of each cell. I decided that, since our world is a two dimensional matrix, dividing the matrix in different block so that each block is only accessed by one thread:

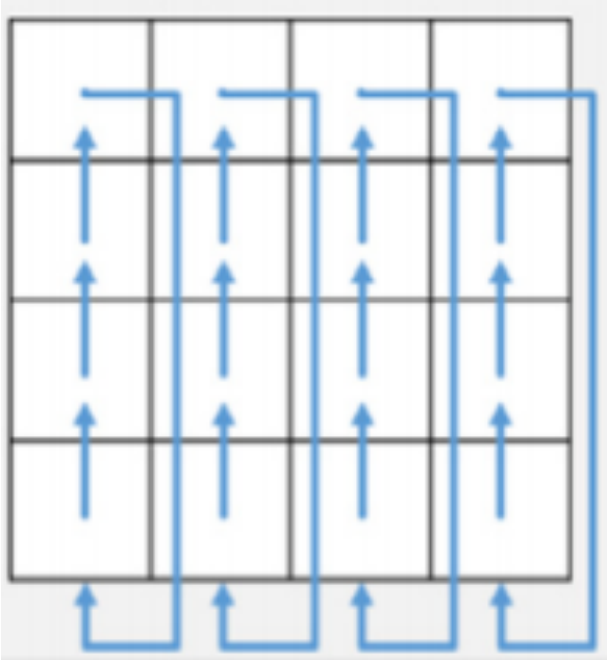

Figure 13: Accessing Columns Simultaneously

In order to obtain the correct flow of information, first all the states in the next generation must be computed, place a barrier to ensure the correct synchronization of all the threads, then proceed to update the current generation and set another barrier again so that all the threads have terminated.

Since Pthreads rely on global shared space model, the following variables are declared as global:

```
//declare variables that will be read as arguments in main function
int N_ROWS, M_COLS , N_STEPS  ,NUM_THREADS;

//set counters and initialize mutex and conditions that will be used in barriers
int counter_update = 0;
int counter_evolve = 0;

pthread_mutex_t lock_update;
pthread_cond_t signal_update;

pthread_mutex_t lock_evolve;
pthread_cond_t signal_evolve;

typedef struct _cell{
  struct _cell** neighbours;
  int current;
  int next;
} cell;

struct parameters{
  int min;
  int max;
};

```

```
25 cell** world;
```

Listing 22: "Global Variables"

The first line contains global variables that correspond to the arguments of the main function. Global variable "world" corresponds to a matrix of our cell structure that will be initialized in the main function so that it can be used within function by every thread. The following function is passed as argument to pthread create:

```
1  void* evolution_threads(void*arg){
2    struct parameters *par = (struct parameters*)arg;
3      for (int i = par->min ;  i < par->max ; i++){
4        for (int j = 0 ; j < N_ROWS ; j++){
5          evolve(i , j);
6        }
7      }
8      barrier_evolving();
9      for (int i = par->min ;  i < par->max ; i++){
10       for (int j = 0 ; j < N_ROWS ; j++){
11         generation_update(i , j);
12       }
13     }
14     barrier_update();
15     return NULL;
16 }
```

Listing 23: "Pthread Create Evolution Function"

It accepts a struct as argument that contains the boundaries for each thread, thus the function knows which columns are accessed by which thread. Then the function iterates over the whole lattice, applying evolve function in every cell. It is the same function as in the serial implementation, but now the lattice is a global variable. Then the first barrier is called:

```
1  void barrier_evolving(){
2    pthread_mutex_lock(&lock_evolve);
3
4    counter_evolve++;
5
6    if (counter_evolve == NUM_THREADS){
7
8      pthread_cond_broadcast(&signal_evolve);
9      counter_evolve = 0;
10   }
11   else{
12
13     pthread_cond_wait(&signal_evolve , &lock_evolve);
14
15   }
16   pthread_mutex_unlock(&lock_evolve);
17   return;
18 }
```

Listing 24: "First Barrier"

This barrier blocks each thread until the counter equals the number of total threads. While waiting, the mutex is released. The broadcast function wakes all threads. This is necessary so that all threads have finished calculating the state of the cell in the next generation before

starting to update world global variable. It is necessary to use another barrier again to ensure synchronization:

```
void barrier_update(){
  pthread_mutex_lock(&lock_update);

  counter_update++;

  if (counter_update == NUM_THREADS){
    //     printf("Releaseing \n");
    pthread_cond_broadcast(&signal_update);
    counter_update = 0;

  }
  else{
  //  printf("Waiting \n");
    pthread_cond_wait(&signal_update , &lock_update);
  }
  pthread_mutex_unlock(&lock_update);
  return;
}
```

Listing 25: "Second Barrier"

This barrier works exactly as the previous one. The next logical step is showing how this is implemented in main function:

```
  for (int t = 0 ; t < N_STEPS ; t++){
    for (int i = 0 ; i < NUM_THREADS ; i++){
      arguments[i].min = i*size_block;
      arguments[i].max = i*size_block + size_block;
  //     printf("i int the loop %d \n" , i);
      pthread_create((void*)&thread_handles[i] , NULL , (void*)evolution_threads
    , (void*)&arguments[i]);
    }
    for (int i = 0 ; i < NUM_THREADS ; i++){
      pthread_join(thread_handles[i] , NULL);
    }
  }
```

Listing 26: "Main Subroutine"

For the total number of steps, n threads are created that execute the function shown above. Now I should reflect on how this implementation works at the level of work sharing and synchronization.

Synchronization in this case is vastly conditioned by the fact that all the threads must finish before starting the next step. With this in mind, I can state that in terms of **synchronization**, this version is the most optimized I could produce.

Since all the parallelized functions carry out the same number of operations, all the threads would have a similar work load. Dividing the work that each thread does statically with equally sized blocks is the most efficient strategy here. Then I could argue that in terms of data sharing, the only variable used within parallelized functions is the world variable, which needs to be shared in order to be accessed by all threads.

The following figure shows the speedup versus different number of cores:

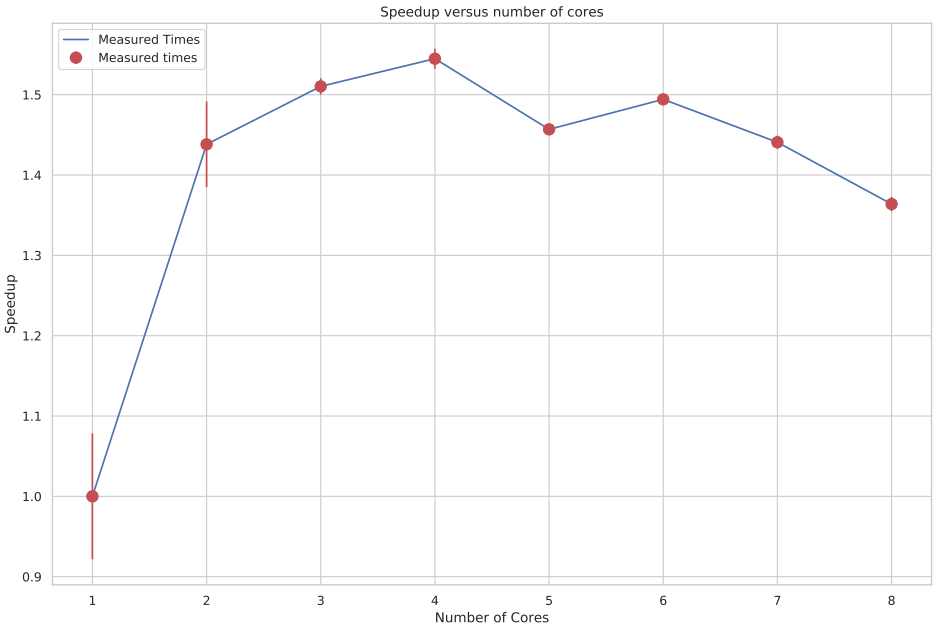

Figure 14: Speedup with different Cores

The speedup has a maximum with 4 cores. There are two key concepts that affect performance in this case **parallel overheads** and **data locality**. The former refers to the fact that when creating different parallel tasks, there is a trade off between the time to coordinate the tasks (synchronize) and the time spent actually doing useful work. The later in this case can be affected because I am dividing work into different block sizes (by rows) and some sizes may lead to a better fit into cache lines. This problems could be solved by increasing the size of the problem, but I am limited by the specifications of my computer.

Even though I obtained a successful result with pthreads, I should explore other options with OpenMP.

9.2 OpenMP

In this section I will try to stick to the same rules as the ones I followed for Pthreads. Starting at my original serial implementation and considering the most time consuming parts of my code, parallelizing the for loops is straight-forward with OpenMP:

```
1  for (int k = 0 ; k < n_steps ; k++){
2    int i , j;
3    #pragma omp parallel for firstprivate(j) schedule(guided,8)
4    for (i = 0 ; i < M ; i++){
5      for (j = 0 ; j < N ; j++){
6          evolution(&cells_grid[i][j]);
7      }
8    }
9    #pragma omp parallel for firstprivate(j) schedule(guided,8)
10   for ( i = 0 ; i < M ; i++){
```

```
11          for ( j = 0 ; j < N ; j++){
12              generation_update(&cells_grid[i][j]);
13          }
14      }
15  }
16
```

Listing 27: "OpenMP Implementation"

I will start measuring the timings with this change and base the corresponding discussion on the results.

Time (s)	Time (s)	Time (s)	Time (s)	Time (s)	Time (s)
-O1	-O2	-O3	-O3 -ffast-math	-O3 -ffast-math -march=native	-O3 -ffast-math -msse4.2
2.153± 0.095	2.021± 0.123	1.99 ± 0.11	2.01 ± 0.12	2.035 ± 0.065	2.12 ± 0.54

9.2.1 Data Sharing

Using OpenMP in this implementation forces some constraints on how data is managed among threads. Since the parallelization comes mostly in terms of for loops, then making it necessary to declare the inner variable as private.

All private variables are allocated on the stack and uninitialized at entry and removed at exit. Since in this case the declared variable (j) is set to 0 in every thread, it would not matter whether I use private, firstprivate or lastprivate.

There is no room for further optimization in terms of data sharing in this implementation, I will now discuss how work sharing affects my code.

9.2.2 Work Sharing

Scheduler

In order to improve the amount of work (**Work Sharing**) used by every thread, I decided to use an scheduler to divide the iteration space into different chunks. First I should explain how every scheduler works.

Static scheduler assigns cyclicly chunks to threads, dynamic means that every time a thread finishes its task, it will be assign a new chunk while guided means the same as dynamic but with chunk size shrinking as the program runs. This behaviour can be better understood with the following picture:

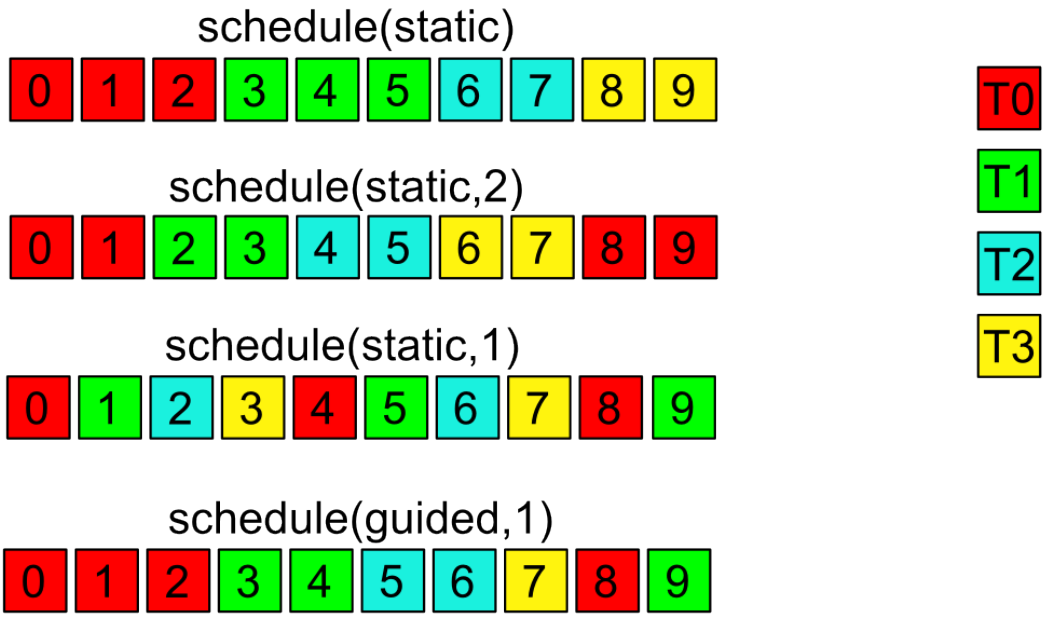

Figure 15: Time vs chunk size

With my implementation, I must expect that static scheduler will show a better performance. Static scheduling preserves data locality, which in this case is a crucial problem. Dynamic and guided scheduling would work better in situations when there is a load imbalance. Since all the members in the grid undergo the same number of operations in every iteration, this should not make a difference. The following fragments show how each scheduling type was implemented:

```
1    #pragma omp parallel for firstprivate(j) schedule(static,x)
```

Listing 28: "Static Scheduler"

```
1    #pragma omp parallel for firstprivate(j) schedule(dynamic, x)
```

Listing 29: "Dynamic Scheduler"

```
1    #pragma omp parallel for firstprivate(j) schedule(guided,x)
```

Listing 30: "Guided Scheduler"

Measuring timings with different chunk sizes could grant a profound insight of the behaviour of this program with respect to different chunk sizes. Running the code with number of threads equal to 8:

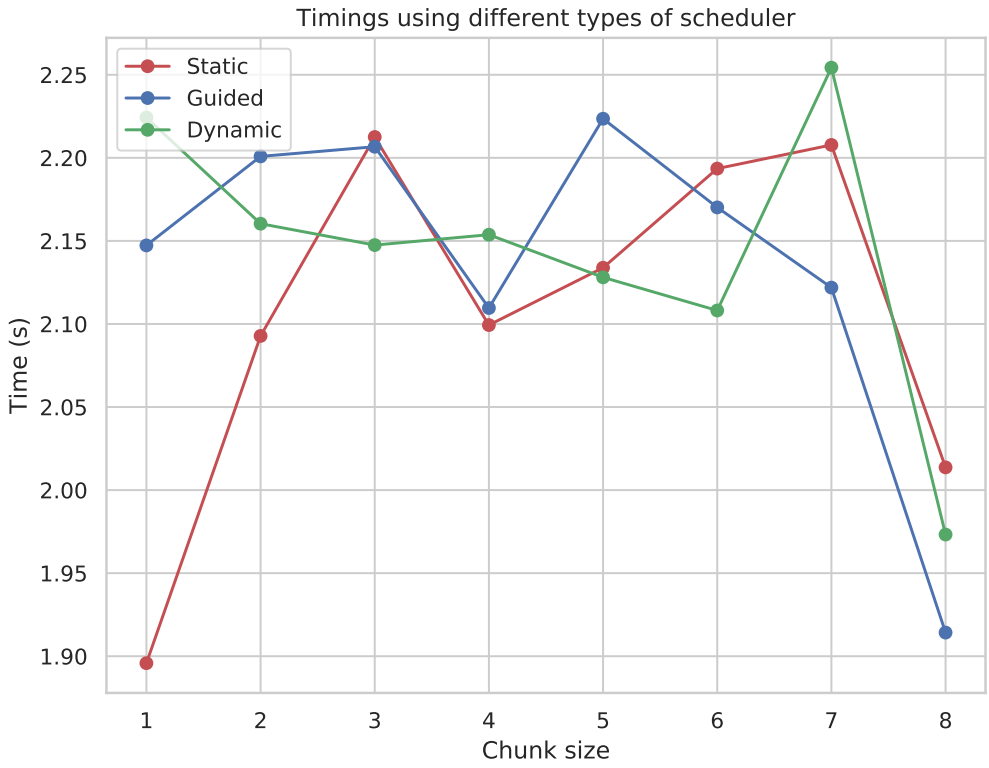

Figure 16: Time vs chunk size

As expected, static scheduler outperformed the other versions, specially when the chunk size equal to 1. Then I measured the timings with different optimization flags to record the speedup:

Time (s)	Time (s)	Time (s)	Time (s)	Time (s)	Time (s)
-O1	-O2	-O3	-O3 -ffast-math	-O3 -ffast-math -march=native	-O3 -ffast-math -msse4.2
1.871± 0.04	2.08 ± 0.23	2.023± 0.078	2.22 ± 0.12	2.022 ± 0.023	2.03 ± 0.64

The calculated speedup was:

$$S = 1.082 \pm 0.012$$

Which is statistically significant and then I decided to keep this modification. It is interesting to notice that the best result in this case was obtained using **-O1** optimization flag. I must conclude from this experiment that a static sheduler in this case enhances **data locality** which leads to a faster execution time.

9.2.3 Serial Sections

If some sections are declared to be executed in a serial fashion (rather than parallelizing everything), performance might be affected. This would mean that threads are not terminated and thus data stored in cache is preserved for different tasks. In this case, it is necessary that most of the parts do not run parallel to each other. Regarding this, I tried to optimize the serial part when the initial conditions are stablished:

```
void seed_cells(cell** cells_grid , int N , int M){
  int j;
  #pragma omp parallel for private(j)
  for (int i = 0 ; i<N ; i++){
    for (j = 0 ; j<M ; j++){
      int cur = rand()%2;
      cells_grid[i][j].current = cur;
      cells_grid[i][j].next = cur;
    }
  }
}
```

Listing 31: "Serial Part"

This lead to the following results:

Time (s)	Time (s)	Time (s)	Time (s)	Time (s)	Time (s)
-O1	-O2	-O3	-O3 -ffast-math	-O3 -ffast-math -march=native	-O3 -ffast-math -msse4.2
1.831± 0.012	1.89 ± 0.24	1.992± 0.057	2.12 ± 0.13	1.875 ± 0.024	2.012 ± 0.543

The measured speedup is:

$$S = 1.023 \pm 0.012$$

Which is statistically significant and thus I resolved to keep this change. Nevertheless, the difference is almost imperceptible (this could be predicted from the output of the profiler, this function was not as time consuming as the others). It would be more relevant for larger sizes, but then again I am constrained by the computer I am using.

9.2.4 Synchronization

In this case the main part where synchronization of the threads occurs is at the end of each for directive. This happens automatically and a memory flush is executed, thus refreshing all the variables shared by the threads. I cannot use the nowait directive in this situation, because each part depends on the previous task. Synchronization with locks would not provide a better solution either, because each thread accesses different parts of the global variable and then there is no need to make this variable accessible to only a specific thread.

The following plot shows the speedup with respect to the number of cores using OpenMP:

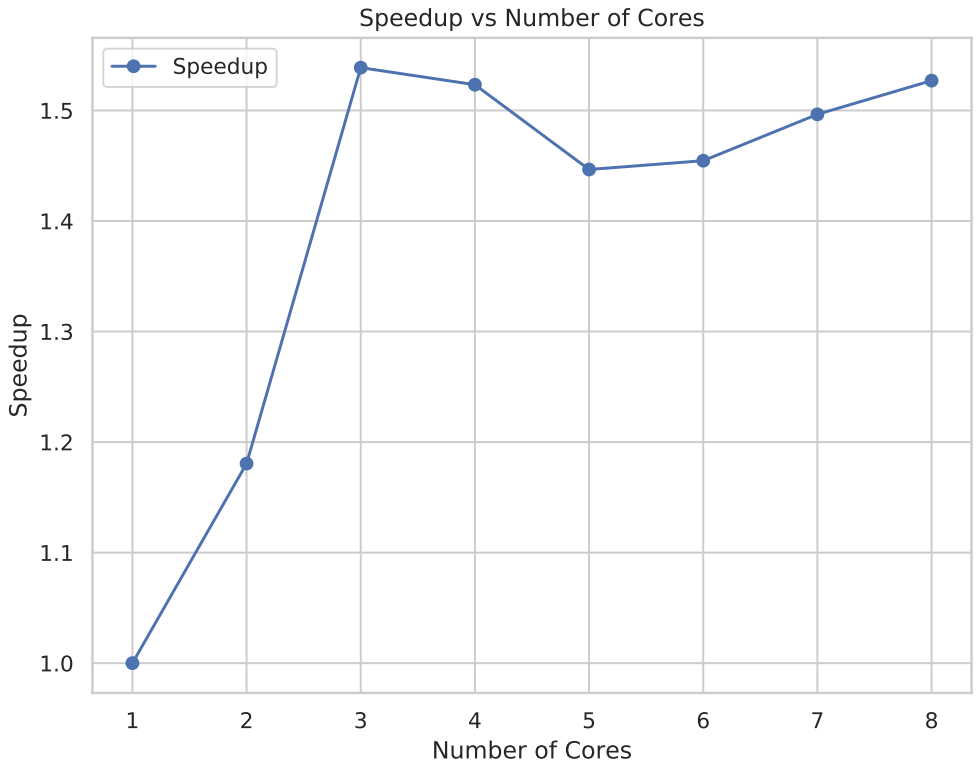

Figure 17: OpenMP Speedup

In this case the speedup suffers from parallel overheads again, making the problem larger would definitely lead to a far better performance.

10 Comparison between Pthreads and OpenMP

Comparing the speedup obtained with different cores for the implementation with Pthreads and the one with OpenMP can inform us about the efficiency of each one. For a more thorough investigation, I will also compute the theoretical maximum.

Then the theoretical maximum speedup can be calculated with Amdahl's law with the following formula:

$$S_{latency}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

Where s is the number of processors and p is the is the proportion of execution time that the part benefiting from improved resources originally occupied. I can find p by resorting to the profiler, which showed that the part benefiting from the parallelization accounts for 81.4 of the work.

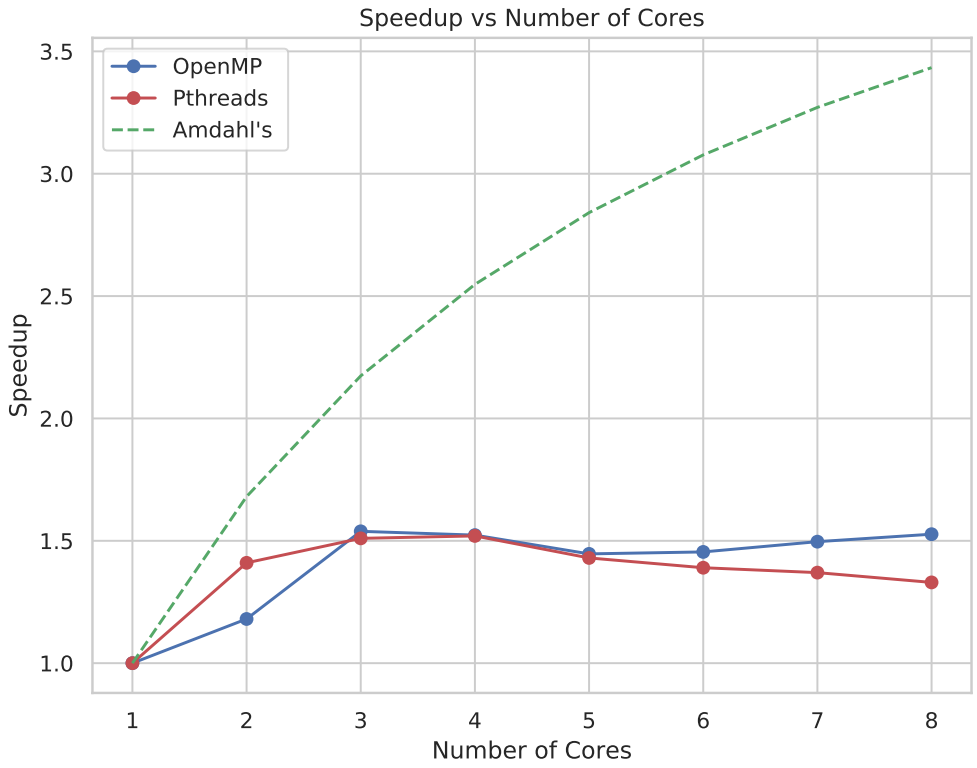

Figure 18: Pthreads vs OpenMP vs Amdahl

The differences between both implementations show that OpenMP scales better with the number of cores. The large difference with respect to the theoretical value can be explained again by parallel overhead. If I could run the problem for a larger size with a computer with more cores, the result would be closer to Amdahl's. Since this plot only takes into account speedup, it is not an honest metric of the difference between the implementation in OpenMP and Pthreads, I will show a plot of the time with respect to the number of elements (considering a square lattice) for different number of cores:

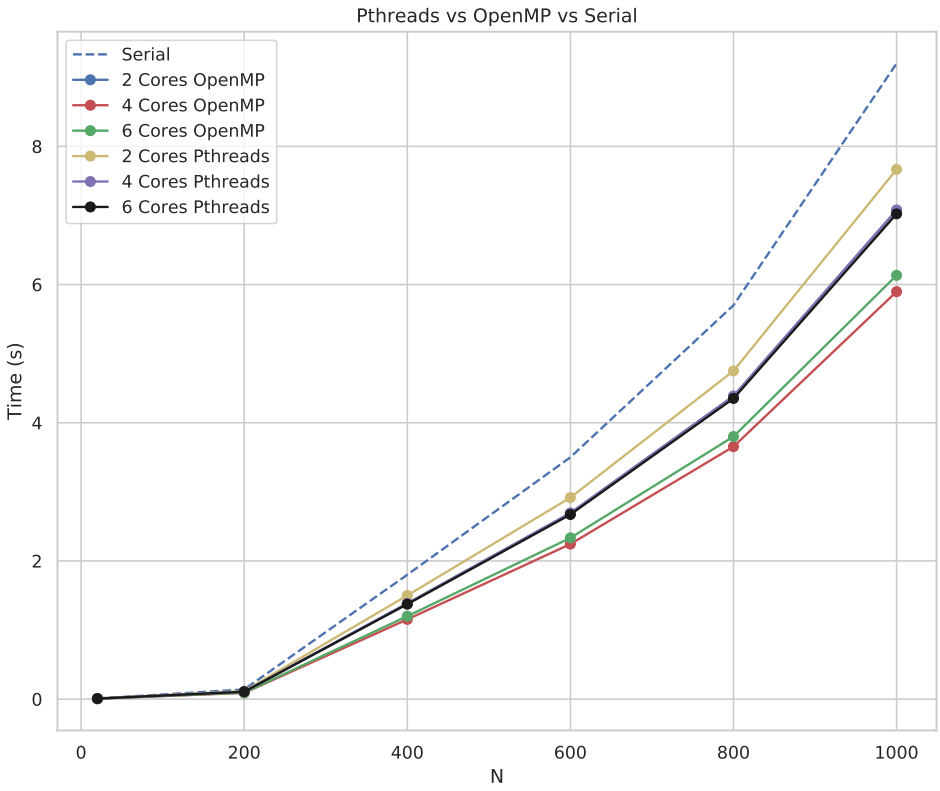

Figure 19: Pthreads vs OpenMP vs Serial

In this case, I can observe that OpenMP outperforms Pthread when the number of cores is higher than 2. Then I can conclude that OpenMP successfully parallelizes the code in a more efficient way in this particular case.

11 Complexity

In order to understand better the performance of the algorithm I designed, it is interesting to plot the timings with respect to different grid sizes. Then, after spotting the tendency and the relation between this variables, I will try to fit the results to a smooth function. The following plot shows the results:

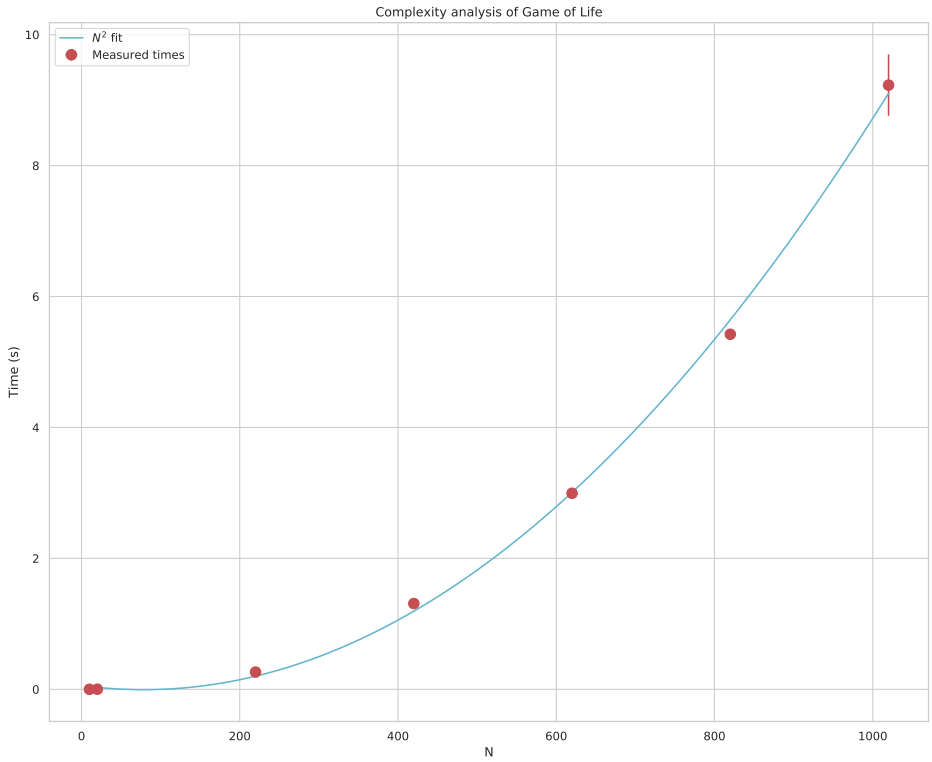

Figure 20: Complexity of Game of Life Algorithm

The function that best fit the experimental data is a quadratic function and therefore I can state that the time complexity of this algorithm $\mathcal{O}(N^2)$.

12 Conclusion

The performance of my design has already been deeply analysed throughout the report, thus I will use this section to collect further improvements and ideas for optimizations. First I shall start by a possible modification to the raw algorithm. When calculating Game of Life long-term simulations of a given starting configuration, the optimal algorithm is called **HashLife**. Hashlife consists of a memoized algorithm (an algorithm that stores results of expensive function calls and returns the cached result when the same inputs occur again) designed by Bill Gosper in the early 1980s.

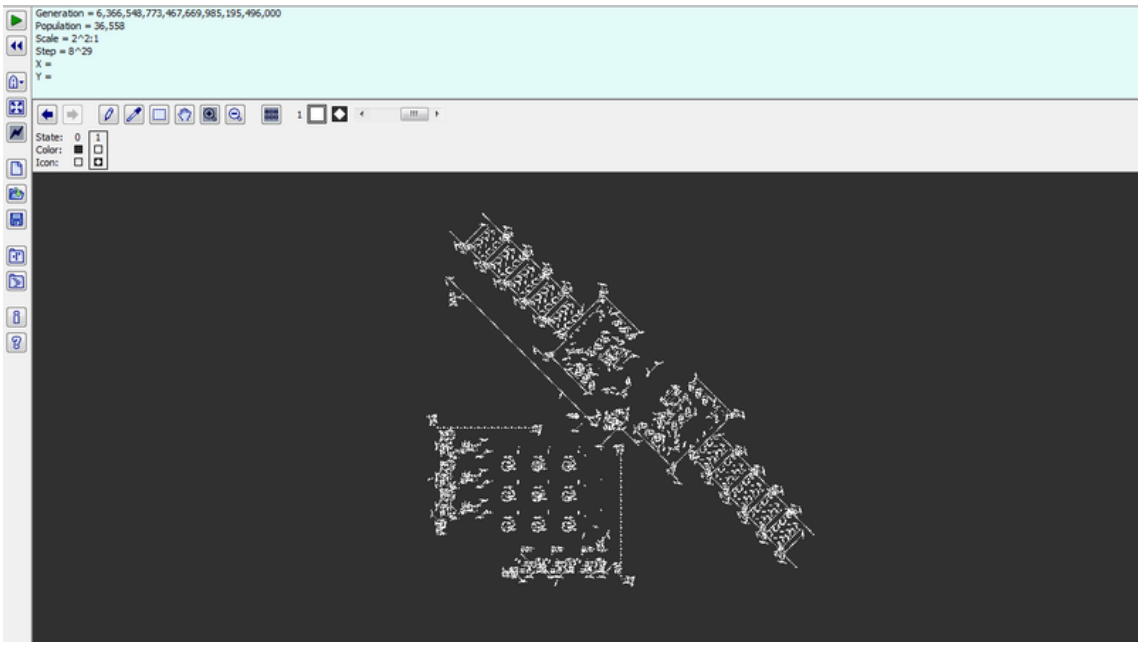

Figure 21: 6 octillionth generation

It capitalizes on the redundancies embedded in Game of Life rules. After a given time, some configurations will be repeated. This is when memoization plays a huge role. The underlying mechanism is that quadtrees are used to represent the field. Then, the algorithm uses hash tables to store the nodes of the quadtree.

Hashlife consumes far more memory resources than my implementation, so it would not be a sensible choice to run in my computer.

Now moving on to my own code, a possible serial optimization would be dividing the matrix into different blocks in order to improve data locality by improving access in cache lines. I did not try this optimization because it is already done in the parallelized version and it would interfere with thread creation, but the experiment could be carried out.

In terms of parallelization, another possible implementation would be, instead of saving all the neighbours inside the structure, another possibility would be to calculate its neighbours in each iteration by using Stencil code. These are a class of iterative kernels that update the elements in an array according to a selected pattern (or stencil).

Last but not least, other possible improvement would be changing the data structure repre-

senting the whole world and instead of using a two dimensional array, I could use a vector of coordinate pairs to represent dead or alive cells. In order to calculate the number of alive neighbours hash tables will become necessary, which may slow down the process. Nevertheless, this approach can allow the pattern to move about the world unhindered.

As a conclusion, I may say that cellular automata problems are very rich in nature and can lead to a wide variety of solutions. The limitation is the creativity of the programmer and its computing power.

13 Bibliography

References

[1] Game of Life
http://beltoforion.de/article.php?a=game_of_life

[2] Game of Life
http://www.physics.drexel.edu/ valliere/PHYS405/Game_of_Life/index.html

[3] SIMD
https://en.wikipedia.org/wiki/SIMD

[4] SSE4
http://softpixel.com/ cwright/programming/simd/sse3.php

[5] Game of Life
https://www8.cs.umu.se/kurser/5DV050/VT12/gol.pdf

[6] Celullar Automata
https://plato.stanford.edu/entries/cellular-automata/

[7] OpenMP
https://computing.llnl.gov/tutorials/openMP/

[8] Pthreads
https://en.wikipedia.org/wiki/POSIX_Threads

14 Appendix

14.1 find_neighbours

```
void obtain_neighbours(cell** cells_grid , int N , int M , cell* b ){
  //obtain neighbours for [0][0]
  cells_grid[0][0].neighbours = (cell**)malloc(8*sizeof(cell*));
  cells_grid[N-1][0].neighbours = (cell**)malloc(8*sizeof(cell*));
  cells_grid[0][M-1].neighbours = (cell**)malloc(8*sizeof(cell*));
  cells_grid[N-1][M-1].neighbours = (cell**)malloc(8*sizeof(cell*));

  cells_grid[0][0].neighbours[0] = b;
  cells_grid[0][0].neighbours[1] = b;
  cells_grid[0][0].neighbours[2] = b;
  cells_grid[0][0].neighbours[3] = b;
  cells_grid[0][0].neighbours[4] = &cells_grid[0][1];
  cells_grid[0][0].neighbours[5] = b;
  cells_grid[0][0].neighbours[6] = &cells_grid[1][0];
  cells_grid[0][0].neighbours[7] = &cells_grid[1][1];


  cells_grid[N-1][0].neighbours[0] = b;
  cells_grid[N-1][0].neighbours[1] = &cells_grid[N-2][0];
  cells_grid[N-1][0].neighbours[2] = &cells_grid[N-2][1];
  cells_grid[N-1][0].neighbours[3] = b;
  cells_grid[N-1][0].neighbours[4] = &cells_grid[N-1][1];
  cells_grid[N-1][0].neighbours[5] = b;
  cells_grid[N-1][0].neighbours[6] = b;
  cells_grid[N-1][0].neighbours[7] = b;


  cells_grid[0][M-1].neighbours[0] = b;
  cells_grid[0][M-1].neighbours[1] = b;
  cells_grid[0][M-1].neighbours[2] = b;
  cells_grid[0][M-1].neighbours[3] = &cells_grid[0][M-2];
  cells_grid[0][M-1].neighbours[4] = b;
  cells_grid[0][M-1].neighbours[5] = &cells_grid[1][M-2];
  cells_grid[0][M-1].neighbours[6] = &cells_grid[1][M-1];
  cells_grid[0][M-1].neighbours[7] = b;


  cells_grid[N-1][M-1].neighbours[0] = &cells_grid[N-2][M-2];
  cells_grid[N-1][M-1].neighbours[1] = &cells_grid[N-2][M-1];
  cells_grid[N-1][M-1].neighbours[2] = b;
  cells_grid[N-1][M-1].neighbours[3] = &cells_grid[N-1][M-2];
  cells_grid[N-1][M-1].neighbours[4] = b;
  cells_grid[N-1][M-1].neighbours[5] = b;
  cells_grid[N-1][M-1].neighbours[6] = b;
  cells_grid[N-1][M-1].neighbours[7] = b;
  for (int i = 1 ; i< (M-1) ; i++){
    cells_grid[0][i].neighbours = (cell**)malloc(8*sizeof(cell*));
    cells_grid[N-1][i].neighbours = (cell**)malloc(8*sizeof(cell*));

    cells_grid[0][i].neighbours[0] = b;
    cells_grid[0][i].neighbours[1] = b;
    cells_grid[0][i].neighbours[2] = b;
```

```
53      cells_grid[0][i].neighbours[3] = &cells_grid[0][i-1];
54      cells_grid[0][i].neighbours[4] = &cells_grid[0][i+1];
55      cells_grid[0][i].neighbours[5] = &cells_grid[1][i-1];
56      cells_grid[0][i].neighbours[6] = &cells_grid[1][i];
57      cells_grid[0][i].neighbours[7] = &cells_grid[1][i+1];
58
59
60      cells_grid[N-1][i].neighbours[0] = &cells_grid[N-2][i-1];
61      cells_grid[N-1][i].neighbours[1] = &cells_grid[N-2][i];
62      cells_grid[N-1][i].neighbours[2] = &cells_grid[N-2][i+1];
63      cells_grid[N-1][i].neighbours[3] = &cells_grid[N-1][i-1];
64      cells_grid[N-1][i].neighbours[4] = &cells_grid[N-1][i+1];
65      cells_grid[N-1][i].neighbours[5] = b;
66      cells_grid[N-1][i].neighbours[6] = b;
67      cells_grid[N-1][i].neighbours[7] = b;
68    }
69    for (int i = 1 ; i < (N-1) ; i++){
70      cells_grid[i][0].neighbours = (cell**)malloc(8*sizeof(cell*));
71      cells_grid[i][M-1].neighbours = (cell**)malloc(8*sizeof(cell*));
72
73
74      cells_grid[i][0].neighbours[0] = b;
75      cells_grid[i][0].neighbours[1] = &cells_grid[i-1][0];
76      cells_grid[i][0].neighbours[2] = &cells_grid[i-1][1];
77      cells_grid[i][0].neighbours[3] = b;
78      cells_grid[i][0].neighbours[4] = &cells_grid[i][1];
79      cells_grid[i][0].neighbours[5] = b;
80      cells_grid[i][0].neighbours[6] = &cells_grid[i+1][0];
81      cells_grid[i][0].neighbours[7] = &cells_grid[i+1][1];
82
83      cells_grid[i][M-1].neighbours[0] = &cells_grid[i-1][M-2];
84      cells_grid[i][M-1].neighbours[1] = &cells_grid[i-1][M-1];
85      cells_grid[i][M-1].neighbours[2] = b;
86      cells_grid[i][M-1].neighbours[3] = &cells_grid[i][M-2];
87      cells_grid[i][M-1].neighbours[4] = b;
88      cells_grid[i][M-1].neighbours[5] = &cells_grid[i+1][M-2];
89      cells_grid[i][M-1].neighbours[6] = &cells_grid[i+1][M-1];
90      cells_grid[i][M-1].neighbours[7] = b;
91
92
93
94      for (int j = 1 ; j < (M-1) ; j++){
95        cells_grid[i][j].neighbours = (cell**)malloc(8*sizeof(cell*));
96
97        cells_grid[i][j].neighbours[0] = &cells_grid[i-1][j-1];
98        cells_grid[i][j].neighbours[1] = &cells_grid[i-1][j];
99        cells_grid[i][j].neighbours[2] = &cells_grid[i-1][j+1];
100       cells_grid[i][j].neighbours[3] = &cells_grid[i][j-1];
101       cells_grid[i][j].neighbours[4] = &cells_grid[i][j+1];
102       cells_grid[i][j].neighbours[5] = &cells_grid[i+1][j-1];
103       cells_grid[i][j].neighbours[6] = &cells_grid[i+1][j];
104       cells_grid[i][j].neighbours[7] = &cells_grid[i+1][j+1];
105
106     }
107   }
```

```
108 }
```

Listing 32: "Find Neighbours"