PROJECT REPORT

FOUNDATIONS OF HIGH-PERFORMANCE COMPUTING 2022-2023

Guido Cera

Table of Contents

0. Introduction

The project consists of a scaling study on a personal and highly parallel implementation of the so-called Game of Life.

0.1 Game

Game of Life is played on a grid, where each cell can be either dead or alive. Starting with a grid in a state - with some cells alive and some dead - the grid "evolves", meaning each cell keeps or changes its state

depending on the state of the neighbouring cells. The specific implementation for the evolution rules used in this project is presented in the assignment description (github.com) and varies slightly from the classical rules of Conway's Game of Life. The grid is virtually infinite, the outer cells of the grid use the cells at the opposite site as neighbours; imagine that the grid multiplicate and each copy sits adjacent to the original in the same orientation. In a computer the cells have to be updated one by one and this offers multiple possibilities in how the cells are navigated and when they are updated; the two methods I implemented are called **ordered evolution** and **static evolution**. The former is perhaps the simplest both conceptually and in code, it is a loop over all cells in order and each is updated right away, before passing on to the next cell. The latter method holds the update of the cells after all cells have been checked and their next state have been decided. While in the ordered case the neighbours of a cell will be half in the old state and half in the new one, for the static evolution all neighbours will be in the same state at the time of determining the new state of a cell.

Interesting fact, the classic implementation of Conway's Game of Life is Turin complete. (Wikipedia)

0.2 Orfeo

This project is being developed to run specifically on the high-performance computer Orfeo located in Area Science Park, Trieste (areasciencepark.it). This is important for the parallelization of the code, which has to be built using the resources available on Orfeo (ofeo-doc.areasciencepark.it). Thus the code will contain a hybridization of the libraries MPI (open-mpi.org) - for multiprocessing - and OpenMP (openmp.org) - for multithreading.

0.3 Report structure

The general structure of the sections in this report adheres faithfully to the one suggested. The subsections in Methodology [1.] and Implementation [2.] often mimic each other. The different subsections with equal name cover the same topic but from a methodology standpoint if they are in section [1.] and from an code implementation point of view if you found them in section [2.]. To get a complete presentation on a topic check both sections.

1. Methodology

From an abstract point of view the core of the project - the Game of Life - is pretty straightforward since it comprises of a matrix, some nested for loops and few simple checks to update the cells. I approached the project implementing this core structure first and then delving in the parallelization; and even if I have experience in multi-processing and multi-threading it felt like going from swimming in a pool to being thrown in the ocean during a storm. The most interesting choice of algorithm I encountered at this stage was approaching the static evolution [1.1].
Now, I do not want to give the impression that no optimization can be done, on the contrary, there are a great number of articles and book's chapters that analyse the problem and make use of clever details of computer architecture or theory on sparce matrices and data structures in general to shrink dramatically the number of operations needed to play the game. Here are some links to provide an overview of the optimization landscape for Game of Life.

- (stackoverflow.com) Post with some good answers and links to useful resources;
- (docplayer.net) Makes use of theory on **sparce matrices**;
- (jaregoy.com) Michael Abrash's Graphics Programming Black Book, chapter 17 spends 33 pages to optimize Game of Life;

- (Wikipedia) **Hashlife** is an algorithm that makes use of hash tables and reaches impressive speeds, with the drawback of occupying large amounts of memory
- (dotat.at) An implementation of Game of Life which uses **bit-wise operations** and **lookup tables**.

I want to stress the many possibilities of optimization since the philosophy of coding for a high-performance computer is all about that, performance, but one of the focuses of this course is parallelization, in particular through MPI and OpenMP, and the objective of this project is to show the scalability of multithreading and multiprocessing. In light of all this I decided to focus on implementing a good parallelization and I left a simple representation of the game underneath; which might actually help showing the power of multiprocessing when scaling.

The following subsections explore the choices made in specific situations or show the methodological approach to specific subproblems.

1.1 Static Evolution

For this approach to evolution, I need first to check all the cells and determine their next state and then, as a separate step, I can update the grid. This way the current state of the table is the only one used to update it. The problem resides in the computed next state, where do I put it while I check the following cells if I cannot use the grid itself? I need a way to store the next state until all cells have been checked and their next state computed. The way I store the next state needs also to be easily accessible and fast to read because as soon as I am done writing all next states there, I will read them and transfer them to the actual grid.
I saw two possible solutions to this problem, each with their own positives and negatives; one being to use middle-states to mark cells to modify and the other being to store two separate lists with the coordinates of cells to kill and to revive. Follows an explanation and then an analysis of the two approaches in terms of memory needed, computational load and ease of use.

- **Middle-states.** With middle-states I mean adding states that a cell can be in other than "alive" and "dead". In particular I add the states "will die" to mark a cell that is alive but will have to die at the next evolution step, and "will live" to mark a cell that is dead but will live at the next evolution step. This approach consists of marking the cells directly on the grid, without any additional data structure, but in a way that still leaves visible the current state of the grid so that checking the following cells will not be influenced by the mark. The drawback of this approach presents itself at the moment of updating the grid for the next state, here I have to change all cells in a middle-state to the state they are meant to have. There are no shortcuts to accomplish this, an additional full scan of the grid is necessary to find the middle-states and write the corresponding final state.
- **Separate Lists.** Two separate dynamic lists store coordinates, one of the cells to revive and the other of the cells to kill. This approach uses additional memory but minimizes the effort to update the grid afterwards, since I only have to loop through the lists and access only the cells that need modifying and no other.

From initial experiments I carried out I formed some expectations on the behaviour of this game.
Unfortunately I did not save the precise numbers but my takeaway is that a significant number of cells change state each evolution step if we start from a random initial state [2.4]. Translated to the two solutions above, this means that the lists will occupy a significant amount of memory each step, and if I update the memory allocated for each list doubling it when it is full, I could have to allocate up to four times the size of the grid at each evolution step. Moreover middle-states are much easier to handle than dynamic memory. All this is why I

opted for the middle-states solution instead of using separate lists. Further implementation details can be found in the next section [2.3].

1.2 Ordered Evolution

The ordered evolution is simple in its implementation and does everything in a couple of nested loops; but this same simplicity is also reason of impairment for parallelization, and consequently scalability performance. Not all parallelization is impossible, multiprocessing works and is as effective as for the static evolution, only multithreading cannot bring any benefit and the reason is in the ordered nature of the approach and the way OpenMP parallelizes the work; let's analyse both in order. The ordered evolution is intrinsically serial because when one cell gets updated its state will affect the other cells coming after that are around it. OpenMP handles multithreading of for loops assigning iterations to the threads, the assignment can be static or dynamic but in both cases a thread with its assigned iterations has to wait for all previous iterations to be done before he can execute its own. In the end, the whole computation is serial even if it is carried on multiple threads. For this reason, I did not implement multithreading for this evolution. MPI implementation is the same for both evolutions and it is discussed in section [1.4].

1.3 Multithreading

I choose to focus my multithreading effort only on the heaviest part of the code, which is the evolution of the board itself. All other components that are necessary but only function as a frame to the main part, even if they could exploit multithreading, I choose to leave single threaded. The reason for my choice was code clarity but most importantly it should not affect overall performance because all initializations and the set up for multiprocessing [1.4] is orders of magnitude faster than the evolution of the board. To cite one of the sources I linked above, Michael Abrash in his book Graphics Programming Black Book says this.

> The first rule of optimization is: Only optimize where it matters.

Where I did apply multithreading, I tried to be strict with the data handling, all outside variables used inside the openMP area are explicitly marked and even the division policy I fixed the one which makes the most sense.

1.4 Multiprocessing

As I anticipated above there is a little set up to do for MPI to work properly and efficiently. Main reason being the need for explicit communication between the processes which for Game of Life, at least as I implemented it, is very much essential; but let's start from the beginning and explain why there is this need. My reasoning is pretty simple, I have a grid that I need to split up in blocks to hand off one per process. So, this I did, the grid gets divided in rectangles and each gets sent to a process. The rectangles are all of the same size, so that each process has the same workload. Here comes the communication problem, the outer cells in a block have some of the neighbours in other blocks and thus in other processes, where they can no longer be accessed. Now I need to carefully propagate all outer cells of all blocks to the right corresponding blocks; top row, right column, bottom row and left column go respectively to top block, right block, bottom block and left block, but the four cells in the corners also need to be sent to their respective block in diagonal of the origin block. All this propagation has to be done at each evolution step before updating the grid and before checking the neighbours.
Last thing to consider for inter-process communication is the need to save a snapshot of the grid during the evolution. When this necessity arises, all blocks are gathered in the root process, the snap is saved and then the blocks are sent back each to the same process it came from, before finally proceeding with the evolution.

There are many operations that need to be computed by a single process, I picked the root process to execute them. Picking the root process specifically ensures that the program will run even if launched with only one process.

I forgot to mention that this way to split the work between processes is called **domain decomposition**, which means that different processes do the same operations on different data.

1.5 Board generation

The assignment requires a mode where a grid gets generated given its dimensions but there is no indication as what the state of this grid should be, so here is what I did. The simplest solution would be to return an image of the requested size with always the same initial state. This would be possible since the minimum size of the grid is 100x100, the requirement would be for the state to be contained in this limit, and all the extra space could be all dead cells. What I did instead is generating a random state each time. Generated the image I iterate through it and with a probability of 0.2 a cell gets marked as alive.

Generating a new grid is one of the tasks my program executes with only one process, but I implemented multithreading since in initialization mode creating a new grid is all the program does. Multithreading brings a problem with the random number generation that I discuss in section [2.4].

To create a different seed for each grid instead of going all fancy with the `time.h` library I implemented a little seed generator using the string containing the name to give the final file. This works on the assumptions that the name of the image would contain some information about the size of the grid and there is no need to generate different grids of the same size. Since I am the only user of this code, I can assure that these assumptions are met.

1.6 Read and write image

I will not comment on any function in the file `read_write_pgm_image.c` since I did not personally write it. I just used the functions in there, hopefully the way they were intended to.

1.7 Time tracking

I used three different timers to gather data on the execution. The first, most general, timer I call "whole evolution timer" because it starts after the initial set up of reading the external parameters, reading the image and initializing MPI and just before launching the evolution. This timer ends just after the evolution. A second timer is all inside the evolution function but still times the whole evolution and is outside all evolution loops. Compared to the first timer it excludes the division of the grid into blocks [1.4]. The third and final timer is actually a measure of average time; the average time to propagate the outer cells of the blocks between processes. I decided to implement the last one because I was worried that it would take a significant portion of the execution; turns out I was wrong as you can see in the results section [3].

1.8 Experimentation Environment

At the time I executed my program and got the tests results Orfeo was pretty busy all the time with, I guess, other students doing their own projects. I had a little hope I could run more tests than the strictly necessary but, in the end, I did not manage to. I ran all tests on EPYC nodes, I only touched THIN nodes for the second exercise. Moreover, I only used OpenMPI and never IntelMPI. I do not have much to say about this because these were not choices, I made but rather constraints I found myself in.

1.9 Data collection

I automated the data gathering process to make it easier to generate plots and such for this report and also to check as soon as a test finishes if the data it created makes sense in relation to the previous data.

2. Implementation

2.1 Repository structure

A list of all relevant files and folders in the repository exercise_1 include gol.h obj snapshots src evolution_parallel.c evolution.c read_write_pgm_image.c utilities.c starts data.csv main.c Makefile report.pdf where,

- `exercise_1` contains all files relevant to the Game of Life assignment
  - `include` contains the personal library
    - `gol.h` is the library to link the functions in the `.c` files
  - `obj` is a folder that stores the object file when the program gets compiled
  - `snapshots` is the folder where the snapshots of the grid get saved during execution
  - `src` is the folder with the program files
    - `evolution_parallel.c` final evolution file with evolution functions and check neighbours function, all parallelized.
    - `evolution.c` first iteration of evolution, no parallelization
    - `read_write_pgm_image.c` contains functions to read and write pgm images
    - `utilities.c` here you can find all the auxiliary functions used by the core of the program
  - `starts` is the folder that contains starting images
  - `data.csv` is the file where I stored the results of the tests
  - `main.c` is the main of the program
  - `Makefile` is the Makefile for the program
  - `report.pdf` this file you are reading!

2.2 Data types

The matrix that stores the grid, and also all auxiliary variables and arrays that store cells, are of type `unsigned char`. Firstable `char` is the type that occupies the least memory in C at 1 Byte, which is plenty to store the numbers from 0 to 255 that the grid uses. Second able since, as just noted, the numbers to be stored are all positive using `usigned` provides a hardcoded guard against negative values; this way the code better resembles the logic behind it.

```
212     unsigned char block[BLOCKROWS*BLOCKCOLS];
```

```
254     unsigned char btm_row[BLOCKCOLS];
255     unsigned char top_row[BLOCKCOLS];
256     unsigned char left_clmn[BLOCKROWS];
257     unsigned char right_clmn[BLOCKROWS];
258     unsigned char temp[BLOCKROWS];
259     unsigned char top_left, top_right, btm_left, btm_right;
```

From `evolution_parallel.c`, use of `unsigned char`

2.3 Static Evolution

The implementation of static evolution is quite long, mostly because of MPI parallelization.

```
323         if(block[i*BLOCKCOLS + j] <= 127) block[i*BLOCKCOLS + j] = 127;
325         if(block[i*BLOCKCOLS + j] >= 128) block[i*BLOCKCOLS + j] = 128;
```

From `evolution_parallel.c`, middle-states and checking the current state

In section [1.1] I explained the theory behind the middle-states, it is here the place to see them in action. The states are `0` for "dead" and `255` for "alive", the middle-states are `127` for "will live" and `128` for "will die". Realistically any number between one and 127 would have sufficed for "will live" state and analogously any number between 128 and 254 would have worked. The important fact about the middle-states is that they still enable checking for the current state of the cell with one single operation, as shown in the image above. If a cell contains a number greater or equal than 128 then it is currently alive. If a cell contains a number lower or equal than 127 then it is currently dead. Important to note is that these middle-states never leave the evolution loops, at the end of the second loop, the one to update the grid to the new state, all cells will have either `0` or `255` as values. When it is time to save a snapshot no middle-states will be present in the grid.

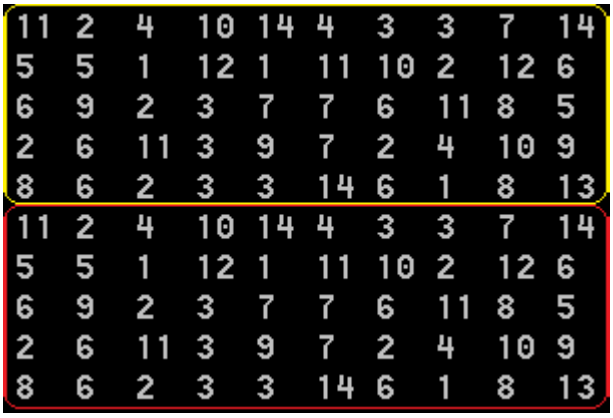
2.4 Board generation

In generating the board I explained in section [1.5] the random approach I took, now I will present an interesting problem I encountered with `srand` and OpenMP.

```
78      #pragma omp parallel shared(board, size)
79      {
80          srand(Seed: easy_seed(string)* omp_get_thread_num());
81          #pragma for schedule(static) collapse(2)
```

From `utilities.c`, `srand` and OpenMP conflict

The problem is not apparent in the code but it will appear looking at the images generated, what happens is that each thread has the same seed. The image shows a pattern that repeats itself for the number of threads used. The image below is taken from a blog post that helped me greatly in solving the problem, it lays out the problem in a very clear way and offers a clever solution. Already present in the code above the solution is to use the thread number which is unique for each thread, to modify the seed, now all seeds will be different and no patterns will emerge.



From a blog post (pvs-studio.com), the pattern in the matrix

This problem is not really critical since the images would have worked even with the repetitions and I could have simply removed the multithreading to solve the problem, but I think it is worth mentioning.

2.5 Check neighbours

This function checks the neighbours of a cell and returns `1` if the cell has to live at the next evolution step, `0` otherwise. The way the propagated cells are handled greatly influences how simple this function is to implement. I thought of two possible ways to manage the propagated cells, follows a presentation of both with an explanation of what would change for `check_neighbours`.

- **New block.** One option is to allocate a new, larger, block to accommodate for the two additional rows and columns that were sent from other processes. Then passed this array to `check_neighbours` it is only a matter of looping through a hard-coded array of off-sets (see figure below) to visit all neighbours because they are already all around the cell.
- **Disjointed arrays.** The opposite approach is to leave all arrays as they are and pass them singularly to `check_neighbours` which will have to handle them with ad-hoc code. Each outer row and outer column need its own lines of code, but the corners need special treatment as well, in total there are eight different possibilities; nine if we consider that a cell might just be not on the edge of the block, then the previous method with the hard-coded array of off-sets can still apply. This last case is actually the most frequent for blocks of a certain size, and it gets more frequent as size increases. Nevertheless, the edge cases (pun intended) have to be handled and to do so a lot of boring, error-prone and difficult-to-maintain lines of code are necessary.

All things considered I picked the second option. The problem with the new block is the new block itself, which uses memory. The amount of memory space used by the program would basically double because a whole new grid (more actually because the new block is slightly bigger) would have to be allocated, and memory allocation also takes time. I repeated already more than once that propagation is not relevant for performance but when it comes to inter-nodes process communication the orders of magnitude of difference between strictly evolution time and propagation time are not that many, so disregarding efficiency altogether is not advisable.

```
29    const int off_sets[NUM_NEIGHBOURS][2] = {[0]={[0]=-1, [1]=-1}, [1]={[0]=-1, [1]=0}, [2]={[0]=-1, [1]=1},
30                                             [3]={[0]=0, [1]=-1},                       [4]={[0]=0, [1]=1},
31                                             [5]={[0]=1, [1]=-1},  [6]={[0]=1, [1]=0},  [7]={[0]=1, [1]=1}};
```

From `evolution_parallel.c`, hard-coded array of off-sets

```
34    /* inner cell */
35    if(i != 0 && j != 0 && i != BLROWS-1 && j != BLCOLS-1){
36        for(int z=0; z<NUM_NEIGHBOURS; z++){
37            if( board[(i + off_sets[z][0])*BLCOLS + (j + off_sets[z][1])] >= 128 ) count++;
38        }
39    }else if(i == 0){     /* outer cell */
```

From `evolution_parallel.c`, code to check neighbours for an inner cell

```
39      }else if(i == 0){     /* outer cell */
40          if(j == 0){
41              if(t_l >= 128) count++;
42              if(top[0] >= 128) count++;
43              if(top[1] >= 128) count++;
44              if(left[0] >= 128) count++;
45              if(board[(i + off_sets[4][0])*BLCOLS + (j + off_sets[4][1])] >= 128) count++;
46              if(left[1] >= 128) count++;
47              for(int z=6; z<NUM_NEIGHBOURS; z++)
48                  if( board[(i + off_sets[z][0])*BLCOLS + (j + off_sets[z][1])] >= 128 ) count++;
```

From `evolution_parallel.c`, code to check neighbours for the upper left cell in the block

2.6 Multiprocessing

I took inspiration for implementing the splitting of the grid from a stackoverflow post about this very topic (stackoverflow.com). I used the answer in the post to check that I used correctly `MPI_Scatterv` to send the blocks to their respective processes.

```
213     MPI_Datatype blocktype;
214     MPI_Datatype blocktype2;
215
216     MPI_Type_vector(BLOCKROWS, BLOCKCOLS, DIM, MPI_UNSIGNED_CHAR, &blocktype2);
217     MPI_Type_create_resized( blocktype2, 0, sizeof(char), &blocktype);
218     MPI_Type_commit(&blocktype);
219
220     int disps[NPROWS*NPCOLS];
221     int counts[NPROWS*NPCOLS];
222     for (int ii=0; ii<NPROWS; ii++) {
223         for (int jj=0; jj<NPCOLS; jj++) {
224             disps[ii*NPCOLS+jj] = ii*DIM*BLOCKROWS+jj*BLOCKCOLS;
225             counts [ii*NPCOLS+jj] = 1;
226         }
227     }
228
229     MPI_Scatterv(board, counts, disps, blocktype, block, BLOCKROWS*BLOCKCOLS,
230                  MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);
231
```

From `evolution_parallel.c`, splitting of the grid

In the code above there is the code pertaining to `MPI_Scatterv`. The use of `MPI_Scatterv` instead of `MPI_Scatter` is really essential even if it is not apparent from the code. The difference between the two functions is that my variable `counts` which specifies the number of elements to send to each processor, is an array in `MPI_Scatterv` while it is only a single variable in the base version. As you can see my `counts` has all ones when passed to the function so one might think that the base version would have sufficed. I though the same but upon making the change I discovered that now the blocks were not sent in order to the processes anymore. Before, with `MPI_Scatterv`, if you numbered the block from zero and proceeded by row each block

would have exactly the rank of the process it would end up in. This ordering of the blocks is of vital importance because it later allows the use of `MPI_Gather` to reconstitute the grid on one process and save a snapshot. So `MPI_Scatterv` is very important for this implementation.
Another piece of code to mention from the image above are the very first rows shown, from `213` to `218`. The variable `blocktype` is a kind of wrapper and represents the type of a block in the grid. This is also taken from the stackoverflow post.

Note that all considerations for splitting the grid are valid both for the ordered evolution and the static.

```
253         /* propagate rows */
254         if((RANK/NPCOLS)%2 == 0){
255             /* send top row */
256             MPI_Send(block, BLOCKCOLS, MPI_UNSIGNED_CHAR,
257                      top_block(RANK), 0, MPI_COMM_WORLD);
258             MPI_Recv(btm_row, BLOCKCOLS, MPI_UNSIGNED_CHAR,
259                      bottom_block(RANK), 0, MPI_COMM_WORLD, &status);
260             /* send bottom row */
261             MPI_Send(block + (BLOCKCOLS*(BLOCKROWS-1)), BLOCKCOLS, MPI_UNSIGNED_CHAR,
262                      bottom_block(RANK), 0, MPI_COMM_WORLD);
263             MPI_Recv(top_row, BLOCKCOLS, MPI_UNSIGNED_CHAR,
264                      top_block(RANK), 0, MPI_COMM_WORLD, &status);
265         }else{
266             MPI_Recv(btm_row, BLOCKCOLS, MPI_UNSIGNED_CHAR,
267                      bottom_block(RANK), 0, MPI_COMM_WORLD, &status);
268             MPI_Send(block, BLOCKCOLS, MPI_UNSIGNED_CHAR,
269                      top_block(RANK), 0, MPI_COMM_WORLD);
270
271             MPI_Recv(top_row, BLOCKCOLS, MPI_UNSIGNED_CHAR,
272                      top_block(RANK), 0, MPI_COMM_WORLD, &status);
273             MPI_Send(block + (BLOCKCOLS*(BLOCKROWS-1)), BLOCKCOLS, MPI_UNSIGNED_CHAR,
274                      bottom_block(RANK), 0, MPI_COMM_WORLD);
275         }
```

From `evolution_parallel.c`, propagation of outer cells

This is only a portion of the whole propagation code and it specifically refers to the propagation of the top and bottom rows of all blocks. It is sufficient to demonstrate the logic of `MPI_Send` and `MPI_Recv`. Both functions require the ranks of the sender process and the receiving process, for this I wrote `top_block` and `bottom_block` which, given the rank of the current process, return the rank of the process managing respectively the upper and lower block. Below you can see the implementation of `top_block` as an example, it does an efficient check of the position of the block and subsequently executes the right formula to get the desired block.
The outer `if` condition ensures that a deadlock is never encountered; it divides the processes, and consequently the blocks, by row. All blocks in an even row will enter the `if` block and the others will execute the `else` block. The difference between the two blocks is only in the order of execution of sends and receives, even rows will execute the send first and odd rows will do the opposite. This prevents deadlocks because a block in an even row will send to blocks in an odd row and they on the other hand, will be ready to receive. Then the roles reverse and, in the end, both blocks will have the row they need. The same concept applies to the outer columns and the four cells at the corners.

```
159   int top_block(const int RANK){
160       return RANK < NPCOLS ? (NPROWS-1)*NPCOLS + RANK : RANK - NPCOLS;
161   }
```

From `evolution_parallel.c`, top_block function, `NPCOLS` and `NPROWS` variables are global and represent the number of blocks per column and per row respectively

---

One last thing to mention about the propagation is that I could have saved space and made the code slimmer by using a matrix to store all the data from other blocks and grouping up the sends and receives into one big send and one big receive. Additionally, I would have probably also gained a little performance by doing this. The reason I did not follow this route is that the logic of sends and receives is not immediate to me and laying out every single operation has been a good exercise in parallelization mentality.

2.7 Multithreading

Since I choose a simple approach to the Game of Life all the heavy computation is inside for loops, nested for loops to be precise. Since each iteration carries out almost the same amount of computation - check neighbours, if the state has to be changed, change it - there is no need to dynamically assign the iterations to the threads, I can save on the overhead and assign statically. The computational differences between iterations depend on the state of the neighbours which in turn, determine if the state of the cell has to be changed or not. The checks for the neighbours are equal for all iterations, what can change is how many times the counter of alive neighbours gets increased. But all things considered the possible difference in load is at most of nine write operations (eight neighbours plus the cell itself), which is really not a big number and it justifies the static assignment.

```
317   #pragma omp parallel shared(BLOCKROWS, BLOCKCOLS, block, top_left, top_row, top_right,
318                               left_clmn, right_clmn, btm_left, btm_row, btm_right)
319   {
320       #pragma omp for schedule(static) collapse(2)
321       for(int i=0; i<BLOCKROWS; i++){
322           for(int j=0; j<BLOCKCOLS; j++){
323               if(check_neighbours(block, BLOCKROWS, BLOCKCOLS, i, j, top_left, top_row, top_right,
324                                   left_clmn, right_clmn, btm_left, btm_row, btm_right, RANK) == 1){
325                   if(block[i*BLOCKCOLS + j] <= 127) block[i*BLOCKCOLS + j] = 127;
326               }else{
327                   if(block[i*BLOCKCOLS + j] >= 128) block[i*BLOCKCOLS + j] = 128;
328               }
329
330           }
331       }
```

From `evolution_parallel.c`, an example of openMP implementation

In the code above there is the first part of static evolution to show how I implemented openMP. There is another nested for loop below in the actual code that is not seen here, that is the reason why I open the parallel region first and then I call the handling of the loops. The variables are shared in the whole parallel region. There are two perfectly nested loops so I can use the `collapse(2)` flag to affect both with one call.

3. Results & Discussion

Three tests were carried out on this code, a test on openMP scalability and two on MPI scalability. Each test has its own subsection were you can find a list of command line instructions to set up and run the test on Orfeo's terminal and the data produced by the test; both with hopefully a satisfactory explanation.

The raw data presented will have the following form.

size	steps	evolution	processes	threads	tot time	evol time	avg propagation time
1000	1000	1	1	1	72635616	72632154	7

- **size** is the dimension of the square grid;
- **steps** indicates the number of evolution steps done by each run;
- **evolution** is the flag passed to the executable `gol.x` to indicate which kind of evolution to use. `1` stands for static evolution;
- **processes** is the number of MPI processes used in each run;
- **threads** is the number of openMP threads used by a run;
- **tot time**, **evolution time** and **avg propagation time** are the three timers, a throughout explanation for them can be found in section [1.7]. All times are in microseconds.

3.1 OpenMP scalability

Test description:

> fix the number of MPI tasks to 1 per socket, and report the behaviour of the code when you increase the number of threads per task from 1 up to the number of cores present on the socket

All relevant command line instructions for this test.

```
    salloc -N1 -p EPYC -n128 --time=1:0:0
    export OMP_PLACES=cores
    srun -n1 --cpus-per-task=64 mpirun --map-by socket gol.x -r -f
start_1000px.pgm -n 1000 -e 1 -s 1000
```

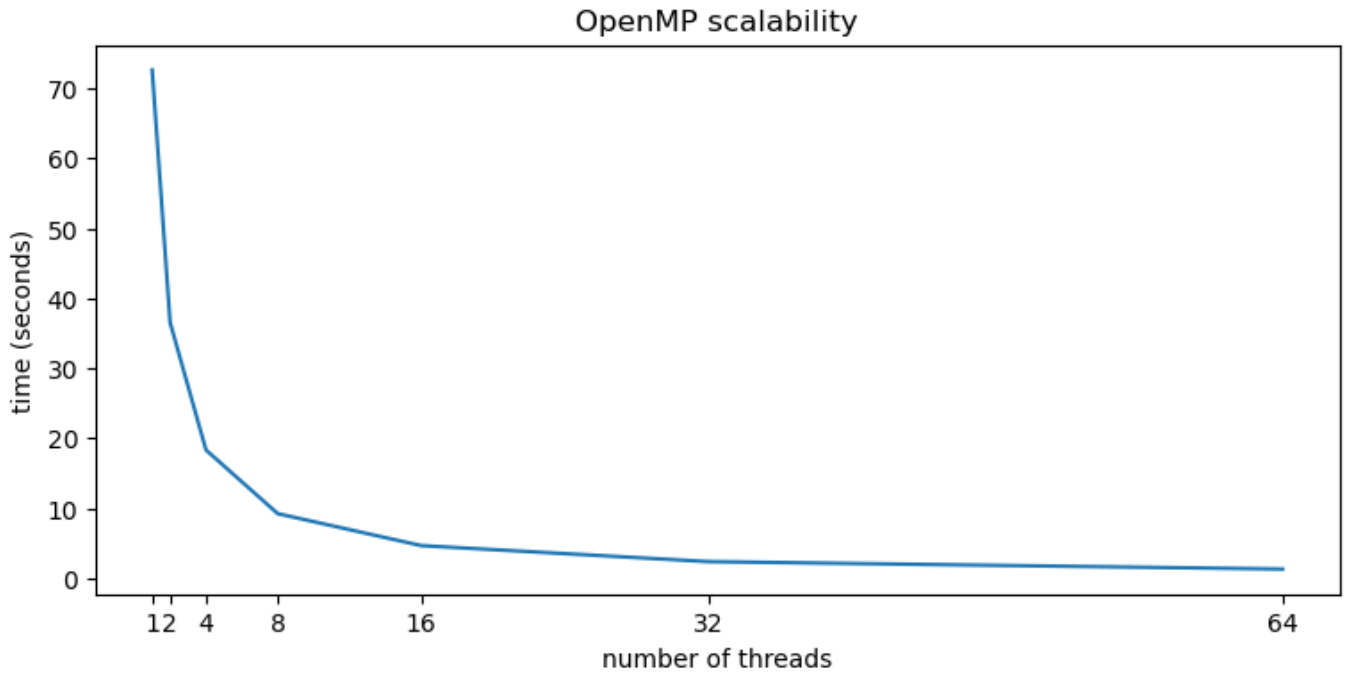
The number of threads is changed inside `gol.x`. As per instructions, the threads populate different cores (`OMP_PLACES=cores`) and fill up the socket where the process runs (`--map-by socket`).

The raw data collected in this test.

size	steps	evolution	processes	threads	tot time	evol time	avg propagation time
1000	1000	1	1	**1**	72635616	**72632154**	7
1000	1000	1	1	**2**	36536630	**36534854**	7
1000	1000	1	1	**4**	18318647	**18317337**	7
1000	1000	1	1	**8**	9218737	**9217823**	15
1000	1000	1	1	**16**	4652385	**4651514**	19

size	steps	evolution	processes	threads	tot time	evol time	avg propagation time
1000	1000	1	1	**32**	2367646	**2366288**	21
1000	1000	1	1	**64**	1302963	**1300866**	24

As the number of threads doubles the evolution time halves, this is even more evident looking at the plot below. This behaviour can be explained by the nature of the evolution. As discussed in section [2.7] each iteration has a very similar computational load to all other iterations, so it is expected that each thread gets an even portion of the overall computational load of the program. Another fundamental point to explain this behaviour is the placement of the threads, and in this experiment each thread has its own core where it does not compete with other threads for cpu time. If this was not the case and multiple threads ran on the same core then some threads could fall behind because they would have to wait for the cpu to free, or in the best case where all threads get an even share of cpu time, there would still be no benefit in increasing the number of threads since they would all compete for the same resources; actually the time would slightly increase along the number of threads because of the overhead of openMP to distribute the iterations and cpu scheduling of the threads.



From `evolution_parallel.c`, plot of strong MPI scalability

3.2 Strong MPI scalability

Test description:

> given a fixed size (you may opt for several increasing sizes) show the run-time behaviour when you increase the number of MPI tasks (use as many nodes as possible, depending on the machine you run on)

All relevant command line instructions for this test.

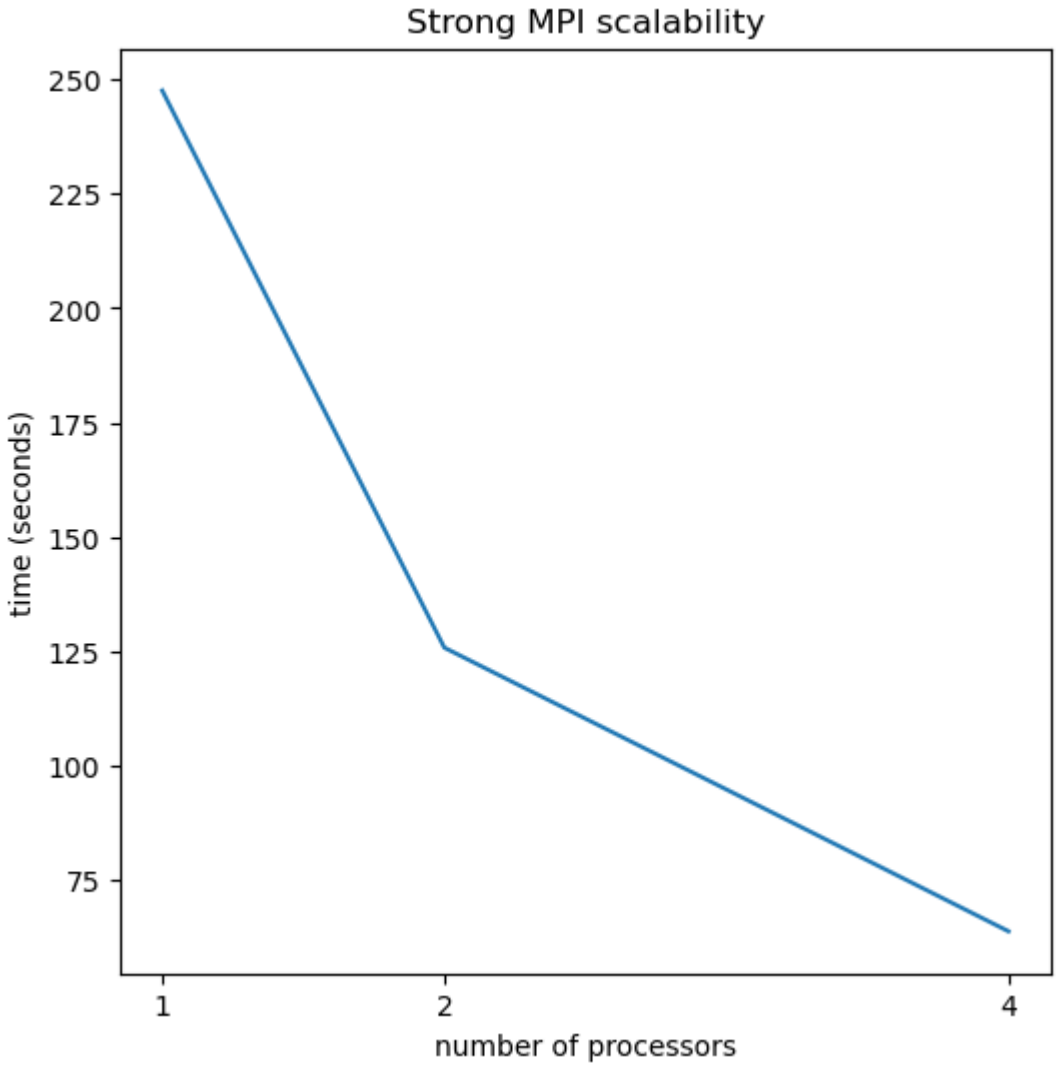
```
salloc -N4 -n4 -p EPYC --time=00:10:00
mpirun gol.x -r -f start_1920px.pgm -n 1000 -e 1 -s 1000
```

The number of nodes (`-N`) is set equal to the number of processes (`-n`), the default behaviour is to spread the processes over the nodes so it results in one process per node as per assignment.
The size of the grid is kept fixed at 1920x1920. The intent is by doing so to show the evolution time reducing as more work gets done in parallel and the amount of work stays consistent. By increasing the size of the grid along with the number of processes the highlight would be the propagation time, since the evolution workload of each process would remain the same but the number of nodes to communicate with would increase; this is the focus of the weak MPI test in section [3.3].
Since the focus of this test is on MPI and multi-processing I decided to not use openMP and multithreading, so the number of threads per process is fixed to one during this test.

The raw data collected in this test.

size	steps	evolution	processes	threads	tot time	evol time	avg propagation time
1920	1000	1	**1**	1	247556594	**247540577**	26
1920	1000	1	**2**	1	125833415	**125822970**	52251
1920	1000	1	**4**	1	63860569	**63848344**	55032

The EPYC nodes where I ran this test are a total of eight, unfortunately despite spending a week trying to allocate more than four nodes I was always unsuccessful. Still the data I collected gives space to considerations and I believe increasing further the number of nodes would have only confirmed the trend in the data.



From `evolution_parallel.c`, plot of strong MPI scalability, time is evolution time

---

The first thing that is clear looking at the plot is that the evolution time tends to halve as the number of processors doubles. This is expected behaviour since the total workload is the same and is has been split evenly between the processes.
Something that should catch the attention is the very sizeable surge in average propagation time jumping from one to two processes. This is explained by the need for inter-node communication at two processes which is not needed when the process is only one and the propagation takes almost no time (remember the times are in microseconds). Propagation time still increases going from two to four nodes, this might be because the number of different nodes a process communicates with goes from one to three.

Some parallels can be drawn between this data and the data of the previous test. In particular the first line of data in both tests differs only for the size of the grid, while other input parameters are the same. There is also a difference in evolution time (and total time) and the two are probably connected since the ratio between sizes (`3,68`) is similar to the ratio between evolution times (`3,40`).
The surge in average propagation time is not present in the data for the openMP test which uses only one process and one node.
Overall these connections increase the credibility of the data since different runs with similar parameters return comparable results.

3.3 Weak MPI scalability

Test description:

> given an initial size, show the run-time behaviour when you scale up from 1 socket (saturated with OpenMP threads) up to as many sockets you can keeping fixed the workload per MPI task

All relevant command line instructions for this test.

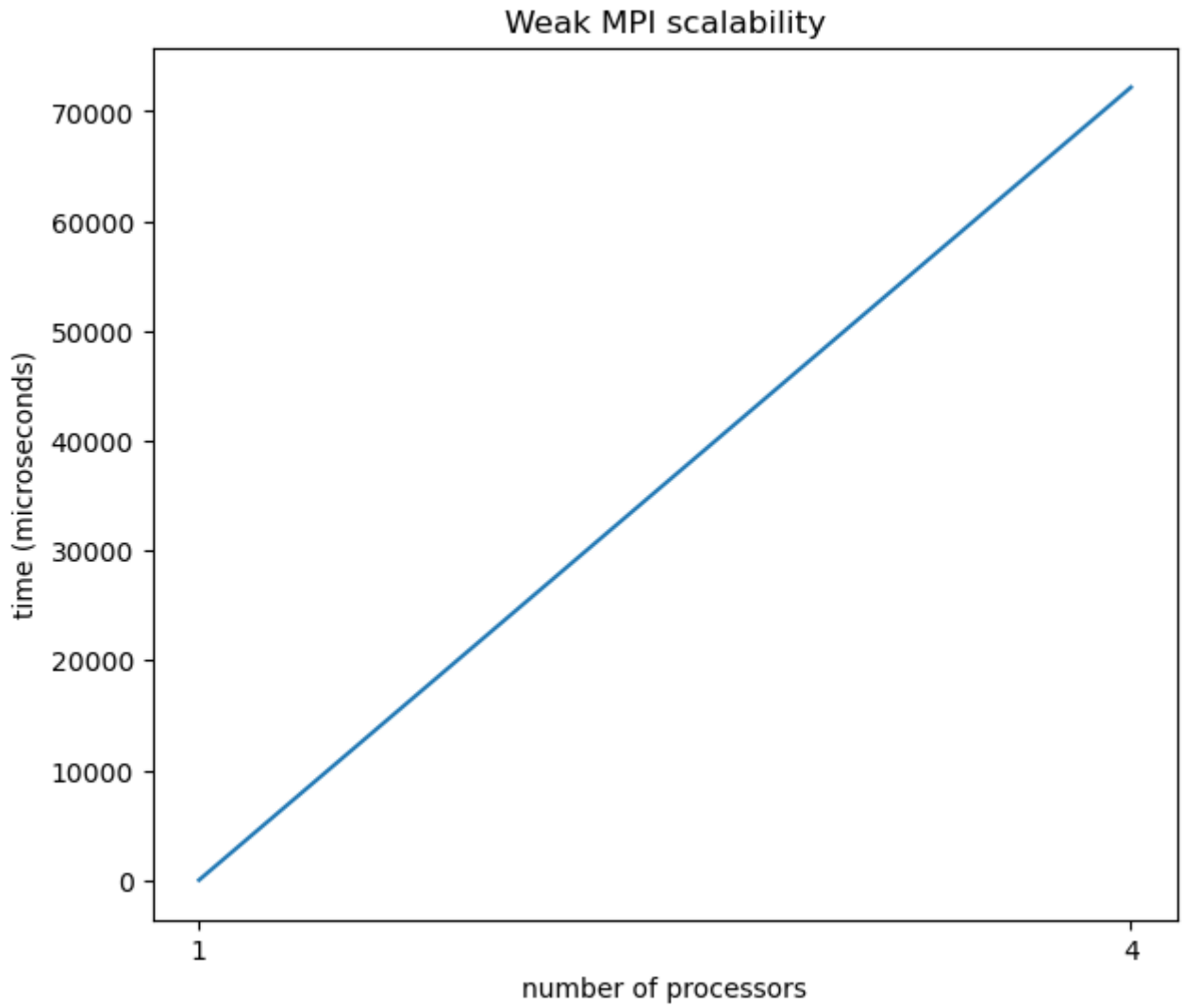
```
    salloc -N2 -n256 -p EPYC --time=00:10:00
    srun -n4 --cpus-per-task=64 mpirun --map-by socket gol.x -r -f start_200px.pgm
-n 1000 -e 1 -s 1000
```

Running on EPYC so number of threads to fill all cores in a cpu is 64 (not using logical threads, which are 2 per core and would make a total of 128 threads). Each node has two sockets so the numbers of nodes to allocate is half the number of processes to have one process per node. Total number of tasks to allocate is number of processes multiplied the number of threads (the model instruction above is allocating 4 processes so 4*64=256).
The size of the grid is such that each MPI process works on a 100x100 block.

The raw data collected in this test.

size	steps	evolution	processes	threads	tot time	evol time	avg propagation time
100	1000	1	**1**	64	1912650	1908870	**1**
200	1000	1	**4**	64	57035128	57023156	**52181**

Here I encountered the same problem of testing strong MPI scalability where allocating more than four nodes has been impossible. My plan was to run the test with 1, 4, 9 and 16 processes. These numbers are all squares because my code does not allow for rectangular grids, it allows for rectangular blocks but I fixed the size of a block to 100x100 for this test, so the only way to accommodate square blocks in a square grid is to have an even number of blocks in the rows and the columns. Anyway, to allocate 9 processes I need five nodes and I could not get a hold of them ever. Nevertheless, some considerations can still be drawn.



From `evolution_parallel.c`, plot of weak MPI scalability, time is average propagation time

---

Opposite of the two previous test this plot does not focus on evolution time but just on the average propagation time. The reason being, that the workload of the evolution per process is still the same using one or four processes. The increase in evolution time is not an actual increase of computation load is actually an increase of propagation time. Consider the first line of data, taking the average propagation time (`1`) and multiplying it by the number of evolution steps (`1000`) we get `1000` microseconds, number 10^3 times smaller than the evolution time so not a relevant portion of code for performance as I have stated so far. Now consider the second line of data, average propagation time = `72181` microseconds, multiplied by `1000` makes `72'181'000` microseconds, just ~ 5 million less microseconds than the logged evolution time. The propagation time is the most relevant portion of code for performance by one order of magnitude. Single digit seconds for evolution time is actually in line with other one process 64 threads runs. Average propagation time being in the low 70 thousand microseconds is consistent with the other two node run. So, the difference must depend on the remaining parameter that changes between these runs, the size of the grid. I used a small grid for this test compared to the others, giving a 100x100 block per process while the other tests had blocks of at least around 450 cells per dimension. This is actually not the first time that this happens, even with the 4 processes run for strong MPI the propagation time is more relevant than the evolution time, only when the block reaches the thousands cells per dimension evolution time is significantly heavier than propagation time. I take from this that using multiprocessing with blocks too small is not advisable.

It would have been interesting seeing how the propagation is affected by increasing the number of nodes a process communicates with. Using four processes two blocks of the eight to communicate with are in the same node, while testing with nine would have lowered the number to the minimum one, and potentially zero depending on the allocation of the processes related to their rank.

4. Conclusions

Overall, I think my implementation of this project has proved to be capable of scaling effectively both in multithreading and in multiprocessing.
I couldn't test the ceiling of multiprocessing capabilities unfortunately, but I will try after the deadline and maybe you would be interested in the results I find.

Each subsection covers one element of the project and contains my considerations regarding what worked well and what could have been improved in light of the data gathered.

4.1 Simpler splitting of the grid

A simpler method for splitting the grid would be to assign to each process groups of `n` rows or columns, depending on which is major, where `n` is the result of the size of the matrix divided by the number of processes. This approach has the same number of cells to propagate as the one implemented but is simpler because they are all grouped in only two rows. There would be no need for external columns propagation. Still all the problems with checking the neighbours would apply.

4.2 Core implementation as base for OpenMP and MPI

The simple implementation of the core game made it simple implementing multiprocessing and almost trivial implementing multithreading. As per its performance the results show that at the dimensions tested it rivalled with MPI propagation in some cases and all tests run in at most few minutes but I cannot calculate the FLOPS because I do not have the number of instructions executed so I do not know if it is actually good performance. Moreover, I am not aware of any real-world application of the Game of Life and as far as I know all the optimization on this problem is done either for the sake of the challenge or for research. Maybe there are MPI functions that would make the propagation more efficient but with my implementation of the game I cannot think of a better way to propagate the data. Overall, I would say that it certainly suffices for these tests but any serious application should look into optimizing it. To give further insight in the optimization I would proceed aiming at minimizing the amount of data to propagate through the processes.