Cellular Automata Framework Report

Student: Simone Baccile (mat. 563289)
Course: Parallel and distributed systems: paradigms and models
University of Pisa
Academic year 2020-2021

Table of Contents

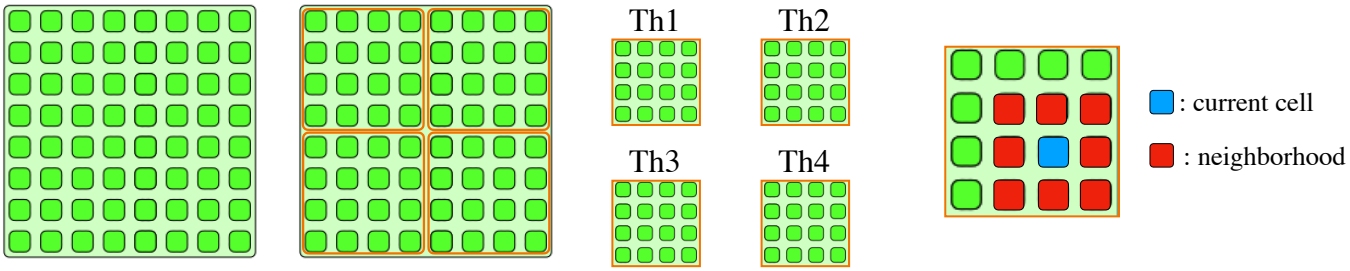
Introduction

The project consists of the implementation of a framework to define and execute a Cellular Automaton (CA). A CA is a discrete model of computation consisting of a grid of cells, each in one of a finite number of states. The grid can be in any finite number of dimensions and for this project, it is a toroidal grid, where the last row (column) must be considered adjacent to the first row (column). An initial state is determined by an assignment of a particular state for each cell. For each cell is defined a set of cells called neighborhood. In this project, the neighborhood is called Moore neighborhood, which for each cell contains the 8 neighboring cells. An initial state is selected by assigning a state for each cell. A new generation is created step by step, according to fixed rules that determine the new state for each cell considering the current state and the states of its neighborhood. The whole grid is updated simultaneously at each step.

Main design choices

The CA is a problem to which you can apply data parallel strategies to parallelize its implementation. Data parallel strategies consist to apply functions (in this case rules) to a collection of data (in this case the entire grid of cells). You have all these data at the starting point, so you don't have to manage data arriving over time. The pattern that fits well in this kind of problem is the stencil pattern and this is what I've implemented in the framework. The main idea of the stencil pattern is the following:

1. Split the grid in chunks, typically proportioned to the number of workers.
2. Assign chunks to threads.
3. Each thread takes care of applying to each cell in the chunk, the rule to update it.



Initial CA grid | Split grid in chunks (1) | Assign chunks to threads (2) | Update cells in each thread (3)

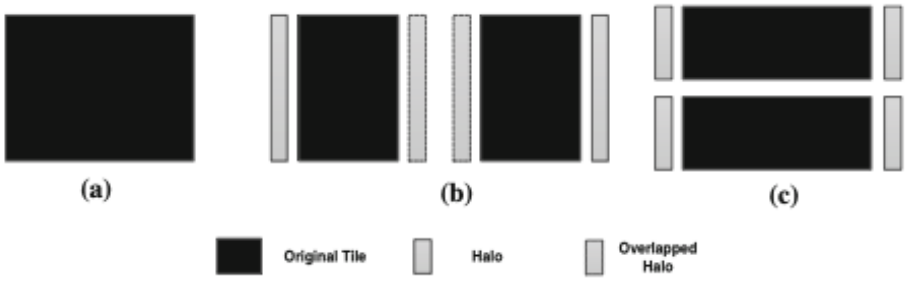
The first two steps are for the initialization, the third step is applied repeatedly for a given number of steps. Applying this pattern to the CA problem you need to manage some issues concerned with shared cells between chunks, synchronization between steps, and analysis of load balancing. Now I analyze all these issues in the following paragraphs.

Shared cells

When you split the whole grid in chunks to compute the new state for a cell, you need to evaluate its neighborhood, but can happen that some neighbors don't belong in the same chunk of the current cell. So you need to manage these shared cells between chunks. I've found two solutions to this problem that I will now describe.

- **Halos:** after dividing the grid into chunks, you have to identify the shared cells between chunks. Now you put in each chunk a copy of these shared cells, that are not updated but only used as

neighbors. These copied cells are called halos. The main advantage is that each thread works with a relatively small grid + relative halos. On the other hand side, as the number of workers increases, so does the number of duplicate cells in the halo. [1]



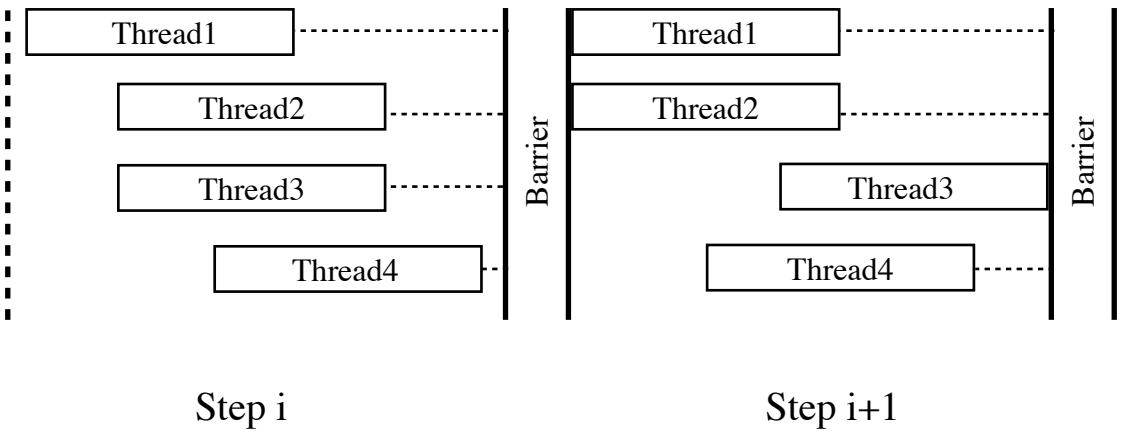
Examples of halo from paper [1]

- **Alternate buffers:** use a duplicate of the initial grid and share to each thread the initial grid only for reading and the duplicate to compute the new states. At the end of each step, you have to save the new generation from the duplicate grid to the initial grid. In this case, you don't have an increase of the halos with the increase of the number of workers, so you don't have any multiple duplicates between threads except the entire duplicate grid.

In the implementation of the CA framework, I chose the alternate buffers solution for two reasons. The first is that this solution doesn't increase its complexity with the increasing of the number of workers. I could have worse results when the number of workers is too low, but in general (especially if the grid is toroidal) the results are better than the halos solution. The second reason is that it is more simple, more readable, and has more understandable code than halo.

Barrier

The second problem that you have to manage when you parallelize the execution of a cellular automaton is related to the synchronization between steps. This problem comes out because if one thread has finished the computation for its chunk at step *i* and it would like to start the computation for step *i+1* it can't. The thread cannot do this until all the other threads gave finished to update their chunk, otherwise, it will work with a grid not updated for the current step, but with previous step cell states. To solve this problem in the project I decided to use a barrier. A barrier synchronizes the thread between each step. If a thread finishes step *i* it will wait on the barrier, until all threads have finished step *i*. When all threads have finished the barrier saves the results from the duplicate grid to the original one, and awakens all threads that can restart the computation for step *i+1*.



Load balancing

To manage the load balancing I've made the following considerations. The grid is divided into chunks based on the number of workers, so all the threads have the same number of cells to update (only the last chunk may have fewer cells). To update each cell the operations are always the same and they don't depend on the configuration of the neighborhood or the current state of a cell. For this reason update, one cell takes more or less the same time for all the grid. Considering these, I chose to divide the grid into chunks using a static method at the beginning of the computation. A dynamic approach wouldn't bring further benefits than the static one.

Expected performances

To analyze the expected performances of the CA framework in an ideal situation, first I'll analyze the performance of the sequential version in terms of the time of completion, then I'll analyze the performance of the parallel version. In both cases, I'll describe the algorithm in pseudocode, which doesn't correspond to the real implementation of the framework, but it's a good approximation to compute ideal performances.

Sequential

Algorithm 1: Sequential CA

Input: CA grid G, number of steps S

```
rows ← G.rows;
columns ← G.columns;
copyGrid ← G;
for i ← 0 to S do
    for r ← 0 to rows do
        for c ← 0 to columns do
            copyGrid[r][c] ← computeNewState(G[r][c]);
        end
    end
    G ← copyGrid;
end
```

In *Algorithm 1* there is the pseudocode of the sequential version. It takes the grid of the CA and the number of steps for the simulation. The alternate buffer is *copyGrid* and it is used to store the new values for each step. At the end of each step, *copyGrid* is saved in the original grid. The function *computeNewState* read the cell from the original grid G and compute the new state considering the neighborhood of the current cell. The completion time can be formulated like:

$$T_{seq} = S \cdot rows \cdot columns \cdot t_{newsState}$$

Where S is the number of steps for the simulation, *rows* is the number of rows of the grid, *columns* is the number of columns of the grid and $t_{newState}$ is the time to compute the new state of a cell.

Parallel

Algorithm 2: Parallel CA

Input: CA grid G, number of steps S, number of workers nw

```
rows ← G.rows;
columns ← G.columns;
copyGrid ← G;
barrier ← initializeBarrier(nw);
δ ← (row·columns)/nw;
for i ← 0 to nw do
    chunk[i] ← (i · δ, (i + 1) · δ);
end
for i ← 0 to nw do
    createThread(CAThreadFunction, chunk[i], S, G, copyGrid,
     barrier);
end
for i ← 0 to nw do
    joinThreads;
end
```

Algorithm 3: CAThreadFunction

Input: CA grid G, chunk C, number of steps S, copy of the Grid *copyGrid*, barrier B

```
first ← C.first;
last ← C.last;
for s ← 0 to S do
    for i ← first to last do
        copyGrid[i] ← computeNewState(G[i]);
    end
    waitOnBarrier(B);
end
```

In *Algorithm 2* there is the pseudocode of the parallel implementation. It takes the grid of the CA and the number of steps for the simulation, like in the sequential version, but also takes the number of workers used to parallelize the work. The alternate buffer is also used here, but in this case, it is saved in the original grid, after each step, by the barrier. In the initialization phase, the algorithm computes the chunks of cells for each thread. Each thread computes the function defined in *Algorithm 3*. It iterates for the number of steps and updates the cells contained in its relative chunk. In this case, the implementation accesses directly to a position of the grid, that can be simply computed considering the grid as an array of *rows*columns* positions. At the end of each step, the thread waits on the barrier until all the threads have finished the current step. The barrier is implemented such that before waking up the threads save the *copyGrid* on the original grid. The completion time of the parallel algorithm is the following:

$$T_{par} = t_{chunk} \cdot nw + S \cdot \left(\frac{rows \cdot columns}{nw} \cdot t_{newState} \right)$$

Where S is the number of steps for the simulation, *rows* is the number of rows of the grid, *columns* is the number of columns of the grid, *nw* is the number of workers, t_{chunk} is the time to compute the chunk for a thread and $t_{newState}$ is the time to compute the new state of a cell. The first part $t_{chunk} + nw$ is negligible because it consists just of a simple numerical computation of the indexes. So the ideal computation cost is given by the second part of the addition.

Implementation

The framework is implemented in 3 classes. The first class *ca.h* contains the methods to define and describe a cellular automaton. It contains the sequential execution method for the simulation of the CA. *ca.h* is the base class and it's inherited from the other two classes, *ca_thread.h* and *ca_ff.h* that implement the parallel version for the simulation overwriting the base execution method inherited from *ca.h*. Now I'll describe in detail all of these implementations.

ca.h

This is the main class of the framework, it implements all the methods to define a CA with its states, rules, and grid.

- **States:** the states are strings saved in a map structure. The map contains pairs (state_tag, state_string) where the state_string is the name given by the user, state_tag is an increasing index assigned to each state. The states can be set by calling the method *setStates* which takes a vector of strings containing the states.

- **Rules:** the rules are defined in a structure called *Rule*. This structure contains the current state, the vector to describe the neighborhood, and the new state to update the cell. The vector has the size equal to the number of states of the CA. Each position corresponds to the index of each state and contains the number of cells in the relative state in the neighborhood. For example, if you have only two states (0,1) and you want to define a Moore neighborhood with 3 (0) states and 5 (1) states the vector should be (3,5). There is also the possibility to insert a special vector containing all 0 to define a rule that is applied in any case, without considering the neighbors. The rules are stored in a map containing pairs ((current_state, neighbors_vector), new_state). I chose to use a map because the rules do not change over the time and I need to access them quickly to find which rule to apply in each step. The rules can be set using the methods *setRules* or *addRule*.
- **Grid:** the grid of the CA is a vector of vectors containing integers, which are the representations of the states. Scanning the first vector you can access rows, whereas in the internal vectors you can access columns. The size of the grid is defined by the constructor of the class, but to initialize the grid you have to call the method *initializeGrid* passing the grid or *initializeRandomGrid* to fill the grid with random states.

The execution method is called *executionCA* and takes as a parameter the number of steps. This method is a virtual function that can be reimplemented by the classes which inherit this class. In the class *ca* it implements the sequential version of the execution.

ca_thread.h

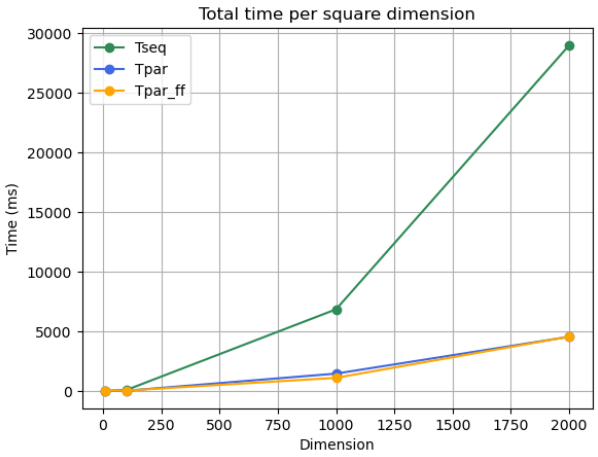
This class inherits the methods from the *ca* class to define a CA and overwrite the execution method to implements the parallel version with the standard library. To implement this version with the standard library, it implements a barrier using a mutex and a condition variable on the number of workers. At the beginning it computes the chunk for each thread and then it spawns threads that execute the function *updateChunk*. This function updates all the cells in the given chunk and puts the threads in waiting until all the threads have completed the step. This is done by calling the method *barrierWait* that maintains a counter with the number of threads that have completed the step. When the counter is equal to zero, the results are stored in the original grid, the counter is reset and all the threads are awaked.

ca_ff.h

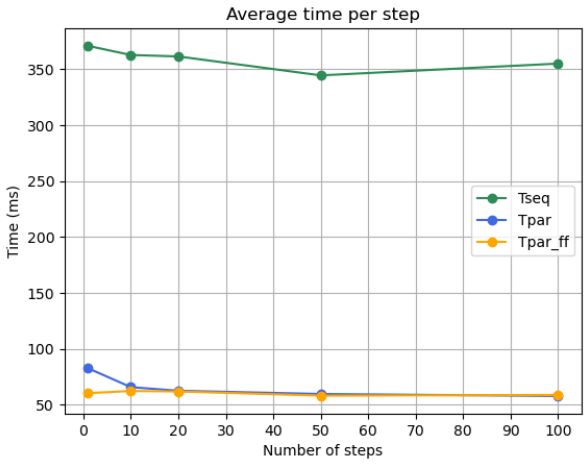
The parallel version that uses the FastFlow library is implemented in the class *ca_ff*. Also in this case the method *executeCA* is overwritten using the *parallel_for* of FF. In particular, for each step, it is called the method *parallel_for_idx* that passes chunks of the grid to threads. Each thread executes the lambda function *map* that iterates over the chunk and updates the cells in the copied grid. At the end of each step, the results are saved in the original grid. The main issue with this implementation is that at each step a new *parallel_for_idx* is called, which means that at each step it's created a farm and then it's destroyed. This could slow down the execution of the simulation.

Performances

I evaluated the performances of all of these implementations and in this paragraph, I'll explain the results obtained. All the evaluations are done considering as a CA, Conway's Game of Life with only two states and its rules. The variable parameters that were evaluated are the size of the grid, number of steps, and number of workers. Now I'll describe the results.

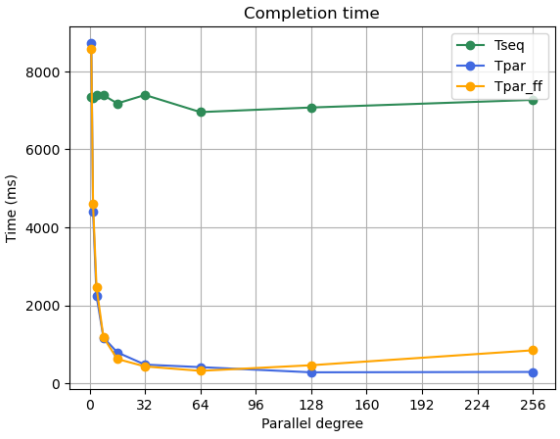


(a)

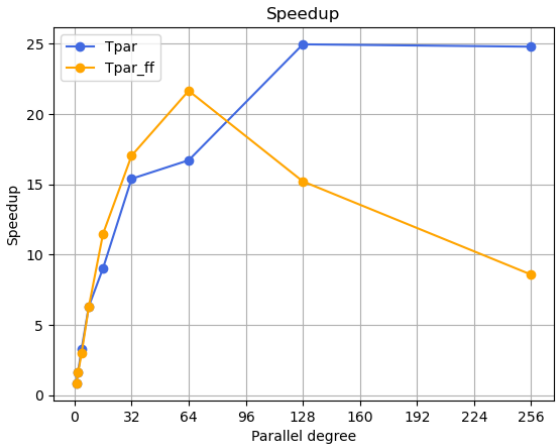


(b)

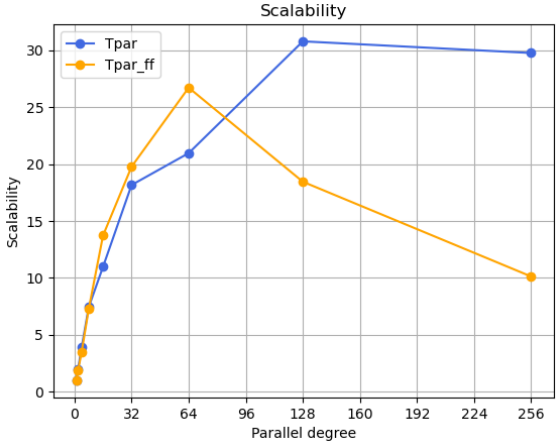
In these two graphs, I evaluated the time completion with different sizes of the grid (a) and the average time per step with a different number of steps (b). In particular, in the graph (a) I considered different square grids from size 10 to size 2000, with 5 steps and 8 workers. In this case, you can see that the time of completion for the two parallel versions is very similar and grow much slower than the sequential. While in graph (b) there is the average time per step, used to analyze if an increasing number of steps can increase the time of completion. This doesn't happen as you can see except when the number of steps is very small (less than 10 for instance). These variations can be attributed to the overhead for the initialization of the algorithm. If I have 1 step I've to pay this overhead for the only step. When I've to compute more steps I can spread this cost. For the plot of this graph, I considered a grid of 500x500 with 8 workers, changing the number of steps from 1 to 100. Both graphs contain the average time of 5 executions.



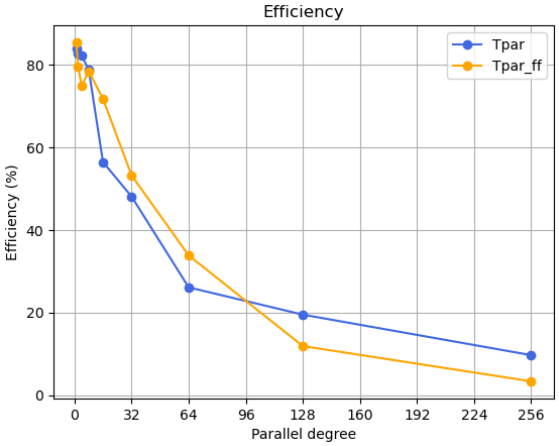
(c)



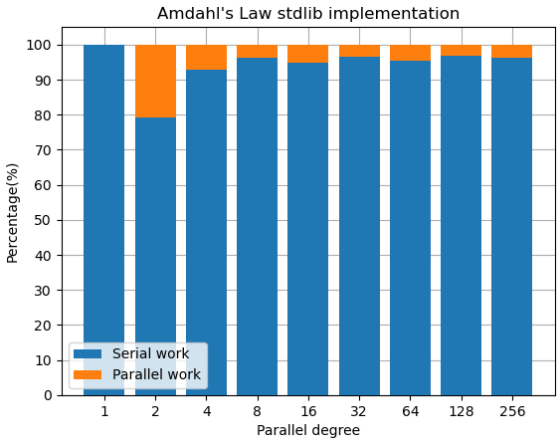
(d)



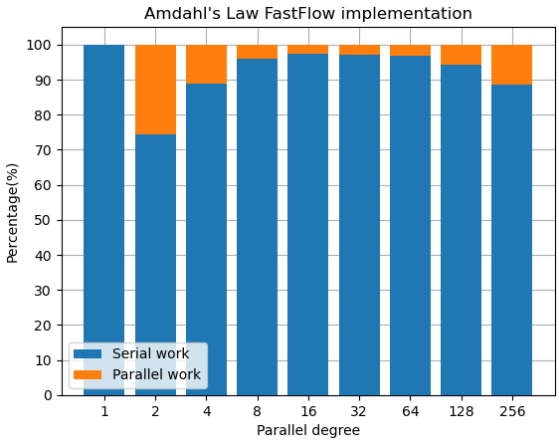
(f)



(e)



(g)



(h)

For all these graphs from (c) to (h), I considered a grid of 1000x1000 cells, with 5 steps, taking an average of 5 repetitions. I used powers of 2 as the number of workers from 1 to 256. In graph (c) you can see the **completion time**. The time of the sequential version is almost the same (except for little fluctuation), whereas the time for the parallel versions decreases a lot when you increase the number of workers. In particular, for this task, you can obtain the best results when the number of workers is between 8 and 32. When the workers exceed 128 there are some differences between the standard library version and the FF version. I think it depends on the different implementations. For instance, when in the standard library implementation I fork the threads one time with all computed chunks for all the steps, in the FF version, at each step, I've to fork new threads (in particular the *parallel_for* set up a farm each time), re-compute chunks, execute and join all. This difference affects all the other graphs.
In graph (d) there is the **speedup** obtained from the two parallel versions. The speedup obtained is between 20-25 (times faster than the sequential version) when the number of workers is between 32-128.
In graph (e) I represented the **scalability** of the parallel versions increasing the number of workers. The better scalability is obtained from the standard library version because when the FF version exceeds 64 workers it has decreasing performances.
The **efficiency** is represented in graph (f). You can see that the algorithms don't reach the ideal efficiency of 100%, but start from a little bit more than 80%, due to overheads like forking more threads, waiting on the barrier, switching buffers, or compute chunks. In particular, the FF version is more efficient than the standard version until the workers are more than 128.
In the graphs (g) and (h) you can visualize the distribution of **serial** and **parallel work** as described in Amdahl's Law. To obtain these charts I used the following formula [2]:

$$parallel\_work = \frac{n}{n-1}\left(1 - \frac{1}{sp(n)}\right)$$

In graph (g) for the standard library implementation, you can see that the law is respected and you see that from 8 workers the fraction of parallel work cannot be further decreased. Instead, the graph (h) doesn't follow the law due to the problem explained previously for the completion time. When the number of workers exceeds 128 the fraction of parallel work grows again.

Differences between C++ and FastFlow solutions

I've already talked about the differences between the standard library implementation and the FastFlow implementation of the framework. In the following table, I'll recap all these differences.

	ca_thread	ca_ff
Steps	• Computed inside threads. • Update chunk of cells. • Barrier to synchronize each step.	• External to threads. • Update chunk of cells. • Independent *parallel_for_idx*.
Chunks	Calculated once for all steps	Calculated for each step
Threads	Forked once at the beginning	Forked and joined at each step. In particular, an FF farm is forked at each step
Synchronization between steps	Barrier	For loop

Build and run

In this paragraph, there are the instructions to build and run the project.

Build

To build the project open the directory of the project and compile it using the command *make*. If the FastFlow library is not linked in the standard include path, you have to modify the *Makefile* and add to INCLUDES variable the FastFlow path (eg. INCLUDES = -I/path/to/ff/dir).

Run

The main application is *game_of_life* that implements Conway's Game of Life with a random grid and prints at the end the times of completion of the sequential version and the two parallel versions of the framework. If you want to execute *game_of_life* here is the syntax:

```
./game_of_life rows columns steps parallel_degree
```

It takes 4 arguments, the number of rows, the number of columns, the number of steps for the simulation, and the number of workers for the parallel implementations. In the end, it prints a string like:

```
Tseq: 7074619 Tpar: 283638 Tpar_ff: 465546
```

Where Tseq is the time for the sequential version, Tpar is the time of completion of the parallel version with thread and Tpar_ff is the time of the FF parallel version. All the times are in microseconds.

If you want to execute the same algorithm more than one time to take an average time of execution you can use the bash script *avg_time.sh*. It takes one more argument than *game_of_life* to define the number of repetitions of the algorithm.

```
./avg_time.sh number_executions rows columns steps parallel_degree
```

To execute a deep test I wrote *script.sh* that implements a script to test the framework with different steps, with different grid sizes, and with a different number of workers. If you want to execute this test execute the command *make script*. It produces three txt files containing the times for each test (this takes 10-15 minutes on the remote machine). You can also personalize this test opening *script.sh*. You can modify the array STEPS to change the steps to test (fixed grid 500x500, 8 workers), the array DIMENSIONS to change the grid dimensions to test (fixed 5 steps, 8 workers), the array WORKERS to test a different number of workers (fixed grid 1000x1000, 5 steps), and the variable N to change the number of repetitions for each test (5 repetitions by default).
If you want to plot graphs about the test done by *script.sh* you can execute the command *make plot*. This command creates a directory with the plots using the Python script *plot_figure.py*. To use this command you need Python and the modules matplotlib, pandas, and numpy.

Examples

In the project directory, you can find a directory called *models*. It contains some cellular automata models implemented with this framework. Use the command *make* inside *models* directory to compile them. The models implemented are the following:

- **Brian's Brain:** two dimensional grid, each cell may be in one of three states on, dying, or off.
- **Wireworld:** cellular automaton to simulate transistors. It uses four states.
- **Seeds:** cellular automaton similar to Conway's Game of Life.
- **Forest-fire:** simply cellular automaton to simulating the spread of fire inside a forest.

For more details, I've put the Wikipedia pages in Bibliography.

API

Now I'll describe the API of the framework. If you want to use this framework you have to include "ca.h" or "ca_thread.h" or "ca_ff.h", according to the type of execution you want of the CA.

- **ca**(int rows, int columns): the constructor for *ca.h*.
- **ca_thread**(int rows, int columns): the constructor for *ca_thread.h*.
- **ca_ff**(int rows, int columns): the constructor for *ca_ff.h*.
- **setStates**(std::vector<std::string> states): method used to set the states of the CA. The states are string passed as vector. The position of the states in the vector is important because determines the index assigned to each state.
- **setRules**(std::vector<Rule> rules): method used to set the rules of the CA. The Rule type is a struct with currentState, neighbors, newState. currentState and newState are the indexes of the cell's state in the current step and if the rule is applied. Neighbors is a vector with the number of positions equal to the number of states containing the sum of cells in each state for the current neighborhood. The sum of all vector's positions must be equal to 8 (Moore neighborhood).
- **addRule**(Rule rule): add one rule to the set of rules.
- **initializeGrid**(std::vector<std::vector<int>> grid): initialize the CA grid passing a vector of vector with this method.
- **initializeRandomGrid**(float density=0): initialize the grid randomly. If you pass a density to this method, the first state is considered as the empty state and the others are non-empty. In this case, the density represents the percentage of non-empty states inside the grid.
- **printStates**(): print states and relative index.

- **printRules**(): print all the rules of the CA.
- **printGrid**(): print the CA grid.
- **getTime**(): print the last execution time of the CA in microseconds.
- **executeCA**(int steps, bool verbose=false): simulate the CA for *steps* steps. If the verbose argument is set to true after each step the grid is printed on the screen.
- **executeCA**(int steps, int parDegree=1, bool verbose=false): simulate the CA for *steps* steps. parDegree determines the number of workers used for the parallel execution. If the verbose argument is set to true after each step the grid is printed on the screen. If this method is called from a *ca_thread* object it executes the thread parallel version, if it is called from a *ca_ff* object it executes the FF parallel version.

For more details the classes are described in *ca.h, ca_thread.h,* and *ca_ff.h*.

Bibliography

[1] "Tuning framework for stencil computation in heterogeneous parallel platforms", Taieb Lamine Ben Cheikh, Alexandra Aguiar, Sofiene Tahar, Gabriela Nicolescu, Springer Science 2015.

[2] "Parallel Programming: Speedups and Amdahl's law", Mike Bailey, http://web.engr.oregonstate.edu/~mjb/cs575/Handouts/speedups.and.amdahls.law.2pp.pdf, Oregon State University, 2021

Other resources consulted:

- "Stencil pattern", https://ipcc.cs.uoregon.edu/lectures/lecture-8-stencil.pdf, University of Oregon.
- https://en.wikipedia.org/wiki/Brian%27s_Brain
- https://en.wikipedia.org/wiki/Wireworld
- https://en.wikipedia.org/wiki/Seeds_(cellular_automaton)
- https://en.wikipedia.org/wiki/Forest-fire_model