Università di Pisa

Computer Science Master Degree

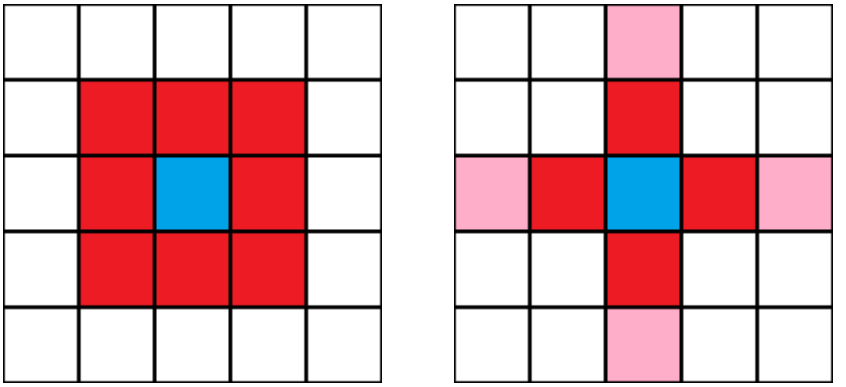
SPM Project
Cellular Automata

Author: Andrea Zuppolini

ACADEMIC YEAR 2020/2021

1 Introduction

This project has been realized for the 2020/2021 final exam of the SPM course. The main purpose is to implement a framework for a CellularAutomata simulation, both using C++ threads and Fastflow library. Here follows a short introduction to what is a Cellular Automata and where it can be used as a descrete model of computation describing complex systems.
The automaton consists in a grid of cells each one having a finite internal state, and the inital one can both be provided by the user or generated from a random probabilistic distribution.
The grid, in this particular case, has been implemented like a toroid where the first row/column is consecutive to the last row/column. Each cell is updated following a determinate rule, provided by the user, and is run for a finite number of timesteps and can change its internal state or leave it unvaried.
It is important to notice that the rule computed on the cell is a function of both the cell state and its neighbours's state. A neighbourhood can be defined in many different ways, the most famous ones are *Von Neumann neighbourhood* and *Moore neighbourhood*, the first is obtained considering the neighbours in the four cardinal directions, the second by enlarging the first with the cells in the diagonal.



(a) Moore neighbourhood (b) Von Neumann neighbourhood

Figure 1: Image from Wikipedia's Cellular automaton page
Two different neighbourhood. The blue square indicates the actual cell while the red ones its neighbourhood. The pink ones are also considered as neighbours in the case of range-2 cross neighbours

2 Cellular Automata

In this section details about the technical implementation will be presented. As requested there're two different implementations of the framework: one using C++ Threads and the other using Fastflow library. The code, in both the implementations, is organized two headers (.hpp files) and their implementation (.cpp files) with an additional .cpp file for the timer implementation.

2.1 Cellular Automata class

This section refers to the Cellular Automata class implemented using C++ threads.
The grid is implemented using C++ std::vector of vectors of integers which, from now on, will be called "grid2D". Grid dimension is determined by the user who also provides the updating rule, the number of steps and the number of threads for the execution.
To compare the performances and to highlight the effects of parallelism, there is also a method providing the sequential execution of the automaton which goes cell after cell, gets their neighbourhood and applies the updating rule to it.
Before going into the details of the thread implementation there is the need to identify to which class of problem the Cellular Automata belongs. The updating rule is applied to all the grid at each timestep and it doesn't change over time. The crucial point is the dependency between the cells: updating a cell's state alters the next cell computation. The grid is divided in equal parts among the threads but, in order to not influence each other during the computation, they should first compute the new states and then synchronize with other threads, and, only after, they can write the results on the original grid. The proposed solution is close to the one just discussed with only one main difference: Threads can write directly on the original grid since, what they're working on, is a deep copy of it which is updated in a synchronization step by the main thread. So at each timestep the threads wait their colleagues to end the computation and, then, wait the main thread to update the deep copy of the grid in order to start a new step.

```
for (int j = 0; j < timesteps; j++)
    {
        pthread_barrier_wait(&barrier1); //Waits the threads
        deep_copy = grid2D(grid);        //Updates the copy
        pthread_barrier_wait(&barrier2); //Unlocks the threads
    }
```

The slice of code above shows how the main threads synchronizes with the workers. All of the threads work on a single copy of the grid, which is passed to them by reference. By doing this we ensure that the time needed to compute the copies stay minimal. The impact of this copy has been measured and, over 1000 samples, the average time needed to copy the grid is 4187 µseconds, hence is many orders of magnitude smaller.

The Cellular Automaton grid computation clearly refers to a MAP computation: the amount of work is divided into equal parts among the threads, they can compute their job indipendently without the need of communicating with the others (Since they work on a copy) and, then, they synchronize on one (actually two) barriers. The solution used in this project follows the spirit of MAP pattern resolution and updates the grid directly:

```
//Computing the rule on the actual cell
grid[i][j] = rule(getNeighbourhood(i, j, grid_copy));
```

The previous version of the framework had the grid as a synchronized data structure. This clearly leaded to much slower performances but, in order to prevent a state of inconsistency it was clearly needed. C++ vectors belong to the container framework and, while getting more information about it, what was found out in the documentation:

"Different elements in the same container can be modified concurrently by different threads, except for the elements of std::vector⟨bool⟩"

Taking this into account, the synchronization mechanism was deleted since there's the possibity to ensure that each thread will be working of different elements of the vector. There's also another way to execute the simulation: using the OpenMP parallel for, which exploits the independence between the iterations and assigns them to different threads. To use this tool the method ompParallelFor() has been provided:

```
#pragma omp parallel for num_threads(num_threads)

    for (int i = 0; i < grid.size(); i++)
        for (int j = 0; j < grid[0].size(); j++)
            //Compute the update
```

Note that there's the chance to collapse the cycles using the *"Collapse()"* option, this is particularly useful when the outer cycle has far more less iterations than the inner one and the load can be better balanced. However, in this case, since the inner loop's instructions depend on the i,j there's not a significant augment in performance.

2.2 Cellular Automata Fastflow

Here follows the discussion about the Fastflow implementation of the Cellular Automaton. There are many ways to implement the same job done with the C++ threads with fastflow there will be provided two: a very simple one using fastflow's parallel for and a more complex one using a farm.

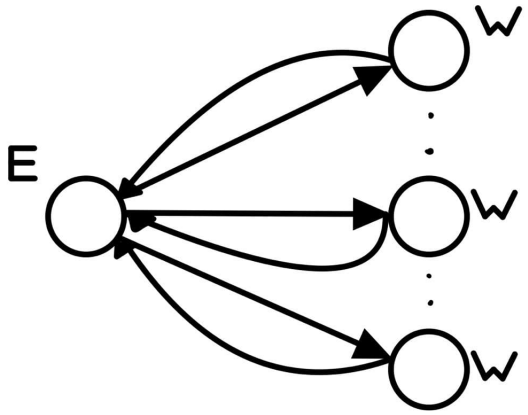
2.3 Parallel For

```
for (int t = 0; t < timesteps; t++)
    {
        //Static division of the job between the workers
        pf.parallel_for(
            0, num_rows, 1, 1, [this,iter](const long i)
            {
                for (int j = 0; j < iter; j++)
                    updatedGrid[i][j] =
                    rule(getNeighbourhood(i, j, &previousGrid));
            },
            num_threads);
        previousGrid = grid2D(updatedGrid);
    }
```

In the fragment of code above we see that some modification in the code have been made with respect to the OpenMP parallel for: here, the pf object requires the function to compute to be defined as a lambda. The outer loop hasn't been taken in account since its iterations aren't independent. Once again the threads will be working on a copy of the grid and will update it directly. Here what's made parallel is the cycle which goes through the rows and the work will be equally divided between the threads following a static scheduling.

2.4 Farm implementation

Here is going to be discussed the farm implementation of the Cellular Automaton using the fastflow library. Usually a farm, in its classic meaning is made of an emitter, who distributes the job, a certain number of worker threads which compute the given jobs, and a collector thread which collects the output results. Since the workers can directly work on the grid there's no necessity for a collector, so it has been removed to save a thread in order to increase the workers.



The structure used is the one in the picture above, which is the classical farm structure without the collector and which channels between workers and emitter. This architecture is in a master-worker version: the emitter assigns the job and the workers execute it giving feedback to the master. Going into details, implementing this architecture means rewriting the emitter and provide a worker, and that's what has been done:

1. The Emitter is initialized and it executes the svc_init() function, where all the PAIRS, an alias for two integers representing an interval, are initialized.

2. The Emitter executes the svc() function, and it sends all of the pairs to the worker nodes, and starts taking trace of the work finished.

3. The worker node execute the updating rule on their section of the grid and return a value to the emitter

4. The emitter wait all the threads, then if it has reached the exact number of steps provided by the user it ends the execution, else it restarts the simulation re-sending out all of the pairs.

Both the structures of first stage (Emitter) and second stage (Workers) are instantiated passing the CellularAutomata as reference. So, in line with what has been done with the Thread implementation, they all look at the current state on a deep copy of the grid and then directly update the grid.
The method showed below, startFarm() has the task of instantiating the stages, create a vector of workers and to pass them to the Farm's constructor. It also removes the collector and creates the back channels from the workers to the emitter. At last, it starts the farm.

```
int startFarm()
    {
        firstStage emitter(this);
        std::vector<std::unique_ptr<ff_node>> Workers;
        for (int i = 0; i < num_threads; i++)
            Workers.push_back(make_unique<secondStage>(this));
        ff_Farm<float> farm(std::move(Workers), emitter);
        farm.remove_collector();
        farm.wrap_around();
        if (farm.run_and_wait_end() < 0)
        {
            error("running farm");
            return -1;
        }
        return 0;
    }
```

3 Experiments

Both the Fastflow and C++ threads framework have been tested on a remote server and a local machine. In this section it will follow the description of the machines and the results obtained on them. Those are the CPUs characteristics of the server:

- Architecture: x86_64
- CPU(s): 256
- Thread(s) per core: 4
- Core(s) per socket: 64
- Socket(s): 1
- CPU family: 6
- Model: 87
- Model name: Intel(R) Genuine Intel(R) CPU 0000 @ 1.30GHz
- CPU MHz: 1235.101
- L1d cache: 32K
- L2 cache: 1024K

N°Threads	Fastflow	Thread execution
2	40431249 µsec	40257615 µsec
4	20449946 µsec	20385704 µsec
8	10227754 µsec	10380796 µsec
16	5458312 µsec	5407741 µsec
32	3112635 µsec	2851768 µsec
64	2298239 µsec	1582616 µsec
128	1520471 µsec	1266126 µsec
256	1162843 µsec	1384389 µsec

In the table above the results of the executions are shown, in particular, those are an average of 20 different executions on the same machine, the simulation refers to a 1000x1000 grid with the rules of the Game of life and it goes on for 40 timesteps.
In order to have a measure of how parallelism increases the speed of execution this is the Sequential time: *65573710 µsec*.
A useful measure of the performance is the so called speedup:

$$speedup(n) = \frac{T_{seq}}{T_{par}(n)} = \frac{65573710sec}{1162843sec} \approx \mathbf{56}$$

To achieve a complete analysis for this result it should be compared with the efficiency:

$$\in (n) = \frac{T_{id}(n)}{T_{par}(n)} = \frac{\frac{T_{seq}}{n}}{T_{par}(n)} \approx \frac{256147sec}{1162843sec} \approx \mathbf{0.22}$$

The first measure, the speedup, refers to how fast the parallel execution is being with respect to the sequential one. While the efficiency shows how good our parallel execution is comparing it with the ideal parallel time. The efficiency measure, which is bounded to 1, is speaking out loud that the performance should be much more better since we're using 256 threads. Before going into the details of what could be the problem with the implementation, it has been taken into account also how much the size of the grid impacts on the execution: The simulation above uses a 1000x1000 grid, a reasonable choice since the simulation had to be run many times. Also much bigger dimensions of the grid, like 10000x10000, up to 100000000 cells, have been taken into account. Their result:
Tseq = 6125613731 µsec, Tpar = 109915914 µsec with the thread execution and a
Tpar = 118675253 µsec with fastflow. The speedup for both the execution is similar and it is still around **56**. It could be concluded that The speedup doesn't really increase with the grid dimension. Different configurations have been tested, changing the updating rule, also trying to run them on a different machine. By the way, only the significant part of all the tests has been reported.

Those are the CPU characteristics of the simulations executed an a second machine, followed by the results obtained.

- Architecture: x86_64
- CPU(s): 12
- Thread(s) per core: 2
- Core(s) per socket: 6
- Socket(s): 1
- CPU family: 6
- Model: 158
- Model name: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
- CPU MHz: 1235.101
- L1d cache: 192K
- L2 cache: 1,5MiB
- L3 cache: 12 MiB

N°Threads	**Fastflow**	**Thread execution**
2	264183640 µsec	252246290 µsec
4	139124768 µsec	142481693 µsec
8	113797856 µsec	123751126 µsec
12	104787436 µsec	111313422 µsec

For the sake of conciseness, it won't be repeated the same thing said above for the other simulation, since the results are very similar from the point of view of speedup and efficiency:

$$\in (n) = \frac{T_{id}(n)}{T_{par}(n)} = \frac{\frac{T_{seq}}{n}}{T_{par}(n)} \approx \frac{39799006sec}{104787436sec} \approx \mathbf{0.38}$$

$$speedup(n) = \frac{T_{seq}}{T_{par}(n)} = \frac{477588078sec}{104787436sec} \approx \mathbf{4.5}$$

Also in this case, efficiency and speedup are small. Even if the CPU frequency is higher and, then, the same results are computed in a smaller time, the improvement in terms of efficiency and speedup hasn't changed a lot.

4 Analysis and possible improvements

Ideally it could be expected a much higher speedup, with 256 Threads reaching a speedup of 56 is not a remarkable result. First of all, some comments about the results provided. The fastflow farm implementation hasn't been taken into account since the Fastflow parallel for is implemented on the top of the farm building block.The two implementation are almost equal from the architectural point of view, but the farm implementation is implemented from "scratch" so it lacks of many optimizations that are present in the Parallel for, hence it achieves slightly worse results.

4.1 False sharing

The memory system inside the machines is hierarchical, there are different levels of memories which come closer to the CPU following the principle of locality. Cache memories are very close to the CPU and, as can be noticed in the machines above, they are internally structured in different levels. The most used part of the data is kept close to the CPU by being copied in the cache memories. In this way, the CPU doesn't need to access central memory and can achieve a nice speedup. When the data is modified coherency has to be conserved. There's a mechanism called cache coherence protocol which is in charge of keeping the data coherent. If the data is modified by many cores (so it's copied in the caches) any of this modification has to be reflected on the other caches. If this kind of operations occurs very often the speedup achieved by the program decays very easily. In the case analyzed above, the grid is shared between all of the cores and it's kept in the same data structure. So it' plausible that the cache coherency protocol is one of the responsible of the very low speedup.

4.2 Arrays instead of vectors

The choice of std::vectors was made because, in the idea of a general Cellular Automata, it could grant states to be whatever, since they allow generic types. There could be examples where states are char or float, and whatever rule could be defined. In this case, it must be said, the arrays would have been a much better choice: they require less memory and less space, the access requires a constant time and contiguous elements are kept contiguous also in memory, plus in this type of problem there's no need to resize the vector.