PDS Report - Parallel Cellular Automata

Gerardo Zinno

Università di Pisa (Artificial Intelligence); g.zinno@studenti.unipi.it

1. Introduction

The objective of this report is to illustrate and analyze the framework for building Cellular Automata [1] on a *toroidal* grid that I developed as a project for the exam of Parallel and Distributed Systems.

Briefly a Cellular Automaton (CA) is defined as a spatially distributed dynamical system where both time and space are discrete. A CA model consists of identical [1] automata (cells) uniformly arranged on lattice points in a D-dimensional discrete space. Each automaton is a dynamical variable, and its temporal change is given by $S_{t+1}(x) = F(S_t(x), N_t(x))$, where $S_t(x)$ is the state of an automaton located at x at time t , F is the state-transition function, and N_t is the neighborhood of x at time t.

2. Problem description and analysis

The problem to solve consists in parallelizing the computation of the new state of each cell of the automaton as a function of the cell itself and its eight neighbors. From this description is clear that this is a data-parallelism problem that can be resolved with an Iterative Stencil Loop (ISL). To solve this problem we need a buffer that will hold the new results and then swap these results at each iteration of the simulation[2]. In the case of a sequential computation, the completion time T_{seq} of a sequential simulation on a grid G consisting of n iterations can be formularized as

$$T_{seq} = T_{alloc} + n * [|G| * \ (T_{getNeigborhood} + T_{updateFun}) + T_{updateState}]$$

.

In this formula T_{alloc} is the time needed to allocate the new space, $T_{getNeigborhood}$ is the time needed to retrieve the neighbors of a cell, $T_{updateFun}$ is the time needed to compute the new value given the neighbors and the current value and $T_{updateState}$ is the time needed to update the state at the end of each iteration (swap buffers and update counters). From now on I define T_{fun} to be $T_{getNeigborhood} + T_{updateFun}$.

To parallelize this problem we need to split the grid according to the number of workers and assign the job to compute the next state of a subset of it to each worker; then we have to wait for each one of them to have completed their task, in order to be able to update the state and swap the buffer before proceeding with the next iteration.

The completion time of the parallelized version using nw workers can be expressed as:

1 New simulations have been made with CA having non-identical cells and this framework can be used to implement them [2].

2 I chose to keep the control simple and use an Alternative Buffer instead of a Temporal one

$$T_{par(nw)} = T_{alloc} + T_{createThreads(nw)} + T_{split(nw)} + n * [\frac{|G|}{nw} * T_{fun} + T_{sync(nw)} + T_{updateState}]$$

$T_{sync(nw)}$ is the time needed to synchronize the threads before being able to proceed with the next iteration. It is equal to the time interval between the fastest thread to reach the barrier and the slowest one.

If we define $T_{computeState} = |G| * T_{fun}$
the maximum speedup $sp(nw)$ that we can achieve is given by:

$$\frac{T_{alloc} + n * [T_{computeState} + T_{updateState}]}{T_{alloc} + T_{createThreads(nw)} + T_{split(nw)} + n * [\frac{T_{computeState}}{nw} + T_{sync(nw)} + T_{updateState}]}$$

Practically speaking we can consider negligible $T_{updateState}$ since it consists of a variable increment and a pointer swap. The same thing can be said of $T_{split(nw)}$.

Now, depending on the size of the grid and the number of iterations of the simulation, one could also consider negligible the time needed to allocate the buffer and the time needed to create the threads, but from the above formula we notice that we have, at the denominator, the overhead of $n * T_{sync(nw)}$. This is an overhead that we have to pay at each iteration of the simulation and it is the main factor that will probably drive the speedup away from the ideal one as the number of threads increases.

3. Framework Structure and API

The framework structure is arranged in a hierarchy of namespaces. The namespace `ca` is the main namespace of the framework and encloses four other namespaces, `seq, par, ffl, omp` containing respectively the implementation of the Sequential, Parallel (C++ threads), Parallel with FastFlow and Parallel with OpenMP versions. All the versions share the same interface so in order to try a different implementation the only thing that needs to be done is to change the namespace. The API of the framework is described in the following section. The documentation of each class and function can be obtained by running `doxygen` in the main folder of the project.

Application Programming Interface

To use the framework you need to 1) create the grid containing the simulation; 2) define the update function; 3) create a cellular automaton and pass to it the grid, the function and, for the parallel versions, the number of workers. 4) run some steps of the simulation The following pseudo-code illustrares how simple is to use the framework. A more detailed description of the API will follow.

```
#include<cellular_automata.hpp>

int main() {
    ca::Grid<int> grid(100,100);

    // init the grid
    for(i,j in 0..99){
        grid(i,j) = something;
    }

    // define update function with this signature
    auto update_fn = [](int c, int tl, int t, int tr, int l, int r, int bl, int b, int br) {
        do something;
    };

    // if the namespace is the one of a parallel implementation you can pass it
    // the number of workers too (harwdare_concurrency() is the default).
    ca::(seq,par,omp,ffl)::CellulatAutomaton<int> aut(grid, update_fn);
    // run 10 simulation steps
    aut.simulate(10);

    std::cout << grid;

}
```

1. Grid creation: The grid is collocated under the main namespace of the framework. It is a class template, this allows to implement more sophisticated automata like the one specified in [2] or implement multi-dimensional automata (e.g. by letting `T` be an array). To create the grid you need to chose the type of the cell and pass to the constructor the number of rows and columns. Its elements can be read or set using the overloaded call operator (`operator()`). It takes as arguments the row and the column of the cell to access.
2. Function definition: The signature of the function for an automaton of type `T` must be `std::function<T(T c, T tl, T t, T tr, T l, T r, T bl, T b, T br)>`. This function has nine arguments, the first is the central cell and the others are it's eight neighbors. The order of the neighbors, relative to the central cell is: Top Left, Top, Top Right, Left, Right, Bottom Left, Bottom, Bottom Right.
3. Automaton Creation: There are four automata, each in its namespace, sharing the same interface. The sequential automaton, located under the namespace `ca::seq`, takes two arguments, the grid and the updated function. The other three automata, in `ca::par`, `ca::ffl`, `ca::omp`,take also a fourth argument, the number of workers. This argument has a default value of 0 which will set the number of workers to `hardware_concurrency`.
4. Simulation: To run the simulation call the `simulate` method of the automaton passing it the number of iterations to run.

Implementation

Of the two techniques presented during the lessons, Alternative Buffer and Temporal Buffer, I chose the former one, in the form of a `vector(T)` of size rows x columns. This allowed me to keep the logic in the main loop of the iterations very simple. This choice is also motivated by the fact that cellular automata are usually used to simulate events on grids of relatively small dimensions since the result is intended to be interpreted by an human. In addition, for the cost of allocating the Alternative Buffer we gain simplicity in the state update phase at the end of each iteration, in which we simply have to swap two vectors.

Sequential version

The sequential implementation allocates the new grid and then starts the iterations of the simulation. In each iteration there are two nested loops to iterate over all the cells of the old grid and compute the value to write in the new grid. After the loops there is a grid swap and a counter update.

```
allocate new_grid;
while (steps > 0)
        {
            for (row in 0..nrows)
            {
                for (col in 0..ncols)
                {
                    new_grid(row, col) = F(grid(row, col), neighbors(grid(row,col)));
                }
            }
            grid.swap(new_grid);
            ++generation;
            --steps;
        }
}
```

OpenMP

To solve the problem using OpenMP I simply added the following line of code above the outermost loop in the pseudocode shown in the sequential section:

```
#pragma omp parallel for collapse(2) num_threads(nw)
```

FastFlow

To solve the problem using FastFlow I used the `ParallelFor` high-level pattern with a minimal change to the sequential code comparable to the one of OpenMP.

C++ <threads>

To solve the problem using the standard `C++` library I implemented a threadpool and a threads synchronization barrier. The use of a threadpool helps reducing the overhead when the program implemented by the user interleaves simulation steps to grid analysis steps. My code is logically equivalent to a parallel for. I split the number of rows in a number of ranges equal to the number of workers, then I create the tasks and pass them to the threadpool. Each thread will use the barrier to wait for others, and only one of them will update the state before proceeding with the next iteration.

```
sync_point(nw); // the barrier

auto step_advancement_fun = [&]() {
        swap and update;
};

// function to be run by each thread
auto work = [&, this](start,end) {
    while (steps > 0)
    {
        for (row in 0..nrows)
        {
            for (col in 0..ncols)
            {
                new_grid(row, col) = F(grid(row, col), neighbors(grid(row,col)));
            }
        }
        sync_point.wait(step_advancement_fun); // wait for other workers.
    }
};
for (i in 0..nworkers)
{
    compute start and end for worker i;
    pool.submit(work, start_i, end_i);
}
```

Threadpool

The threadpool is a work-stealing threadpool. Each worker will first try to pop from its own queue, then from the threadpool global queue and then from other workers' queues. To improve cache locality, each thread will submit a work in its `thread_local` queue if available.

The signature of the submit function is the following one:

```
template <class F, class... Args>
auto submit(F &&f, Args &&...args) -> std::future<typename std::result_of<F(Args...)>::type>
```

It is a template function that allows to submit a function with any kind of signature to it and its arguments. In addition, thanks to the use of the `packaged tasks` we split the execution of the function from its the return value. This allows for a great flexibility since we can submit a function and wait for the return value or its completion only when and if we need it.

Barrier

The barrier is implemented using a counter, a mutex and a condition variable. It offers also the possibility to busy wait by continuously checking the condition instead of using the condition variable. In addition to the classic `wait` method, it exposes a `wait(function<void()>)`. Using this latter method, the last thread to reach the barrier will call the function before resetting the barrier status and waking the waiting threads. This allows to atomically synchronize the state at the end of each iteration.

Code

The code can be found on this repository along with detailed instructions: https://github.com/gerzin/parallel-cellular-automata. In each folder there is a `README` containing a description of its content.

The directories `include` and src contain the implementation of the framework. Under `tests` there are the unit tests (developed using the Catch2 C++ framework) and the benchmarks.

To compile the code you need to use CMake.

```
mkdir build
cd build
cmake .. -DCMAKE_BUILD_TYPE=Release
make
```

this will create a static library containing the compiled code needed to compile programs using the parallel version. The sequential one and the parallel versions using OpenMP and FastFlow can be used as header-only libraries.

To reproduce the unit tests and the benchmarks just go in the restepcive folders, and use CMake and make to compile them as already mentioned above. In order to compile the benchmarks you first need to get and compile the Google benchmark framework and put it under `tests/ext`. All this can be done automatically by "cd-ing" into the `tests/scripts` directory and from there run the `get_benchmark_framework.sh`.

4. Benchmarks

To develop the benchmarks I used the Google benchmark framework https://github.com/google/benchmark, a library to benchmark code snippets which offers some nice utilities for argument passing, automatic inference of the number of iterations needed to stabilize the result and preventing that a section of code you want to time gets optimized out by the compiler.

settings

I run the benchmarks on the remote machine made available to us, which from now on I'll refere to as `Xeon` and my personal machine using an AMD Ryzen 7 5800H, 16gb of ram and running Ubuntu 21.04, which from now on I'll refer to as `Ryzen`. In all the plots I indicete with *Parallel* the parellel version using the C++ standard library and with *ParallelBW* the same version with the busy wait at the synchronization barrier. The function used to run the simulations is a simple function consisting in a sum of it's arguments and a few if statements[3].

In the figures 1 and 2 is shown the completion time in nanoseconds of a simulation consisting of 10 iterations on square grids of various sizes, whose side length is reported on the x-axis, with the default number of workers, set using the std::thread::hardware_concurrency returning an hint of the number of concurrent threads supported by the implementation. 256 for the Xeon machine and 16 for the Ryzen. The y-axis is plotted using a logarithmic scale.

For the grid with small size, the overhead in which the parallel versions incur is not negligible, and in this case the sequential version has been faster than the Parallel and Fastflow versions, but only on the Xeon machine. As the size of the problem grows, the overhead of setting up the parallel computation becomes negligible compared to the rest of the computation to perform and, remembering that the scale used to plot is logarithmic, we can notice how the parallel versions are up to two orders of magnitude faster. Always on the Xeon the Fastflow version has been a little bit faster than the Parallel ones, while on the Ryzen machine the results are virtually identical among all the parallel versions. The version using OpenMP is the one who required less effort to write and the one that produced better results, also on the smallest grid.

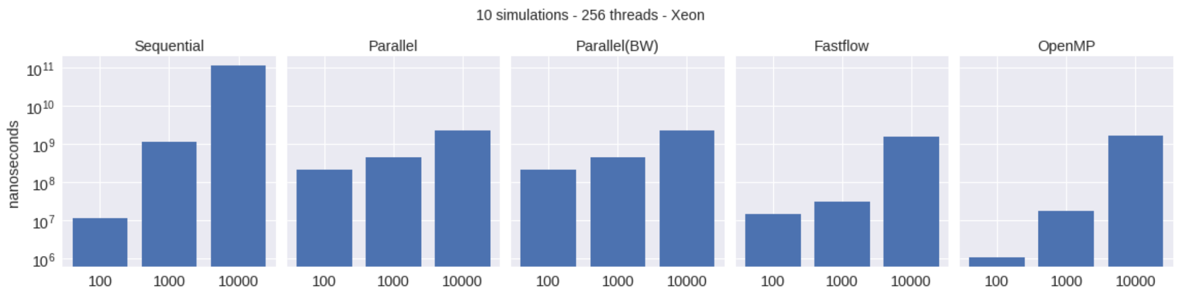

Figure 1

[3] The function implements the rules of the Game of Life

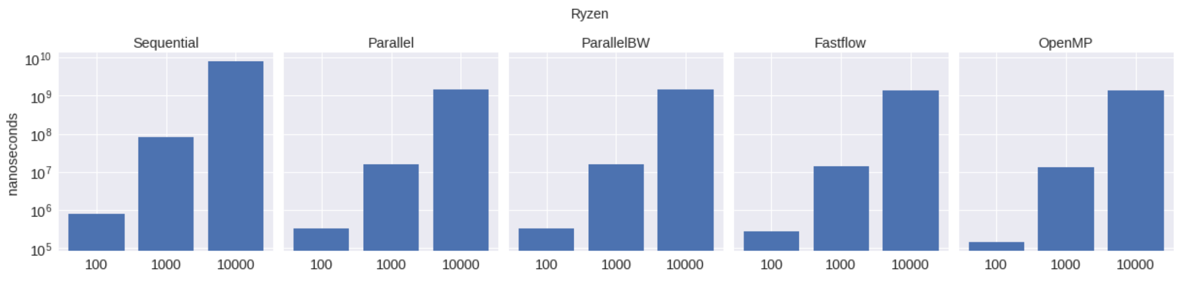

Figure 2

The figures 3 and 4 show the completion time of ten iterations of the simulation on a square grid of side 1000, varying the number of workers. The first thing to notice is that we are gaining performance up to a relatively higher number of threads, even on problems of medium-small size. On the Ryzen machine the behaviour is consistent across all the versions. The completion time decreases until we reach nine workers. On the Xeon machine, after the number of workers reaches 16 we can see a considerable gain in performance when using the busy wait mechanism in the Parallel(BW) version, compared to the one using the condition variables. Again the fastflow version is a bit faster.

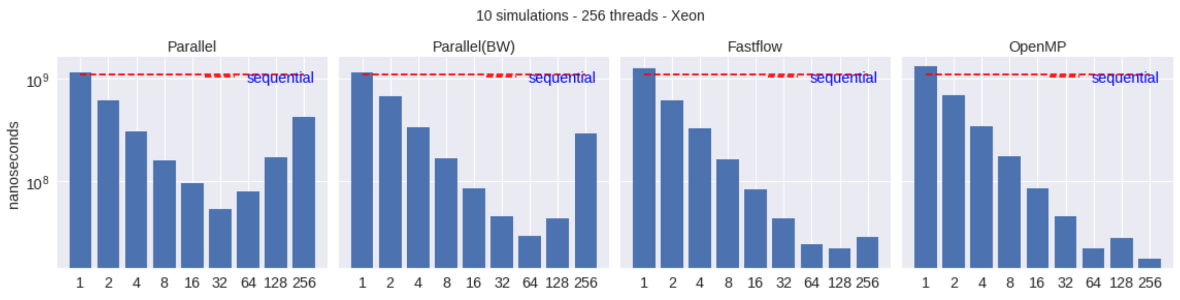

Figure 3

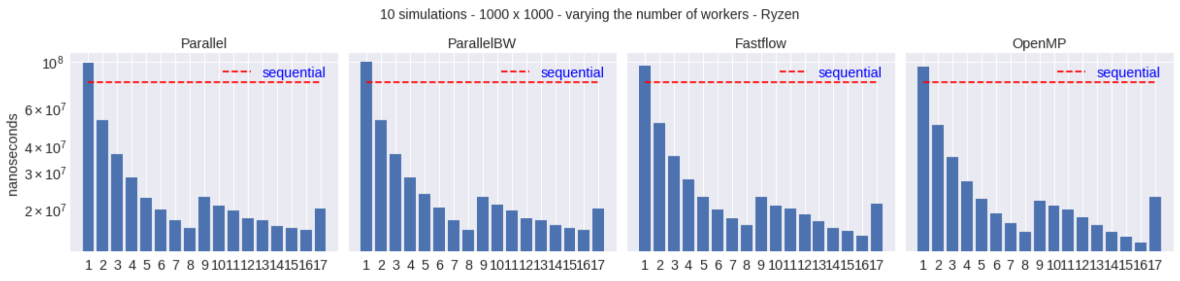

Figure 4

Finally the figures 5 and 7 show the speedup achieved. On the Xeon machine, with up to 4 threads the speedup is close to the ideal one.

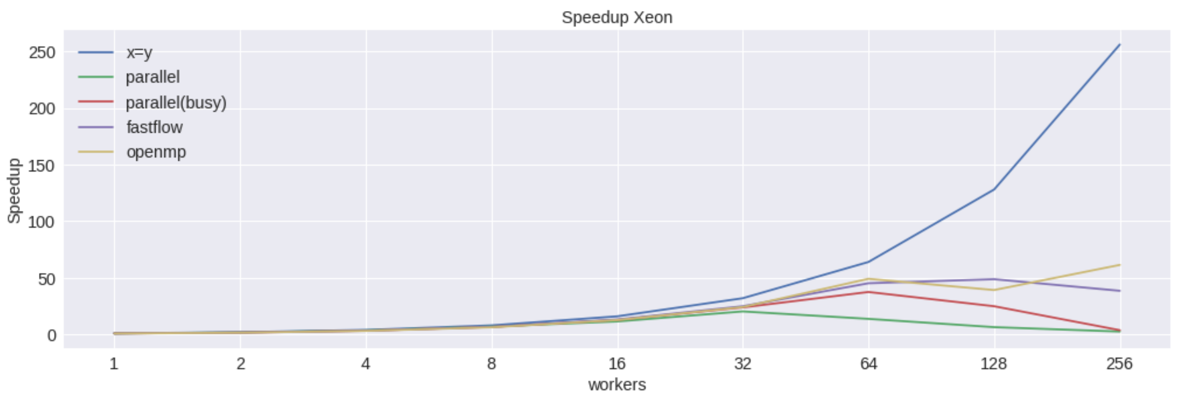

Figure 5

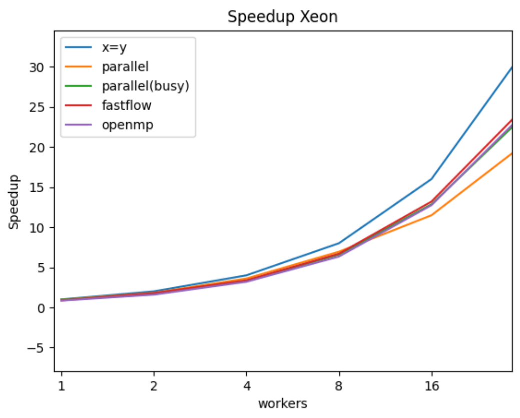

Figure 6

On the Ryzen machine we can see that the speedup is almost identical for all the versions and never really close to the ideal speed up, but this was expected.

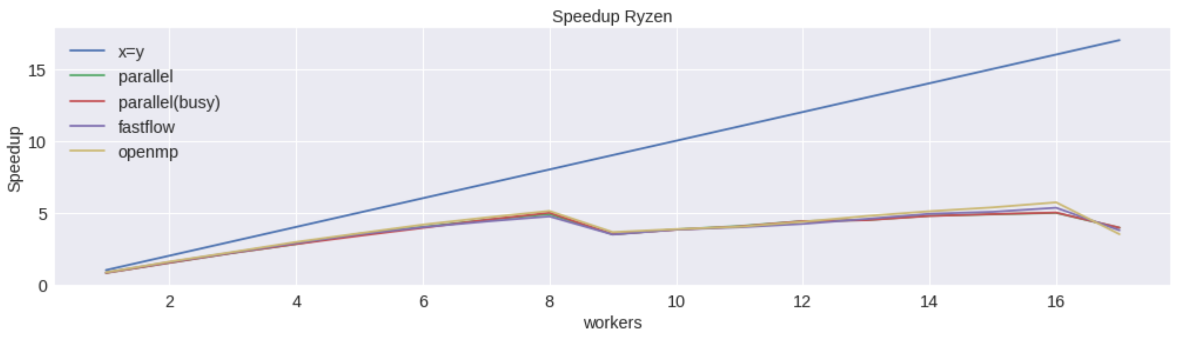

Figure 7

The results are in line with what said in the section 2. The higher the number of threads, the more we are driven away from the ideal speedup due to the cost of synchronizing the workers.

1. Wikipedia contributors. Cellular automaton — Wikipedia, The Free Encyclopedia, 2022. [Online; accessed 12-January-2022].
2. Gerlee, P.; Anderson, A. An evolutionary hybrid cellular automaton model of solid tumour growth. *Journal of Theoretical Biology* **2007**, *246*, 583–603. doi:https://doi.org/10.1016/j.jtbi.2007.01.027.