MPHYG002: Research Computing with C++

• Coursework 2 - Conway's Game of Life •

Sebastiano Ferraris SN: 14108168 s.ferraris@ucl.ac.uk August 15, 2016

Introducing the Game

Conway's Game of Life is a deterministic evolutionary discrete event dynamical system played on an infinite 2-dimensional grid. At each position of the gird there is a cell, that can be in two states: *alive* 1, or *dead* 0. The state of cell C at time t, indicated with $\text{state}_t(C)$ evolves over a discrete time according to its degree. The degree of a cell, $\deg_t(C)$ is the number of alive cells into its eight neigbours, and the evolution model from the state t to the state $t+1$ happens according to the following rules, called update rule, applied simultaneously for all the cells in the grid for a given time-state:

1. $\text{state}_t(C) = 1$ and $\deg_t(C) < 2$ then $\text{state}_{t+1}(C) = 0$.
2. $\text{state}_t(C) = 1$ and $2 \leq \deg_t(C) \leq 3$ then $\text{state}_{t+1}(C) = 1$.
3. $\text{state}_t(C) = 1$ and $\deg_t(C) > 3$ then $\text{state}_{t+1}(C) = 0$.
4. $\text{state}_t(C) = 0$ and $\deg_t(C) = 3$ then $\text{state}_{t+1}(C) = 1$.

The initial state is called *seed*, and it completely determines the subsequent steps of the game.

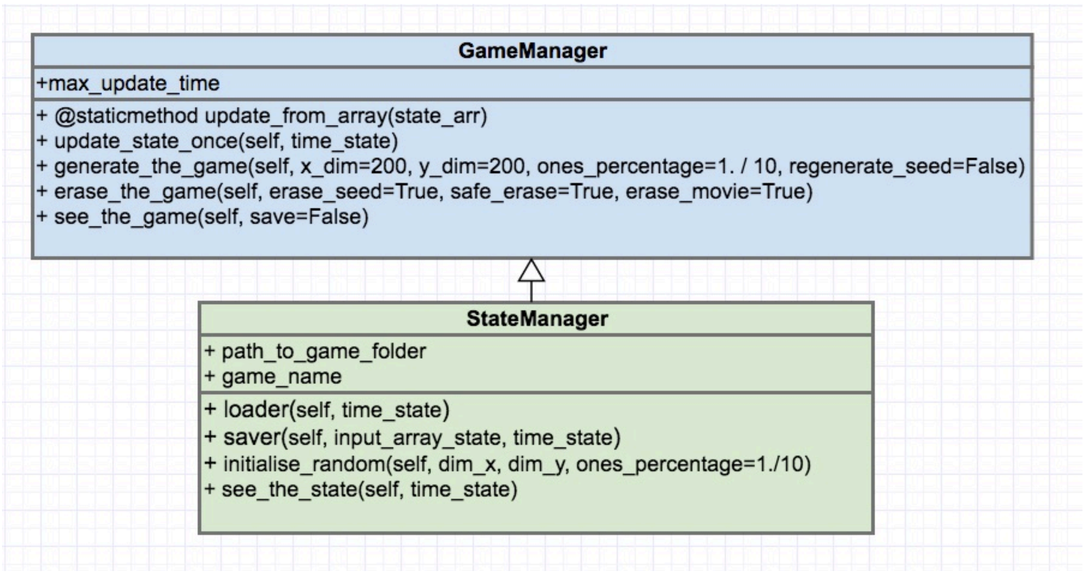
GameManager
+max_update_time
+ @staticmethod update_from_array(state_arr)
+ update_state_once(self, time_state)
+ generate_the_game(self, x_dim=200, y_dim=200, ones_percentage=1. / 10, regenerate_seed=False)
+ erase_the_game(self, erase_seed=True, safe_erase=True, erase_movie=True)
+ see_the_game(self, save=False)
StateManager
+ path_to_game_folder
+ game_name
+ loader(self, time_state)
+ saver(self, input_array_state, time_state)
+ initialise_random(self, dim_x, dim_y, ones_percentage=1./10)
+ see_the_state(self, time_state)

Figure 1: UML diagram representing the structure proposed for the StateManager and the GameManager classes implemented in python.

Part 1: Serial Solution

In the simple implementation here proposed, a *state* is represented by a *.txt* file that contains a binary matrix whose zeros represents the dead cells and ones represents the living cells. A *game*, in the perspective here proposed, is a sequence of *.txt* file where in the filename appears the name of the game, shared for each state, and an integer time parameter, starting from 0 for the seed. Two states with consecutive indexes are related to each others by the update rule, and the finite grid is considered with a toroidal equivalence relation on the edges, to make it if not periodic, at least infinite. Given the need of plotting computing statistics for computational comparisons purposes and of obtaining an animation of a game (see the README.md on github repository), I opted for wrapping the C++ code inside python with boost::python, and to call the shared object (.so) generated with the C++ compiler as libraries inside python code.

Design and code structure

The object oriented structure is driven by the definition of state and game. To reduce computational costs and the length of the code, there is no class implemented for a state or a game, being structured .txt files. The classes are implemented for the operations between states, as well as to create random seeds, visualize states and games.

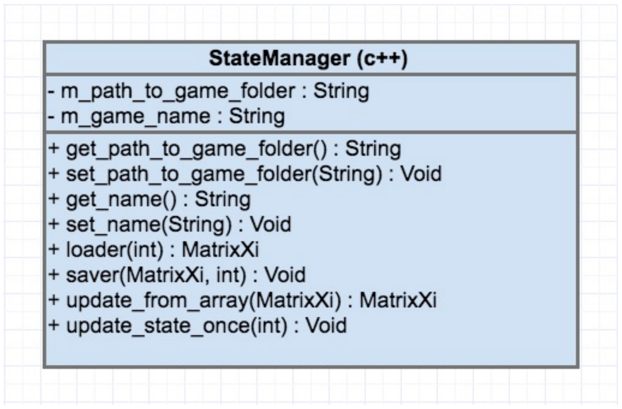
StateManager (c++)
- m_path_to_game_folder : String
- m_game_name : String
+ get_path_to_game_folder() : String
+ set_path_to_game_folder(String) : Void
+ get_name() : String
+ set_name(String) : Void
+ loader(int) : MatrixXi
+ saver(MatrixXi, int) : Void
+ update_from_array(MatrixXi) : MatrixXi
+ update_state_once(int) : Void

Figure 2: UML diagram representing the structure proposed for the StateManager and the GameManager classes implemented in C++.

The folder structure is organized as follows:

examples - contains some hello-world style examples and some code to create games of life, from random seeds and from seeds created ad hoc.

report contains this report and the figures produced with statistical methods.

src is divided into:

- **core** where the core libraries having the classes, in python and C++ are stored.
- **utils** where auxiliaries functions and visualisation methods are stored.

stats contains the methods that produced the statistical results.

test contains the tests, performed with the library Nosetest. The C++ code is tested here as well, in its main functions.

Part 2: Parallel computation - OpenMP

A simple code with a parallelised version of the C++ StateManager is implemented and its performance is compared with the non parallel version. The main for-cycle in the code, for loading the *.txt* files and for update the states are parallelized with a simple OMP environment

```
#pragma omp parallel for
```

If OpenMP is not available to the compiler then the parallelisation is ignored, thanks to the following workaround proposed in the code:

```
#if defined(ENABLE_OPENMP)
  #include <omp.h>
#else
  typedef int omp_int_t;
  inline omp_int_t omp_get_thread_num() { return 0;}
  inline omp_int_t omp_get_max_threads() { return 1;}
#endif
```

Part 3: Remote computation

To run a python script of the project on a cluster (for example *play_a_game_cpp_omp.py*) we propose to copy the code on the cluster with the command *scp*, and then to type

```
nohup ./MasterRun.sh &
```

This command run a script that calls the *qsub* as in the following code:

```
qsub -l h_rt=49:00:00 -l tmem=15G -l h_vmem=15G  -cwd  -v FOLDER=path-to-data-folder
path-to-script-folder/play.sh;
```

In this way the *qsub* command has its own script that includes the chosen parameters and the path to the data folder in the cluster. The bash script *play.sh* has finally the command for the cluster with the python code:

```
#!/bin/bash
/home/path_to_python/python/bin/python2.7 path-to-examples/play_a_game_cpp_omp.py ${FOLDER}
```

And to include the path with the data as an input parameter, we have to add the following line in the file *play_a_game_cpp_omp.py*

```
path_to_game_folder = sys.argv[1]
```

Conclusions

In an experiment, we compared the computational time of a pure python implementation, with the one implemented with C++ and the one parallelised with OpenMP. Results show that the serial C++ version is slower than the python implementation for board larger than 45×45, while the parallelised is the fastest implementation for any of the chosen board dimension. The discrepancy between the python and the serial C++ can be explained by the fact that the *loader* method to store each state in .txt in a numpy.ndarray or in an Eigen::MatrixXi is optimised for python, while it is not for the case of the serial C++.

A fair computational complexity analysis that is not biased by this problem, needs to have methods in C++ that can have numpy.ndarray as input, or vice versa, methods in python that can work with Eigen C++ library matrix as input. An option is given by the library eigency of Cython, that provides an interface between these two data structure. And in general a comparison between Cython version of game of life an the optimised C++ library can be a good option for future work.

Please see the link include in the README.md on the github repository for references.

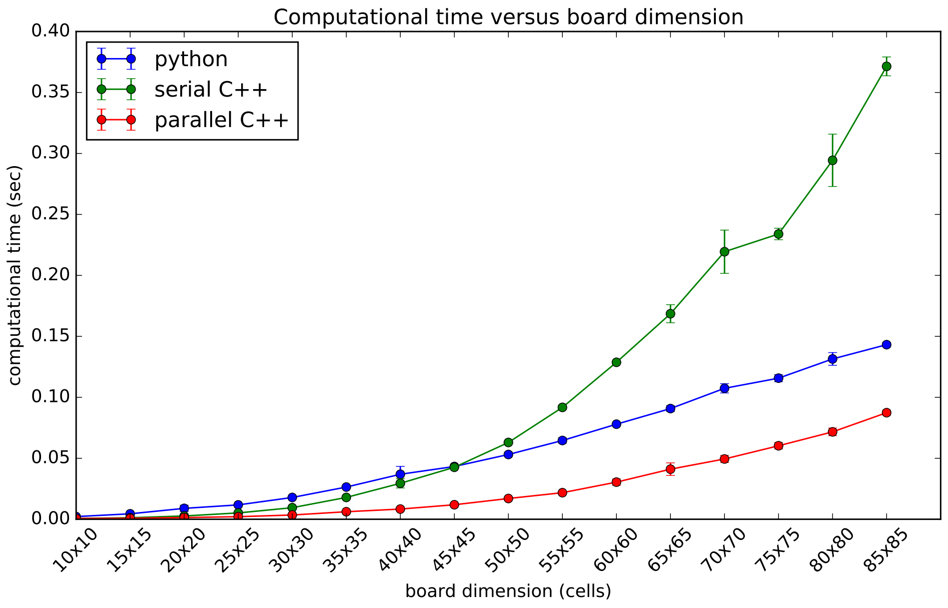

Figure 3: Comparison in computational time for an increasing grid. The computational time is the one computed to run a full game, and the mean is computed for a random sample of 10 initial seeds. Each game is computed for 30 time points.