CellAutoCpp

an efficient Cellular Automaton library implemented in C++

Authors:

Haomin Long (hl3107)

Tongfei Guo (tg2616)

Xiaotian Hu (xh2332)

Introduction

A cellular automaton is a widely used computation model in fields of Mathematics, Physics, Biology, Computer Science and Complexity Science, and the concept of which predicates on the underlying progression can be modelled as a finite number of discrete grid cells whose states are changed constantly at discrete timestamps according to some fixed rule. Some of the commonly known models that could be constructed with cellular automata are Conway's Game of Life model, sheep-wolves predation model from Mathematics, and Ising model from Physics.

Our goal of this project is to implement a computationally efficient library which allows user to create a cellular automaton with specifying only the updating rules, to control the execution of the models with a set of provided APIs, and to retrieve or visualize the result of the execution.

Previous work

The are some similar libraries that have been implemented in higher level languages such as javascript and Haskell, some of the notable projects are Cellauto.js and terra.js. Though they are all well-done and well-received libraries, the functionality of them are limited, and performance of javascript running on node.js v8 (the fastest JIT compiler of js) is still outperformed by C++.

Our work

Our project was initially adapted from cellauto.js, which provides basic functionality of a cellular automaton, and gradually improved in terms of functionality and performance.

i. Functionality

Except for the basic functionality of the execution of cellular automata, some other nice-to-have functionalities that were implemented are:

1. Measurement

Addable feature that provides real-time statistics on the quantity of each type of cell in each of the state as illustrated below: (it should be turned off if performance is the primary concern)

```
=========my metric=========
Epoch 0:
        Cell Type = "cell":
                State Name = "alive" :
```

```
                ("0" = 181,718)
                ("1" =   68,282)

=========my metric=========
Epoch 100:
        Cell Type = “cell”:
                  State Name = “alive” :
                ("0" = 200,096)
                ("1" =   49,904)
```

2. Visualization

Two flavor of visualization methods are provided in this library, which is described below: (they should be turned off if performance is the primary concern)

(i). Addable feature that allows animated plotting of grid with user defined palette (i.e. user can define what color is associated with what cell type in what state). External dependency is needed for this animated color plotting.

Click on this link to view some of the animated plotting:

https://github.com/Tongfei-Guo/CellAutoCpp

(ii). Gird of size less than 60 by 60 could be plotted on the console with less visual effect. This method does not require any external dependency.

3. Timestamp

Addable feature that allows user to specify a fixed number or unlimited of timestamps to store. User can choose the buffersize when creating a model which subsequently used to create a world, and a optional parameter “buffersize” could be pass when initializing the model to specify how many timestamps user want to keep. Keeping only current state incurs no extra overhead, while keeping a fixed number of timestamps or unlimited timestamps incur some overhead.

4. Model combining

Allowing user to combine grid results from two instances of model (put some grid on the top of the other) and continue to progress, which can be used in a recursive manner. For instance, when we are modelling the formation of a cave, we might want to model the top half with less pressure and the bottom half with more pressure (or with even more layers if preferred), and after the formation of the caves, then combine them all together, and model the formation of water within the caves.

5. State snapshot

Allowing user to save current state to a file and later restore current state by loading from a file.

6. Pre-defined models

Allowing user to set up a model from a list of our pre-defined models (such as Game of Life, Lava, Ising, etc) by specifying only a few parameters.

7. Flexible updating rule

Allowing user to update a cell by looks at the states of all other cells in the grid instead of just its neighboring cells (which is typically done in other cellular automaton library), and allow updating a single cell at a time in additional to updating all cells at once (which is typically done in other cellular automaton library).

ii. Performance

For cellular automaton, one of the primary concern is performance, while initialization, construction and destruction of the model might take time, they are one-time-cost, while any overhead regarding cell state transition can occur every time a cell is updated and is far more damaging. Therefore, throughout the design of this library, our main focus of optimization is to reduce overhead on *step()* function call. (*step()* function is called to calculate next state from current state), and our performance measurement is benchmarked on when *step()* is called with measurement, visualization and timestamp turned off (as many of the serious users of cellular automata will oftentimes only want the end results of the simulation without auxiliary functionalities such as measurements, visualization or timestamps). Also, we have given thought and made effort to keep our library's memory footprint low.

List of optimizations that we have attempted are described below in chronological order:

A couple of notes regarding the discussion below : (i) all the instances of optimization that will be discussed below are assuming that measurement, visualization and timestamp are turned off, all the specs that might be system or compiler dependent are assume under 64bit machine with GNU g++7 compiler optimization level -O3) (ii) all the performance statistics on running time was measured by four CAVE model on 500*500 grid for 100 steps, repeated for 10 times, and taking the average running time.

1. The major cost of memory comes from storing the grid of cells. Each of the cell stores a virtual table pointer (8 bytes), two **int**s storing the coordinates of cell (4bytes + 4 bytes), a **string** to store type name (32 bytes assuming short string), an **map<string, int>** storing a map from state name to state value (48 bytes + 32 btyes + 4 bytes assume map one entry is stored). We attempted to a static map that maps type name and state name from **string** to **int**, and so instead of storing **string** for type name and state name in each cell, we can store **int** instead. Thus, the memory footprint would be reduced from 132 bytes per cell to 76 bytes per cell, which is almost by half, and the memory saving would be more substantial if there are more entries of state. However, we had to later on discard this memory optimization because it costed a huge overhead by accessing the map every time when a state value is sought, as it would need to map the type name from a string to its corresponding int, and use that int to loop up the state value, and convert back from int to string when returning, which cost the running time to increase by 75%. In favor of running time, we chose not to do this memory optimization.

2. As we are using class inheritance for our **Cell** type, our grid is stored as a vector of vector of **Cell** pointers. Initially, we stored grid as **vector<vector<unique_ptr<Cell>>>** which is natural to do following c++11 standard. Consequently, the initialization of a grid would be initializing a **unique_ptr<Cell>** by **unique_ptr<Cell>(new Cell()) (**or equivalently

make_unique<Cell>()), and **push_back** it into the vector, but the problem with this way of initialization is that in a highly concurrent environment, there will be little or no spatial locality, and thus when running *step()*, it could be observably slowed down by around 98%. After realizing this, for construction of a grid, we decided to pre-allocate a chunk of contiguous memory block using **Cell *p =new Cell[height * width]**, and using placement new (i.e. **new (p+offset) Cell()**) to construct each instance of Cell, and this guarantees spatial locality.

3. As we observed that for many cases of cellular automata, their *reset()* and *process()* functions are highly parallelizable. Each call to *reset()* and *process()* oftentimes only modify the current value of a cell based on the previous states of the whole grid, which means the computation of the current value of a cell is independent of the computation of another cell. Thus, we parallelize the computation of both *reset()* and *process()* into multiples threads (depending on the number of cores you machine has). Running on a quad-core Linux machine, it increase the speed by 118%.

4. Oftentimes, the *reset()* function is not used, for instance in our Ising model simulation, and what's even worse here is that unlike many of the cellular automata that update every cell of the grid at each iteration (and hence can be parallelized as mentioned above), Ising mode updates only one cell at a time, and the ramification of which is that for just updating one cell, we need to call *reset()* on every cell, especially if the *reset()* function has an empty body, it would practically have no effect but just run a empty statement for O(n^2) times, which dramatically slows down Ising model. On side note here, just to give you some insight on how *reset()* works, it's stored as **function<void(Cell*)>** which allows user to initialize it with any type of callable object, so initially, when user do not use *reset()*, it just pass in something like **function<void(Cell*)>([](){})** to initialize *reset()* with empty body. How **function<void(Cell*)>** works is that it will access the stored function through a pointer to the callable object, which causes a layer of indirection, and then it calls the underlying function, and transfer control, which also add overhead, so even if it's just an empty-body function, the overhead of calling it on each cell every time before we updating just one cell is astronomical. Thus, we allow user to initialize *reset()* to some default value if they do not intend to use it, and skip *reset()* step altogether. After doing this, the running time of Ising running 1 million iterations is less 1/1000 as before. Similar optimization is done in a number of places.

5. This is a snippet of code from Cellauto.js project that we adapted from, it defines an array of lambda functions to allow easy computation of a cell's neighboring cells' coordinates

```
var NEIGHBORLOCS = [
        { diffX : function() { return -1; }, diffY: function() { return -1; }},  // top left
        { diffX : function() { return 0; }, diffY: function() { return -1; }},  // top
        { diffX : function() { return 1; }, diffY: function() { return -1; }},  // top right
        { diffX : function() { return -1; }, diffY: function() { return 0; }},  // left
        { diffX : function() { return 1; }, diffY: function() { return 0; }},  // right
        { diffX : function() { return -1; }, diffY: function() { return 1; }},  // bottom left
        { diffX : function() { return 0; }, diffY: function() { return 1; }},  // bottom
        { diffX : function() { return 1; }, diffY: function() { return 1; }}  // bottom right
];
```

A more direction approach for adaption in C++ would be

```
std::vector<std::function<int()>> CAWorld::diffX = std::vector<std::function<int()>>(8),
                                  CAWorld::diffY = std::vector<std::function<int()>>(8);
diffX[0] = [](){return -1; }; diffY[0] = [](){return 1; };// top left
diffX[1] = [](){return 0; }; diffY[1] = [](){return 1; };// top
diffX[2] = [](){return 1; }; diffY[2] = [](){return 1; }; //top right
diffX[3] = [](){return -1; }; diffY[3] = [](){return 0; }; //left
diffX[4] = [](){return 1; }; diffY[4] = [](){return 0; };//right
diffX[5] = [](){return -1; }; diffY[5] = [](){return -1; }; // bottom left
diffX[6] = [](){return 0; }; diffY[6] = [](){return -1; }; // bottom
diffX[7] = [](){return 1; }; diffY[7] = [](){return -1; }; // bottom right
```

Normally, it won't be much of a problem, but if these function get used repeated in *step()* every time a cell get updated, it cause a huge overhead just by indirection from accessing callable through its pointer and control transfer from function call, so it would make much more sense just do define a static vector to do the same job.

```
static std::vector<int> diffX = {-1, 0, 1, -1, 1, 0, -1, 1};
static std::vector<int> diffY = {-1, -1, -1, 0, 0, 1, 1, 1};
```

with minimum change to the interface, and this reduce running time by more than 6%.

6. In an auxiliary function that we provide has the following interface

```
std::vector<Cell*> get_neighbors(const grid_type &grid, int x, int y)
```

and it will take in a grid and a pair of coordinates and return a vector of pointers to its neighboring cells, and how it does this is that it will initialize a **vector<Cell*>**, find all the neighboring cells' addresses, store them in the vector, and return the vector, and this function is called repeatedly each time a cell gets updated. A potential optimization is that we could change the interface to
void get_neighbors(vector<Cell*> &neighbors, const grid_type &grid, int x, int y)
and ask user to pass in a **vector<Cell*>** by reference and return neighboring cells' addresses by modifying that variable, the benefit of this is that this would get rid of the construction and destruction of a **vector<Cell*>** variable, and depending on whether compiler performs copy elision optimization, it could have be down by 2 instances of construction and destruction, and if all these overhead occur every time a cell is updated, it could be quite costly, and it would reduce running time by 2%. However, the would result in a more complex and less natural interface for user to call, and so we decided not to implement it.

7. For the **Cell** class, which now stores **map<string, int>** to store the states and their corresponding values in each cell, it was initialized stored as **unordered_map<string, int>**, but we realized that for a typical cellular automaton, the number of states is less than 5, and there seldomly is one with more than 10 states, while **map** actually outperforms **unordered_map** when the number of keys is trivial, and the number is typically around 10 (the number might vary a little depending on the implementation of compiler). Therefore, we decided to switch off **unordered_map** and use **map** instead. This turns out to boost our performance by 28%.

Further work

1. Allow grid to store timestamps of different size at different locations.

2. Allow template cell type name, state type name, state value, now cell type and state type are restricted to string, and state value is restricted to int.
3. Add the feature to allow storing timestamps at a specified sub-region.
4. Optimize memory footprint on storing timestamp, a potential idea is to store diff from previous timestamp instead of storing the whole timestamp, which could be extremely useful when running Ising where each timestamp is different from previous one at only one cell.
5. Implement hexagonal grid, for now we only allow square grid.
6. Further optimization on execution with add-on features, for now we mainly focus on optimizing overhead on *step()* function call without any add-on features.
7. Work on 3-D implementation of cellular automaton.

Conclusion

Our cellular automaton library provides a fairly complete set of APIs to allow both flexibility and efficiency. Compared to typical cellular automaton libraries like Cellauto.js, our one has more sophisticated features like timestamp, statistic measurement, combining, save&load, etc, and our library gives a better performance, as measured by executing a 500*500 simulation with 100 frames, it is 67% faster than cellauto.js running with node.js v8.

Acknowledgements

We want to thank Prof.Stroustrup for his invaluable instructions throughout the course, course teaching assistants Kai-Zhan Lee and Abhishek Shah for their suggestions on this project, and Cellauto.js for inspiring us to implemented a cellular automaton library in C++.

References

Cellauto.js : https://sanojian.github.io/cellauto/

Cppreference: http://en.cppreference.com/w/