MPHYG002: Research Computing with C++

Parallel Implementation of the Game of Life

Michael Ebner
(Student Number: 15001660)

April 25, 2015

Contents

1 Conways Game of Life

Based on the lecture notes and `http://en.wikipedia.org/wiki/Conways_Game_of_Life`[1] the concept of the game can be understood as a simulation of a population change over time. The rules read as follows:

- The game is represented by an infinite two-dimensional grid of cells.
- Every cell can represent two possible states: dead or alive.
- The grid with its cells represent a population at a discrete point of time.
- The state of each cell in the consecutive time step depends on its 8 adjacent neighbours.
- The transition of the each cell's state are described by these rules (Summary is given in table 1.1):
  - A live cell remains alive in case it is surrounded by either two or three living cells. Otherwise it dies which relates to the cases of under-population or overcrowding respectively.
  - A dead cell with exactly three surrounding living cells becomes alive associated with the idea of reproduction.

		Number of neighbour cells alive								
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Current state	**0**	0	0	0	1	0	0	0	0	0
	1	0	0	1	1	0	0	0	0	0

Table 1.1: Transition rules: "0" indicates a dead and "1" a live cell

[1] Retrieved: April 25, October 2015

2 Implementation

2.1 Structure of project-folder

The project can be downloaded from `https://github.com/renbem/RCCPP-coursework02`. The code is structured in several folders within the main directory:

- `documentation`: Contains all documentation of the project including this report and the code documentation generated by doxygen[1].
- `include`: Contains all header files (`*.h`)
- `matlab`: Contains the MATLAB-files used for the random generation of initial boards (or grid according to the terminology in chapter 1), the creation of the Game of Life video[2] and statistical evaluation of the results.
- `source`: Contains the code files (`*.cc`).
- `test`: Contains all data and source files for the unit tests.

2.2 Choice of design

The implementation follows the design as illustrated in fig. 2.1 which translates the concept of a having a game consisting of boards which in turn consist of the cells.

The classes are designed so that a new iteration is triggered by the member function `computeNextStep` of `Game`, cf. fig. 2.2. I decided to connect the "rules of the game" with the board via the two methods `determineNeighbourCells` and `applyTransitionRules`. One could argue that this might be better situated in the class `Game` since the rules can be considered as an intrinsic property of the game and the board could be of arbitrary shape. However, since the Conway's Game of Life relies on well-established rules in order to have a "meaningful game" (i.e. no extinction or overcrowding of cells) the definition of a neighbour and the propagation rules depend strongly on the shape and therefore on the board.

2.2.1 Parallelisation

The code was parallelised via **OpenMP** within the method `computeNextStep`, cf. line 10 in listing 2.1. The line 9 and 11 was added since I could not compile it without any error messages on Mac (further instructions in section 2.3) but it works on Ubuntu. Therefore, the code runs on non-linux platforms only in the serial version.

Listing 2.1: Game::computeNextStep

```
1 void Game::computeNextStep(){
2     Board currentBoard = boardHistory.back();
```

[1] `www.doxygen.org`
[2] Provided at `https://www.youtube.com/watch?v=AeIm2I_9n5g&feature=youtu.be`

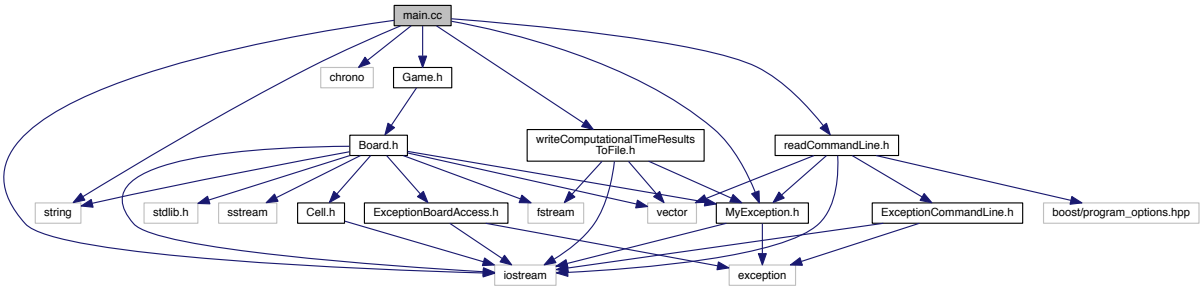

Figure 2.1: Overview of implementation

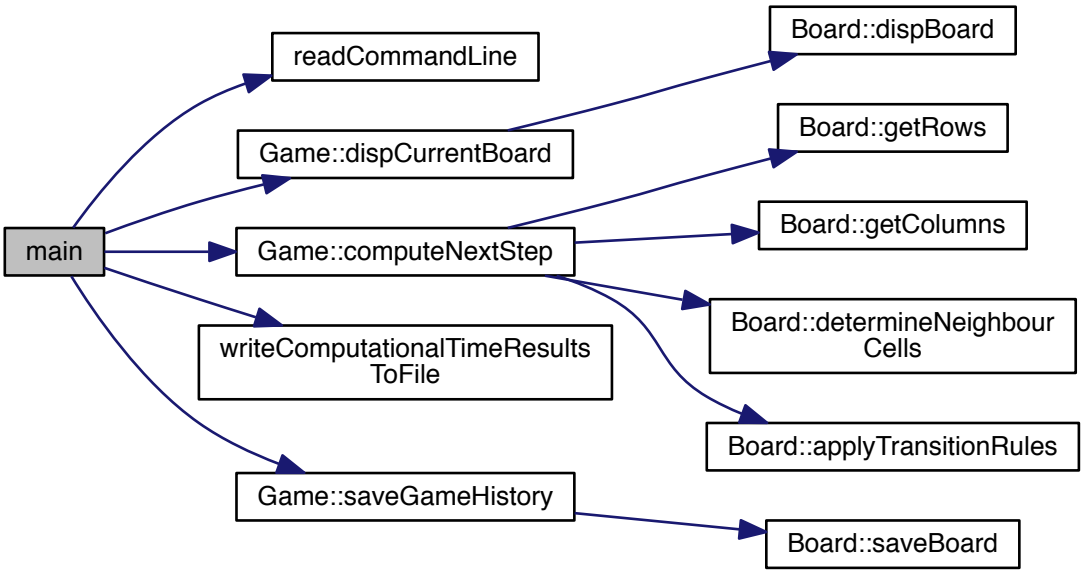

Figure 2.2: Call graph of `main.cc`

```
3       Board updatedBoard = currentBoard;
4
5       unsigned int irows = currentBoard.getRows();
6       unsigned int icolumns = currentBoard.getColumns();
7       std::vector<Cell> neighbourCells;
8
9       #ifdef __linux
10          #pragma omp parallel for private(neighbourCells)
11      #endif
12      for (unsigned int row = 0; row < irows; ++row){
13
14          for (unsigned int col = 0; col < icolumns; ++col){
15              neighbourCells
16                  = currentBoard.determineNeighbourCells(row, col);
17              updatedBoard.applyTransitionRules(row,col,neighbourCells);
18          }
19      }
20      this->pushBoard(updatedBoard);
21  }
```

2.2.2 Unit tests

Unit tests were written to test the following cases:

- Check input file[3].
- Several tests to check malformed command lines.
- Several tests to check whether the correct neighbours are returned by `Board::neighbourCells`. (The "infinite board" was implemented by assuming periodic boundary conditions.)
- Attempted access to non-existing indices of board throws an exception.
- Test whether one step of Conway's Game of Life is computed correctly.
- Test whether the parallel computation via OpenMP returns the same board as the serial version.

The respective file `main_UnitTests.cc` is located in `test/`

2.3 Build, run and test

The respective `CMakeLists.txt` file in the parent directory to build the code was tested on two operating systems:

- OS X 10.10.3 (Macbook Pro, 2.5 GHz Intel Core i7)
- Ubuntu 14.04 (Virtual machine via VirtualBox[4] within OS X 10.10.3)

[3] It is only tested whether the input file exists. Many other cases should be tested in addition. E.g.: Format of file appropriate?, Number of columns always identical?, Minimum number of rows and columns alright? etc.

[4] `https://www.virtualbox.org/`

Unfortunately, I couldn't run OpenMP on Mac without any errors (comments are provided in the `CMakeLists.txt` file). Therefore, I used VirtualBox to build it on Ubuntu where everything went without any problems. Consequently, I configured the CMake file (and added the respective lines in `Game.cc`) so that the code runs on Mac in serial and on Ubuntu with OpenMP. The results in section 2.4 are based on the computation in Ubuntu.

2.3.1 Build instructions

In order to build the code run the following lines in the main directory:

```
mkdir build
cd build
cmake ..
make
```

After building the code two binary files will be generated:

- `conwaysGameOfLife` located in `build/test/bin/`
- `conwaysGameOfLife_UnitTests` located in `build/test/bin/`

2.3.2 Running the programme

For running the code several example data sets are provided in `build/test/exampleData/`. A possible command of the programme within the `build`-folder reads

```
bin/conwaysGameOfLife --i "test/exampleData/InitialBoardRandom_10Times10.txt"
--o "out_BoardHistory.txt" --s 100
```

which uses the initial board defined in `InitialBoardRandom_10Times10.txt` and performs 100 steps by applying the transition rules given in table 1.1. A concatenated history of all computed boards is then stored in `out_BoardHistory.txt` and the information of the corresponding computational time per iteration is saved in `out_BoardHistory_ComputationalTime.txt`. Alternatively any other file `InitialBoardRandom_*.txt` in `test/exampleData` can be used as input file[5].

2.3.3 Run unit tests

To run the unit tests, execute the command

```
./conwaysGameOfLife_UnitTest
```

in `build/test/bin/`.

2.4 Results

[5] A "on-screen-simulation" and its corresponding computational time can also be enabled by setting the respective flags `flagDisplayGame` and `flagDisplayComputationalTime` to `true` within `main.cc`.

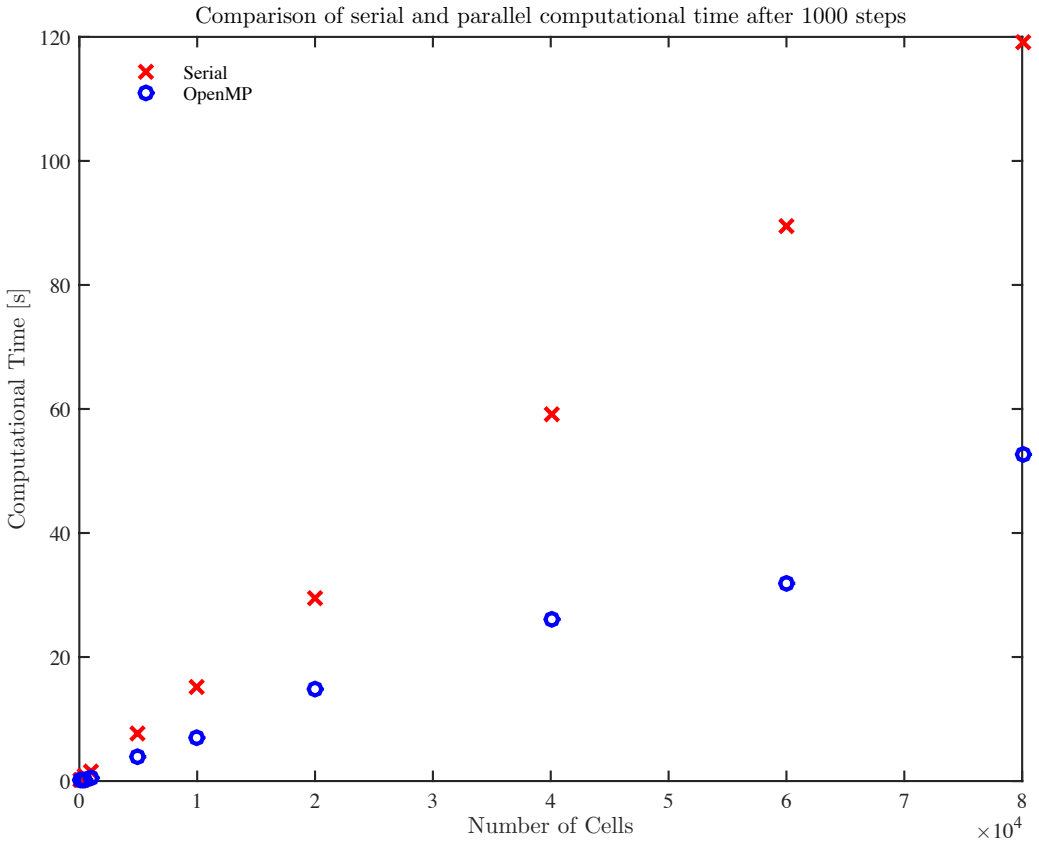

Figure 2.3: Computational time over number of cells

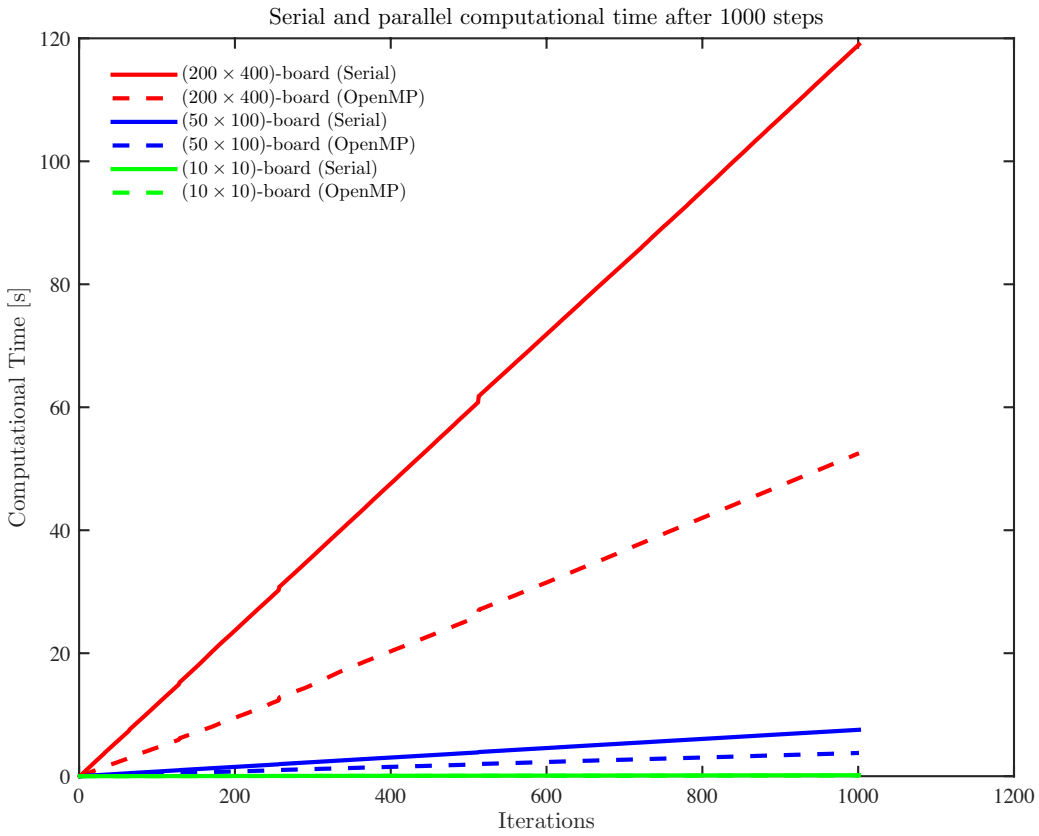

Figure 2.4: Computational time over number of iterations

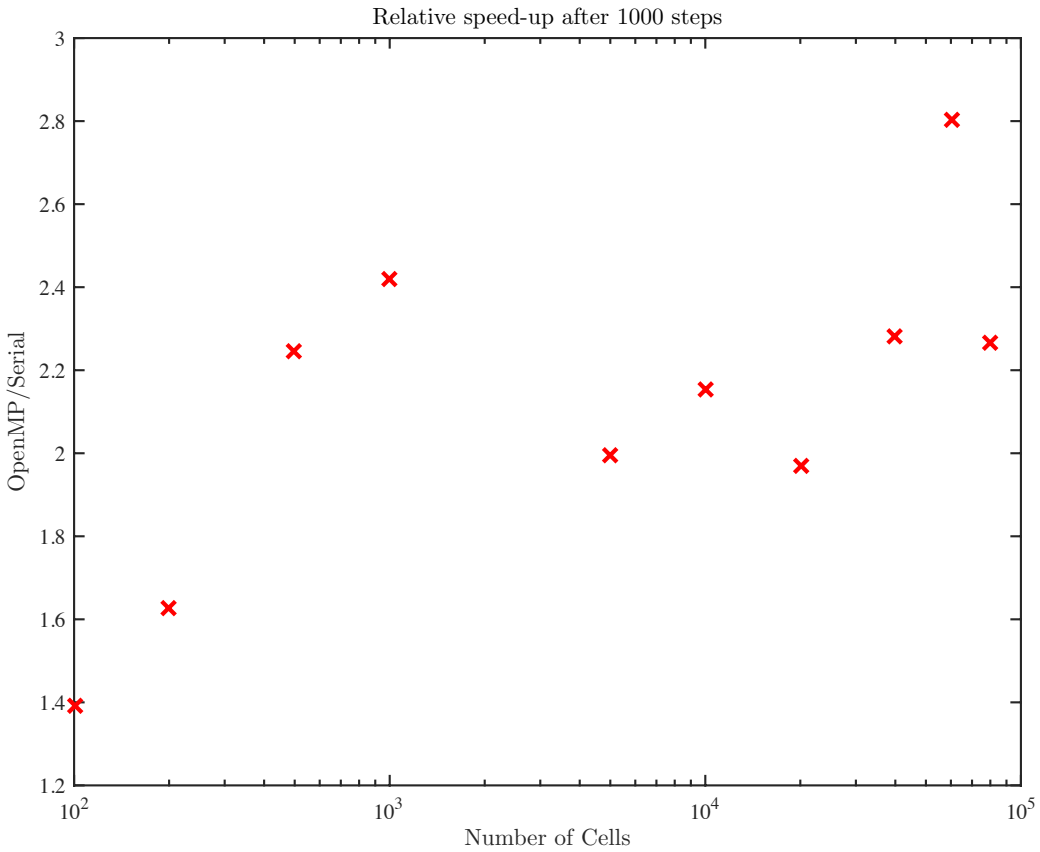

Figure 2.5: Relative speed-up