C++ course – Exercises Set 11

# Wouter Verkerke (Jan 2023)

(Optional) Exercise 11.1 – Parallel Game of Life

*The goal of this exercise is to parallelize a given serial implementation of Conway’s Game of Life using OpenMP.*

Conway's Game of Life is a popular cellular automaton simulation created by mathematician John Horton Conway in 1970. It consists of a grid of cells, each of which can be in one of two states: "alive" or "dead." The state of the cells evolves over time according to a set of rules based on the number of live cells in the cells' immediate vicinity. You can play around with it yourself on playgameoflife.com.

The rules of the cellular automaton are simple:

* For a space that is already populated
	+ Each cell with one or no neighbors dies, as if by solitude
	+ Each cell with four or more neighbors dies, as if by overpopulation
	+ Each cell with two or three neighbors survives
* For a space that is empty or unpopulated
	+ Each cell with three neighbors becomes populated

Step 1 – Copy and understand the serial implementation

* Copy the file ex11.1/ConwaysGame.cpp, this is a serial implementation of Conway’s game of life that outputs a grid in a text file at each timestep. Make sure you understand the code.

Step 2 – Parallelize parts of the code. This code leaves much room for implementing different parallelization strategies. The next bullet points give some suggestions, but feel free to experiment more.

* The initialization of the grid is currently serial. It consists of two nested for loops to initialize every index randomly to 0 or 1 individually. Can you think of a way to parallelize this?
	+ - * Do you need a barrier behind this parallelized section? Do you need to make the operation atomic? These questions are relevant to all subsequent bullet points.
		- In perform\_iteration() the current grid is swapped out for the new grid on every iteration with std::vector::swap(). Rewrite this operation as a parallel operation.
		- In both perform\_iteration() and in count\_neighbours() there are double for loops. The former also calls the latter within its inner loop, so the innermost for loop within count\_neighbours() is nested within three other for loops. Find a suitable strategy to parallelize this section of code and implement it.
			* Which for loop would you parallelize, the innermost for loop (in count\_neighbours()) or the outermost for loop (in perform\_iteration())? Consider the overhead spent in spawning and killing threads.
			* Would it be a good idea to parallelize multiple of these for loops simultaneously?

Step 3 – Improve the write\_to\_grid() function

* Time the write\_to\_grid() function after you have parallelized all of the aforementioned bullet points. Does this function significantly impact the total execution time when you use a single thread, how about when you use more threads?
* Parallelize this function.
	+ - * Note that only a single thread at a time can write to any given file. Do you need synchronization between threads to properly parallelize this function?
			* The << operator that streams to the output file takes a relatively long amount of time and (if implemented correctly) blocks all other threads from writing to the same file at the same time. Think of an implementation that limits the number of times the file write operation occurs.
* Finally, if you really want to go the extra mile, you could also convert the written file to an image in your code. The bitmap file format is a nice solution to this, as it does not require many changes to the raw file beyond the existing serial implementation.