ECE 4122/6122 Lab 2: John Conway's Game of Life

(100 pts)

Category: Multithreading and OpenMP
Due: Saturday October 7th, 2027 by 11:59 PM

Objective:

To understand and apply the principles of multithreading using std::thread and **OpenMP** for parallel calculations in a computationally-intensive problem related playing John Conway's Game of Life.

Game Description:

The Game of Life (an example of a cellular automaton) is played on an infinite two-dimensional rectangular grid of cells. Each cell can be either alive or dead. The status of each cell changes each turn of the game (also called a generation) depending on the statuses of that cell's 8 neighbors. Neighbors of a cell are cells that touch that cell, either horizontal, vertical, or diagonal from that cell.

The initial pattern is the first generation. The second generation evolves from applying the rules simultaneously to every cell on the game board, i.e. births and deaths happen simultaneously. Afterwards, the rules are iteratively applied to create future generations. For each generation of the game, a cell's status in the next generation is determined by a set of rules. These simple rules are as follows:

- If the cell is alive, then it stays alive if it has either 2 or 3 live neighbors
- If the cell is dead, then it springs to life only in the case that it has 3 live neighbors

There are, of course, as many variations to these rules as there are different combinations of numbers to use for determining when cells live or die. Conway tried many of these different variants before settling on these specific rules. Some of these variations cause the populations to quickly die out, and others expand without limit to fill up the entire universe, or some large portion thereof.

Assignment:

1) Write a C++ application that takes up to 5 command line arguments to dynamically change the number of processing threads (>= 2), cell size, the image size and the type of processing. Below is an example

➢ ./Lab2 -c 5 -x 800 -y 600 -t OMP

The flags

-n is the number of threads (must be larger than 2),

-c is used to denote the "cell size" with cells being square (c >=1),

-x is the window width,

-y is the window height

-t is either SEQ, THRD, or OMP where SEQ is sequential, THRD is std::thread, and OMP is OpenMP

The grid size used for calculations and display is the (window size)/(cell size).

If one of the flags above is missing then automatically use the defaults:

-n defaults to 8 (-n value is ignored if the processing type is SEQ)

-c defaults to 5

-x and -y default to 800 by 600 respectively.

-t defaults to THRD

2) Write your code using three functions: one for the sequential processing, one for multithreaded processing using std::thread, and one for OMP processing. Your code will always be compiled with the OMP flag enabled.

3) Your code needs to use a random number generator to initially set the individual grid element to either "alive" or "dead".

4) Your code then runs continuously generating new results until either the window is closed or the "Esc" key is pressed.

5) While your code is running you need to display to a graphics window the current state of the Life game. Cells that are alive are white and dead cells are black. You don't need to draw the dead cells.

6) While your code is running you need to constantly output to the console window the processing time in microseconds of the last 100 generations of the game and the type of processing. Do not include the time it takes to display the results.

For example:

100 generations took ??? microseconds with single thread.

100 generations took ??? microseconds with 8 std::threads.

100 generations took ??? microseconds with 12 OMP threads.

Turn-In Instructions

Place your files in a zip file called **Lab1.zip** and upload this zip file on the assignment section of Canvas.

Grading Rubric:

If a student's program runs correctly and produces the desired output, the student has the potential to get a 100 on his or her homework; however, TA's will look through your code for other elements needed to meet the lab requirements. The table below shows typical deductions that could occur.

AUTOMATIC GRADING POINT DEDUCTIONS PER PROBLEM:

Element	Percentage Deduction	Details
Does Not Compile	40%	Code does not compile on PACE-ICE!
Does Not Match Output	Up to 90%	The code compiles but does not produce correct outputs.
Runtime and efficiency of code setup	Up to 15%	The code generates the correct output but runs slower than expected. score = 15 x (Tmax-T)/(Tmax-Tmin) for 100 game generations.
Clear Self-Documenting Coding Styles	Up to 25%	This can include incorrect indentation, using unclear variable names, unclear/missing comments, or compiling with warnings. (See Appendix A)

LATE POLICY

Element	Percentage Deduction	Details
Late Deduction Function	score – 0.5 * H	H = number of hours (ceiling function) passed deadline

Appendix A: Coding Standards

***Indentation*:**

When using *if/for/while* statements, make sure you indent **4** spaces for the content inside those. Also make sure that you use spaces to make the code more readable.
For example:

```
for (int i; i < 10; i++)
{
    j = j + i;
}
```

If you have nested statements, you should use multiple indentions. Each { should be on its own line (like the *for* loop) If you have *else* or *else if* statements after your *if* statement, they should be on their own line.

```
for (int i; i < 10; i++)
{
   if (i < 5)
   {
       counter++;
       k -= i;
   }
   else
   {
       k +=1;
   }
   j += i;
}
```

Camel Case:

This naming convention has the first letter of the variable be lower case, and the first letter in each new word be capitalized (e.g. firstSecondThird).

This applies for functions and member functions as well!

The main exception to this is class names, where the first letter should also be capitalized.

Variable and Function Names:

Your variable and function names should be clear about what that variable or function represents. Do not use one letter variables, but use abbreviations when it is appropriate (for example: “imag" instead of “imaginary”). The more descriptive your variable and function names are, the more readable your code will be. This is the idea behind self-documenting code.

File Headers:

Every file should have the following header at the top

```
/*
Author: your name
Class: ECE4122 or ECE6122 (section)
Last Date Modified: date

Description:

What is the purpose of this file?

*/
```

Code Comments:

1. Every function must have a comment section describing the purpose of the function, the input and output parameters, the return value (if any).
2. Every class must have a comment section to describe the purpose of the class.
3. Comments need to be placed inside of functions/loops to assist in the understanding of the flow of the code.