School of Computer Science: assessment brief

Module title	Parallel Computation
Module code	XJCO3221
Assignment title	Coursework 1
Assignment type and description	OpenMP Programming Assignment
Rationale	To implement parallel loops on a shared memory CPU system, and to identify and handle data dependencies.
Guidance	See overpage
Weighting	15%
Submission deadline	9:30am, Friday 28th February
Submission method	Gradescope
Feedback provision	Marks and comments returned *via* Gradescope
Learning outcomes assessed	Apply parallel design paradigms to serial algorithms. Evaluate and select appropriate parallel solutions for real world problems.
Module lead	Peter Jimack

1. **Assignment guidance**

   This coursework specification is designed to work on `cloud-hpc1.leeds.ac.uk`. We cannot guarantee it will work on any other environment.

   Conway's Game of Life is a classic computer science problem and is one of the most well-known cellular automata. It consists of a two-dimensional grid in which each grid square can either be occupied by a 'cell,' or be empty. Once the initial configuration of cells is given, they are updated by applying the following rules at every time step:

   - For each grid square, the number of neighbouring squares occupied by a cell, n_{occ}, is counted. This includes grid squares immediately to the left, right, above and below, and also the diagonals, otherwise known as the Moore neighbourhood; see Fig. 1(a).
   - A cell that is occupied 'dies' – that is, becomes empty – on the next time step, if $n_{\text{occ}} < 2$ ('underpopulation') or $n_{\text{occ}} > 3$ ('overpopulation').
   - Otherwise, the cell remains alive in the next time step.
   - Conversely, an empty cell with $n_{\text{occ}} = 3$ becomes 'alive' (occupied by a cell) in the next time step, otherwise it remains empty.

   Code has been provided that implements the Game of Life in serial, and you should start by downloading the code from Minerva and familiarising yourself with it. The files are

   - `cwk1.c` which contains the code relevant to this assessment.
   - `cwk1_extra.c` which contains extra code for display *etc.* You do not need to look at this code, and should not make any changes as it will be replaced with a different version for assessment.
   - `makefile` which works on `cloud-hpc1.leeds.ac.uk`.

   Once built, execute as *e.g.*

   ```
   > ./cwk1 30 500 50 0
   ```

   where:

   - The first command line argument ('`30`') sets the grid size `N`, *i.e.*, grids are `N`×`N`.
   - The second argument ('`500`') specifies the initial number of occupied cells.
   - The third ('`50`') is how many iterations to perform before stopping.
   - The final ('`0`') selects the rule set as explained later.

   Build the code and run. You will notice there is a textual output to the terminal showing the grid and the number of cells after each iteration. In addition, there is the option for graphical output using OpenGL and GLFW that can be selected when the executable is built:

   ```
   > make GRAPHICS=GLFW
   ```

This works on `cloud-hpc1.leeds.ac.uk` (but remember to use "-Y" when connecting via `ssh`), however the makefile may require modification on personal devices. Note that this graphical output is entirely optional and does not form part of the assessment.

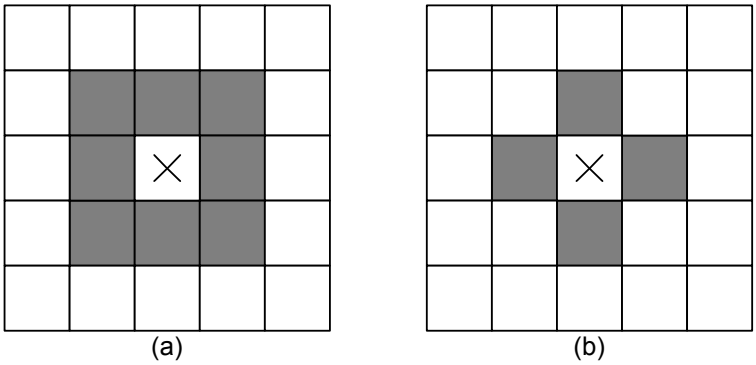

Figure 1: (a) Moore and (b) von Neumann neighbourhoods. For each, the central grid square with the cross '×' is the square being updated, and the surrounding grey squares are the neighbours from which cells should be counted.

2. **Assessment tasks**

   This assignment consists of 4 tasks that involve modifying or designing functions to run in parallel using OpenMP directives as covered in lectures.

   1. The file `cwk1.c` contains the function `countCells` that returns the current number of cells in the grid. Currently this is a double loop, in serial. Replace this with a parallel loop (or loops) that gives the same result.
   2. The function `initialiseGrid` adds `M` cells to the system, ensuring that no cell is added to a grid square that is already occupied, so the total number of cells initially in the grid is exactly `M`. As before, you should modify this function to make it parallel. For the purposes of this assignment, you should assume that the library routine `rand()` is thread-safe.
   3. The function `iterateWithOriginalRules` implements the standard rules for Conway's Game of Life described above. Again, you should modify this function to make it parallel. **Important: You should only modify the array `grid` and, if necessary, copy the data to the separate array `gridCopy` prior to performing calculations.** Some of you will realise that it would be more efficient to instead have a single grid pointer that alternates between `grid` and `gridCopy` between iterations – however, you should *not* do that here, and instead create copies if/when necessary.
   4. The file `cwk1.c` contains the function `iterateWithModifiedRules` that is called when the final command line argument is set to `1` rather than `0`. These modified rules differ from the standard rules as following:

- Rather than the Moore neighbourhood with 8 cells as before, n_{occ} should be calculated using the von Neumann neighbourhood, which includes the 4 adjacent left/right/up/down cells but *not* diagonals; see Fig. 1(b).
- Since the maximum value of n_{occ} is now 4, the update rules should be modified so that:
  * A cell dies if $n_{\text{occ}} = 0$ or $n_{\text{occ}} = 3$, other it remains alive.
  * An empty grid square becomes occupied by a cell if $n_{\text{occ}} = 2$.

  Implement a parallel version of this modified algorithm using the red-black pattern that was covered towards the end of Lecture 5, putting your solution in the provided function `iterateWithModifiedRules`. You should *not* make a copy of the grid first, which means that your implementation will differ slightly from the serial equivalent, as one sub-grid (*e.g.*, the 'red' one) will be updated before the other (*e.g.*, the 'black' one), but this is acceptable for this assignment. It does not matter which subgrid you update first – the autograder will accept either choice.

For all tasks, your parallel implementation should be as efficient as you can using the material covered up to and including Lecture 6. You do not need any material covered in Lecture 7 or later to achieve full marks.

3. **General guidance and study support**

   If you have any queries about this coursework then please ask in person at one of the timetabled lab sessions (held each afternoon during the week of the assignment).

4. **Assessment criteria and marking process**

   Your code will be checked using an autograder on Gradescope to test for functionality. Staff will then inspect your code then allocate the marks as per the mark scheme (see below).

5. **Submission requirements**

   You should upload the file `cwk1.c` to Gradescope using the portal for this coursework. This will test your submission for correct functionality, and return immediately with the results. If any of the checks were failed, modify your code and re-submit until all checks are passed. Late penalties will be calculated from the date and time of your final submission.

   Note that your submission may appear to work on your test system but fail on the autograder. This means that your solution does not work on all systems and is therefore not robust, which means there was an error in your code (see Lecture 3 for a discussion of non-deterministic behaviour for improper parallel code). **You will lose marks if your submission does not pass all of the tests performed by the autograder**, even if it (fortuitously) appears to work on your system.

   You should only modify and upload `cwk1.c`, ensuring that your modified code still works with the makefile provided.

   Do not modify `cwk1_extra.h`, **or copy any of the content to another file and then modify**, as this file will be overwritten with an alternative version for assessment.

6. **Academic misconduct and plagiarism**

   Academic integrity means engaging in good academic practice. This involves essential academic skills, such as keeping track of where you find ideas and information and referencing these accurately in your work.

   By submitting this assignment, you are confirming that the work is a true expression of your own work and ideas and that you have given credit to others where their work has contributed to yours.

   There is a three-tier traffic light categorisation for using Gen AI in assessments. This assessment is **amber** category: AI tools can be used in an assistive role. Use comments in your code to declare any use of generative AI, making clear what tool was used and to what extent.

   Code similarity tools will also be used to check for collusion.

7. **Assessment/marking criteria**

   There are 15 marks in total.

4 marks	:	Correct functionality of operations as tested by the autograder; 1 mark for each of the 4 tasks.
6 marks	:	Efficient parallel implementation of `countCells` and `initialiseGrid`.
5 marks	:	Efficient parallel implementation of `iterateWithOriginalRules` and `iterateWithModifiedRules`.