Concurrent Programming

COMP 409, Winter 2018

Assignment 4

Due date: Thursday, April 12, 2018
6pm

All code should be well-commented, in a professional style, with appropriate variables names, indenting, etc. Your code must be clear and readable. **Marks will be very generously deducted for bad style or lack of clarity.**

All shared variable access must be properly protected by synchronization—there must be *no race conditions,* but also no unnecessary use of synchronization. Unless otherwise specified, your programs should aim to be efficient, and exhibit high parallelism, maximizing the ability of threads to execute concurrently. Please stick closely to the described input and output formats.

1. A simple computer game allows game characters to move around a 2D space discretized into a grid, with each character or obstacle fully occupying a grid cell, and each character moving atomically to adjacent cells (using an 8-way neighbourhood).

   Characters choose a random, non-obstacle goal point within an 8x8 radius (clipped to the interior of the map), and attempt to move directly toward it. Once they get to their destination, or if they cannot step toward their destination at some point (either because of an obstacle or another character being in the way) they pause for one move, and then choose a new destination.

   Maps are generated with different obstacle densities by randomly placing r non-overlapping obstacles on the map, for a given r, never within one square of the perimeter.

   Each character is placed, non-overlapping, at a random point on the perimeter of an $m \times m$ grid. Movement is discrete, based on a k ms rate, but characters move at different speeds relative to that—choose a random number $w \in \{1, 2, 3, 4\}$ for each character, and make them wait wk ms after each move.

   Your application should take 4 parameters, n p r and k, in that order and construct a simulation of n characters on an 30×30 grid. A visual depiction is not required, although you may find it helpful for debugging.

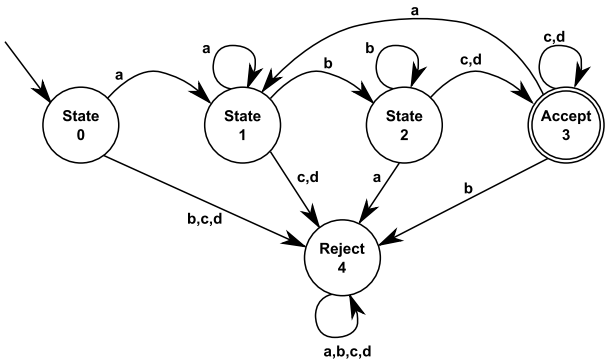
   It should always be possible for characters at least 2 squares away from each other to move concurrently, and movement steps must be atomic—a character moving from a to adjacent cell b must be guaranteed of control over both a and b until their movement is done.

   You must use a *thread pool* to control and move the characters. You may use either a fixed pool, or a cached one, but must limit the maximum number of threads used to p.

   Try your simulation for $n \geq 30$ and 3 values of r that represent few obstacles, a medium density of obstacles, and a high density of obstacles. Let each simulation run for 2 minutes. Once the simulation has ended report the total number of moves made by each character. Plot this value over p for $p \in \{2, 4, 8\}$. **15**

2. In order to match a regular expression (RE), it may first be converted into a deterministic finite automaton (DFA). Matching the RE to an input string is then a matter of processing each character of the input string, making state transitions according to the character. If the DFA is at an accept state once at the end of the string is reached, then the RE matched. **15**

   For example, the regular expression `^(a+b+(c|d)+)+$` can be represented by the following automaton, assuming a string alphabet of $\{a, b, c, d\}$.



The matching process is inherently sequential, since we need to process the string one character at a time. It may be parallelized, however, through the use of an optimistic approach.

Assume we have 1 normal thread, and n optimistic threads. We divide the input string into $n + 1$ pieces. The normal thread gets the first piece of the string, and performs matching as above.

The n optimistic threads each perform matching on their own portion of the string, but since they are not sure what state the DFA will be in to start with, they simulate matching from every possible state simultaneously. For instance, in the above example, an optimistic thread would be checking its fragment 4 times, assuming it started in state 0, 1, 2, or 3 (you do not really need to model starting in the reject state, as reject has no transitions out). In effect, each optimistic thread computes a mapping from each possible state of the DFA to the resulting state for the associated input fragment. Note that this means the optimistic threads might do more work than the normal thread, even on the same size input string fragment.

Once the normal thread reaches the end of its input fragment i, it looks at the mapping produced by the thread handling the next fragment $i + 1$. It knows the the ending state of i in the DFA, and so can use the optimistic map to compute the resulting ending DFA state of the $i + 1$ fragment. This repeats until the matching process is completed for the entire string.

Implement and test this design in **OpenMP** on top of C/C++. Hard-code the example DFA shown above (you do not need to do any RE→DFA conversion), and include a function that generates a (very long) string that would match; an example code fragment is provided. Your simulation should accept a command-line parameter for controlling the number of optimistic threads, and should run your test 10 times, timing the total matching time (excluding the string construction time). Show data for 0–3 optimistic threads and explain your results in relation to the number of processors in your test hardware. Your solution must demonstrate speedup for some non-0 number of optimistic threads!

What to hand in

Submit your assignment to *MyCourses*. Note that clock accuracy varies, and late assignments will not be accepted without a medical note: **do not wait until the last minute**. Assignments must be submitted on the due date **before 6pm**.

Where possible hand in only **source code** files containing code you write. Do not submit compiled binaries or .class files. For any written answer questions, submit either an ASCII text document or a .pdf file *with all fonts embedded*. Do not submit .doc or .docx files. Images (plots or scans) are acceptable in all common graphic file formats.

This assignment is worth 10% of your final grade. **30**