Concurrent Programming
COMP 409, Winter 2021

Assignment 3

Due date: Tuesday, March 30, 2021
6pm

General Requirements

These instructions require you use Java or OpenMP (via C/C++). 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.**

There must be no data races: all shared variable access must be properly protected by synchronization. Any variable that is written by one thread and read or written by another should only be accessed within a synchronized block (protected by the same lock) or critical section, or marked as volatile/atomic. At the same time, avoid unnecessary use of synchronization or use of volatile/atomic. 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.

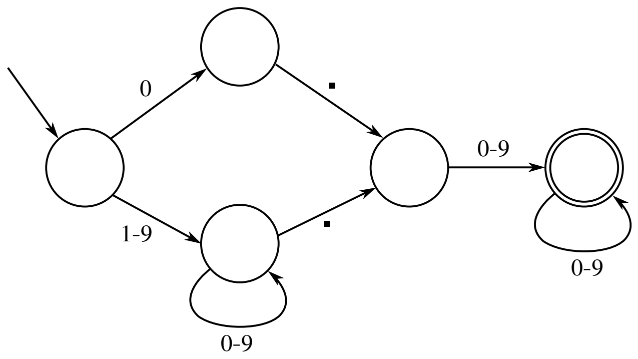
Your assignment submission **must** include a separate text file, `declaration.txt` stating “This assignment solution represents my own efforts, and was designed and written entirely by me”. Assignments without this declaration, or for which this assertion is found to be untrue are not accepted. In the latter case it will also be referred to the academic integrity office.

Questions

1. A simple representation of floating point numbers can be captured by the regular expression **20**

`(0|[1-9][0-9]*)\.[0-9]+`

This regular expression can be applied efficiently by first converting it into a deterministic finite automaton, as so:



We can then look for the longest match, greedily making transitions as we look at each character in sequence. Once we are unable to make a transition we either have found (the end of) a floating point number (i.e., we are in the state with concentric circles), or the characters considered do not form a legal number.

Suppose you have a large text input, and you want to recognize and extract every legal, non-overlapping floating point value. Doing this kind of matching is normally done sequentially. The task here is to improve performance by using multiple threads

First, your input string should be created and stored internally prior to creating any threads. The string should then be divided among the threads for matching; each thread must convert any character that would not be be part of a longest match from a sequential search into an underscore ('_') character.

Unfortunately, dividing the work evenly among the threads is insufficient, as the boundary between recognized floating point values may not occur at the boundaries used for thread partitioning. To address this, each thread other than the first must compute *speculatively*. This works as follows:

Assume we have 1 *normal* thread, and $n \geq 0$ *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. For instance, in the above example, an optimistic thread would consider the processing of its fragment to start in any of the 5 states, and thus cannot be sure of how many characters match until it knows the initial state. Of course if/when all 5 possibilities converge, it may start normal, sequential recognition. In effect, each optimistic thread is computing 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 may need to do more work than the normal thread on a given size of input string (fragment).

To complete the processing, the computation of each thread needs to be merged with its neighbours. For the optimistic threads this extends the input/output (and conversion) mapping over a longer sequence. For the normal thread merging with a neighbour allows it to complete the actual match/conversion over a longer sequence, and eventually the entire sequence.

Implement and test this design in **Java**. Hard-code the example DFA shown above and include a function that generates a string consisting of random characters uniformly chosen from the set

$$\{\texttt{0,1,2,3,4,5,6,7,8,9,.,a}\}$$

Print out the string twice (separated by a newline): first before doing any processing, and second after all unrecognized characters have been converted to underscores. The string should be long enough that your single-threaded simulation runs for at least 100's of milliseconds (not including I/O or string construction).

Your program should be called `q1.java` and should accept one or two command-line parameters: $t \geq 0$ specifying the number of optimistic threads, and an optional random seed (to ensure the same random string can be tested in different runs). Time the matching (post-construction and non-I/O) part and run your test several times (discarding the first run). In a *separate document* show timing data (plot or table) for each of 0–7 optimistic threads and explain your results in relation to the number of processors in your test hardware. Your solution must demonstrate speedup for at least some inputs and some non-0 number of optimistic threads!

2. *Graph-colouring* requires assigning a (usually small) set of colours to nodes in a graph, such that no two adjacent nodes have the same colour. Finding a minimal number of colours to do this is difficult, but heuristic solutions can perform well, and are known to benefit from parallelism. **15**

   The following algorithm, based on the design of Gebremedhin and Mann is quite simple. We need an input undirected graph, with vertices ordered in some fashion (e.g., from unique ids). We will assign colours as integers > 0 (using 0 to mean uncoloured). Our main loop is as follows:

```
Require: A graph formed of a set $V$ of Node objects, each containing an integer colour value, and an adjacency
    list. Each vertex should have an initial colour of 0.
 1: Conflicting = $V$
 2: while Conflicting $\neq \emptyset$ do
 3:     Assign()
 4:     Conflicting = DetectConflicts()
 5: end while
```

Within that main loop are two parallelizable functions. The first is `Assign`:

```
Require: Conflicting must be partitioned among the threads.
 1: for each subset Conf$_i$ of Conflicting concurrently do
 2:     for all $v$ in Conf$_i$ do
 3:         Set $v$.colour to the smallest colour not used by any adjacent Node
 4:     end for
 5: end for
```

The second multithreaded function is `DetectConflicts`:

```
Require: Conflicting must be partitioned among the threads. We also need a new (global) empty set, New-
    Conflicts
 1: for each subset Conf$_i$ of Conflicting concurrently do
 2:     for all $v$ in Conf$_i$ do
 3:         (Atomically) add $v$ to NewConflicts if it has the same colour as any adjacent node $u$, and $u < v$.
 4:     end for
 5: end for
 6: return NewConflicts
```

Define an **openMP** program `q2.c` which accepts 3 command-line parameters: $n > 3$ (the number of nodes in the graph), $0 < e \leq n * (n-1)/2$ (the number of undirected edges in the graph), and $t > 0$ (the number of threads to use). It should do the following,

(a) Sequentially construct a random graph of n nodes and e edges. Edges are made between random pairs of different nodes (and a pair of nodes should not have more than one edge between them).

(b) Colour the graph, according to the above algorithm, sharing the work among the t threads.

(c) Print out the time taken to colour the graph (not including the time to construct it).

(d) Verify the graph is properly coloured. Print out the maximum node degree, and the maximum colour used.

Your design should achieve speedup over $t = 1$ for the colouring part. Select n, and e such that it takes significant time to execute with $t = 1$ (at least a few 100ms; note that you will need a big and relatively dense graph). Provide a *separate document* comparing and discussing performance for $t = 1, 2, 4, 8$, based on average time over at least 5 runs of each configuration (but discard the first run in each case).

What to hand in

Submit your declaration and assignment files 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. Avoid .doc or .docx files. Images (plots or scans) are acceptable in all common graphic file formats.

This assignment is worth 15% of your final grade. $\overline{\textbf{35}}$