Concurrent Programming
COMP 409, Winter 2017

Assignment 4

Due date: Tuesday, April 4, 2017
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 (no race conditions). 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. An *L-System* is a string-rewriting system. It consists of an initial string and a set of rewrite rules. Each rule specifies how a single string character should be transformed into another sequence of string characters or arbitrary length. The system operates by applying all rules concurrently, wherever possible, producing a new string. This process iterates, generating a sequence of strings as output. For instance, given rule $(a \rightarrow bca)$ and rule $(b \rightarrow cab)$, the string ab is successively transformed into **20**

gen 0	ab
gen 1	$bcacab$
gen 2	$cabcbcacbcacab$
gen 3	$cbcacabccabcbcaccabcbcacbcacab$

Use OpenMP in C/C++ to model some simple L-Systems. Represent your string as a linked list of single characters, and implement the rules through separate, hard-coded functions. Use n threads to efficiently create each generation of the string. Once fully transformed the resulting string should be (atomically) printed out, and once fully emitted, the next generation begins.

Use appropriate synchronization, avoid deadlock, and ensure sections of the string that are being independently modified can be worked on concurrently and that all threads are well-utilized.

Your program should take three integer arguments. $n \geq 1$ is the number of threads to use, $g > 1$ indicates how many generations to produce (not including the initial input (gen 0)), and m indicates which of the following initial strings and rules to apply:

hard-code each of these rules

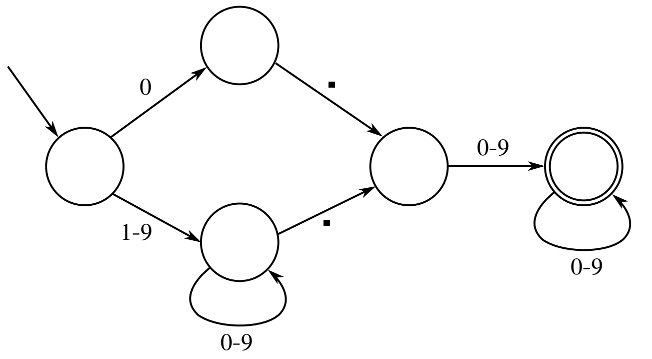
m	Initial String	Rules
1	a	(a → ab), (b → ba)
2	a	(a → b), (b → ba)
3	a	(a → aba), (b → bbb)
4	b	(a → ac), (b → abc), (c → ca)

In a separate document give a brief explanation of your parallelization strategy.

2. A simple representation of floating point numbers can be captured by the regular expression **25**

```
(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 a floating point number (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 write an OpenMP in C/C++ for doing this with multiple threads.

First, your input string should be created and stored internally as an array 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 a space 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 leftmost must compute *speculatively*. This works as follows:

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 normal matching.

The n optimistic threads each apply the regular expression to their own portion of the string, but since they are not sure what state the DFA should be in to start with, they simulate matching starting from *every* possible state simultaneously. 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.

Once the normal thread reaches the end of its input fragment i, the thread handling the next fragment $i+1$ will know the starting state it should've started in, and can thus complete its recognition. 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 and include a function that generates a string consisting of randomly, characters uniformly chosen from the set {`0`,`1`,`2`,`3`,`4`,`5`,`6`,`7`,`8`,`9`,`.`,`x`}. Print out the random string before doing any processing, and after all unrecognized characters have been blanked (separated by a newline). The string should be long enough that your 1-threaded simulation runs for at least 500ms (not including I/O or string construction).

Your simulation should accept a command-line parameter for controlling the number of optimistic threads. Time the matching (post-construction, non-I/O) part and run your test several times. Show timing 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.

Note that for written answers you must show all intermediate work to receive full marks.

This assignment is worth 10% of your final grade. $\overline{\textbf{45}}$