Concurrent Programming

COMP 409, Winter 2024

Assignment 4

Due date: Wednesday, April 10, 2024
Midnight (23:59:59)

General Requirements

These instructions require you use Java or C with OpenMP. 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. This means all shared variable access must be properly protected by synchronization in Java, and your OpenMP code should use appropriate OpenMP constructs. At the same time, avoid 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.

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. The Karatsuba algorithm is a divide-and-conquer algorithm for multiplying two numbers that reduces the number of elementary (single-digit) multiplications necessary over the standard algorithm you (hopefully) learned in grade school. This makes it particularly suitable for efficiently multiplying very large numbers. **10**

   Consider numbers x and y, both expressed as base-10 integers. We can divide x (and y) into two pieces, separated at a given power of 10:

   $$x = x_H 10^i + x_L$$
   $$y = y_H 10^i + y_L.$$

   The product xy can then be derived by multiplying these quantities, along with some extra additions and subtractions. First, we define sub-quantities a, b, and c:

   $$a = x_H y_H$$
   $$b = x_L y_L$$
   $$c = (x_H + x_L)(y_H + y_L) - a - b.$$

   Then we can compute the final product by adding together a shifted up by 10^{2i}, c shifted up by 10^i, and b:

   $$xy = a10^{2i} + c10^i + b.$$

   Assuming a power of 10 that separates x (and y) into pieces (x_H and x_L, and y_H and y_L) of approximately equal numbers of digits, applying the above procedure recursively (with a base case of just multiplying two single-digit numbers) gives an efficient algorithm.

As `q1.java` define a recursive implementation of the Karatsuba algorithm using Java's fork-join thread pooling framework. Your program should accept 3 parameters, consisting of first the number of threads to use in the pool (≥ 1), and then the two numbers in base-10.

Measure and compare performance of your implementation, comparing a pooled version limited to one thread in the pool and a pooled version with a number of threads corresponding to the number of processors in your system. Generate 10 pairs of increasingly larger numbers, choosing a starting number size that takes at least a few 100 ms with a single-threaded version. Test both the single and multithreaded versions for each number pair (use an average over several runs for each number pair, discarding the first run to reduce timing noise). Your multithreaded pool implementation must show non-trivial speedup.

In a separate document `q1.txt` (or `q1.pdf`), show your experimental data and give a brief explanation of your results.

Helper code is provided that will add or subtract two integers expressed as base-10 strings and which you can integrate into your solutions. Note as well that you will need very larger numbers in order to observe performance differences; the following command-chain will emit a random integer of 10000 digits on linux:

```
tr -dc "[:digit:]" < /dev/urandom | head -c 10000
```

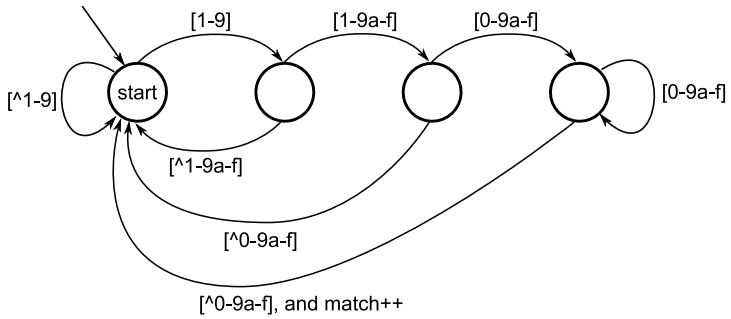
2. Counting all instances of a substring within a string is a common operation in many contexts, and many systems support using a regular expression to match substring instances. Regular expression matching, however, is a very sequential process, since in order to know if a given character is part of a match we also need to know where the previous character left us in terms of the state of the regular expression. **15**

   Optimistic techniques, however, can allow matching to proceed in parallel. Suppose we want to count instances of a regular expression within a string using t threads. First, we convert the regular expression into a DFA consisting of states $s_0, s_1, \ldots, s_m$. The length n string is then divided among the threads, giving each thread a segment of the string. We'll assume that allocation is ordered, so thread 0 gets a segment starting from the first character in the string, thread 1 gets the next segment, and so on. Thread 0 will do a naive matching, while threads $1\ldots$ are optimistic threads. Note the segments do not necessarily need to be of the same size.

   Each thread can then test its segment. Of course all but thread 0 do not know which state of the DFA to start in, since that requires knowing which state the DFA would have been left in by the concurrently computing previous thread. All the optimistic thread thus need to test their segment multiple times: for each possible DFA starting state, they test their segment, counting matches and recording the resulting ending state that would be reached under that starting state assumption.

   Once all threads have finished, counting the actual number of matches merely involves iterating through the thread data in sequence, initializing the count and ending state from the naive thread and then for each optimistic thread using the ending state computed by thread $i-1$ to identify which count and ending state is correct for thread i.

   Build an implementation of this optimistic parallelization approach to match-counting **in OpenMP**. Use the following DFA, which you can embed into your approach (i.e., your design does not need to take in different DFAs or do any RE conversion). Note that each labelled transition implies consuming a character, matches are counted during one of the transitions, and EOF should be treated as a character symbol.



For testing, include a function that generates a random string. The characters should be uniformly chosen from the set:

$$\{\texttt{0},\texttt{0},\texttt{1},\texttt{1},\texttt{2},\texttt{3},\texttt{4},\texttt{5},\texttt{6},\texttt{7},\texttt{8},\texttt{9},\texttt{a},\texttt{b},\texttt{c},\texttt{d},\texttt{e},\texttt{f},\texttt{x},\texttt{x},\texttt{y}\}$$

Your program should accept two integer parameters, $t \geq 0$ being the number of *optimistic* threads to use, and $n > t$ the size of the string to test. As output it should emit three lines, first the the input string, then the count of matches, and finally the number of milliseconds taken by the matching process (do not time the string creation).

You will need a very long string, such that a single-threaded simulation runs for at least 20 to 30 ms or more. *In a separate document* `q2.txt` (or `q2.pdf`) give your timing data for 0–4 optimistic threads averaged over 10 runs each. Give a brief explanation for your results. Your solution must demonstrate speedup for some non-0 number of speculative threads!

What to hand in

Submit your declaration `q1.java`, `q1.txt/pdf`, and `q2.c` and `q2.txt/pdf` files to *MyCourses*. Note that clock accuracy varies, and late assignments will not be accepted without a special permission: **do not wait until the last minute**. Assignments must be submitted on the due date **before midnight**.

Where possible hand in only **source code** files containing code you write. Do not submit compiled binaries or .class files, but do include a `readme.txt` of how to execute your program if it is not trivial and obvious. 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 10% of your final grade. **25**