Concurrent Programming
COMP 409, Winter 2025

Assignment 4

Due date: Tuesday, April 1, 2025
Midnight (23:59:59)

General Requirements

These instructions require you use 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.**
Your OpenMP code should use appropriate OpenMP constructs to ensure proper use of shared and/or private variables. 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. "Sparse matrices" are matrices characterized by having a high proportion of 0 values in the array. Storing these as regular 2D arrays can be quite wasteful, and thus special formats exist that compress the storage, storing only the non-0 values. One popular format is *Compressed Sparse Row* or CSR. In this format three 1D arrays are used to store the actual values, their column indices, and a compressed vector describes the rows. For example, given a matrix (indexing columns and rows with the origin at the upper-left): **10**

1	0	0	2
0	3	4	0
0	0	0	0
5	6	7	0

the CSR format would consist of the following 3 arrays:

rowptr	0	2	4	4	7		
cols:	0	3	1	2	0	1	2
values:	1	2	3	4	5	6	7

The `values` array lists the actual (non-0) array element values. The `cols` array gives the column index of each non-0 value. The `rowptr` gives the offset to the `cols` (and `values`) array of the first non-0 entry in that row. Thus the difference between `rowptr[i]` and `rowptr[i+1]` tells us the number of non-0 values in row i. Note that the `cols` and `values` arrays are the same length as the number of non-0 entries, while the `rowptr` array has the same length as the number of array rows (in the example above one extra entry (7) is shown at the end of the `rowptr`, but we could infer that if we otherwise know the total number of non-0s).

The task in this question is to convert a regular array into CSR format. You must do this in C using OpenMP. Your (C) program should be invoked as:

csr n p s

The $n > 3$ parameter is the array dimension (assume a square, $n \times n$ matrix). The array should be filled with random 0 or 1 values, with $0.0 \leq p \leq 1.0$ used as the probability of a given value being a 0. This means that if $p == 1.0$ then the array should be all 0s, and if $p == 0.0$ the array should be all 1s. The s parameter is a random seed—initialize the random number generator to this value in order to get reproducible results.

Once the array is constructed, print it out to the console in a readable array form (one row per line, column values space-separated). Serially determine the number of non-0s and allocate the 3 1D arrays required. Then convert it to CSR format, and print out the `rowptr`, `cols`, and `values` arrays, each on one line.

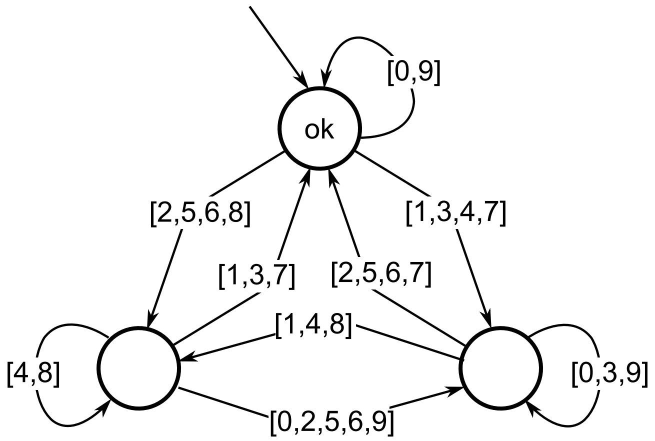
Your approach should use OpenMP to parallelize the initial array construction as well as the conversion. Disable the I/O and time the array creation and conversion phases, using a reasonably large n, for $p \in \{0.05, 0.2, 0.5\}$ for both the sequential execution and an OpenMP execution (using the same seed values for comparability). *In a separate document* report your performance data and comment on the relative performance.

2. Regular expression matching on strings is efficiently performed by converting the expression into a DFA and then testing each character of the input string in sequence, following DFA transitions to see if/when it goes into an accept state or not. This is an inherently sequential process though—in order to know whether/which DFA state an input character takes us to we need to know which DFA state the previous character left us in. **10**

   Optimistic or speculative techniques, however, can allow matching to proceed in parallel. Suppose we have a fixed DFA and divide up the input string among threads somehow, so each thread has a contiguous segment of the input string to examine. The thread with the first segment can examine its segment deterministically, starting with the DFA in its initial/default state. Each other thread, though, is unsure which DFA state to start in, since that depends on the terminating DFA state of the previous segment. These other threads thus compute optimistically or speculatively—they process their segment multiple times, once for each possible DFA starting state. Each optimistic thread thus generates a map from each possible starting DFA state to the resulting ending DFA state for its segment.

   Once all threads have finished, determining the actual final DFA state involves iterating through the mappings in sequence, using the mapping to see what the ending DFA state is given the ending DFA of the previous segment.

   Build an implementation of this optimistic parallelization approach **in OpenMP**. Embed the following DFA into your code.



   For testing, include a function that generates a random string of base-10 digits. 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 either “true” or “false” depending on whether the DFA ended up in the “ok” state or not, 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 measurable time. *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.c`, `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. **$\overline{20}$**

Programming assessment

	Mastery	Proficient	Developing	Beginning
Correctness	The solution works correctly on all inputs and meets all specifications.	The solution meets most of the specifications; minor errors exist.	The solution is incorrect in many instances.	The solution does not run or is mostly incorrect.
Readability	Well organized according to course expectations and easy to follow without additional context.	Mostly organized according to course expectations and easy to follow for someone with context.	Readable only by someone who knows what it is supposed to be doing.	Poorly organized and very difficult to read.
Algorithm Design	The choice of algorithms, data structures, or implementation techniques is very appropriate to the problem.	The choice of algorithms, data structures, or implementation techniques is mostly appropriate to the problem.	The choice of algorithms, data structures, or implementation techniques is mostly inappropriate to the problem.	Fails to present a coherent algorithm or solution.
Documentation	The solution is well documented according to course expectations.	The solution is well documented according to course expectations.	The solution documentation lacks relevancy or disagrees with course expectations.	The solution lacks documentation.
Performance	The solution meets all performance expectations	The solution meets most performance expectations	The solution meets few performance expectations	The solution is not performant.