Concurrent Programming

COMP 409, Winter 2019

Assignment 2

Due date: Wednesday, February 20, 2019
6pm

These instructions require you use Java. 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 not be any *data races,* 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. An 8×8 chessboard is initialized with p chess pieces, either *knight*s or *queen*s. Pieces are random selected and placed in non-overlapping locations on the board, with each piece controlled by a separate thread. **20**

   Threads repeatedly sleep for 10–30ms, then move their piece to a random new location on the board following the standard movement model of the given chess piece (queens can move in a straight line, knights move in an L-shaped pattern). Movement is conceptually instantaneous, but pieces may not occupy the same spot at the same time, and must be guaranteed that their movement will be successful before leaving their existing position. Note that while knights can jump over other pieces, queens traverse all spots between their current location and their destination. Pieces should also be able to move as independently and concurrently as possible.

   Implement a program, `q1.java`, to simulate this scenario. Your program receives two numeric command-line arguments, the first is the number of pieces (being a value between 1 and 64 inclusive), and the second argument is a simulation time in seconds.

   Some additional caveats and constraints to consider in your implementation design:

   (a) Spin-locks are not allowed—use blocking synchronization only, and be careful to avoid deadlock!

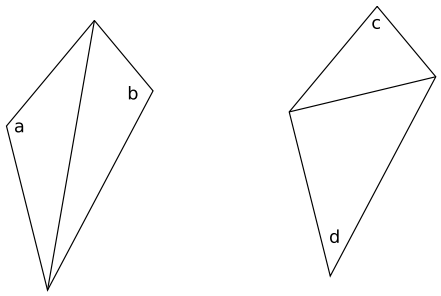
   (b) All pieces should be placed on the board before any piece can move.

   (c) At the end of simulation, and after every 1s of real simulation time as well, the program should emit the number of moves that have been made (in total, by all threads) (followed by a newline). This count should be "instantaneously accurate"— during the time it takes to emit the count it must be an accurate count of all moves completed.

   As a *separate document* (e.g., `q1.txt` or `q1.pdf`), include a brief description of your locking strategy, arguing how/why deadlock is avoided.

2. A *triangulation* of a point set in the plane can be constructed in various ways. A conceptually simple, if "brute-force" solution is to consider every pair of vertices and attempt to add an edge between them if it does not intersect an existing edge. **20**

   A triangulation can be made *Delaunay* by using the "edge-flipping" algorithm. In this algorithm we look for pairs of adjacent (edge-sharing) triangles forming a convex quadrilateral where the sum of opposite angles is $> 180^\circ$. If so, we convert the triangle pair into a different pair (on the same set of 4 vertices) by flipping the internal edge to instead connect the other two vertices. This algorithm is known to converge, reaching a point where no edge flips are possible.



Your program should accept three command line arguments, n and t, and an optional r. Parameter n specifies the number of points, and t specifies the number of threads to use. You can assume $t \geq 1$, and $n \geq 4$. The 3rd parameter, if present, gives the value to use to seed the random number generator. This is used to ensure experiments are repeatable, acting on the same set of points.

The program first creates n random points within a unit square, including the four outer corners of the square. It then triangulates them using a brute-force approach, looking at all pairs of points and add an edge between them if that new edge would not intersect any existing edge. This generates the baseline triangulation.

Draft code is provided that performs the above steps. Your task is to produce a parallel version of the edge-flipping algorithm.

Using t threads, look for and flip non-Delaunay pairs of adjacent triangles. Each thread should keep track of the number of edge-flips done, and the program terminates when all threads are satisfied that no more edge-flips are possible. Final output should be two numbers on separate lines, first the time taken by the multithreaded portion of the code, and second the total number of edge-flips done (sum of all threads).

Use only blocking synchronization; do not use spin-locks for synchronization.

Choose an appropriate value of n for testing, and compute a speedup curve using $t = 1, 2, 3, 4$ threads.

As *separate documents* from your code, submit your plots and a brief explanation of your performance results.

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. 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. **40**