CSCI-GA.3033-017 Special Topic: Multicore Programming

Lab 2 Assignment (Part 1: A Thread-Pool-Driven Genetic Algorithm Solver)

This is the second of four labs in which you'll be building a multithreaded system capable of using genetic algorithms to solve interesting problems, on top of thread-safe primitives you created in Lab 1. Genetic algorithms are a sometimes-powerful, sometimes inefficient method of solving a problem when parameters of the problem are not known a priori. Genetic algorithms have been used for everything from designing complex chip design problems to teaching robots to walk to calculating protein folding arrangements. In this lab, you will use genetic algorithms to solve polynomial equations. Specifically, your program will pick a set of *N* random points for which your program will compute the coefficients for an *N*+1-degree polynomial that goes through the line.

As a simplified example, imagine that you are solving for a second-degree polynomial (that is, $Ax^2 + Bx + C = y$), for which you'll need 3 random points, and your program chooses the points:

(1, 3); (10,4); (-4, 5)

Analytically, we can use a system of equations to find that the matching polynomial is:

A = 0.03651; B = 0.290476; C = 3.25396

We can teach a genetic algorithm to solve this equation. If we pick a random set of starting coefficients, we can repeatedly (1) compute some "fitness" metric (in this case, something like a summed squared distance between each point and the line) that we want to minimize towards zero, (2) randomly mutate the coefficients of the polynomial; (3) keep our changes if and only if they yield a closer solution.

We will do this by creating a thread pool and a driving "main" thread. The thread pool workers will repeatedly try to find a set of better coefficients, and the main thread will be responsible for both determining if a thread has indeed found a better set of coefficients, and determining when a "close enough" solution has been found (allowing the program to terminate).

Further information, hints, tips, and tricks will be / were given in Lecture 6. Specific requirements are as follows; everything else is up to you. Please **strongly** consider documenting the design you create for structuring your classes and their methods, including the functionality you're delegating to each class and method, in a document for your and our reference in coding and grading, respectively. It can only help you write better-structured code the first time.

1. You must implement a thread pool. Among the command line arguments your program must take be a single integer specifying the number of threads to spawn. Your threads will now be thread pool workers.
2. Your program must be able to handle polynomials of any degree, which should be another command line argument. See Appendix 1 for a recommended (but not required) skeleton of what your server should look like.

Testing: A testing framework will be provided during/after Lecture 7.

Grading: This lab will contribute at most 25% to your lab score. It will be graded on functionality, adherence to the specification, thread safety, code style, and commenting. Proper thread safety will be given an inordinate share of the grade, for obvious reasons.

Getting Help: If you're struggling, your first recourse should be the class mailing list. *Important:* you should solicit your fellow students for high-level help, such as "how can I iterate through a vector?". You should not ask for specific help with your code, such as "why doesn't the following code work?". If you're still stuck in a week, come to my office hours on Monday. See the first lecture for the policy on collaboration with other students (tl;dr: don't, outside of the mailing list). As is university policy, instances of cheating will be taken very seriously. If you believe there's an omission or error in this document, you're welcome to email me directly.

Due date: November 5, 2018, by 11:59:59pm EDT. See the first lecture for the late policy.

Submission: Push your code to your **private** MulticoreProgramming repository in a folder entitled "lab2". Please also send me and the grader an email once you have committed and pushed your final submission with the tag (a 7-character hexadecimal string) of that commit; I will use the timestamp GitHub records for that commit to determine whether your submission is on-time or late.

Appendix A: Suggested (*Not* Required) Code Structure

- The threads in the thread pool will repeatedly pull work (a set of current-best coefficients and the fitness or sum square distance metric for those coefficients) from the thread-safe queue, repeatedly perform random mutations *restarting from the given best coefficients* each iteration until that thread finds a better set of coefficients, and then pushing the best result (the coefficients plus the fitness or sum squared distance metric) onto a second queue.
- The "main" thread will start by selecting the *N*+1 random points for a degree *N* polynomial, and will store them in some global data structure that the threads should never mutate. The main thread will then start with random coefficients, compute the fitness metric (which should probably be a function, don't you think?), and push as many items containing the starting

coefficients and fitness metric onto a thread-safe queue as there are threads in the thread pool. It should then listen to a second thread-safe queue that the threads in the pool push their results onto. Every time a thread returns a set of new coefficients with a distance metric, it should see if that's better than some local best it is currently maintaining, replace the coefficients and best fit it's currently locally tracking if so, and either way, push a new item onto the thread-safe queue to the workers to force the now-idled thread to get back to work.

When the main thread is given a set of coefficients that are within some epsilon of an ideal fit (e.g., 1, or 0.1, or 0.001), it should print the result and terminate.