CSCI-GA.3033-017 Special Topic: Multicore Programming
Lab 3 Assignment (Part 2: Genetic Algorithm Solver Performance)

This is the third of four labs in which you'll be building a multithreaded system capable of using genetic algorithms to solve interesting problems. In Lab 2, you implemented a thread-pool-based solver capable of using several worker threads driven by a main thread to search for the correct coefficients to fit a line to a set of points. Your work in this lab will be optimizing your program, and then quantifying the performance. As discussed in class, you should make sure you leave enough time after optimizing to quantify performance: even if you aren't able to greatly optimize your program, you can receive a perfect grade on this lab through a good report that explains the performance you see.

Deliverables

1. Code that is as optimized as reasonably possible, and implements measurements of the below performance metrics.
2. A report of no more than 4 pages, graphs and tables included, tabulating performance, and more importantly, explaining why you see the performance that you do.

Optimization

Try to reduce the number of iterations necessary to find the coefficients for a line of best fit. You should work on your own to try to find techniques to reduce the number of iterations. A few hints that may help: try to offset from existing coefficients instead of picking new random coefficients on each iteration, try to come up with a principled approach to gradually reducing that offset as the sum squared distance for each corresponding coefficient decreases, try modifying only one coefficient per iteration, and try to determine the correct "direction" in which to move each coefficient (larger or smaller) in order to reduce the sum squared distance. You will *not* be penalized if you make (and document) an effort to improve your program's performance but are unsuccessful. Much of the optimization for this program relates to the underlying solution methodology rather than to the multicore programming techniques used.

Storing and Printing Statistics:

1. Add a way to record and print the total number of coefficient guesses across all threads, and the total number of "best" coefficients returned to the main thread for testing.
2. You should also find an *efficient* way to compute the mean, median, minimum, and maximum *time* taken by the coefficient guess iterations across your program's threads: you should try to impact the performance of the program as little as possible adding these computations! If

you find that they have a significant impact on the performance of the program, try to explain why, and suggest what could be done to reduce this impact.

3. Finally, measure the total time taken to fit an equation to a set of points (i.e., the entire runtime of your program). It's acceptable to simply use Linux's `time` program for this.

Measuring and Reporting Performance: Measure and report (via tables/graphs) performance per coefficient guess iteration, in total numbers of guesses needed to find a solution, and in total program time. Measure from 1 to at least 4 threads and for equation degrees from 2 to (if possible) 5. If you find that any equation degree is prohibitively slow, omit it, and explain why it was omitted. Try to measure at least as many threads as your testing machine has cores (try to find machines with at least 4). Report aggregate statistics for as many runs as is feasible,

Baseline: Sample points and sample solving times will be provided during/after Lecture 11.

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 26, 2018, by 11:59:59pm EDT. See the first lecture for the late policy.

Submission: Push your code (and a PDF of your report) to your **private** MulticoreProgramming repository in a folder entitled "lab3". 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 1: Sample Performance

My solution was not heavily optimized. In particular, it performs notably better for functions with small-magnitude first derivatives (shallow slopes). Each of the following samples is executed on 4 threads, with a maximum allowed sum squared distance of 0.1. Note that these results are collected with point x and y bounds of [-5, 5], and produces similar results with any *square* bounds (e.g., x and y in [-1000, 1000]).

Degree	Equation	Points	Real Times (m:s)
1	y = 73.8045x + -252.932	(3.44266, 0.928446) (3.47252, 3.57946)	0:58.84 0:58.18 0:56.75
1	y = 0.325432x + -1.68596	(-0.0541007, -2.01239) (3.31911, -0.569851)	0:0.02 0:0.03 0:0.04
2	y = 15.949x^2 + -36.4841x + -62.269	(3.44266, 0.928446) (3.47252, 3.57946) (-1.15618, 1.23564)	1:29.45 1:23.77 1:23.92
3	y = 0.0964522x^3 + 0.257209x^2 + -1.52385x + -1.87369	(-0.0541007, -2.01239) (3.31911, -0.569851) (-4.74828, 0.833217) (-2.34434, 2.09208)	0:0.13 0:0.14 0:0.18
3	y = 1.72558x^3 + -2.10124x^2 + -35.8631x + 48.0942	(4.54412, 3.87412) (1.21489, 4.52278) (4.50452, 1.40356) (-4.58827, 1.72866)	13:23.55 12:20.15 12:34.69