Lab3 Report:

1 Optimization Methods

1.1

Since I observed that lower degree coefficients tend to fluctuate more frequently than higher degree coefficients, everytime when I add an offset to the current coefficient, I divide the offset by the degree number so that the extent to which coefficients would be affected is inversely proportional to the degree. This would make the possibility of finding a better solution more likely.

1.2

Sometimes, due to the inherent randomness involved, a thread might get stuck in the loop for a long time before it finds a good enough solution and sends it back to the main thread. And this would make the whole process progress slowly because it could've given up earlier and receive a new task to optimize against. So I added a timeout variable for each thread so that when it processes long enough without finding a good enough solution, it would send what it had so far back to the main thread immediately in hope for getting a newer, updated solution for next round. This would somehow prevent the bad case where the queue has lots of better coefficients but each thread is still working on the outdated solution.

1.3

Since as the program runs, the changes to each coefficient tend to be smaller, keeping a constant random variable range to find new offset would become less efficient as our results become closer and closer to the true answer. So I make each thread own a different random number generator with distinct ranges, which could make the progress speed more uniform and make the overall process less likely to get stuck at certain point.

1.4

I tried annealing algorithm in Lab2, which could help get out of the situation where the program might get stuck at a local max or min point without being able to find the global max or min. And the idea of this is to not always adapt the better solution, but rather to give worst solutions a chance with certain probability which changes in each epoch. But it turned out not to be really efficient, not even as good as the vanilla version. So I got rid of it from Lab2 and didn't consider revising it to make it more optimized.

2 Performance Chart

2.1 Charting the Performance

I ran the program several times with fitness set to 0.1, which is the sum of squared distance between all computed y and real y value. The range of random sample points is between -5 and 5. Program time is measured as total CPU clock time (more detailed in 2.2.1).

DEGREE	SAMPLE_POINTS	RESULT_COEFF	TOTAL_ITERATION	PROGRAM_TIME	NUM_OF_THREADS
2	(0.652475,-4.39765) (0.919633,1.5912) (-1.3337,1.14881)	10.425 4.39287 -11.5047	1221	0.005497s	1
2	(-4.31685,1.49474) (2.56056,0.995586) (-2.05491,-2.73603)	0.397037 0.605594 -3.35276	2417	0.006477s	2
2	(2.68814,-0.948165) (1.11373,1.07084) (0.395055,-0.282049)	-1.28 3.47962 -1.17655	499	0.003201s	3
2	(-4.32794,-4.18864) (0.048152,4.35491) (-4.03676,1.59337)	-4.25572 -16.3023 5.00334	13273	0.034276s	4
3	(0.672042,0.151069) (2.21996,-0.377869) (-0.118023,2.89463) (-3.46507,-2.17985)	0.39722 0.0285656 -3.35466 2.33048	2777	0.007084s	1
3	(3.19757,-0.230029) (-3.73096,2.63142) (1.20326,2.75614) (1.57901,0.26924)	0.547896 -0.560624 -7.38501 11.2891	110109	0.164125s	2
3	(-3.46798,-2.74517) (-2.34117,-0.81162) (2.29216,4.94457) (-0.757318,3.47732)	-0.177232 -0.670579 2.16732 5.55227	3704	0.067255s	3
3	(-2.21766,1.30086) (3.99643,1.87446) (2.21461,1.58015) (-4.77738,1.26963)	0.0018418 0.0266734 0.0400161 1.23107	514996	0.799594s	4
3	(2.25303,-4.0488) (0.342269,1.67802) (-0.434242,-1.31966) (-1.28146,2.76908)	-2.06492 2.14839 3.94166 -0.0616038	1097	0.005806s	4
3	(4.58947,-1.56614) (0.659466,3.08361) (-1.2425,3.45865) (2.30307,1.58305)	0.0201345 -0.281807 -0.253574 3.52976	148394	0.257657s	4
4	(-3.22614,0.909698) (2.04444,-2.95729) (-1.48182,1.11719) (-1.95451,3.73962) (-4.61576,2.59405)	0.405454 2.84769 2.4867 -13.5338 -16.9275	16190573	19.0559s	1
4	(1.34679,0.803751) (-2.14298,-3.84713) (-4.69,-0.874701) (-2.692,4.25609) (-4.07237,2.77102)	0.708291 7.5155 20.9038 -4.93419 -51.1503	19427651	27.0654s	2

4	(-1.0504,-3.67368) (-2.23719,-1.45367) (-0.0138011,3.80849) (-0.588923,2.05846) (3.90483,0.2862)	1.42333 -1.77162 -13.9655 -4.08838 3.72508	7211207	10.5747s	3
4	(3.60107,-2.54529) (-3.89956,4.42658) (-4.65746,-0.0974264) (-2.86828,-4.53461) (-0.160759,-0.0130053)	-0.351503 -1.59924 4.32655 20.0195 3.07721	18444414	29.4227s	4
4	(-1.32219,-3.33598) (-0.878799,-0.516273) (-1.02298,-0.108296) (2.41151,-4.1706) (0.904758,3.57332)	-3.2316 3.55717 10.686 -0.811206 -4.98286	2676528	4.25202s	4
4	(3.45343,0.613576) (-2.05585,2.18267) (-0.116299,-3.87003) (-0.0065364,4.49059) (-4.89432,-3.71604)	-2.22813 -7.82834 30.6531 78.0759 4.83862	763944537	1183.72s	4
4	(1.42832,-3.75899) (-0.0966148,-3.69463) (-3.38354,3.10568) (4.45623,-3.29891) (-1.48854,-3.15288)	0.0134865 -0.0883848 0.153975 -0.0421568 -3.78887	109078384	162.889s	4

2.2 Performance Analysis

2.2.1 Time Calculation

At first glance, more threads didn't increase the latency that much as shown in the chart. And the time fluctuates a lot from few seconds to hundreds or even a thousand seconds. When I divide the number of guesses all threads in total have done by the number of threads, the rounds with more threads sometime even have smaller number, which means threads are more efficient in single threaded program than multithreaded. But then after I run time command in the shell, I figured that the way I recorded the "time" is a little bit different and misled me as I didn't realize the fact that I was calculating the CPU clock time.

Here is the method I used to calculate the time:

```
startTime = clock();
endTime = clock();
total = (endTime - startTime) / double(CLOCKS_PER_SEC);
```

And it turned out to be the total CPU time consumed in seconds by all the threads that this program spawns. EndTime minus startTime is the sum of clock cycles of all threads in this process, and we divide it by the macro which gives us the number of seconds this program actually runs. So ideally, the wall clock time (real time from start to end) could be estimated by dividing the clock time by the number of threads we use, assuming that each thread does average amount of work. And after the division, the result is very close to what time command

gave me. At the end, I decided not to change the calculation method (like gettimeofday), because I think this way, even though initially a little bit misleading, ends up being more accurate as it doesn't care about other concurrently running processes.

2.2.2 Randomness

In some cases, when I run two rounds with the same degree and one with one thread and the other one with more threads, the single-threaded program seems to take less time (even after the division). This is due to the inherent randomness in this algorithm. Sometimes, a single flip of the sign of one coefficient could result in a hundred-times slower execution, so does a less ideal set of sample points. A better way to look at the performance (throughput) is to look at the number of guesses all threads have done during the execution divided by the execution time. We can take two rows (two runs) from the chart:

A. degree 4, 16190573 guesses, 19s, 1 thread
B. degree 4, 763944537 guesses, 1184s, 4 threads

We first divide A's time by 1 and B's time by 4 as times are measured as total clock time by all threads. Then the two normalized rows become the following:

A. 16190573 guesses, 19s
B. 763944537 gusses, 296s

Right now, we can see that run A's throughput is around 852,135 guesses per second, and B's throughput is around 2,580,893 guesses per second. If we divide A's by B's throughput, we can get 852,135 / 2,580,893, which roughly equals to 0.33. So B with four threads has roughly three times the throughput than A with a single thread.

2.2.3 Bottleneck

As shown in 2.2.2, we expect B to run about four times faster than A in terms of throughput because B has four threads and A uses one, but we ended up getting B three times faster than A. The are multiple reasons that might cause the overhead. The concurrent queue is not every efficient as it only allows single access. So when the thread's individual task becomes relatively easy, the time it takes for each thread to send its answer and receive new tasks via the queue becomes longer. One workaround is to set a higher standard for each task, making each send more significant to our final result. But this might at the end make our program get stuck. Using a better queue implementation can surely speed up the program. Second, it is hard to improve the performance without changing the fundamental algorithm methodology is that as the fitness improves over time, the distribution engine becomes less likely to find a proper result if it's distribution range doesn't change. So we need to either change way to generate new coefficients or dynamically adjust the range. Third, the naive algorithm, which gives an offset in each iteration, might not be able to find a solution for higher degree polynomial as it lacks some mechanisms to get out of local maximum, which makes it impossible to ever terminate if the fitness requirement is really high.