Ryan Margono
Lab 4 Report

Pre-optimized Solution

The most naive implementation of the lab would be mutating the coefficients by replacing them with a completley new set of coefficients. The combination of possible coefficients generated this way is incredibly large even within a small range. Only a tiny subset of these combinations will result in an acceptable fitness score, so the number of times a thread has to iterate is expectedly large. In fact, there is no guarantee that the program will ever find said combination. Because of this, running trials of this approach is almost impossible.

After lab 2, I have already implemented some optimizations compared to the most naive approach. Firstly, rather than select completley random coefficients, mutations were generated by altering the existing coefficients within a certain range. This was further optimized by having this range scale according to the fitness - the smaller the fitness, the smaller the range. The idea behind this was that the closer the coefficietns were to the fitness, the less they had to be changed.

Optimized Solution

Other than the mutex logic, I believed that the other area of improvement could be with multithreading. The preoptimized solution had each thread listen to the best coefficients queue, 'caching' its fitness in a local variable. They would then keep looping and mutating the coefficients until a better solution is found and push that to the main thread to process. The problem with this is that the thread's local fitness goal could be vastly different than the best global fitness goal. Therefore, its work could be wasted, resulting in more thread iterations and longer run times.

To optimize this, the main thread will handle determining whether or not a solution is better or not rather than the individual threads. Then, main will always push to the best coefficients to the worker queue. Because threads don't check themselves and keep looping to find better coefficients, their local fitness goal is constantly being updated to the most up to date best fitness. So, when they mutate coefficients to find a better solution, its more likely to be an actual better solution rather than one from a while ago.

Protocol

The trials were ran on the 64 core courant crunchy1 machine with the same coords, threshold, and random starting coefficients. They were ran with 32 threads and 64 threads three times, and their averages were recorded in the charts below.

Optimized

Polynomial Degree	Number of Threads	Equation	Coordinates	Fitness Result Threshold	Real Time	Thread min time	Thread max time	Thread median time	Thread mean time	Total Thread iterations
1	**32**	0.771775x + 4.12156	(-1, 3), (-4, -2)	2	0.03	0	0.03	0.01	0.0136842	39
1	**64**	0.513002x + 3.48543	(-1, 3), (-4, -2)	2	0.008402	0.00011	0.006628	0.002415	0.00279782	15
2	**32**	1.47922x^2 + 7.81684x + 9.47589	(-1,3), (-4, -2), (-3, -2),	2	1.75	0	1.74	0.82	0.835808	1481
2	**64**	1.61633x^2 + 8.29816x + 9.0006	(-1,3), (-4, -2), (-3, -2),	2	0.060874	0.000148	0.060516	0.032265	0.0312586	1313
3	**32**	0.600362x^3 + 6.16054x^2 + 18.7787x + 15.9613	(-1,3), (-4, -2), (-3, -2), (-5, 1)	2	943.97	0	943.59	468.12	471.659	459231
3	**64**	0.598349x^3 + 6.10997x^2 + 18.4193x + 15.4823	(-1,3), (-4, -2), (-3, -2), (-5, 1)	2	809.55	0.01	809.3	408.4	406.165	294764

Pre Optimization

Polynomial Degree	Number of Threads	Equation	Coordinates	Fitness Result Threshold	Real Time	Thread min time	Thread max time	Thread median time	Thread mean time	Total Thread iterations
1	**32**	2.22579x + 6.62299	(-1, 3), (-4, -2)	2	0.4	0	0.31	0.01	0.0359091	697
1	**64**	0.604373x + 4.64991	(-1, 3), (-4, -2)	2	0.37	0	0.11	0.03	0.030.0377941	479
2	**32**	1.56581x^2+ 8.0593x + 9.10997	(-1,3), (-4, -2), (-3, -2),	2	22.24	0.01	22.19	0.03	2.45978	16717
2	**64**	1.35991x^2 + 6.9903x + 7.84959	(-1,3), (-4, -2), (-3, -2),	2	17.13	0	16.83	0.11	1.1881	11689
3	**32**	0.710774x^3 + 7.08776x^2 + 20.7833x 16.8005	(-1,3), (-4, -2), (-3, -2), (-5, 1)	2	2239.2	0.02	558.35	0.155	49.7536	568135
3	**64**	0.654376x^3 + 6.66841x^2 + 19.975x + 16.4201	(-1,3), (-4, -2), (-3, -2), (-5, 1)	2	1267.9	0.01	631.69	0.075	17.4663	469518

Comparison

Polynomial Degree	Number of Threads	Optimized Runtime	Pre-Optimized Runtime	Difference	Improvement %	Optimized Number of Thread Loops	Pre-Optimized Number of Thread Loops	Difference	Improvement %
1	**32**	0.03	0.4	0.37	13.3333333333333	39	697	658	17.8717948717949
1	**64**	0.008402	0.37	0.361598	44.0371340157105	15	479	464	31.9333333333333
2	**32**	1.75	22.24	20.49	12.7085714285714	1481	16717	15236	11.2876434841323
2	**64**	0.060874	17.13	17.069126	281.400926503926	1313	11689	10376	8.9025133282559
3	**32**	943.97	2239.2	1295.23	2.372109283134	459231	568135	108904	1.23714426944174
3	**64**	809.55	1267.9	458.35	1.56617874127602	294764	469518	174754	1.59286072926138

Observations

As for the lower polynomial degrees, there are significant improvements in speed when comparing the two solutions. This is extremely evident in the 64 thread runs, where there was a 40% improvement in first degree polynomials, and 200% improvement in second.
However, as the polynomial degrees increase, the two solutions become closer and closer to the same efficiency. In third degree polynomials, there is only about a 1-2% improvement in efficiency.

Explanations

I believed that the change of main processing the best solution would be more efficient because it would lead to better guesswork from the threads. However, I think that as the coefficients increase and the chances of finding the right combination of coefficients shrink, the guesswork done by the threads in both solutions near the same in either solution.