Casey Adams, Kevin Connell
ECEC 622
Project 4 - Particle Swarm Optimization with OpenMP
2/14/2021

Particle Swarm Optimization - OpenMP

Implementation of Particle Swarm Optimization using OpenMP

Particle swarm optimization is a technique designed to optimize nonlinear functions without the use of a function gradient. This is useful in situations where the gradient cannot always be easily determined. The process works by generating a large number of "particles" with separate starting positions. Each dimension of a particle's position corresponds to a parameter of the function being optimized. These particles are given velocities in this space and each iteration the velocity is changed slightly based on the direction of the particle's optimal solution and the collective swarm's optimal solution. The net effect is that the swarm slowly congregates around a potential solution. While this solution is not guaranteed to be a globally optimal one, it tends to be a relatively good local minima.

Updating the position of each particle can be a time-consuming process when there are thousands of particles used for the approximation. For this reason, parallelization can provide a useful speedup in performance when the particle count and dimensionality of the problem is high. Parallelization was achieved here using OpenMP, and API/standard for defining parallelizable parts of a program such as for loops. Parallelizable for loops were tagged with pragmas which tells OpenMP how many threads to use when processing the loops. OpenMP then divides these loops in the background into a set of different threads. Any variable that needs to be written to simultaneously across multiple threads in the same for loop was set to private so that each thread could have a separate instance of that variable. Additionally, any code that could only be processed by 1 thread at time was tagged with a critical pragma. An example of what a pragma looks like for parallelizing a loop can be seen here:

```
#pragma omp parallel for num_threads(num_threads) private(fitness, i, j, status, particle)
```

This pragma says the following for loop should be run with 'num_threads' threads and that the variables fitness, i, j, status, and particle should be separate for each of those threads. Many of these variables are temporary ones used in the for loop and must be kept separate from other threads to ensure proper operation.

This idea of parallelization was applied to not only the main particle update loop by also the pso_init loop which creates the initial particle states and the pso_get_best_fitness loop which determines which particle has the optimal solution.

Performance

To test the performance of the multithreaded solution when compared to the single threaded solution, tests were run with 4, 8, 16, and 32 threads on the Rastrigin and Schwefel functions with dimensionalities of 10 and 20. In all situations, 10,000 particles were used and the iteration limit was also set to 10,000. For Rastrigin the initialized particles were given positions between -5.12 and 5.12 on each axis and -500 to 500 for the Schwefel function. The results can be seen in the two tables below.

Rastrigin Function			
Dimensionality (D)	**Threads**	**Single-threaded solution (Gold)**	**Multi-threaded Solution (OMP)**
10	4	41.29	23.68
	8	41.76	28.21
	16	41.62	16.54
	32	41.76	14.80
20	4	80.11	32.41
	8	80.25	32.27
	16	78.36	27.81
	32	76.46	15.96

Schwefel Function			
Dimensionality (D)	**Threads**	**Single-threaded solution (Gold)**	**Multi-threaded Solution (OMP)**
10	4	45.26	24.15
	8	44.97	28.61
	16	44.77	16.78
	32	45.52	10.09
20	4	83.81	31.52
	8	84.24	32.42
	16	84.79	19.79
	32	84.63	16.39

Based on the data found here, the multithreaded solution sees increased improvement over the single-threaded one when the dimensionality of the function is high. This makes sense as the initial overhead of the extra threads becomes less significant as the problem gets harder. Additionally, it was found that going from 4 to 8 threads offered little improvement but going from 8 to 16 and 16 to 32 offered more meaningful improvements in performance. In no situation did the multithreaded solution do worse than the single threaded one.

Conclusion

Using OpenMP for parallelizing the problem of particle swarm optimization was found to offer significant improvements over traditional sequential implementations. This is because particle updates can be performed on each particle independently which benefits greatly from parallelization. The improvements are even more obvious when the amount of computation per iteration is high (high dimensionality and particle count). This is because the overhead associated with having a large number of threads becomes less computationally expensive when compared to the complex problem being solved. With this problem, the more threads provided, the better the system performed. It is possible even more threads would show even more improvement on a server that had more CPU cores.