Dominic Erardi, Marc DeCarlo

ECE 622

Particle Swarm Optimization

Summary:

We created a parallel section surrounding the while loop so that threads do not get re-created for each iteration, which would lead to much longer run times. In here, we set the number of threads along with the private and shared variables. Each thread maintains its own copy of I, j, r1, r2, particle, curr_fitness, gest, local_g, and local_fitness as these values are updated by each of the threads, but their values do not need to be shared amongst them. The variables g and best_fitness, however, do need to be shared as the threads need to check against the latest values in pso_get_best_fitness.

We parallelized the particle for-loop using #pragma omp for. Each thread will do an approximately equal amount of work before moving on to another parallelized for-loop that replaces pso_get_best_fitness. For this loop, each thread maintains their own copy of local_fitness and local_g. Instead of updating best_fitness and g directly, they update their own local_fitness and local_g. After the threads finish their work, they move into a critical section in which they update best_fitness and best_g according to the lowest local_fitness value. The threads then move into a final parallelized for-loop for setting each particle's value 'g' to the new g value. Finally, one thread will increment iter.

We also parallelized the swarm's initiation using omp. To do this, we established a parallel section. Each of the threads maintain their own copies of I, j, fitness, status, and particle for the initial for-loop. Then, the same process for parallelizing the pso_get_best_fitness function are followed again.

Results:

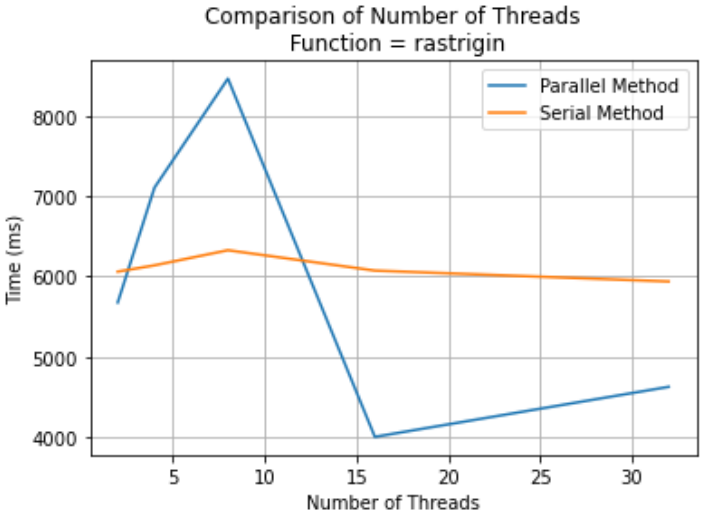

Figure 1: Number of threads vs runtime for the Rastrigin function.

Parameters: 5000 iterations, 10,000 particles, 10 dimensions

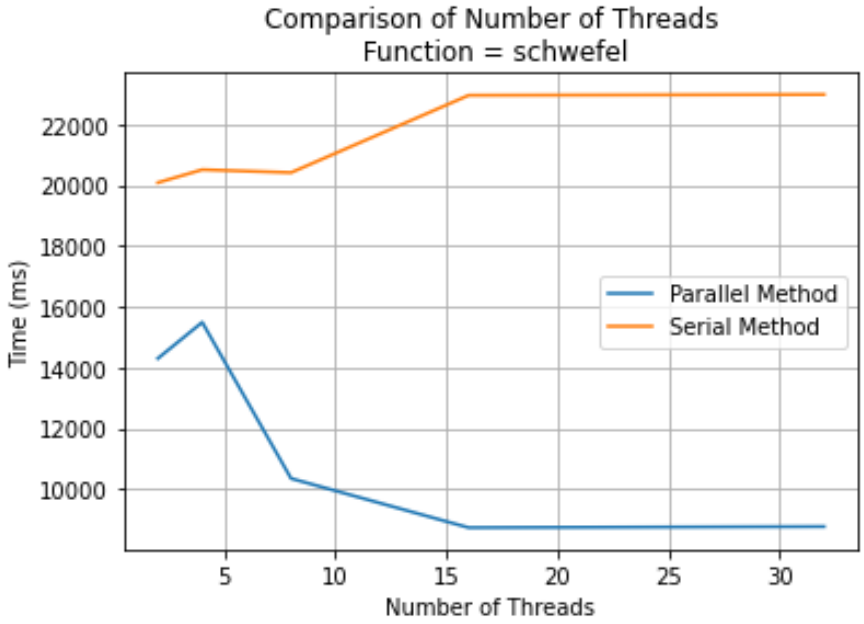

Figure 2: Number of threads vs runtime for the Schwefel function.

Parameters: 5000 iterations, 10,000 particles, 20 dimensions

Conclusion:

Particle Swarm Optimization was one of the more challenging parallel problems faced so far. It required a deep knowledge of Open MP to reach a substantial speedup. As this is extrema discovery problem like Gradient Descent, it is highly parallelizable and most likely faster speedups can still be achieved through the use of more resources and more efficient code.