Aleksandar Aleksandrov
ECEC413

Particle Swarm Optimization

Introduction

This assignment is focused on converting the Particle Swarm Optimization algorithm from a single threaded algorithm to multithreaded one using OpenMP. The PSO algorithm uses a set number of particles and through every iteration it generates velocity for every particle and moves it to a new position. Every position is then compared to the current best fit position. If the specific particle has reached a position that is a better fit than the current best fit, it updates the best fit position. Also, every particle contains the particle index which currently holds the position for best fitness. By randomly generating the velocity for each particle and moving them towards the direction of current best fit; through many iterations, the particles "swarm" around the solution.

Multithreaded Implementation

The multithreaded implementation starts with the function "optimize_using_omp". The function initializes the swarm with "pso_init_omp" and then proceeds to call the "pso_solve_omp" function which contains the multithreaded algorithm to find a for the corresponding function.

The "**pso_solve_omp**" function starts off by declaring all the necessary variables for the execution of the algorithm:

```
    int i, j, iter = 0, g = -1, private_g;
    float w = 0.79, c1 = 1.49, c2 = 1.49;
    float best_fitness, partial_fitness;

    float r1, r2;
    particle_t *particle, *gbest;
    float curr_fitness;

    unsigned int seed = 2021;
```

Every thread will share the values of the variables – g,w,c1,c2,xmax,xmin,max_iter,function. The threads will have their own private variables – r1 ,r2 , j ,i , curr_fitness ,particle ,gbest ,partial_fitness ,private_g. The "iter" variable is also private, since all threads will have to go through the algorithm #iter times. The iter variable is already initialized, so it is passed as a "firstprivate" variable.

```
#pragma omp parallel private(j,i,r1,r2,curr_fitness,particle,gbest,partial_fitness,
private_g) firstprivate(iter)
shared(g,best_fitness,w,c1,c2,xmax,xmin,max_iter,function) num_threads(num_threads)
```

Then, the function proceeds to create a parallel section over the while loop, all threads will execute on the while loop and computation heavy parts of that loop will be split between the threads.

Every thread starts off the execution by resetting their local private_g and partial_fitness variables. One thread resets the best_fitness variable since it is shared:

```
            private_g = -1;
            partial_fitness = INFINITY;

            //one thread updates the best_fitness
            #pragma omp single
                best_fitness = INFINITY;
```

Then by using OpenMP, the for loop that iterates over all particles that are generated is parallelized between the threads:

```
#pragma omp for schedule(static)
```

The static scheduling is used to distribute the chunks evenly across the threads (small performance improvements were noticed through testing). Then every thread operates on its own chunk of particles, generates new random velocity, updates the positions, and evaluates the fitness of the particle.

After a thread is done operating on the particles in its chunk, the thread proceeds to execute the parallelized version of "pso_get_best_fitness". All threads first wait for each thread to complete their particle calculations. Then, the for loop going through the whole swarm is again chunked up between the threads and every thread stores the results from its own chunk in its private variables – private_g and partial_fitness. Next, the threads go into the critical section one at a time to update the best_fitness result for all particles and the index of the particle.

```
            #pragma omp critical
            {
                if(partial_fitness < best_fitness){
                    best_fitness = partial_fitness;
                    g = private_g;
                }
            }
```

Lastly, all threads again go over the chunked-up particle swarm to update the best_fitness particle index on all particles.

The "pso_init" function is also parallelized. The for loop which creates the particles is split between the threads. The best fitness is parallelized in a similar way as the compute function and all particles are again updated in parallel.

Performance Improvements

1. Rastrigin Function

./pso rastrigin 10 1000 -5.12 5.12 10000 num_threads

Particles	Iterations	Synchronous	4 threads	8 threads	16 threads
1000	10000	6.81s	4.16s	4.94s	6.65s

For the Rastrigin function, we can see a performance improvement for the 4 threads and the 8 threads multithreaded versions. The 16 threaded version does not perform much better than the synchronous one due to the larger overhead. In addition, to get the best results the code had to be executed multiple times. We can say that for the Rastrigin function, the multithreaded version gives a small improvement however it is not significant.

2. Schwefel Function

./pso schwefel 20 1000 -500 500 10000 num_threads

Particles	Iterations	Synchronous	4 threads	8 threads	16 threads
1000	10000	25.42s	14.88s	7.89s	16.63s

For the Schwefel function, we can see higher performance gains between the multithreaded version and the single threaded implementation. The 8 threaded version performed the best by running ~3 times faster than the single threaded version. Again, we can see a loss in performance with the higher 16 thread count due to overhead.

Conclusion

Overall, the multithreaded version provided better gains in performance for the Schwefel function over the Rastrigin function. In addition, the performance gained varied largely between every execution. The correctness of the program was compared with the all the functions, and it returned proper “fitness” and “pbest” results. A way to improve the performance would be to find a way to reduce the places at which the threads synchronize, and parallelization is lost.