Tam Phi (thp29)
Hanh Phung (hdd29)

Assignment 3 Report - PSO OMP

Introduction:

Particle Swarm Optimization is a simple method to optimize non-linear functions using population-based search algorithm. The algorithm initializes a population of random solutions then move those particles around the search space until best solution is found. In other words, those particles will swarm toward local best solution. The size of each swarm is determined by number of particles. The implemented program ***optimize_using_omp.c*** applies multi-thread in OMP fashion to improve the performance, time-wise, of its serial counterparts. Each thread is responsible for certain number of particles within each swarm.

Design and Implementation:

Overview of structure

The structure of a swarm that contains multiple particles is as follows:

```
/* Structure of a particle */
typedef struct particle_s {
    int dim;            /* Dimension of particle */
    float *x;           /* Particle position */
    float *v;           /* Particle velocity */
    float *pbest;       /* Best particle position seen */
    float fitness;      /* Fitness of particle */
    int g;              /* Index of best performing particle in swarm */
} particle_t;

/* Structure of the swarm */
typedef struct swarm_s {
    int num_particles;          /* Number of particles */
    particle_t *particle;       /* Particle within swarm */
} swarm_t;
```

In addition to these parameters, number of iteration, how many times the these particles will swarm, is an user-defined input.

Implementation of multi-thread:

There are 4 functions called in the implementation of OMP

1) Exec func: call other functions
2) Pso_init(): initialize a swarm and vectors
3)pso_solve_omp: solve for a solution
4) pso_best_fitness_omp: helper funciton of pso_solve_omp to find the best fitness and its location

Function Implementation:

1) Exec function:

Optimize_using_omp()

```
{
   call pso_init_omp();
  /* g is the index of best solution */
   int g;
   g = pso_solve_omp(function, swarm, xmax, xmin, num_iter, num_threads);
   if (g >= 0)
      pso_print_particle(&swarm->particle[g]);
    return g;
}
```

2) Initialize PSO:

Pso_intit_omp()

{

#OMP thread with each thread containing private particle information (quantity, x_vector, v_vector, pbest)

Malloc space for swam_t struct;

Generate position vector of random number of uniform distribution;

Generate velocity vector of random number of uniform distribution;

Initialize best position for particle;

Evaluate fitness

Get best fitness

}

3) Solve for solution:

Pso_solve_omp()

{

While (iter < max_iter){

#Each OMP thread is responsible for update the particle velocity, position, and evaluate current fitness, update best position

Begin for loop {

Generate unique seed for each thread;

Update best performing particle from previous iteration

If (particle position > max position) position = max position

If (particle position < min position) position = min position

// initial particle-> fitness = INF

If (current_fitness < particle->fitness) particle_fitness = current_fitness

} /* end of for loop for thread */

} /* end of iteration */

4) Find best fitness

Pso_get_best_fitnes_omp()

{

// both array elements can be indentified with thread id

Allocate array of g that contain index of particle associated with each thread //g = tid

Allocate array of best_fitness that contains the best_fitness value of each thread //initialized to INF

#Each OMP thread is responsible for updating the best_fitness and store it in the array at the index = tid

{

Begin for loop

Best_fitness = particle->fitness (initialized to INF)

G = tid

} /* end of multi-threading */

Compare the best_fitness among all threads

} /* end of function */

Results and Discussion:

Comparison of fitness of eggholder, holder table, and booth function

Fitness Value	D	Serial	Thread 1	Thread 4	Thread 8	Thread 16	Particle	Iteration
Booth	2	0	0	0	0	0	100	10000
Holder table	2	-19.2085	-15.1402	-19.2085	-19.2085	-19.2085	100	10000
Eggholder	2	-959.6407	-786.526	-959.641	-959.641	-959.641	100	10000

Multi-threaded version, when compared with its serial counterparts, produces the same result which confirms the correctness of implementation and functionality. However, the result for one thread of holder table and eggholder functions have different values from the reference solution. This is caused by the high value of fitness with only one thread. A simple fix to that is to increase number of iterations and/or number particles when using only one. The result below confirms the suggestion:

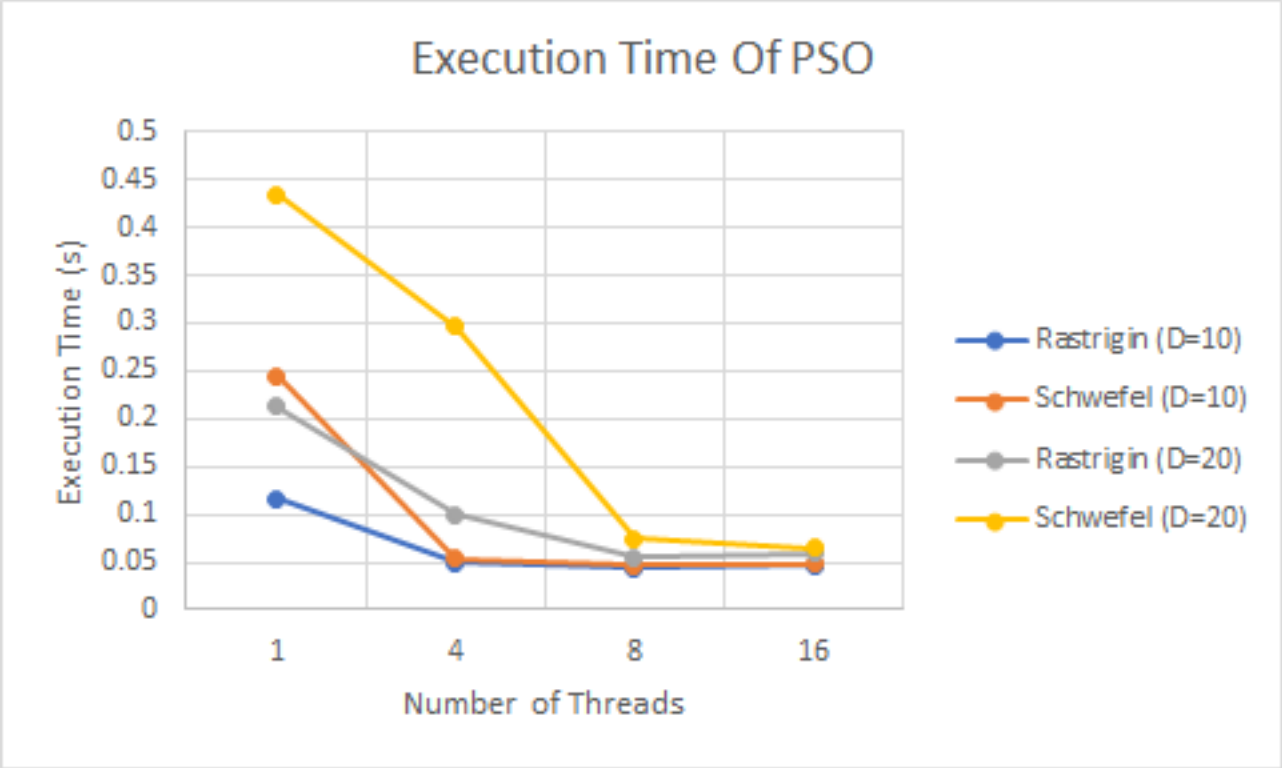
Number of Particles: 1000

```
[thp29@xunil-03 a3]$ ./pso holder_table 2 1000 -10 10 1000 1
position: 8.06 -9.66
velocity: -0.00 -0.00
pbest: 8.06 -9.66
fitness: -19.2085
g: 0
Gold Execution time = 0.309791s
position: 8.06 9.66
velocity: 0.00 0.00
pbest: 8.06 9.66
fitness: -19.2085
g: 6
OMP Execution time = 0.275087s
[thp29@xunil-03 a3]$ ./pso eggholder 2 1000 -512 512 1000 1
position: 512.00 404.23
velocity: 0.00 0.00
pbest: 512.00 404.23
fitness: -959.6407
g: 0
Gold Execution time = 0.690370s
position: 512.00 404.23
velocity: 0.00 -0.00
pbest: 512.00 404.23
fitness: -959.6407
g: 1
OMP Execution time = 0.487538s
```

Comparison of Execution Time

Execution Time	D	Serial	Thread 4	Thread 8	Thread 16	Particle	Iteration
Booth	10	0.190017	0.028952	0.029461	0.039033	100	10000
Holder table	10	0.217214	0.037834	0.039516	0.042946	100	10000
Eggholder	10	0.161797	0.046534	0.040765	0.04233	100	10000
Rastrigin	10	0.11572	0.049758	0.043501	0.046458	100	10000
Schwefel	10	0.244575	0.053026	0.047127	0.048642	100	10000
Execution Time	D	Serial	Thread 4	Thread 8	Thread 16	Particle	Iteration
Booth	20	0.223182	0.043285	0.03712	0.043899	100	10000
Holder table	20	0.257013	0.054924	0.055266	0.06107	100	10000
Eggholder	20	0.364798	0.073163	0.05557	0.050572	100	10000
Rastrigin	20	0.212494	0.099802	0.05504	0.059605	100	10000
Schwefel	20	0.433535	0.295736	0.074056	0.06462	100	10000

(*) Execution time was recorded after multiple runs to account for cache warmup.



(*) Visualization of execution time. (1) denotes the serial version, not 1 thread.

The evaluation will only be performed on Rastrigin and Schwefel function, as indicated by the assignment handout. Though, the execution time for first three functions will be provided for reference.
The execution time is improved for multi-threaded version comparing to the serial one. However, when D= 10, as the number of threads get larger than 8, the overhead from thread creation trumps the benefits for a small dimension size. With D= 20, this is not a problem; in fact, as thread increases to 16, the execution time vastly increases.

Conclusion

With smaller dimension size (< 20), the program benefits from a shorter execution time with smaller number of threads approach. Diminishing returns are present, although only observed within D=10 or smaller: when number of threads are larger than 8, the execution time is longer for parallelization method of < 8 threads. On the other hand, as the size grows, implementation of parallelization allows the programs to run faster. As observed in the execution time graph, the time is significantly less comparing to the serial version.