Particle Swarm Optimization

Prof. Naga Kandasamy
ECE Department, Drexel University

Develop a parallel version of a particle swarm optimizer using OpenMP. The problem is due May 22, 2023, by 11:59 pm. You may work on this problem in a team of up to two people. Please see the submission instructions later on in the document. Submit original code.

Particle swarm optimization (*PSO*) belongs to the class of evolutionary computational techniques that have been developed to optimize nonlinear functions.[1,2] It is a population-based search algorithm that solves the optimization problem by initializing a population of random solutions called *particles* over the desired search space; and then by moving these particles around the search space in iterative fashion until the best solution is found. Particles fly through the search space with *velocities* which are dynamically adjusted using simple mathematical formulae according to their historical behavior. Each particle's movement is influenced by its local best-known position but is also guided toward the best-known positions within the search space, which are updated as better positions are found by other particles. Therefore, particles tend to *swarm* toward the best solutions over the course of the search process.

PSO is an extremely simple algorithm that is quite effective for optimizing a range of functions. It is a *metaheuristic* in that it may provide a sufficiently good solution to the underlying optimization problem. PSO makes no assumptions about the structure of the problem and can search very large spaces of candidate solutions. Due to its metaheuristic nature, however, PSO does not guarantee that an optimal solution will be found. Also, PSO does not use the gradient of the function being optimized to guide its search, meaning that the function need not be differentiable as is required by classic optimization methods such as gradient descent and quasi-newton methods.

Consider N particles spread over a D-dimensional search space. The state of the i^{th} particle is given by its position within this search space, X_i, as well as by the rate of change of this position or velocity V_i:

$$X_i = \{x_{i1}, x_{i2}, \ldots, x_{iD}\}$$
$$V_i = \{v_{i1}, v_{i2}, \ldots, v_{iD}\}$$

The vector

$$PBEST_i = \{p_{i1}, p_{i2}, \ldots, p_{iD}\}$$

[1] Y. Shi, "Particle Swatm Optimization," *IEEE connections*, vol. 2, no. 1, pp. 8–13, February 2004.

[2] J. Kennedy and R. Eberhart, "Particle swarm optimization," *Proc. Int'l Conf. Neural Networks* (*ICNN*), vol. 4, pp. 1942–1948, 1995.

represents the best previous position of the particle, that is, the position that resulted in the best *fitness* value, and g denotes the index of the best particle, in terms of fitness, among all the particles in the population. Then, the basic PSO equations are

$$v_{id} = v_{id} + c_1 rand()(p_{id} - x_{id}) + c_2 rand()(p_{gd} - x_{id}) \quad (1)$$

$$x_{id} = x_{id} + v_{id} \quad (2)$$

where c_1 and c_2 are positive constants, and the *rand()* function generates a value in the range $[0, 1]$. The equation that dynamically updates the velocity vector of a particle consists of three parts:

- *Momentum*: The velocity cannot be changed abruptly; any changes must be made from the current value.
- *Cognitive*: A particle learns from its private experience flying around the search space by remembering the location $PBEST$ within the space that provided the best fitness value.
- *Social*: This part represents the collaboration among particles, that is, learning from the group's flying experience. Particles keep track of the index, g, of the best performing particle at all times.

Returning to (1), if the velocity along any dimension exceeds a user-specified maximum value $\pm V_{\max}$, then it is clamped to $-V_{\max}$ on the negative side and to $+V\max$ on the positive side. If $V_{\max}$ is large, particles may fly far away from good solution areas whereas if $V_{\max}$ is chosen to be too small, they may be trapped within local minima, unable to fly into better solution areas. Our implementation uses a fixed value for $V_{\max}$. However, dynamically changing $V_{\max}$ has the potential to improve PSO performance, especially as we get closer to a good solution.[3]

The structure of the swarm is defined as follows in our code. The position and velocity vectors X and V, respectively, are defined for each particle, as are $PBEST$ and g.

```
typedef struct particle_s { /* Structure of a particle */
    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;

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

swarm_t *swarm;
```

[3]X. Cai and Y. Tan, "A study on the effect of vmax in particle swarm optimisation with high dimension," *Int'l Journal Bio-Inspired Computation,*, vol. 1, pp. 210–216, 2009.

The *pso_init()* function within the *pso_utils.c* source file initializes each particle's state:

- The D-dimensional position vector X is initialized to lie on a random location within the evaluation space for the function to be optimized. Each component of this vector is initialized to a value uniformly distributed within $xmin$ and $xmax$, which are the bounds of the evaluation space (lines 9–11).
- Each of the components in the velocity vector V are initialized to random values uniformly drawn from the distribution $-|xmax - xmin|$ and $|xmax - xmin|$ (lines 13–16).
- The best position, *pbest*, seen by each particle is initialized to its initial location within the search space (lines 18–20).
- The current fitness is evaluated for each particle for the function to be optimized and the index g corresponding to the particle with the best fitness value is stored (lines 27–31).

```
1  /* The pso_init() function */
2  swarm = (swarm_t *)malloc(sizeof(swarm_t));
3  swarm->num_particles = swarm_size;
4  swarm->particle = (particle_t *)malloc(swarm_size*sizeof(particle_t));
5  for (i = 0; i < swarm->num_particles; i++) {
6      particle = &swarm->particle[i];
7      particle->dim = dim;
8      /* Generate random particle position */
9      particle->x = (float *)malloc(dim * sizeof(float));
10     for (j = 0; j < dim; j++)
11        particle->x[j] = uniform(xmin, xmax);
12    /* Generate random particle velocity */
13     particle->v = (float *)malloc(dim * sizeof(float));
14     for (j = 0; j < dim; j++)
15         particle->v[j] = uniform(-fabsf(xmax - xmin),
16                                  fabsf(xmax - xmin));
17     /* Initialize best position for particle */
18     particle->pbest = (float *)malloc(dim * sizeof(float));
19     for (j = 0; j < dim; j++)
20         particle->pbest[j] = particle->x[j];
21     /* Initialize particle fitness */
22     pso_eval_fitness(function, particle, &fitness);
23     particle->fitness = fitness;
24     particle->g = -1;
25 }
26 /* Get index of particle with best fitness */
27 g = pso_get_best_fitness(swarm);
28 for (i = 0; i < swarm->num_particles; i++) {
29     particle = &swarm->particle[i];
30     particle->g = g;
31 }
```

The *pso_solve_gold()* function is shown below. It runs for a user-specified number of iterations before returning the index, g, of the best performing particle.

```
/* The pso_solve() function */
w = 0.79;
c1 = c2 = 1.49;
iter = 0;
g = -1;
while (iter < max_iter) {
    for (i = 0; i < swarm->num_particles; i++) {
        particle = &swarm->particle[i];
        gbest = &swarm->particle[particle->g];
        for (j = 0; j < particle->dim; j++) {
            r1 = (float)rand()/(float)RAND_MAX;
            r2 = (float)rand()/(float)RAND_MAX;
            /* Update particle velocity */
            particle->v[j] = w * particle->v[j]
                    + c1 * r1 * (particle->pbest[j] - particle->x[j])
                    + c2 * r2 * (gbest->x[j] - particle->x[j]);
            if ((particle->v[j] < -fabsf(xmax - xmin))
                 || (particle->v[j] > fabsf(xmax - xmin)))
                     particle->v[j] = uniform(-fabsf(xmax - xmin),
                                              fabsf(xmax - xmin));
            /* Update particle position */
            particle->x[j] = particle->x[j] + particle->v[j];
            if (particle->x[j] > xmax)
                particle->x[j] = xmax;
            if (particle->x[j] < xmin)
                particle->x[j] = xmin;
        } /* Update state for each particle */
        pso_eval_fitness(function, particle, &curr_fitness);
        if (curr_fitness < particle->fitness) { /* Update pbest */
            particle->fitness = curr_fitness;
            for (j = 0; j < particle->dim; j++)
                particle->pbest[j] = particle->x[j];
        }
    } /* Particle loop */
    g = pso_get_best_fitness(swarm); /* Update best particle */
    for (i = 0; i < swarm->num_particles; i++) {
        particle = &swarm->particle[i];
        particle->g = g;
    }
    iter++;
} /* End of iteration */
```

The major steps within each iteration of the outer *while* loop are as follows.

- Update the velocity and position vectors for each particle according to (1) within the *for* loop in lines 10–27. We introduce a new term w called the *inertia weight* within the basic PSO update equations as follows:

$$v_{id} = wv_{id} + c_1 rand()(p_{id} - x_{id}) + c_2 rand()(p_{gd} - x_{id}) \quad (3)$$

$$x_{id} = x_{id} + v_{id} \quad (4)$$

  The inertia weight is introduced to balance between the global and local search abilities of the algorithm: a large value for w facilitates global search whereas smaller values facilitate local search. The introduction of w also eliminates the requirement for carefully setting the maximum velocity of the particles each time PSO is used. In our case, we simply clamp the velocity to lie within the range $\pm|xmax, xmin|$ in lines 17–20. We set $w = 0.79$ for the inertia weight, and $c_1 = c_2 = 1.49$ for the constants based on previously published studies.

- Compare the particle's fitness with its previous best, *pbest*, and if the current value is better than *pbest*, set *pbest* equal to the current value (lines 28–33).

- Identify particle in the neighborhood — we assume globally — with the best success so far, in terms of fitness, and assign its index to g (lines 35–39).

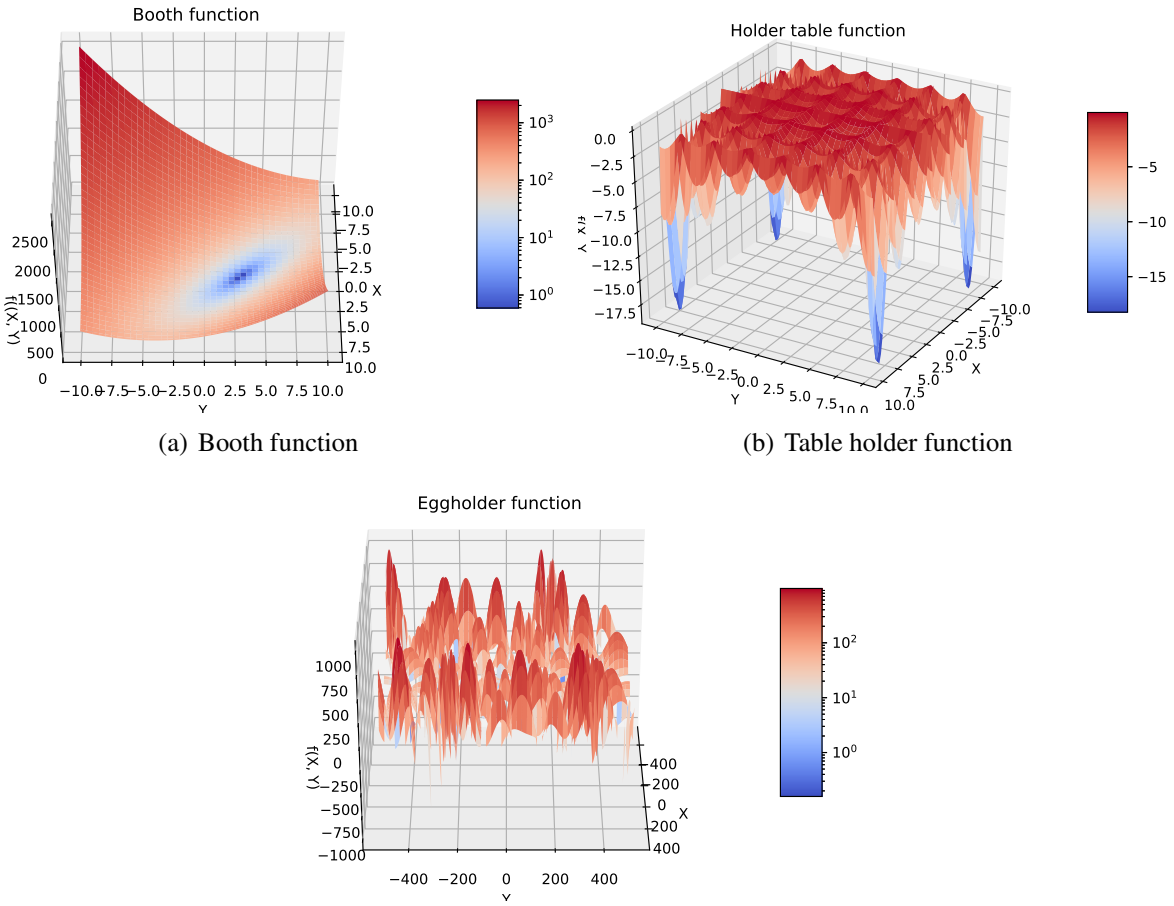
The *optimize_gold()* function is very straightforward to implement. Once the PSO is initialized, it calls *pso_solve_gold()*, which returns the index of the best performing particle after a set number of PSO iterations.

```
srand(time(NULL));
swarm_t *swarm = pso_init(function, dim, swarm_size, xmin, xmax);
int g;
g = pso_solve_gold(function, swarm, xmax, xmin, max_iter);
if (g >= 0) {
    fprintf(stderr, "Solution:\n");
    pso_print_particle(&swarm->particle[g]);
}
pso_free(swarm);
return g;
```

The reference implementation provided to you takes the following command-line parameters:

```
$./pso function-name dimension swarm-size xmin xmax max-iter num-threads
function-name: name of function to optimize
dimension: dimensionality of search space
swarm-size: number of particles in swarm
xmin, xmax: lower and upper bounds on search domain
max-iter: number of iterations to run the optimizer
num-threads: number of threads to create
```

Using the reference implementation as the starting point, edit the *compute_using_omp()* function to complete the PSO functionality using OpenMP. Add other helper functions and data structures as



(a) Booth function

(b) Table holder function

(c) Eggholder function

Fig. 1: Visualization of the Booth, Holder Table, and Eggholder optimization test functions.

necessary to your code. Similar in vein to *optimize_gold()*, your function must also print the final solution to the screen using *pso_print_particle()* prior to returning to the *main()* function.

Some test optimization functions, shown in Fig. 1, have been provided in *pso_utils.c* to test the correctness of your multi-threaded implementation.

- *Booth function.* As visualized in Fig. 1(a), the formula is defined over two dimensions as

$$f(x_1, x_2) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2,$$

where the function is usually evaluated on the square $x_i \in [-10, 10], \forall i = 1, 2$. The function has a global minimum of $f(1, 3) = 0$. Following is the output of the reference implementation when PSO is run using 100 particles for 1000 iterations on the Booth function:

```
./pso booth 2 100 -10 10 1000 1
Solution:
position: 1.00 3.00
velocity: 0.00 -0.00
pbest: 1.00 3.00
```

```
fitness: 0.0000
g: 0
```

- *Holder table function.* The formula, defined over two dimensions, is

$$f(x_1, x_2) = -\left|\sin(x_1)\cos(x_2)\exp\left(\left|1 - \frac{\sqrt{x_1^2 + x_2^2}}{\pi}\right|\right)\right|,$$

evaluated on the square $x_i \in [-10, 10], \forall i = 1, 2$. It has many local minima, with four identical global minima defined at:

$$\begin{aligned} f(8.06, 9.66) &= -19.21, \\ f(-8.06, 9.66) &= -19.21, \\ f(8.06, -9.66) &= -19.21, \\ f(-8.06, -9.66) &= -19.21 \end{aligned}$$

The Holder table function is visualized in Fig. 1(b). Following is a possible output of the reference implementation when PSO is run using 100 particles for 1000 iterations:

```
./pso holder_table 2 100 -10 10 1000 1
Solution:
position: 8.06 9.66
velocity: 0.00 0.00
pbest: 8.06 9.66
fitness: -19.2085
g: 0
```

- *Eggholder function.* It is defined over two dimensions as

$$f(x_1, x_2) = -(x_2 + 47)\sin\left(\sqrt{\left|x_2 + \frac{x_1}{2} + 47\right|}\right) - x_1\sin(\sqrt{|x_1 - (x_2 + 47)|}),$$

evaluated on the square $x_i \in [-512, 512], \forall i = 1, 2$. As seen in Fig. 1(c), the function has a large number of local minima with a global minimum of $f(x_1, x_2) = -959.6407$ occurring at $(512, 404.2319)$. Following is the output of the reference implementation when PSO is run using 100 particles for 1000 iterations:

```
./pso eggholder 2 100 -512 512 1000 1
Solution:
position: 512.00 404.23
velocity: 0.00 -0.00
pbest: 512.00 404.23
fitness: -959.6407
g: 0
```

Uncomment the `SIMPLE_DEBUG` flag in *pso.h* to see the evolution of PSO, in terms of the best performance particle per iteration.

Answer the following questions:

- **(10 points)** Parallelize the PSO update process, specifically the *for* loop between lines 7–34 within the *pso_solve_gold()* function discussed earlier.
- **(5 points)** Staying within *pso_solve()*, during each PSO iteration, the *pso_get_best_fitness()* function in line 35 compares all particles in the swarm to obtain the best performing one. If done in single-threaded fashion, this may represent a potential choke point. Parallelize the *pso_get_best_fitness()* function.
- **(5 points)** Finally, the *pso_init()* function initializes the swarm in single-threaded fashion. Parallelize this function as well.

Once you have parallelized the key PSO functions, test the correctness of your multi-threaded code using the Booth, Holder table, and Eggholder functions, for various combinations of particles, PSO iterations, and threads.

Report the execution time achieved by your multi-threaded version for 4, 8, and 16 threads, on `xunil` for the following many-dimensional functions:

- *Rastrigin function.* It is defined as

$$f(\mathbf{x}) = A * D + \sum_{i}^{D} (x_i^2 - A * cos(2 * \pi * x_i)),$$

  evaluated on the square $x_i \in [-5.12, 5.12], \forall i = 1, 2, \ldots, D$, where $A = 10$ and D denotes the dimensionality. The function has a global minimum of $f(x) = 0$ at $x = 0$.

- *Schwefel function.* It is defined as

$$f(\mathbf{x}) = 418.9829 * D - \sum_{i=1}^{D} x_i sin(sqrt(abs(x_i))),$$

  evaluated over the square $x_i \in [-500, 500], \forall i = 1, 2, \ldots, D$, where D is the dimensionality. The function has many local minima and the global minimum is $f(x) = 0$ at $(420.97, \ldots, 420.97)$.

For each of the above functions, fix the number of dimensions to $D = 10$ and $D = 20$, respectively; and for each value of D, tune both the number of particles and PSO iterations appropriately to achieve the best results possible in terms of the final fitness functions. Set the upper limit for both the number of particles as well as iterations to 10,000.

Note. Comment out the `DEBUG_SIMPLE` flag in *pso.h* prior to running the timing tests since the system calls enabling the *fprintf()* and *printf()* functions incur high overhead.

Submission Instructions

Once you have implemented all of the required features described in this document submit your code by doing the following:

- Run `make clean` in your source directory. We must be able to build your code from source and we don't want pre-compiled executables or intermediate object files.
- Zip up your code, including complete instructions in a separate README file on how to compile and run your program on the *xunil* cluster.
- Name the zip file *abc123_pso.zip*, where *abc123* is your Drexel ID.
- Upload your zip file using the BBLearn submission link found on the course website.
- Provide a short report describing the parallelization process, using code or pseudocode to help the discussion.
- If you are working as a group of two, one submission will suffice. Please indicate the two authors in the cover page of your report.