Case Study: Implementing the Particle Swarm Optimization algorithm and comparison with other metaheuristics

Wilbert Pumacay, *Catholic University San Pablo*, wilbert.pumacay@ucsp.edu.pe
Gerson Vizcarra, *Catholic University San Pablo*, gerson.vizcarra@ucsp.edu.pe

Abstract

Heuristic-based swarm algorithms emerged as a powerful family of optimization techniques, inspired by the collective behavior of social animals. Particle Swarm Optimization (PSO) , part of this family, is known to solve large-scale nonlinear optimization problems using particles as a set of candidate solutions. In this paper we test the PSO on 3 common benchmarks functions, present a parallel implementation of the algorithm, and compare it with the results from other metaheuristics using the Wilcoxon rank sum test.

Index Terms

Metaheuristics, Particle Swarm Optimization, Convergence Behavior, CUDA.

I. Introduction

THE Particle Swarm Optimization algorithm is a population-based optimization method. It is based on the social behavior of birds and fish when moving as a group and it has been applied to many areas, such as artificial neural network training, function optimization, fuzzy control, and pattern classification because of its ease of implementation and fast convergence to acceptable solutions.

II. Particle Swarm Optimization (PSO)

A. Basic concepts

Particle Swarm Optimization, first introduced by Kennedy and Eberhart [1] is a stochastic optimization technique that is based on two fundamental disciplines [3]: social science and computer science. In addition, PSO uses the swarm intelligence concept: the collective behavior of unsophisticated agents that are interacting locally with their environment to create coherent global functional patterns.

B. Description of the algorithm

PSO is a population based optimization technique, where the population is called a *swarm*. Earch particle represents a posible solution to the optimization task at hand. During each iteration each particle accelerates in the direction of its own personal best solution found so far, as well as the direction of the global best solution discovered so far by any of the particles in the swarm, as described in Fig. 1. This means that if a particle found a promising new solution, the other particles will move closer to it. The whole algorithm can be described in Algorithm 1.

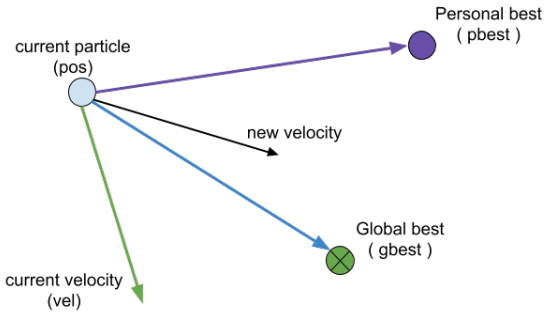

Fig. 1: PSO overview

Algorithm 1 Particle Swarm Optimization Algorithm

In : *objFunction, populationSize, w,* c_1, c_2, k
Out : *Solution optima* $P_{g.cost}$
$P_{g.pos} = \mathbf{0}$
$P_{g.cost} = Inf$
$particles =$ **CreateParticles()**
while *!stopCondition()* **do**
  for p **in** $particles$ **do**
    UpdateParticlesVelocities()
    UpdateParticlesPositions()
    UpdatePersonalBest()
    UpdateGlobalBest()
return $P_{g.cost}$

Each individual particle has the following attributes:

- A current position in search space x.
- A current velocity v.
- A personal best position in the search space x^{pbest}.

As desribed earlier, each particle updates its velocity according to its personal best and the global best. The update rule (based in the social behaviour mentioned earlier) can be described with the following equation:

$$v \leftarrow wv + c_1 r_1[x^{pbest} - x] + c_2 r_2[x^{gbest} - x] \tag{1}$$

Where w (inertia coefficient), c_1 (cognitive coefficient), c_2 (social coefficient) and k are the hyperparameters of the algorithm; and r_1 and r_2 are random numbers in the range $[0, 1]$. We then just update the particle position with the just calculated velocity.

$$x \leftarrow x + v. \tag{2}$$

The personal best position of each particle, x^{pbest} is updated using the following equation :

$$x^{pbest} \leftarrow x, \quad if\ Cost(x) \leq Cost(x^{pbest}) \tag{3}$$

and the global best solution found by any particle during all previous steps, x^{gbest}, is defined as :

$$x^{gbest} = arg\min_{x}(Cost(x)) \tag{4}$$

C. Implementation details

The algorithm is quite easy to implement, and that is one reason why it is still used. We implemented both a serial and a parallel version of the algorithm and test it using manually tuned hyperparameters. The best configuration that work in our case was :

$$w = 1.0, c_1 = 2.0, c_2 = 2.0, k = 0.5, populationSize = 100000 \tag{5}$$

For the CPU implementation we followed a simple variation of the asynchronous PSO algorithm, which is described in Algorithm 2; and for the GPU implementation we followed the same variation of the PSO algorithm in a synchronous way, by making the comparison with the global best outside of the particles update loop. Both implementations are shown in figures 2 and 3.

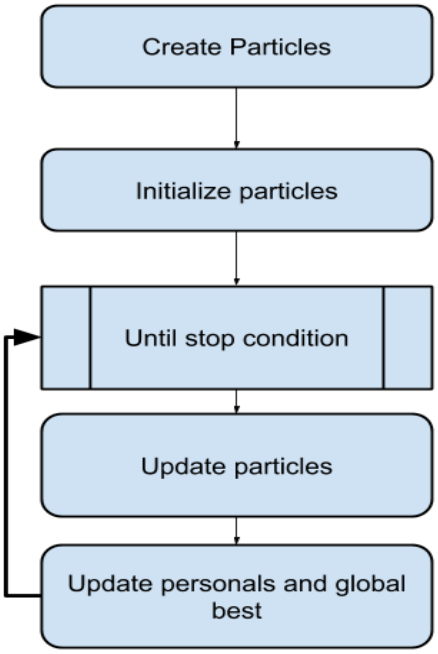

Fig. 2: PSO cpu implementation

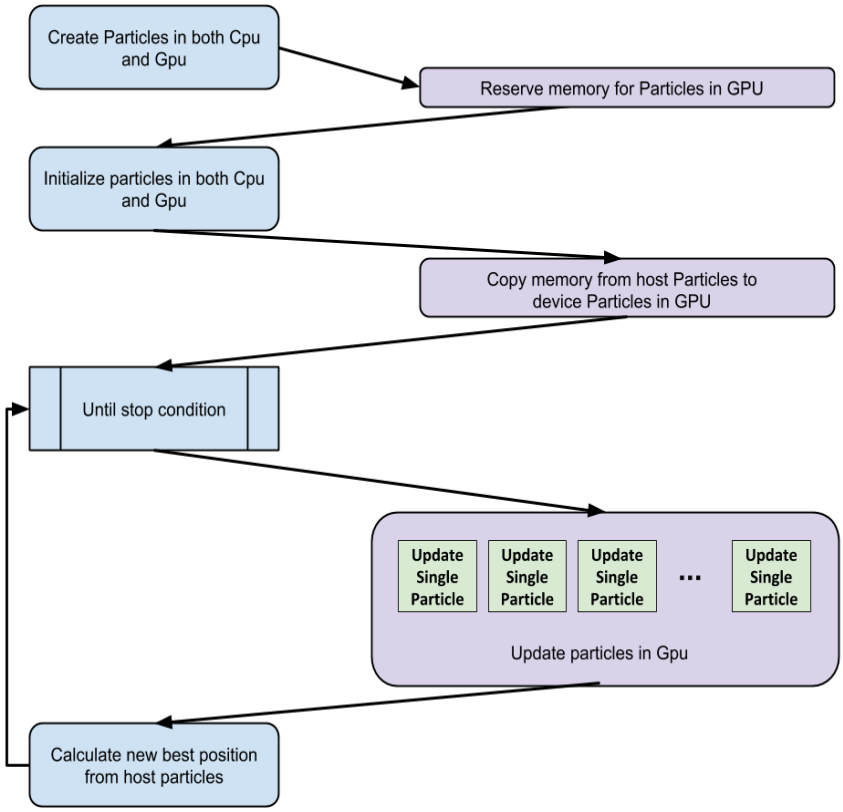

Fig. 3: PSO gpu implementation

Algorithm 2 Particle Swarm Optimization - Asynchonous Serial Version

In : *objFunction, populationSize, w,* c_1, c_2, *k, dimensions, domain*
Out : *Solution optima* $P_{g.cost}$
$vMin = -k * (domain.max - domain.min)/2.0$
$vMax = k * (domain.max - domain.min)/2.0$
$P_{g.pos} =$ *zeros(dimensions)* ▷ Position that gives best cost
$P_{g.cost} = Inf$ ▷ Best cost so far

$Particles = \{\}$ ▷ Empty array of particles
for $i = 1$ **to** $PopulationSize$ **do** ▷ Particles Initialization
    $p =$ **Particle()**
    $p.pos =$ **randUniform(domain)**
    $p.vel =$ **zeros(dimensions)**
    $p.cost =$ **objFunction(p.pos)**
    $p.bestpos = p.pos$
    $p.bestcost = p.cost$
    if $p.cost \leq P_{g.cost}$ **then**
        $P_{g.cost} = p.cost$
        $P_{g.pos} = p.pos$

while *!stopCondition()* **do** ▷ Optimization process
    for p **in** $Particles$ **do**
        $p.vel = w * p.vel + c_1 * (p.bestpos - p.pos) + c_2 * (P_{g.pos} - p.pos)$ ▷ Velocity update
        $p.vel =$ **ClampVector**(*p.vel, vMin, vMax*)
        $p.pos = p.pos + p.vel$ ▷ Position update
        $p.pos =$ **ClipPosition**(*p.pos, domain*)

        if $p.cost \leq p.bestcost$ **then**
            $p.bestcost = p.cost$ ▷ Update personal best
            $p.bestpos = p.pos$
            if $p.cost \leq P_{g.cost}$ **then**
                $P_{g.cost} = p.cost$ ▷ Update global best
                $P_{g.pos} = p.pos$
return $P_{g.cost}$

Algorithm 3 Particle Swarm Optimization - Synchonous Parallel Gpu Version

In : *objFunction, populationSize, w,* c_1, c_2, *k, dimensions, domain*
Out : *Solution optima* $P_{g.cost}$
$vMin = -k * (domain.max - domain.min)/2.0$
$vMax = k * (domain.max - domain.min)/2.0$
$P_{g.pos} =$ *zeros(dimensions)* ▷ Position that gives best cost
$P_{g.cost} = Inf$ ▷ Best cost so far

$hostParticles = \{\}$
$deviceParticles = \{\}$
GpuCreateParticles(*hostParticles, deviceParticles, objFunction, populationSize, dimensions, domain*)
GpuInitParticles(*hostParticles, deviceParticles, objFunction*)

while *!stopCondition()* **do** ▷ Optimization process
    GpuUpdateParticles(*hostParticles, deviceParticles, w,* c_1, c_2, *k*)s
    for p **in** $hostParticles$ **do**
        if $p.cost \leq P_{g.cost}$ **then**
            $P_{g.cost} = p.cost$ ▷ Update global best
            $P_{g.pos} = p.pos$
return $P_{g.cost}$

Algorithm 4 Particle Swarm Optimization - GpuUpdateParticles

In : *deviceParticles, coreIndx, w,* c_1, c_2, *k,* P_g
$p =$ **getDeviceParticle**(*coreIndx*)
$p.vel = w * p.vel + c_1 * (p.bestpos - p.pos) + c_2 * (P_{g.pos} - p.pos)$ ▷ Velocity update
$p.vel =$ **ClampVector**(*p.vel, vMin, vMax*)
$p.pos = p.pos + p.vel$ ▷ Position update
$p.pos =$ **ClipPosition**(*p.pos, domain*)

if $p.cost \leq p.bestcost$ **then**
    $p.bestcost = p.cost$ ▷ Update personal best
    $p.bestpos = p.pos$

III. Results

To test our implementation, we used the following benchmark functions :

1) **Ackley**: This is a function with a minimum global optima at $\mathbf{X} = (0, 0, \ldots)$ with cost of 0.0.

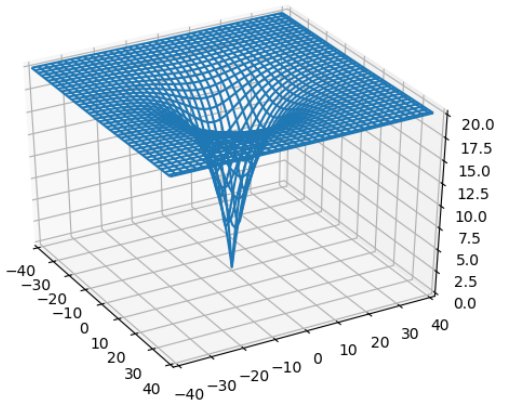

Fig. 4: Ackley benchmark function

2) **Schwefel**: This is a function with a minimum global optima at $\mathbf{X} = (420.9687, 420.9687, \ldots)$ with cost of 0.0.

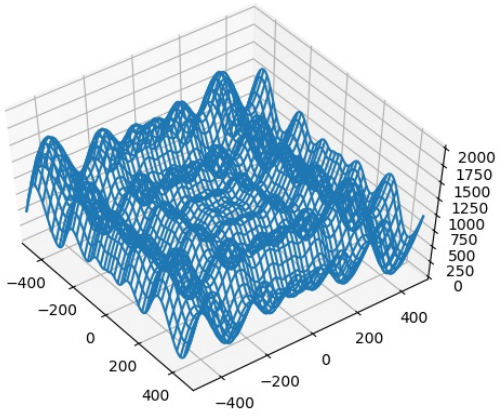

Fig. 5: Schwefel benchmark function

3) **Schaffer6**: This is a function with a maximum optima at $\mathbf{X} = (0, 0, \ldots)$ with cost of 1.0.

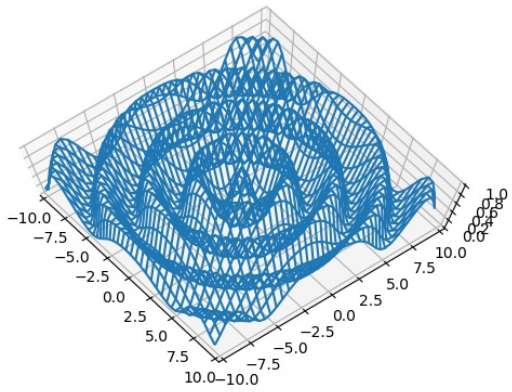

Fig. 6: Schaffer6 benchmark function

A. *Convergence behaviour*

We tested the implementation for both 2 and 8 dimensions of the search space. In the case of 2 dimensions, we plotted the results in the surface and contour curves, and got the following behaviour:

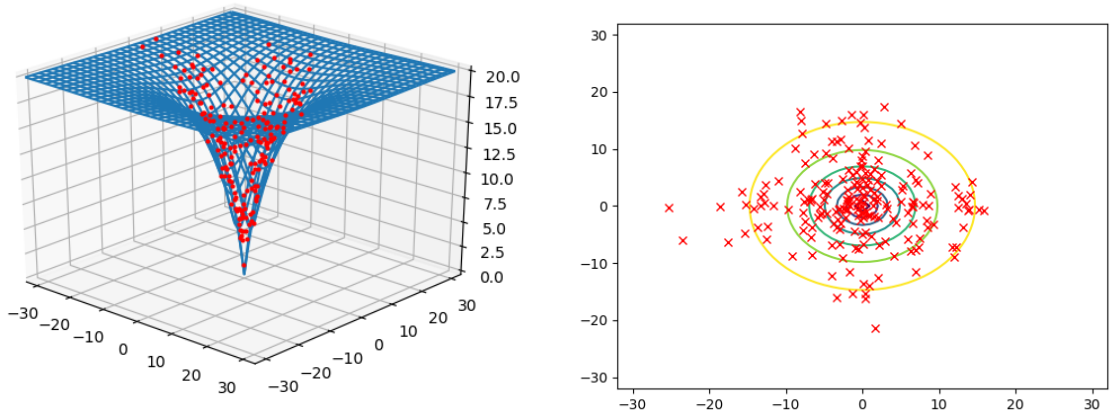

Fig. 7: Particles behaviour for the Ackley benchmark functions

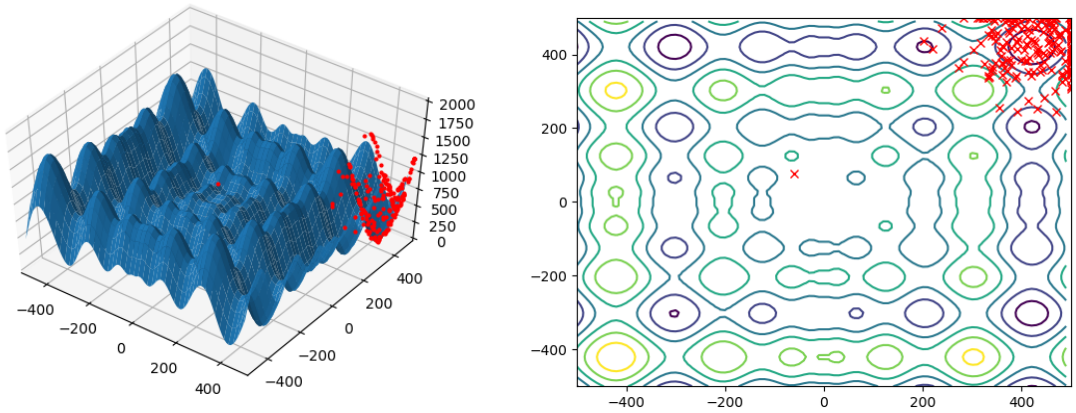

Fig. 8: Particles behaviour for the Schwefel benchmark functions

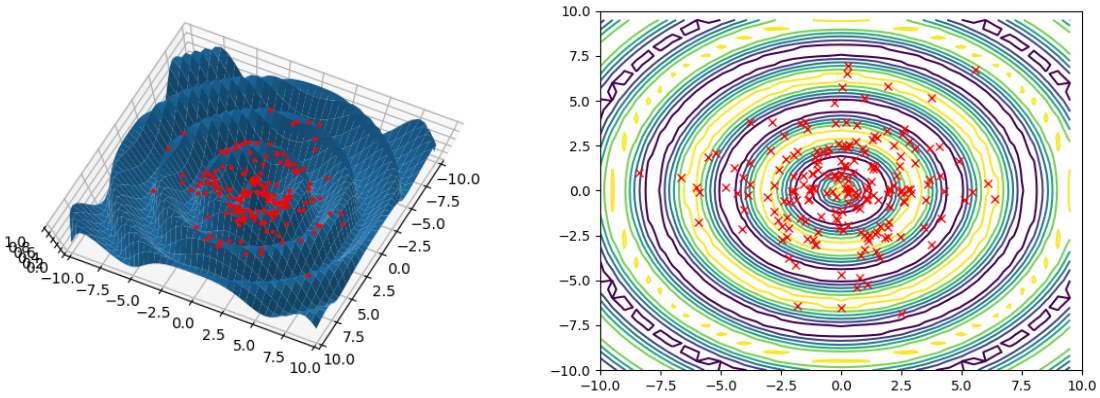

Fig. 9: Particles behaviour for the Schaffer6 benchmark functions

Also, we plotted the optima at each iteration for both 2d and 8d, and got the following results :

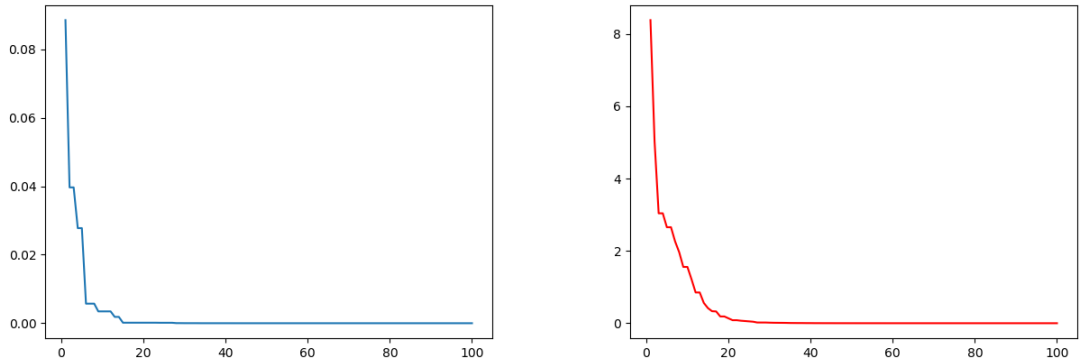

Fig. 10: Convergence behaviour during optimization in the Ackley function. Left: 2d, Right: 8d

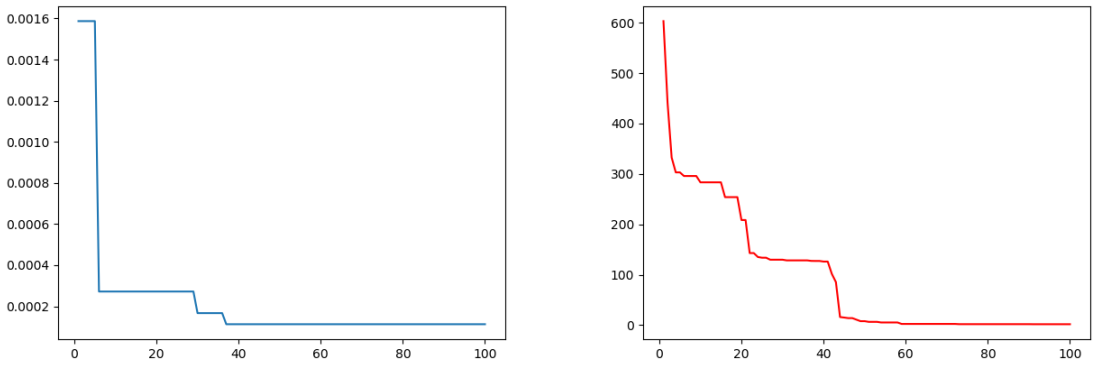

Fig. 11: Convergence behaviour during optimization in the Schwefel function. Left: 2d, Right: 8d

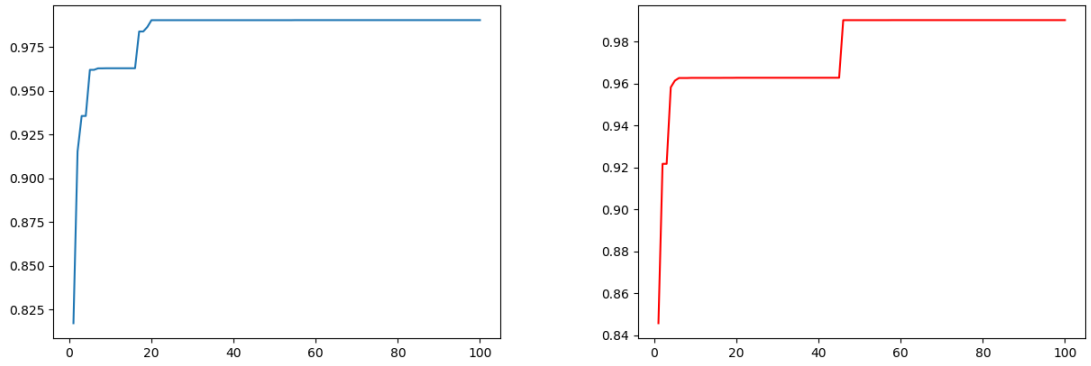

Fig. 12: Convergence behaviour during optimization in the Schaffer function. Left: 2d, Right: 8d

B. *Comparison with other metaheuristics*

We run the Wilcoxon test to compare our implementation with the following metaheuristics :

- **CS** Cuckoo Search algorithm
- **BA** Bat Inspired algorithm
- **GC** GENOCOP algorithm
- **FPA** Flower Pollination algorithm
- **DE** Dynamic Differential Evolution algorithm
- **GSA** Gravitational Search algorithm
- **HS** Harmony Search algorithm
- **GWO** Grey Wolf Optimization algorithm

The **p-val** found by running the Wilcoxon test in **Octave** (with a vector of 100 executions) can be found in the following table.

Algoritmo	Ackely 2d	Ackely 8d	Schewfel 2d	Schewfel 8d	Schaffer6 2d	Schaffer6 8d
CS	3.89656e-18	3.89656e-18	0	0	3.45096e-16	0.00393593
BA	0	3.89656e-18	4.01616e-18	0.0114986	1.78129e-17	7.69167e-10
GC	3.89656e-18	3.89656e-18	0	0	0	3.89656e-18
FPA	3.89656e-18	3.89656e-18	0	0	3.45096e-16	3.89656e-18
DE	3.89656e-18	2.61518e-12	0	0	1.1236e-11	3.89656e-18
GSA	3.89656e-18	3.89656e-18	2.28706e-14	9.86809e-16	0	0
HS	3.89656e-18	3.89656e-18	3.36398e-13	2.14496e-10	0	0
GWO	3.89656e-18	3.89656e-18	7.5806e-06	3.89656e-18	0	0

IV. Conclusions

- The algorithm works quite fast in finding an optima. As the results suggest, the algorithm finds a local optima quite fast, but it gets stuck in that optima, as the other particles are attracted to that optima. Once that optima is found, the exploration is restricted to the neighborhood of that optima, which does not allow to escape it in cases of complicated objective functions.
- One way to escape the optima is by using more particles, which was an option in our CUDA implementation. This allows us to search the space in order to escape local optima, as was the case of the Schwefel function, in which we got stuck because of lack of a bigger population size in our first test when tuning the hyperparameters.

References

[1] James Kennedy and Russell Eberhart.
Particle Swarm Optimization. - 1995

[2] Simon Garnier, Jacques Gautrais, Guy Theraulaz
The biological principles of swarm intelligence. - 2007

[3] Yamille del Valle, Ganesh Kumar, Salman Mohagheghi, Jean Hernandez, Ronald Harley
Particle Swarm Optimization: Basic Concepts, Variants and Applications in Power Systems. - 2008