Particle Swarm Optimization using Spark framework

Abishek Kumar[1], Megha Jayakumar[1], Ravikant Tyagi[1]

s6abkuma@uni-bonn.de, s6mejaya@uni-bonn.de, s6ratyag@uni-bonn.de

Supervisors: Dr.Hajira Jabeen, Gëzim Sejdiu

Informatik II, Universität Bonn, Germany

Abstract

This paper presents a detailed report on the implementation of Particle Swarm Optimization (PSO) algorithm utilizing the concept of parallel programming using spark RDD. Evaluation of the implementation has been done using various metrics such as number of cores and workers in a cluster, particle size, dimension and also by providing a comparison between a sequential and distributed implementation of PSO.

1 Introduction

Swarm Intelligence (SI) is an artificial intelligence based on collective behavior of decentralized, self organized systems. SI systems generally consists of a population of simple agents interacting locally with one another and with their environment[1]. The natural examples include bird flocking, ant colonies, fish schooling etc. Particle swarm optimization is a part of swarm intelligence.

1.1 Particle Swarm Optimization

Proposed by James Kennedy and Russel Eberhart in 1995, particle swarm optimization is a computational method that optimizes a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality. It consists of a number of agents which is referred to as "particles" that forms a swarm moving continuously in the search space looking for the best solution. The position of the particles are adjusted in every step considering its own experience and that of its peers towards an optimum solution.

for *each particle i = 1, ..., S* **do**

Initialize the particle's position with a uniformly distributed random vector: xi ˜ U(blo, bup)

Initialize the particle's best known position to its initial position: $p_i = x_i$

if $f(p_i) < f(g)$ **then**

update the swarm's best known position: g = pi

end

Initialize the particle's velocity: vi ˜ U(-|bup-blo|, |bup-blo|)

end

while *a termination criterion is not met* **do**

for *each particle i = 1, ..., S do* **do**

for *each dimension d = 1, ..., n do* **do**

Pick random numbers: $r_p, r_g \tilde{U}(0, 1)$

Update the particle's velocity: $v_{i,d} = \omega v_{i,d} + \phi prp(p_{i,d} - x_{i,d}) + \phi_g r_g (g_d - x_{i,d})$

end

Update the particle's position: $x_i = x_i + v_i$

if $f(x_i) < f(p_i)$ **then**

Update the particle's best known position: $p_i = x_i$

if then

update the swarm's best known position: $g = p_i$

end

end

end

end

2 Approach

Optimizing the implementation of PSO algorithm using the concept of parallel programming using spark RDD. Each particle is modelled using a Scala class and a swarm is created using multiple instances of this class processed into an RDD. Update functionality of the particles is done in parallel. After particles are updated for a certain number of times in the worker node, the global best values from all the worker nodes are collected at the driver node and the best global-best value is determined, after which it is broadcast to the worker nodes. The process is carried out until stop criteria is met.

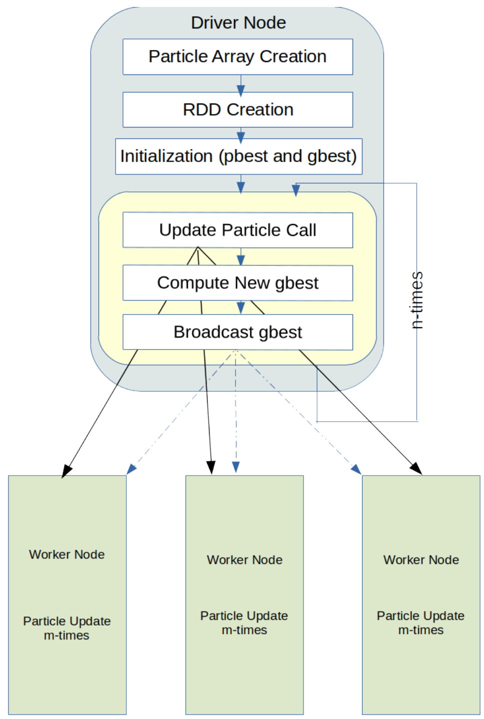

Figure 1: Work-flow of the project

3 Implementation

The below sections cover the steps involved in the development of our project.

3.1 Creation of particle class

The class *Particle* consists of the properties *p_id*, *p_position*, *p_velocity*, *p_best*, *particle_fitness* and *p_best_fitness* which stands for the particle's id, particle's current position, particle's velocity, particle's best known position, particle's current fitness and particles best fitness respectively. The properties except particle's id are declared as an array and initialized with random values.

3.2 Main method object

Main method object named *PSO_RDD* consists of the variable declarations for *dimension*, number of particles (*no_of_particles*), *g_best* and number of iterations (external and internal) (*no_of_iteration_internal*, *no_of_iteration_external*). It consists of the main method where all the computational logic of our project is included.

3.2.1 Creation of an array buffer of the type *Particle*

An array buffer *swarm* of the type *Particle* is created and it is initialized by invoking the *Particle* class iteratively based on the number of particles (*no_of_particles*) and by passing the *dimension* and iteration index i as the *p_id* (particle id).

3.2.2 RDD creation

The array buffer *swarm* of the type particle consists of the entire particle data and this is transformed into an RDD *swarm_rdd*

var swarm_rdd = sc.parallelize(swarm,optional:number of partition)

3.2.3 Initializing *p_best*, *g_best* and Broadcast *g_best*

The initial value of *p_best* at the start of the program is set to *p_position*. The *init_pbest* method is invoked for the same. *p_fitness* and *pbest_fitness* are computed. The data of each particle is passed to the method *global_best_position* which computes the best position among all the particles in the swarm. Here we compare the result of the *pbest_fitness* with the *g_best*'s fitness and assign the position that returns the minimum value into *g_best*. The *g_best* is delivered to all workers using *sc.broadcast()*.

3.2.4 Particle updates upto the *no_of_iteration_external*

The *update_particle* method is invoked *no_of_iteration_external* times using the .map() for each element of RDD. Initially Broadcast-ed *g_best* is stored locally and the following computations are performed for each of the particles *no_of_iteration_internal* times inside the function.

1. Velocity update:
   For each *dimension* update the particle velocity *p_velocity* as per the velocity update formula of PSO algorithm given in section 1.1. The velocity update implementation of our project is shown below. C1 and C2 are accelerator coefficients. C1 value gives the importance of personal best value and C2 is the importance of social best value. W is an inertial parameter. This parameter affects the movement propagation given by the last velocity value.[3]

   ```
   for i = 0 to dimension-1 do
       var toward_pbest = c1*math.random*(p.p_best(i) - p.p_position(i))
       var toward_gbest = c2*math.random*(g_best(i) - p.p_position(i))
       p.p_velocity(i) = w*p.p_velocity(i) + toward_pbest + toward_gbest
   end
   ```

2. Position update:
   With the obtained velocity value from the above section, the current position *(p_position)* is updated as shown below.

for *i = 0 to dimension-1* **do**
| p.p_position(i) = p.p_position(i) + p.p_velocity(i)
end

3. Fitness value is computed for the new position.

4. Global-best and particle-best update:
   New *p_best* position as well as its fitness value is computed and then *g_best* position is updated as shown below.

   /* p_best update */
   if *p.p_fitness <p.pbest_fitness* **then**
   | p.p_best = p.p_position
   end
   p.compute_pbest_fitness()
   /* g_best update */
   if *p.pbest_fitness <obj_func(g_best)* **then**
   | g_best=p.p_position
   end

3.2.5 *g_best* computation and broadcasting it to other workers

The *g_best* value is computed by passing the *p_best* of all the particles and computing the value which results in the minimum value for objective function. Again the value is sent to all the workers by using the broadcast variable.

4 Evaluation

4.1 Parallel PSO using spark RDD

Figure 2 shows the event time-line from the spark history server of our implementation. From this we can infer that the code utilizes all the workers and cores in parallel for computation.

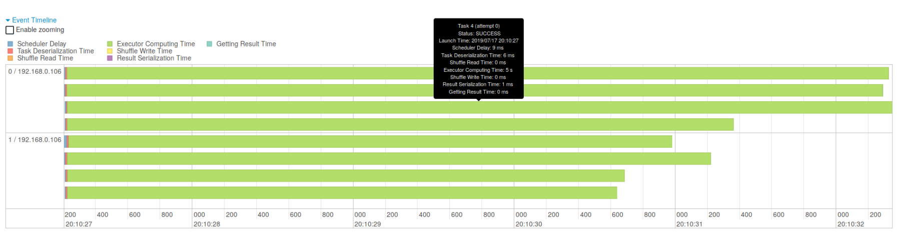

Figure 2: Event time-line from spark history server

4.2 Objective functions used for the experiment

The objective functions used for the evaluation are shown in table 1 and the results for the same are shown in table 2. Both the functions are run using the same configuration: 2 dimensions, 1000 particles, 2 workers and 4 cores.

Table 1: Objective functions

f1	Sphere Function	$F_1 = \sum_{i=1}^{n} x_i^2$
f2	Rosenbrock Function	$F_1 = \sum_{i=1}^{n-1}(100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2)$

Table 2: Results for the objective functions

Sphere Function	**Rosenbrock Function**
0.270595241250681	77.4439212938086
3.66134535616401E-41	0.0294075671486228
1.65302625444563E-105	0.0100001390838247
2.47297441789503E-172	0.00598758282330703
8.44135864687224E-236	0.00373256775651358
5.85972454378006E-306	0.00238114618131031
0	0.00164916014729307
0	0.00112468629145819
0	0.000722519360102357
0	0.000476493799876262

4.3 Performance evaluation based on the number of workers and cores

The number of workers and cores were modified keeping the *dimension* (15), *no_of_particles*(1000) and *no_of_iterations* constant for all the cases and the results were compared. The objective function used for this evaluation is the sphere function. From the graphs in figure 3 (utilizing the data from table 3), it can be inferred that increasing the number of cores and workers results in faster convergence of the objective function to the minimum value. Another observation made in this scenario was on the time taken for computation which is shown in table 4. With an increase in the number of workers the execution time of the program is very less with good convergence results where as, for an increase in the number of cores, the results obtained are better but with a compromise on the execution time.

Table 3: Convergence to minimum values for different number of workers and cores

1 worker-8 cores	1 worker-2 cores	3 workers-2 core
6.43224766823076	5.86582338945573	4.92267294874303
0.0000120796655538064	0.808820483963611	0.771122703788966
1.11720053602022E-15	0.115227468662645	0.0361475398385379
4.82461168175844E-22	0.0186535300869684	0.00339068382992434
2.47194687946337E-30	0.00216894471778122	0.0000857443982413093
2.00488955029556E-38	0.000218739706491391	8.63627877064445E-06
2.7014621010484E-45	0.0000295488780515095	3.0640308540069E-07
3.86790217028661E-54	3.24417157883386E-06	7.45412516431952E-09
2.08077724176654E-61	5.49201217198881E-07	4.06409407405483E-10
3.60716295590847E-69	8.64113101369839E-08	3.36328483119157E-11

Table 4: Time taken for the analysis

	Time taken
1 worker - 8 cores	7.2 minutes
1 worker - 2 cores	2.9 minutes
3 worker - 2 cores	59 seconds

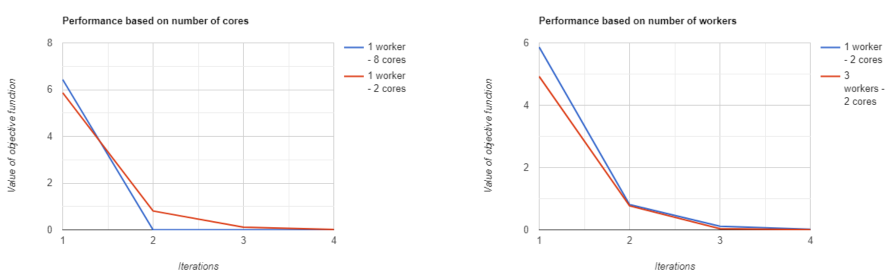

Figure 3: Convergence to the minimum value for sphere function based on the number of cores and workers

4.4 Performance based on the number of particles

With the increase in the *no_of_particles* the results showed better convergence. The implementation was run with a particle size of 250, 500 and 2500 for the sphere function of *dimension* 25. For 2 workers and 4 cores the results are shown in table 5.

Table 5: Convergence to minimum value for different number of particles

250 Particles	**500 Particles**	**2500 Particles**
9.03795132284362	7.9793782064223135	4.894248722807026
0.10065775537379154	0.16833220778733726	0.012479417827545781
0.001020808866580307	1.8856770152304498E-4	1.0316646133844685E-6
8.605743826154334E-6	7.320664696076117E-7	9.679724605155795E-11
7.0105177140880885E-9	2.8681249695061763E-10	1.1274546780082091E-14
4.5378585829206145E-11	5.3883814410184036E-15	4.091474202632229E-17
4.84272852360384E-14	1.0176139269258964E-17	1.4157219516553926E-19
4.2353911902758717E-16	2.9719640852213652E-19	1.1332352536148681E-24
6.578168941195027E-19	5.469940248677021E-22	7.345929102664538E-29
9.892366200088468E-21	1.4530207641414299E-24	1.6432761871346766E-31

4.5 Performance comparison between PSO using scala and PSO using Spark RDD

By comparing the overall performance of PSO using scala and PSO using spark, the later gave better results by good convergence to the minimum value. But for smaller dimension functions, the PSO using scala took less execution time and gave good results but did not converge to an exact minimum value unlike PSO using spark RDD. For dimensions higher than 20, even with a high particle size, PSO using scala did not give good results.

Table 6: Time taken by PSO using scala

	Time taken by PSO using scala
10 dimension - 250 particles - 10000 iterations	11 seconds
15 dimension - 250 particles - 10000 iterations	15 seconds

5 Project Time-line

Week	**Activity**	**Person Responsible**
Week 1	Research upon Particle Swarm Optimization algorithm. Learning phase of Scala and Spark	Abishek, Megha, Ravikant
Week 2	Understanding the working of RDD, analyzing approach and preparation for the first presentation	Abishek, Megha, Ravikant
Week 3	Required library setups and installation	Abishek, Megha, Ravikant
Week 3-4	Defining and initialization of particles and the update particle functionality	Abishek, Megha, Ravikant
Week 5-6	Changes in the implementation of parallelizing the update particle functionality.	Abishek, Megha, Ravikant
Week 7	History Server Setup	Abishek
Week 7	Implementation of PSO using scala	Megha
Week 8	Testing the implementation, Started with the documentation	Abishek, Megha, Ravikant
Week 9	Finalized the evaluation metrics	Abishek, Megha, Ravikant
Week 10	Code optimization and documentation	Abishek, Megha, Ravikant
Week 11	Git setup	Ravikant

6 Future work

The scope of future work would be evaluating the implementation using accumulators instead of broadcast variables with which we believe would give better results.

References

[1] khashayar Danesh Narooei. *Particle Swarm Optimisation.* National University Malaysia, 2013.

[2] Wikipedia
`https://en.wikipedia.org/wiki/PSO`

[3] Iran Macedo. *Particle Swarm Optimisation Algorithm.* Analytics Vidhya, December 2018.