HADOOP MAPREDUCE TO PARALLELIZE SOCIAL SPIDER OPTIMIZATION

Rama Naga Kiran Kumar. K[1*], Dr. Ramesh Babu. I[2]

[1*]Research Scholar, [2] Professor,
Dept. of Computer Science & Engineering,
Acharya Nagarjuna University, Guntur, Andhra Pradesh, India.

Abstract

Social Spider Optimization (SSO) has received attention in many research fields and real-world applications for solving optimization problems. Factor that affects the performance of SSO is its imbalance of exploration and exploitation. Its ability of the exploration in a multi-dimensional solution pace increases the execution time quite significantly. To reduce the execution time, parallel implementation of SSO should be implemented. In this paper, we implement and compare the parallel implementation of SSO using two different parallelization techniques using MapReduce programming, 1) all nodes in the cluster work on the same population, and 2) each node in cluster has its own population. Both parallel implementations are compared based on performance and speedup. Parallel implementation of the SSO algorithm makes the algorithm faster in case of both low and high dimensional datasets.

Keywords*: SSO, Data Analytics, Parallelization, Map Reduce, Hadoop, Boot's Function.*

Introduction

Data analytics is attracting more and more attention. Technological advancements have enabled us to capture very high volumes of data since space is not such a vital problem anymore, however, now analyzing and processing the very large amount of data (big data) is the biggest challenge. There are four main objects involved: capturing, storing, managing, and analyzing the data. Researchers have proposed many data mining algorithms to address the main objective of data analysis. However, the performance of an algorithm depends on the number of dimensions [1]. Nature inspired algorithms can explore multi-dimensional search spaces to find optimal solutions. In order to search for the minimum or maximum in a problem domain, a swarm intelligence algorithm processes a population of individuals [2] [3]. These algorithms are population-based algorithms, which consists of a population of individuals. Everyone represents a potential solution of the problem being optimized. The population of individuals is expected to have high tendency to move in high dimensional search spaces in order to find better solutions from iteration to iteration through cooperation or competition among themselves. Figure 1 shows the conceptual diagram depicting the workings of a population of individuals in a swarm. In the representation, the algorithm is initialized with random spiders within a problem space and the spiders are iteratively moving to find the optimum. However, the solution space of the problem often increases exponentially with the problem dimension and more efficient search strategies are required to explore all promising regions within a given period. The search performance of most algorithms is based upon the previous search experience. Considering the limitation of computational resources, the performance of the algorithm is affected by increasing of problem dimensions. This paper concentrates on the parallelization of the

Social Spider Optimization algorithm to optimize and compare the performance of the algorithm with other algorithms.

Background

There are many nature inspired algorithms available. In this article, we parallelized social spider optimization using Hadoop map reduce. In a social spider colony, each spider, depending on its gender, performs various tasks such as designing communal web, mating, killing the other spiders etc. The communal web acts as both communicational channel and common environment as shown in Figure 1. The spiders use vibrations to pass information in the communal web [4].

Figure 1: A Communal web of social spiders (Ahmed Fouad Ali, 2015)

Cuevas E simulated the behavior social spiders and proposed SSO. The solution space is a collection of spiders. A spider will be considered as globally best spider sgbs if its fitness is better than all other spiders [5]. Likewise, a spider will be treated as worst spider sws if all other spiders are having more fitness than it. The weight of a spider s can be computed using equation (1).

$$w[s] = \frac{\text{fitness}(s) - \text{fitness}(s_{ws})}{\text{fitness}(s_{gbs}) - \text{fitness}(s_{ws})} \qquad (1)$$

Defining Search Space

Initialization of all dimensions of a spider is performed using equation (2). The types of spiders are specified in Figure 2. The low and up functions return the lowest and the highest value in dimension i respectively.

$$spid[s,i]=low(i)+rand(0,1)*(up(i)-low(i)) \qquad (2)$$

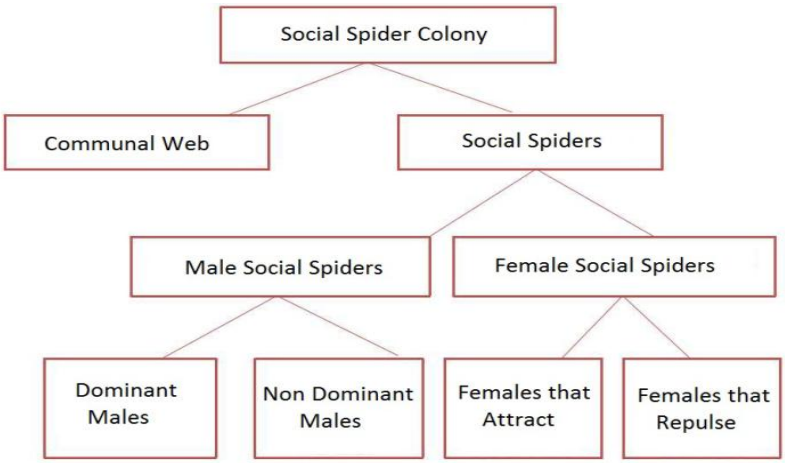

Figure 2: Types of spiders

Updating the positions of spiders

The attributes of the spiders are specified in Figure 3. The next positions of the spiders mainly depend on the weights and distances of spiders with highest fitness values, spiders at nearest distance with better fitness, and nearest female spiders. The amount of vibrations that spider S_j produces to spider S_i can be estimated using equation (3).

$$\text{vibrations}[s_i, s_j] = w[s_j] * e^{-\text{Dist}(s_i, s_j)^2} \quad (3)$$

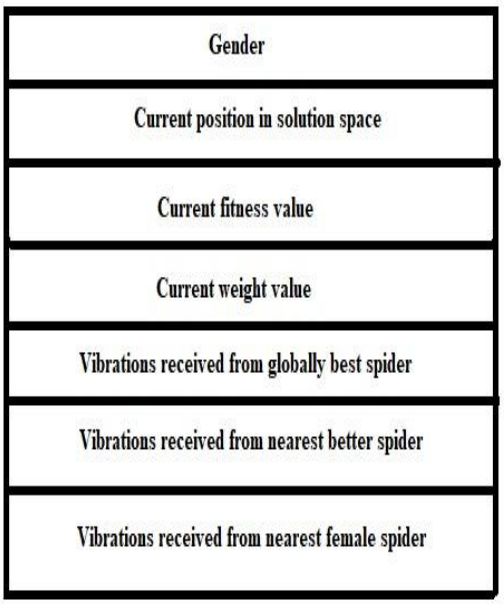

Figure 3: Characteristics of a spider

Evaluating subsequent locations of female spiders

A female spider always searches for better and best spiders .as shown in Figure 4. The updation of the position of a female spider s_f is calculated using equation (4). If female

spider does not like other spiders, the updation of its position happens using equation (5).

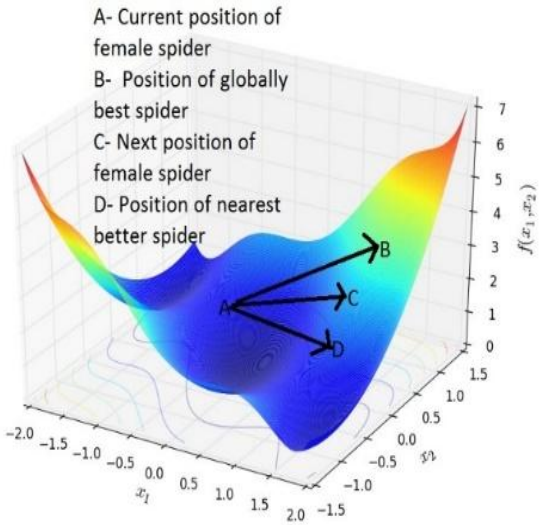

Figure 4: Updation of the position of a female spider

$$spid[s_f, i] = spid[s_f, i] + r_1 * (spid[s_f, i] - spid[s_{gbs}, i]) * w[s_{gbs}] * e^{-Dist(s_f, s_{gbs})^2} + r_2 * (spid[s_f, i] - spid[s_{nbs}, i]) * w[s_{nbs}] * e^{-Dist(s_f, s_{nbs})^2} + r_3 * (r_4 - 0.5) \quad (4)$$

$$spid[s_f, i] = spid[s_f, i] - r_1 * (spid[s_f, i] - spid[s_{gbs}, i]) * w[s_{gbs}] * e^{-Dist(s_f, s_{gbs})^2} - r_2 * (spid[s_f, i] - spid[s_{nbs}, i]) * w[s_{nbs}] * e^{-Dist(s_f, s_{nbs})^2} + r_3 * (r_4 - 0.5) \quad (5)$$

Evaluating subsequent locations of male spiders

The subsequent location of a dominant male spider S_{dm} can be computed as per equation (6). The vibrations from a female spider S_{nfs} at minimum distance plays an important role in estimating the subsequent position of male spiders that have better fitness values as shown in Figure 5. The weighted mean of spiders whose gender is male, W is used to compute subsequent positions of male spiders having low fitness values. It is obtained as per equation (7).

$$spid[s_{dm}, i] = spid[s_{dm}, i] + r_1 * (spid[s_{dm}, i] - spid[s_{nfs}, i]) * w[s_{nfs}] * e^{-Dist(s_{dm}, s_{nfs})^2} + r_3 * (r_4 - 0.5) \quad (6)$$

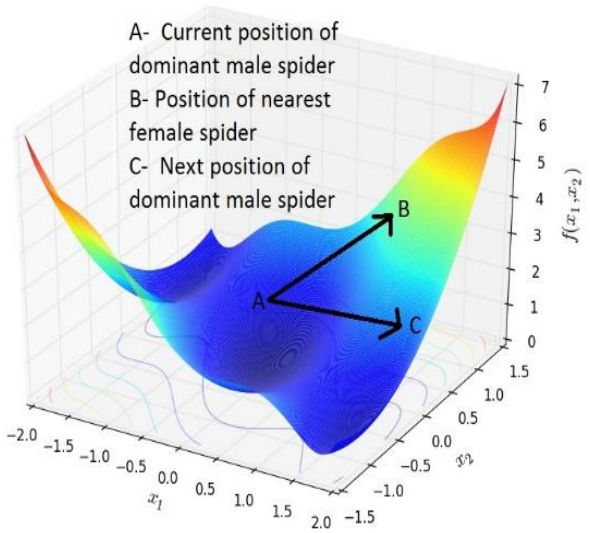

Figure 5: Updation of the position of a dominant male spider

Then the female spiders can be represented as $s_{f_1}, s_{f_2}, s_{f_3}, \ldots\ldots\ldots, s_{f_{Nf}}$ and the male spiders can be represented as $s_{m_1}, s_{m_2}, s_{m_3}, \ldots\ldots\ldots s_{m_{Nm}}$.

$$W=\frac{\sum_{j=1}^{N_m} spid[s_{m_j}, i]*w[s_{m_j}]}{\sum_{j=1}^{N_m} w[s_{m_j}]} \quad (7)$$

The position of a non-dominant male spider s_{ndm} is done using equation (8).

$$spid[s_{ndm}, i]=spider[s_{ndm}, i]+r_1*W \quad (8)$$

Creation of new spiders

New spiders are generated using the Roulette wheel method [6]. Each dominant spider finds its female spiders and generates new spider after the mating operation is over [7]. The arrival of new spider makes the spider with lowest fitness dead.

Parallelization of Social Spider Optimization Algorithm

In SSO, we must evaluate fitness values of large number of spiders sequentially. This problem can be avoided using a parallel implementation of SSO. For the parallelization, Hadoop is one of the most widely known and used runtime environment using the MapReduce paradigm. The SSO algorithm can be expressed with MapReduce and developed as a simple and robust parallel implementation. SSO has been parallelized using the following two different implementations. They are parallelization on the algorithm level and parallelization on the population level [8]. In parallelization on the algorithm level, the entire population is considered as a single node. In parallelization on the population level, each node will have some portion of the population. In this paper, we implement the SSO algorithm with MapReduce using both methods, 1) all the nodes in the cluster work on an entire population, and 2) each node in cluster has a portion of the entire population. In reducer step, the parallelism is achieved by performing reducers on several keys simultaneously [9][10]. We analyzed the

performance of both implementations on a Hadoop cluster measuring the execution time and speedup.

The SSO Algorithm

Algorithm1: SSO()
Input: Spiders and their Positions
Output: Globally best spider Gbest
1.Initialize the population (swarm) with random individuals (spiders)
2.For each spider, calculate the fitness value
3.Calculate vibrations received by each and every spider
4.Update position of each spider
5.Choose the spider with the best fitness value of all the spiders as the Gbest
6.Repeat Steps 3-5 until stopping criterion (maximum number of iterations) is met.
7.Return Gbest as best spider.

Implementation I: Parallelization on Algorithm Level

In this implementation, we generate the swarm of spiders and provide it as input. In addition, all the nodes of the Hadoop cluster work on an entire population of spiders independently. The mappers will take the complete population and evaluate the fitness of the spiders.

Map Function
The Map evaluates the fitness of the given spiders and update their positions. It also keeps track of the best fitness achieved and then passes it to reducer.

Algorithm 2: MAP()
Input: Spiders and their Positions
Output: Globally best spider Gbest

1. While (number of iterations<=max. iterations)
{
Evaluate fitness of each of the spider
For n = 1 to the number of spiders
Update the position of the spidern
}
2. Return the spider with best fitness Gbest to reducer

Reducer Function
The Reducer function collects the Gbest values from all algorithm runs that are performed on different nodes in the cluster. After comparing all Gbest values, the reducer returns the best Gbest value and writes it to HDFS.

Algorithm 3: Reduce()
Input: Gbest values
Output: Gbest value

1.Gbest=$Gbest_1$
2. For (i=2; i<=number of runs; i++)
{

If (fitness of $Gbest_i$ > fitness of Gbest)
Gbest=$Gbest_i$
}
3. Write (Gbest) to HDFS

Implementation II: Parallelization on Population Level

In the second implementation, we split an entire population to several nodes available on the cluster. In the first iteration, we randomly generate spiders and write the data into HDFS; the input file contains the spiders with their positions.

Initialization
Algorithm 4: Initialization
Input: swarm size, no. of dimensions
Output: Spiders and their positions

1.PopulationSize = SpecifyPopulationSize()
2.NumberofMaps=SpecifyNo.ofMaps()
3.GeneratePopulation(PopulationSize)
4. Write Population to HDFS
5. While (number of iterations<=max. iterations)
{
Map and Reduce functions are called
}
6. Write the data to HDFS

Map Function
We split the input file and each mapper takes a fragment of file (i.e., portion of the population) and evaluates several spiders in a mapper by evaluating the fitness of each spider. Each spider position is updated and written back to the input file in HDFS, and each spider's current position is sent to the Reducer.

Algorithm 5: Map()
Input: File containing spiders
Output: Pbest
1.Evaluate fitness of each of spider
2.Update position of each spider
3.Write spider to HDFS
4.Send position of the spider to Reducer

Reducer Function
The Reducer function collects the currrent positions of all spiders after each iteration and compares those values to identify Gbest, which is written to HDFS. This value represents the best value achieve so far. When a better Gbest value comes in, the reducer replaces the old Gbest value that was saved in the output file with the new Gbest value achieved.
Algorithm 6: Reduce()
Input: Current positions of spiders
Output: best spider: Gbest
1.For n = 1 to the number of spiders
{

Update Gbest
}
2.Update Gbest to HDFS
3. Write (Gbest) to HDFS

Results & Discussion

In this section, we present the implementation details, the experiments that were performed with outcomes of both MapReduce implementations of the SSO algorithm. The parallel algorithms were executed on the departmental Hadoop cluster 2.7.1. In order to evaluate the algorithm, the Booth's benchmark function has been used.

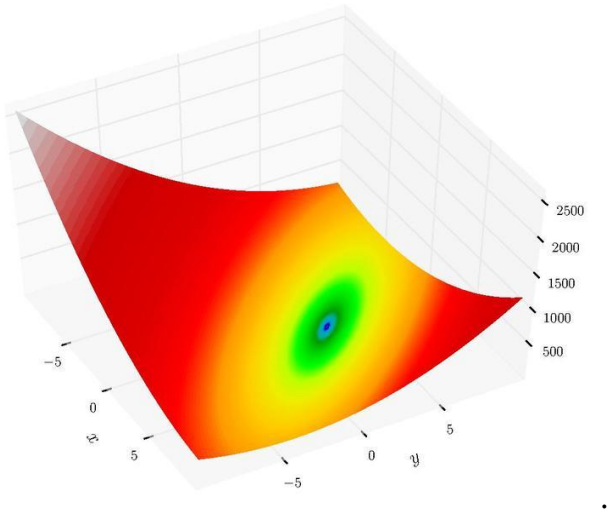

Figure 6. Booth's function

Experiments

Our experiments involve both implementations of the SSO algorithm. We have tested the performance of both the MapReduce implementations on the Hadoop cluster 2.7.1, Java version
1.6 with the focus on speedup and performance.

Firstly, we ran both implementations of SSO algorithm for the following settings.
Iterations: 5000
Spiders: 50000
Dimensions: 50
Evaluation function: Booth's function.

Secondly, we ran both implementations of the SSO algorithm by increasing number of iterations, spiders and dimensions. The settings are as follows.
Iterations: 10,000
Spiders: 100,000
Dimensions: 70
Evaluation function: Booth's function.

In Table 1 and Table 2, the execution time and speedup taken by both implementations when the number of cluster nodes is increased are specified. As we increase number of

cluster nodes, execution time gets decreased in both implementations. And it is found that second implementation takes more time for the execution and more speedup value than first implementation.

Table 1. Comparison of both implementations of SSO with respect to execution time and Speedup (5000 iterations, 50000 spiders, 50 dimensions)

Number of Nodes	Execution time (in seconds)		Speedup	
	First implementation of SSO	Second implementation of SSO	First implementation of SSO	Second implementation of SSO
2	14500	18000	1	1
4	12145	13590	1	1
6	10590	11166	1	1
8	8020	9504	1.5	1.7
10	6401	8550	2.1	2.5
12	4015	6250	3.0	3.2
14	3587	5256	3.5	4.1
16	3245	4215	4.4	5.2
18	1890	3555	5.5	6.8

Table 2. Comparison of both implementations of SSO with respect to execution time and Speedup (10000 iterations, 100000 spiders, 70 dimensions)

Number of Nodes	Execution time (in seconds)		Speedup	
	First implementation of SSO	Second implementation of SSO	First implementation of SSO	Second implementation of SSO
2	27002	32678	3.4	4.3
4	19456	25790	4.7	6.6
6	14678	18445	5.1	7.5
8	12003	15266	6.4	8.5
10	10345	13267	7.2	10.4
12	9000	12690	8.1	13.5
14	7345	9555	10.4	15.2
16	5288	7250	12.5	16.7
18	4210	6245	15.6	18.9

Figure 7 shows how execution times of both implementations vary for the change of number of spiders when iterations, nodes, and dimensions are fixed.

Fixed parameters: 6,000 iterations, 10 nodes, 50 dimensions.

It is obvious that in both implementations, as we increase number of spiders, execution time also gets increased.

Hadoop implementations the performance has improved when we increased the number of nodes. Thus, adding more resources while keeping the problem size fixed and with increasing the population size decreases the execution time. However, with increasing population size and keeping number of nodes, iterations and dimensions constant gradually increases the execution time. We also observed that the performance of the first implementation is faster than the second implementation. For larger problem dimensions, if we would increase the hardware and resources, both implementations can scale well.

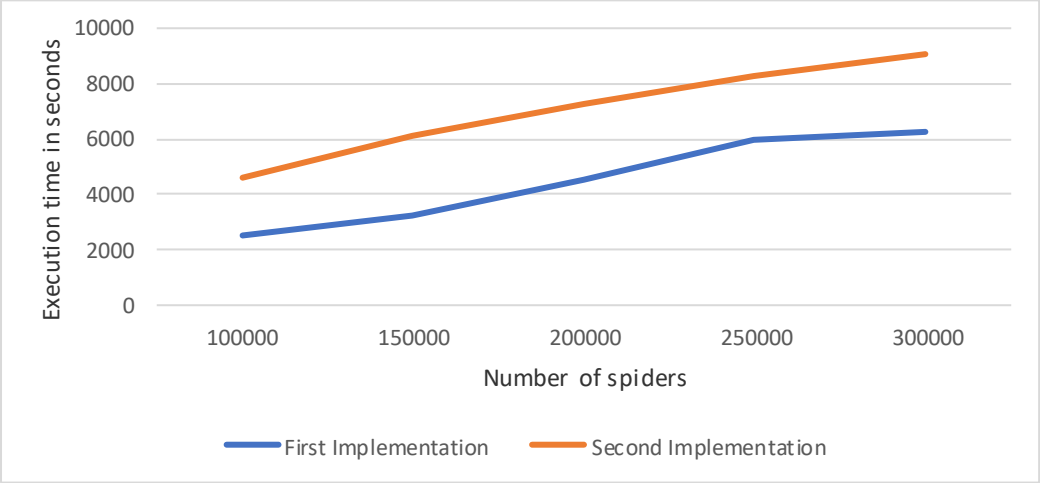

Figure 7: Comparison of execution time of both implementations for fixed parameters

Conclusion & Future Work

The performance of algorithms degrades significantly as the dimension of the search space increases. In this paper, we parallelized the SSO algorithm using MapReduce and executed the code on Hadoop, which is the most widely used implementation of the MapReduce programming paradigm. We used two different techniques for the parallelization, 1) entire population is evaluated on one node, 2) each node in cluster evaluates a portion of the population. In the first implementation, all the nodes available in the cluster receive an entire population as the input and work on this data. In the second implementation, each node in the cluster has a portion of the population. In this implementation, the input file (population) gets divided among nodes and each node available on the cluster gets its own part of the population to perform the evaluations. We analyzed both models using different combinations of iterations, population size, dimensions, and nodes. Based on our experiments, we realized that both parallel implementations can easily be scaled and used for large swarm populations to address more complex optimization problems. If we want to scale the problem size, we can easily add the hardware and scale both implementations. We also observed that the performance of the first implementation, parallelization at the algorithm level, performs faster than the second implementation where we implemented parallelization on the population level. Regardless, both implementations showed promising results and scalable nature to provide feasible solution for complex optimization problems in a reasonable time.

As for future work, we would like to implement the SSO algorithm using the Apache spark framework, which is a promising framework for iterative programming to observe the difference.

References

[1] S. Cheng, Y. Shi, Q. Qin and R. Bai, "Swarm Intelligence in Big Data Analytics",Intelligent Data Engineering and Automated Learning", p. 417-426, 2013.

[2] J. Brownlee, "Clever Algorithms: Nature-Inspired Programming Recipes", Lulu.com publication, p. 414, 2011.

[3] A.P.Engelbrecht, Computational Intelligence An Introduction, 2nd ed. John Wiley & Sons Ltd, p. 630, 2007.

[4] Cuevas E, Cienfuegos M, Zaldvar D, Perez Cisneros M, A swarm optimization algorithm inspired in the behavior of the social-spider, Expert Systems with Applications, Volume 40, Pages 6374-6384,Year 2013

[5] Cuevas E, Valentin Osuna, Diego Oliva, Evolutionary Computation techniques: A comparative perspective, Springer, Year 2016

[6] T. Ravi Chandran, A. V. Reddy and B. Janet, Text clustering quality improvement using a hybrid social spider optimization, Int. J. Appl. Eng. Res.12 (2017), 995–1008.

[7] T. Ravichandran, A. V. Reddy, and B. Janet, A Novel Bio-inspired algorithm based on Social spiders for improving performance and efficiency of data clustering, Journal of Intelligent Systems, https://doi.org/10.1515/jisys-2017-0178, year 2018

[8] R. Shonkwiler, "Parallel Genetic Algorithms", Georgia Institute of Technology Atlanta, p.1-7, 2016.

[9] M. Usuelli, "An example of MapReduce with rmr2 - MilanoR", MilanoR, 2013. [Online]. Available: http://www.milanor.net/blog/an-example-of-mapreduce-with-rmr2/.

[10] A.McNabb, C. Monson, and K. Seppi. "Parallel PSO Using Mapreduce". IEEE Congress on Evolutionary Computation, p1-9, 2007.