Comparing Spark with MapReduce - Glowworm Swarm Optimization applied to Multimodal Functions

Goutham Miryala and Simone A. Ludwig

North Dakota State University

ABSTRACT

Glowworm Swarm Optimization (GSO) is one of the optimization techniques, which need to be parallelized in order to evaluate large problems with high-dimensional function spaces. There are various issues involved in the parallelization of any algorithm such as efficient communication among nodes in a cluster, load balancing, automatic node failure recovery, and scalability of nodes at runtime. In this paper, we have implemented the GSO algorithm with the Apache Spark framework. Even though we need to address how to distribute the data in the cluster to improve the efficiency of algorithm, the Spark framework is designed in such a way that one does not need to deal with any actual underlying parallelization details. For the experimentation, two multimodal benchmark functions were used to evaluate the Spark-GSO algorithm with various sizes of dimensionality. We evaluate the optimization results of the two evaluation functions as well as we will compare the Spark results with the ones obtained using a previously implemented MapReduce-based GSO algorithm.

Keywords *Glowworm Swarm Optimization, Apache Spark, Hadoop, MapReduce.*

INTRODUCTION

Optimization is a process whereby the best possible solution to a particular problem is being sought. Thinking in terms of function optimization, one is usually interested in finding the minimum or maximum value of a particular function. There a many different kinds of optimization algorithms and one subgroup belongs to the stochastic algorithms, namely nature-inspired algorithms. Nature-inspired algorithms are algorithms that are inspired by natural processes such as evolution or swarming behavior. Within the nature-inspired algorithm category there are two, namely evolutionary algorithm and swarm intelligence approaches (Engelbrecht, 2007). Evolutionary algorithms take inspirations of processes in nature such as natural selection, crossover and mutation. Examples of evolutionary algorithms are genetic algorithm, evolutionary strategy, genetic programming, etc. (Eiben, 1994).

Swarm intelligence methods' core concept is that interacting swarm members are exchanging information to achieve a particular goal, which then is used to find the optimum value of a particular problem. Particle swarm optimization is one of the swarm intelligence methodologies whereby the concept of finding food sources based on the birds' current movement, the flocks' best food source ever found, and an individual bird in the flock experiencing the best food source is used. In ant colony optimization, the actions of ants performed for example during the process of finding the shortest path is applied (Stützle, 2009). In particular, the ants' indirect communication ability such as secreting pheromone on various paths is used in the algorithm. Another example of a swarm optimization algorithm is the bee colony optimization algorithm (Wong, Low, & Chong, 2008). In bee colony optimization, the concept of using local and global searching honeybees to build honeybee colonies is applied.

Yet another swarm intelligence algorithm, inspired by the characteristics shown by glowworms, called Glowworm Swarm Optimization (GSO) (Krishnanand & Ghose, 2009a), has been introduced. To mimic goals such as attracting a mate during the breeding season, the glowworms govern the emission of light. Application areas of GSO are for example, hazard sensing in ubiquitous environments (Krishnanand & Ghose, 2008), robotics and portable sensor networks (Krishnanand & Ghose, 2005), and data clustering (Aljarah & Ludwig, 2013b). All these applications can make use of the GSO algorithm due to its simplicity and small number of parameters that are required for tuning (Krishnanand & Ghose, 2009a) (Krishnanand & Ghose, 2008) (Krishnanand & Ghose, 2005).

GSO has been applied to many different application areas, however, one area that GSO is well suited for are multi-modal function optimization (Barrera & Coello, 2009). The aim of multi-modal function optimization is to find all maxima (peaks) or minima (valleys) given some constraints. However, one problem multi-modal function optimization suffers from is the relatively long runtime of the algorithm when searching for the peaks/valleys. In particular, the running time of the algorithm is significantly increased when the peak count is increased for higher dimensional spaces. The count of swarm members is also increased in order to have a better chance to find all the peaks or valleys in higher dimensional spaces. One approach to speed up the processing is to divide the search into several groups as to carry out the optimization process. Thus, a parallelized solution is required to achieve the optimization task in a feasible amount of time.

One solution that has been implemented using the MapReduce framework was proposed in (Aljarah & Ludwig, 2013b). The benefit of the MapReduce framework is that no knowledge of parallel programming is required to implement the parallelization. A conceptual understanding of MapReduce allows one to program a parallel implementation. Besides that, the MapReduce framework comes with built-in fault-tolerance, load balancing, and data locality.

The latest parallelization technology is Spark (Apache Spark^{TM}, 2017). The Spark development was initiated by the UC Berkeley RAD lab that started as a research project in 2009. The main goal behind the development is to provide iterative in-memory computing support for MapReduce. Companies like Yahoo, Databricks and Intel are major contributors to the Spark development (Matei Zaharia, Holden Karau, Andy Konwinski, 2015). Spark is used for data intensive applications that companies such as Facebook work with (Apache Spark @Scale, 2017).

Apache Spark can handle large sets of data and scale well by increasing the number of nodes at runtime. Apache Spark applications can be developed in various programming languages including Java, Python, Scala and R. A Spark application in general is mainly operated by a driver program that controls parallel operations on a cluster. A Resilient Distributed Dataset (RDD) is the main abstraction provided by Spark, which is a collection of elements that can be partitioned and distributed over a cluster of nodes for parallel processing. RDD provides reusability through caching capabilities and node failure recovery.

In this paper, a parallel GSO algorithm is proposed using Apache Spark. The following contributions are presented implementing parallel glowworm swarm optimization on Spark (Spark-GSO):

- Apache Spark concepts have been successfully applied to Glowworm Swarm Optimization to enable parallelization.
- Higher dimensional multimodal functions have been evaluated in Spark-GSO and compared with the MR-GSO algorithm.

The following sections in this paper are organized as follows: Background and Related work in the areas of parallel computing, MapReduce and parallel optimization algorithms are presented in Section 2. GSO, MR-GSO and the proposed Spark-GSO algorithm are presented in Section 3. The experiments with results are provided in Section 4 followed by conclusions.

BACKGROUND

Despite the fact that Genetic algorithms (optimization algorithm) can be used to solve difficult problems, they demand a lot of computational power and memory, thus the demand for computation in order to solve large-scale problems is increased (Venter & Sobieszczanski-Sobieski, 2006) (Ismail, 2004). In order to achieve a speedup against the execution of genetic algorithms in a single processor, parallel algorithms utilize multiple processing nodes (Grama, 2003). Several methods have been proposed to tackle the various challenges in implementing parallel optimization algorithms.

Message Passing Interface (MPI) methodology has been extensively used for various algorithms for parallel computing (Snir, 1998). A master slave paradigm on a Beowulf Linux Cluster using the MPI library has been proposed by Ismail et. al in 2004 (Ismail, 2004). A parallel PSO algorithm based on MPI has been introduced by Venter and Sobieszczanski-Sobieski in 2005 (Venter & Sobieszczanski-Sobieski, 2006). Since MPI has a large function set, it makes programming MPI complex (Snir, 1998), while algorithms using MapReduce are easy to design and develop (Dean & Ghemawat, 2004), although finer granular level parallel processes can be reused in MPI. MapReduce uses a distributed file system such as HDFS (Hadoop Distributed File System) to achieve faster file access whereas the message-passing model is used for communication in MPI. When a node fails in MPI the processes are terminated, whereas fault-tolerance of nodes is automatically built-in using MapReduce.

McNabb et al. in 2007 (McNabb, Monson, & Seppi, 2007) have created a model called MRPSO and implemented PSO with MapReduce. Radial basis benchmark functions have been used to verify and validate the efficiency of the execution of data-intensive optimization functions in MapReduce.

In 2008, Jin et al. (Jin, Vecchiola, & Buyya, 2008) have successfully implemented genetic algorithms with MapReduce. They provided proof that a genetic algorithm can be parallelized using MapReduce. Later, it has been proved that designing and implementing an ant colony optimization (ACO) algorithm can be done using the MapReduce framework (B. Wu, Wu, & Yang, 2012). Later, ACO has been implemented for different optimization problems such as 0-1 knapsack problem and the Travelling Salesman Problem (TSP) (B. Wu et al., 2012) to prove that larger problems can be solved using MapReduce. A MapReduce version of ACO applied to the Max-Min problem has been used (Tan, He, & Shi, 2012), which resulted in better results when compared to the sequential Max-Min ACO problem.

A differential evolution algorithm has been implemented in MapReduce (Zhou & Chi, 2010) to improve scalability. The population is divided into several partitions, and the sub-population is updated by a task in each partition. This improvement resulted in faster execution time when compared to the traditional sequential version.

Paduraru et al. (Paduraru, Melemciuc, & Stefanescu, 2017) implemented a genetic algorithm for a test function to evaluate the parallelization features capabilities. The PSO algorithm is implemented with a large amount of data (greater than $3 * 10^7$ data-points have been executed and evaluated) which resulted in a better performance than the traditional approaches (K. Wu, Zhu, Li, & Han, 2017).

Most of the MapReduce implementations above were used to optimize single objective functions. In (Aljarah & Ludwig, 2013a), GSO was used to deal with multimodal functions. The GSO algorithm was implemented using MapReduce (MR-GSO). The benefit of the parallel version of GSO is that higher dimensions can be executed (Aljarah & Ludwig, 2016) thus, achieving scalability and efficiency.

Although Apache Spark is not as efficient as MPI, it reduces the gap in performance in terms of speed and scalability when compared with MapReduce (Big Data Analytics, 2015). Apache Spark on Hadoop provides various additional features such as failure and data replication management, runtime addition of new nodes, and also provides tools for easy implementation. At times, these features make Apache Spark more preferable over the MPI methodology.

Performance on moderately sized datasets is substantially slower because of the overhead scheduling. In addition, there is no support for iterative computation in MapReduce, while excellent results are obtained in terms of scalability and performance using MLLIB libraries provided by Apache Spark (Meng et al., 2016) (Gopalani & Arora, 2015). MLLIB in Spark provides off-the-shelf algorithms for classification, regression, recommendation, clustering, etc. in conjunction to the use of streaming services for real-time analysis, which MapReduce does not provide (MLlib, 2017) (Spark Streaming, 2017) (Miryala, 2017).

In this paper, we have taken the GSO algorithm and implemented it on Apache Spark as Spark-GSO to improve the efficiency of GSO. The evaluation is done by comparing the optimization process, runtimes and speedup for both MapReduce (MR-GSO) and Spark (Spark-GSO).

GLOWWORM SWARM OPTIMIZATION

(Krishnanand & Ghose, 2005) have introduced a new swarm intelligence method called Glowworm Swarm Optimization (GSO) in 2005. Initially, the algorithm randomly places N glowworms in the workspace. $X_i(t)$ is the position at time t in the function search space, $L_i(t)$ is the Luciferin level and $rd_i(t)$ is a local decision range for a glowworm *i*. Based on the objective function *J*, an objective value of an individual's position is defined, which is associated with a luciferin level.

A glowworm closer to the peak has a higher fitness value holding a higher luciferin level (emits more light) than the others. If a glowworm has a higher luciferin value than the neighboring glowworms within the local decision range, then they try to attract the other glowworms towards it. The glowworms with a lower luciferin value and within its local range move towards a glowworm with a high luciferin value. This is a continuous process and requires several iterations to complete the process of movement towards several peaks in the given search space.

The GSO algorithm can be broadly divided into four stages:

Initially, all the required variables for the optimization are declared and initialized. Then, the algorithm randomly deploys *N* glowworms in the given workspace. L_0 (constant) is used to initialize the luciferin level for all glowworms. Finally, in the first stage, r_0 is used to initialize both r_d (local decision range) and r_s (radial sensor range).

The *second stage* deals with updating the luciferin levels. In this stage, the fitness is evaluated using the glowworm position (X_i). For all swarm glowworms, the luciferin levels are updated using the fitness values. The equation to update the luciferin level is based on:

$$L_i(t) = (1 - \rho)L_i(t-1) + \gamma J(X_i(t) \quad (1)$$

For a glowworm *i*, $L_i(t\text{-}1)$ and $L_i(t)$ are previous and updated luciferin levels, respectively, the luciferin decay constant is ρ, $\forall \rho \in (0,1)$, luciferin enhancement fraction is γ, and for iteration *t* and the current glowworm position, the objective function to measure the fitness is represented as $J(X_i(t))$.

The *third stage* defines the glowworm movement in the search space. For each glowworm *i*, based on L_i and L_j (luciferin level of another glowworm) and r_d (local decision range), the glowworms neighbor group $N_i(t)$ is extracted using:

$$j \in N_i(t) \;\; iff \;\; d_{ij} < rd_i(t) \;\; and \;\; L_j(t) > L_i(t) \quad (2)$$

where the neighbor group is represented as $N_i(t)$, one of the glowworms other than *i* is glowworm *j*, the Euclidean distance between the i^{th} and the j^{th} glowworm is d_{ij}, the local decision range for glowworm *i* is *rdi(t)*, and the luciferin levels for the j^{th} and i^{th} glowworm are *Lj(t)* and *Li(t)*, respectively.

The best neighbor is identified from the existing neighbor group by applying the roulette wheel method on the probability-based values. By applying the roulette wheel selection method, only higher probability glowworms in the neighbor group have a good chance to be chosen as the best neighbor. The probability calculation is done by:

$$Prob_{ij} = \frac{L_j(t)-L_i(t)}{\sum_{k\in N_i(t)} L_k(t)-L_i(t)} \quad (3)$$

where for glowworm i, a neighbor from neighbor group $N_i(t)$ is represented as j. The glowworm i does not update its location if the denominator in the equation becomes zero (case where no neighbors are found).

At the end of third stage, based on the selected neighbor position, the position of the current glowworm is updated which is the following:

$$X_i(t) = X_i(t-1) + s\frac{X_j(t)-X_i(t)}{\delta_{ij}} \quad (4)$$

Here, the new and old positions for glowworm i is represented as $X_i(t)$ and $X_i(t-1)$, respectively, the step size constant is s, and the distance between the i^{th} and the j^{th} glowworm is δ_{ij}.

The *final stage* deals with updating the local decision range. This adds flexibility while formulating a neighbor group in the successive iterations. To update rd_i, the following equation is used:

$$rd_i(t) = \min\{rs, \max[0, rd_i(t-1) + \beta(nt - |N_i(t-1)|)]\} \quad (5)$$

Here for glowworm i, the new and previous local decision range are represented as $rd_i(t)$ and $rd_i(t-1)$, respectively, the radial sensor range constant as rs, model constant as β, constant to govern neighbor count as nt, and the number of neighbors as $Ni(t)$.

PROPOSED SPARK GSO ALGORITHM (SPARK-GSO)

Initially, a swarm of glowworms of a specific size is created. In the swarm, each glowworm is associated with a random positional vector (X_i) in the given search space and is generated using uniform randomization. For each X_i vector, the fitness is calculated using the objective function J. Using Equation (1), the luciferin level (L_i) is evaluated for each glowworm with the provided default luciferin value L_0, $J(X_i)$, and other constants. The initial local decision range, r_0, is used as a local decision range r_d for the first iteration. Once the entire swarm is initialized with the updated information, the glowworms are added to a list. This list is used, broadcasted, and updated during every iteration of the algorithm.

In the next phase of Spark-GSO, the iterative process of RDD operations is performed. Each iteration (RDD action) updates the glowworm swarm and the updated swarm is used as the input for the next iteration for processing.

Before the transformations are applied, the entire swarm is sent to each task using a broadcast variable, a feature provided by Spark is to send and cache an object on each node before starting the tasks. The broadcast variable is initialized and broadcasted as a list of glowworms for the processing in the mappers, which is required to calculate the Euclidian distance between each worm in the swarm. The GSO constants such as $\beta, \rho, \gamma, s, rs, nt$ which are used in the process of movement of the glowworm swarms are retrieved.

There are two mapper transformations used in the architecture. The first transformation is used to find the best neighbor from all the glowworms in the swarm. To find the neighbors, an $O(n^2)$

algorithm is used. The algorithm involves calculating the Euclidian distance and the luciferin level comparisons between the given glowworm and all the other glowworms in the search space to locate a neighbor group as given in the Equation 2. Once the neighbor group is found, Equation 3 is used to find the best neighbor in that group. A technique called roulette wheel selection method is used to find the best neighbor. At the end of the first transformation, the best neighbor is attached to the original glowworm. Finally, the glowworm with an attached neighbor glowworm is emitted (returned) for further processing in the second transformation. The first transformation algorithm is outlined in Algorithm 2.

The second transformation picks up the glowworm swarm with each glowworm attached with a best neighbor glowworm. This transformation mapper is used to update the luciferin level L_i for each glowworm by evaluating the fitness for the new glowworm position. In this phase, the glowworm and its best neighbor position (X_j) is extracted at the start. Using Equation 4, the next step is to update the glowworm positional vector. Then, the fitness is evaluated for the new positional vector for the luciferin level calculation using Equation 1. In the last step before emitting the new glowworm, $\underline{rd_i}$ is calculated using Equation 5. Finally, the glowworm with the updated information is emitted. The second transformation algorithm is outlined in Algorithm 3.

Then, an Apache Spark action *Collect* is implemented in the driver class. As Spark transformations are "lazy", no transformation is applied until the action is implemented. The *collect* supplies the actual updated glowworm swarm to the driver program. At the end of each iteration, the updated glowworm swarm is collected and broadcasted for the next iteration processing. Also, this updated glowworm is used for RDD operations in the next iteration.

```
main()
      List swarm = createSwarm()
      RDD = SparkContext.Parallelize(swarm)
      for each iteration in numberOfIterations
            SparkContext.broadcast(swarm)
            RDD.map(mapper1)
            RDD.map(mapper2)
            swarm = RDD.collect()
function createSwarm()
      return List of glowworms
```

Algorithm 1 Spark Driver Program

```
function mapper1(Glowworm)
      broadcastSwarm = read (BroadcastVariable)
      for each glowworm in BroadcastSwarm do
            Xj=extractPosition(glowworm)
            Lj=extractluciferin(glowworm)
            EDist=returnEDistance(Xi,Xj )
            if (EDist <rdi and Lj >Li) then
                  NeighborsGroup:add(j)
            end if
      end for
      if (NeighborsGroup:size() > 0) then
            for each glowworm j in NeighborsGroup do
                  //calculate the probabilities from
                  the NeighborsGroup using Equation (2)
                  prob[j]=calculateProbability(i,j)
            end for
      end if
      nj=selectBestNeighbor(prob) //using roulette wheel selection
      Glowworm.setNeighborSize(NeighborsGroup.size())
      Glowworm.addNeighbor(nj)
      return Glowworm
end function
```

Algorithm 2 Transformation Mapper 1

```
function mapper2(Glowworm)
      Glowworm newGlowWorm
      if(neighborSize != 0)
            //Extract the neighbor glowworm information from the attached
glowworm
            extractNeighbor(Glowworm)
      else
            //Make the existing glowworm as new glowworm
            newGlowWorm = Glowworm
      end if
      //calculate the new position for glowworm using Equation (4)
      newX=calculateNewX(Xi,Xj )
      //update luciferin level for glowworm i using objective function
formula J
      newJx=calculateNewJx(newX)
      //update luciferin level for glowworm  using Equation (1)
      newL=calculateNewX(Li,newJx)
      //calculate the new rd for glowworm using Equation (5)
      newrd=calculateNewrd(rdi,nbSize)
      newGlowWorm.update(newX,newJx,newL,newrd)
      return (newGlowWorm)
end function
```

Algorithm 3 Transformation Mapper

EXPERIMENT AND RESULTS

In this section, we provide the details about the computing environment and the benchmark functions used for the experiments as well as give a brief description of the MR-GSO algorithm. We also discuss the optimization quality, running time of the measurements for the MR-GSO and Spark-GSO algorithms.

Environment

We executed the MR-GSO and Spark-GSO algorithms on the Wrangler Hadoop cluster hosted by the Texas Advanced Computing Center (TACC). Each node of the Wrangler cluster has 24 cores (Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz), and 128 GB of memory. The Hadoop environment, which we have used is Hadoop 2.6.0-cdh5.7.1 to run the MR-GSO algorithm, while Apache Spark version 2.1.0 is used to execute the Spark-GSO algorithm.

Benchmark Functions

We have used two multimodal benchmark functions to evaluate the MR-GSO and Spark-GSO algorithms. The description of the benchmark functions is as follows (Li, Engelbrecht, & Epitropakis, 2013) (Liang, Qu, Suganthan, & Chen, 2014):

F1: A highly multimodal function called Equal-peaks-B, which can be spanned into an m-dimensional search space is chosen as Function F1. The Equal-peaks-B function has equal function values at all local maxima. When X_i, $i = 1, \ldots, m$, is considered as a multidimensional vector, the function search space used is $(-\Pi \leq X_i \leq \Pi)$. The function has 2^m peaks and the definition is:

$$F_1(X_i) = \sum_{i=1}^{m}[sin^2(X_i)] \quad (6)$$

F2: The Rastrigin function is a highly multimodal function, which is generally used for optimization algorithms as a performance test problem. The minima and maxima of locations are regularly distributed in this function. Due to its large number of local minima and large search space, this function has difficulty in achieving the solution. When X_i, $i = 1, \ldots, m$, is considered as a multidimensional vector, the function search space used is $(-1 \leq X_i \leq 1)$. The function has 2^m peaks with the following definition:

$$F_2(X_i) = 10m + \sum_{i=1}^{m}[X_i^2 - 10\cos(2\pi X_i)] \quad (7)$$

Evaluation Measures

The Peaks Capture Rate (PCR) and the average minimum distance from each glowworm to the peak locations (D_{avg}) are used to determine the optimization quality (Krishnanand & Ghose, 2009b). If the distance of three nearest glowworms to a peak is less than or equal to ε, then we say that the peak is captured. As recommended by Krishnanand et al. (Krishnanand & Ghose, 2009b), $\varepsilon = 0.05$ is used in our experiments.

The Peak Capture Rate (PCR) is calculated using:

$$\text{PCR} = \frac{Number\ of\ Peasks\ Captured}{Number\ of\ All\ Peaks} \times 100\% \quad (8)$$

The average minimum distance, D_{avg}, to the peak locations is calculated using:

$$D_{avg} = \frac{1}{N} \times \sum_{i=1}^{N} \min_{\{1 \leq j \leq Q\}} \{\delta_{i1} \ldots \delta_{iQ}\} \quad (9)$$

where the number of glowworms in the swarm is N, the Euclidian distance between glowworm i and peak j is represented as δ_{ij}, and the number of available peak locations is represented as Q.

When high PCR and low D_{avg} values are achieved, it is considered as the best result. For example, if the result achieved has a low PCR, it means the glowworms are gathered at a few peaks only ignoring the rest of the peaks, which is not a good solution. While when PCR is close to 100%, it means that the glowworms are actually gathered at all the peaks available, and a low D_{avg} means the glowworms are actually gathered very close to the peaks, which is an optimal solution.

The experiments which we executed uses the default GSO settings as specified in (Krishnanand & Ghose, 2009b). We used ρ (luciferin decay constant) = 0.4, γ (luciferin enhancement constant) = 0.6, β (constant parameter) = 0.08, *nt* (number of neighbors' limit) = 5, L_0 (Luciferin rate) = 5.0, *s* (step size) = 0.03. The r_d (local decision range) and r_s (radial sensor range) values are adjusted depending on the function chosen. In our executions, r_d is constant throughout the optimization process such that $r_s = r_d = r_0$.

MR-GSO Algorithm

The MR-GSO algorithm is implemented based on the work published by Aljarah and Ludwig (Aljarah & Ludwig, 2016). The implementation of Spark-GSO is similar to MR-GSO except with some modifications to make use of features available in Spark. In MR-GSO, the glowworms in the swarm are initially written to the distributed file system with the <Key, Value> structure. The key-value structure is described in Figure 1.

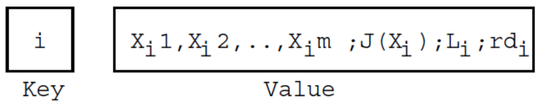


Figure 1 Representation of Glowworm in MR-GSO Algorithm

Here, during each iteration a MapReduce job is executed which produces the updated swarm, which is used for the next iteration. The mapper is used for finding the neighbor group and the best neighbor from the workspace. The mapper emits the current glowworm and the best neighbor glowworm as <Key, List of Values> at the end. Once the reducer is started, the emitted <Key, List of Values> pairs from the mapper is consumed and the update of the Luciferin level is carried out. The Glowworm positional vector is also updated in the reducer and a newly updated glowworm is emitted at the end. More details can be found in (Aljarah & Ludwig, 2016).

RESULTS

To evaluate and compare both MR-GSO and Spark-GSO algorithms, various experiments have been conducted measuring PCR, D_{avg}, running time and speedup for both the Equal-peaks-B and the Rastrigin benchmark.

The optimization quality for the Spark-GSO algorithm for the *F1* function with 2 dimensions is shown in Figure 1. For each swarm size varying from 10,000 to 60,000, PCR and D_{avg} for every iteration have been evaluated and presented. Although we can see that the minimum distance is reduced at each iteration, we cannot see any significant improvement in D_{avg} when the swarm size is increased. Also, for the 2-dimensional glowworms swarm, we cannot see a significant improvement in PCR when the swarm size is increased as the PCR converges to 100% at the lowest swarm (10,000) size (Figure 2(b)).

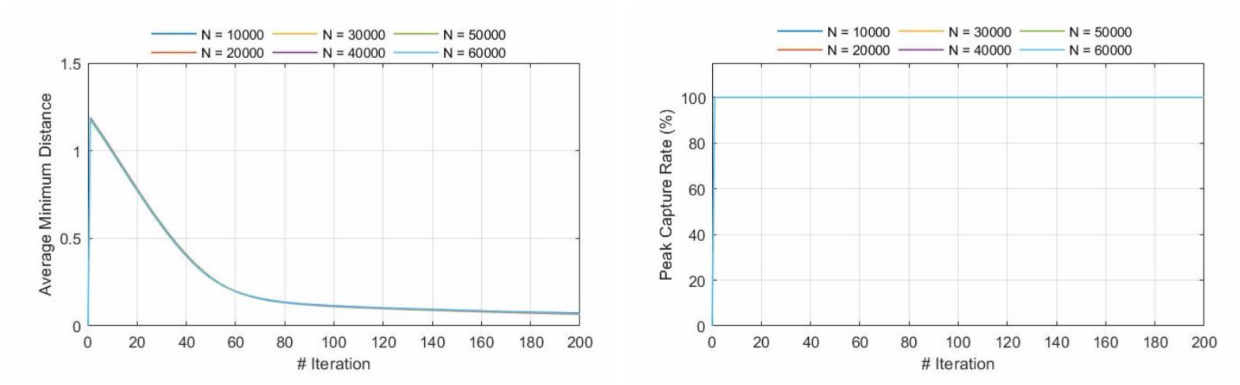


Figure 2 Spark-GSO: Equal-peaks-B (F1) 2-dimensional optimization process, iterations=200, r0=2.0 (a) Average Minimum Distance. (b) Peaks Capture Rate.

Figure 3 represents the optimization quality for the MR-GSO algorithm for the *F1* function with 2 dimensions. Similar to Spark-GSO, the PCR converges at the second iteration for a swarm size of 10,000 and D_{avg} significantly reduces at each iteration until the first 60 iterations, and shows a slow reduction after that. There is no visible difference between Spark-GSO and MR-GSO for D_{avg} in Figures 2(a) and 3(a).

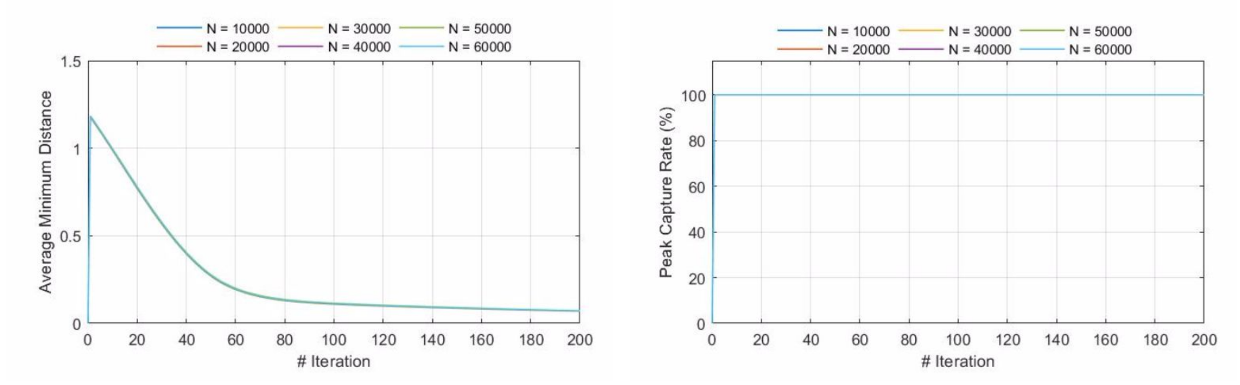


Figure 3 MR-GSO: Equal-peaks-B (F1) 2-dimensional optimization process, iterations=200, r0=2.0 (a) Average Minimum Distance. (b) Peaks Capture Rate.

The optimization quality for the Spark-GSO algorithm for the *F1* function with 4 dimensions is shown in Figure 4. Here, we can see that the average minimum distance is almost equal for all swarm sizes and low values are achieved over time (iterations). However, the PCR values convergence rate to 100% varies for each glowworm swarm size. From Figure 4(b), we can see that the PCR converges to 100% at the 42nd iteration for a swarm of 10,000 glowworms, while the PCR converges to 100% at the 25th iteration for a swarm of 60,000 glowworms.

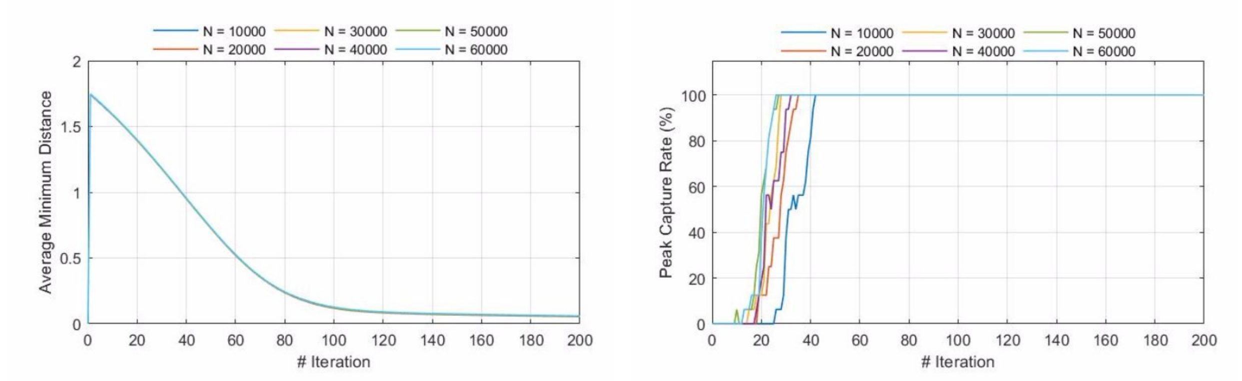


Figure 4 Spark-GSO: Equal-peaks-B (F1) 4-dimensional optimization process, iterations=200, r0=2.0 (a) Average Minimum Distance. (b) Peaks Capture Rate.

Figure 5 represents the optimization results for the MR-GSO algorithm for the *F1* function with 4 dimensions. From Figure 5(b), it shows that the difference in achieving a low D_{avg} is almost similar to the Spark-GSO execution. PCR converges to 100% at the 47th iteration using MR-GSO (Figure 4(b)) while it is achieved at the 42nd iteration for Spark-GSO (Figure 4(b)), which is a minute difference.

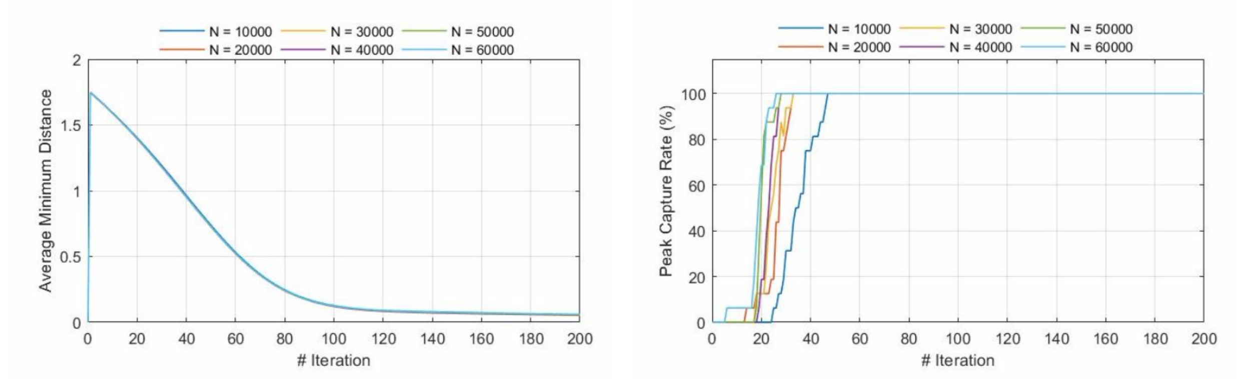


Figure 5 MR-GSO: Equal-peaks-B (F1) 4-dimensional optimization process, iterations=200, r0=2.0 (a) Average Minimum Distance. (b) Peaks Capture Rate.

Figure 6 and Figure 7 represent the optimization quality for the *F1* function with 6 dimensions for the Spark-GSO and MR-GSO algorithm, respectively. In both, Spark-GSO and MR-GSO, the average minimum distance is better for the swarm size with 60,000 glowworms than the smaller swarm sizes. For both algorithms, the PCR does not converge to 100% when the swarm size is

10,000. For the rest of the swarm sizes, the PCR converges to 100% for 6 dimensions. The Spark-GSO algorithm captured 98.4% of the peaks, while MR-GSO captured 95.3% of peaks in the search space.

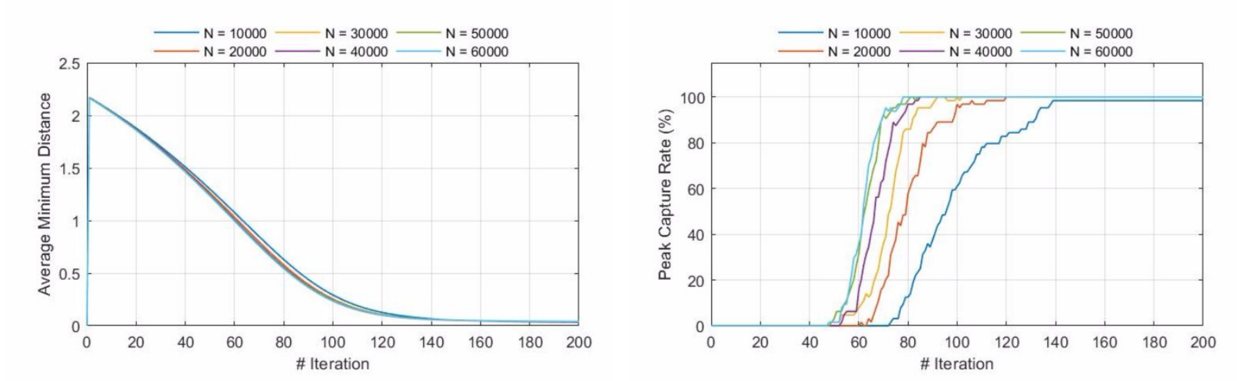


Figure 6 Spark-GSO: Equal-peaks-B (F1) 6-dimensional optimization process, iterations=200, r0=2.0 (a) Average Minimum Distance. (b) Peaks Capture Rate.

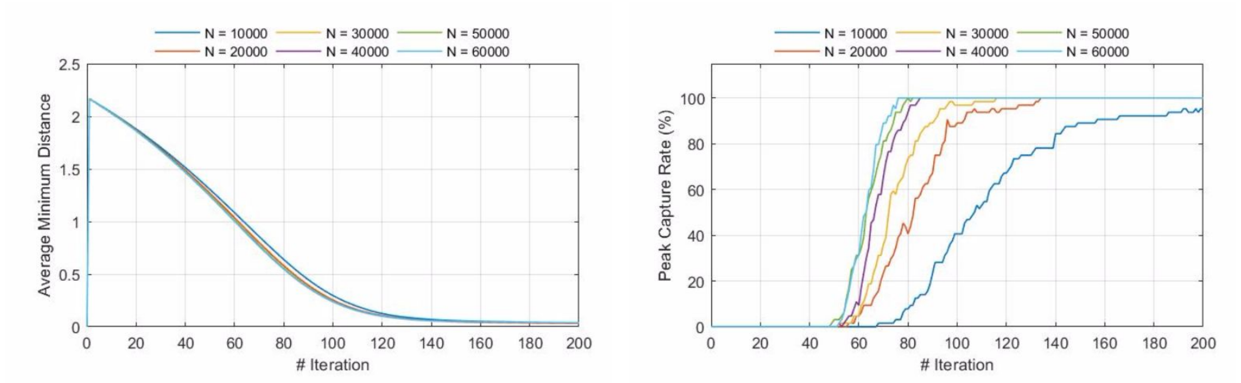


Figure 7 MR-GSO: Equal-peaks-B (F1) 6-dimensional optimization process, iterations=200, r0=2.0 (a) Average Minimum Distance. (b) Peaks Capture Rate.

Figure 8 and Figure 9 represent the optimization quality for the *F1* function with 8 dimensions for Spark-GSO and MR-GSO, respectively. In both, Spark-GSO and MR-GSO, the average minimum distance is better for the swarm size with 60,000 glowworms than the smaller swarm sizes, and for a 10,000 sized swarm the D_{avg} is considerably larger. For both algorithms, the PCR does not converge to 100% irrespective of the swarm size. Spark-GSO and MR-GSO captured only around 5% peaks for swarm of 10,000 glowworms after 200 iterations, while for a swarm of 60,000 glowworms only around 70% peaks are captured.

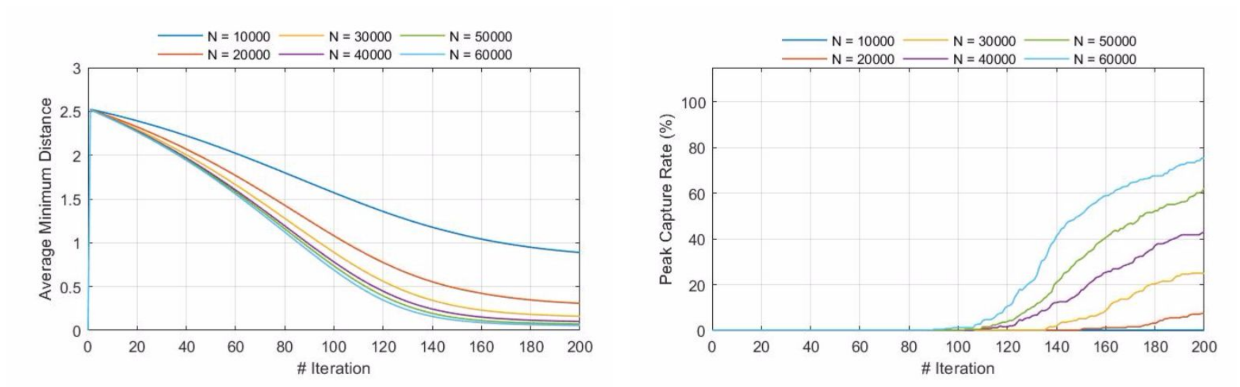


Figure 8 Spark-GSO: Equal-peaks-B (F1) 8-dimensional optimization process, iterations=200, r0=2.0 (a) Average Minimum Distance. (b) Peaks Capture Rate.

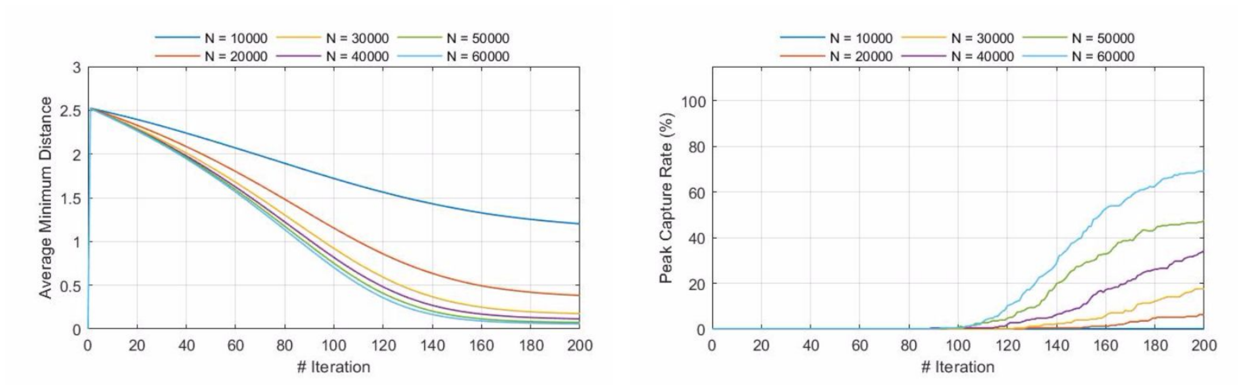


Figure 9 MR-GSO: Equal-peaks-B (F1) 8-dimensional optimization process, iterations=200, r0=2.0 (a) Average Minimum Distance. (b) Peaks Capture Rate.

For the Rastrigin (*F2*) function with 2, 4, 6 and 8-dimensions, the optimization quality results for both Spark-GSO and MR-GSO are represented in Figures 10 to 17. For the 2-dimensional glowworms swarm of various sizes, 100% of the peaks are captured at the 1st iteration for both Spark-GSO and MR-GSO (Figure 10(b), Figure 11(b)). Spark-GSO achieved 100% PCR at the 10th iteration for 10,000 glowworms with 4 dimensions, while MR-GSO achieved the same at the 13th iteration. The same has been achieved at the 7th iteration for 60,000 glowworms for both algorithms (Figure 12(b), Figure 13(b)). For the 6 dimensions, the maximum peaks capture rate is 98.4% after 200 iterations with a swarm of 10,000 for Spark-GSO and MR-GSO, while for 60,000 glowworms it is achieved at around the 22nd iteration (Figures 14(b) and 15(b)). For both Spark-GSO and MR-GSO with 8-dimensions, not even a single peak is captured for a glowworms swarm of 10,000. But when using 60,000 glowworms, 85.9% of the peaks are captured for the Spark-GSO algorithm while only 63.2% of the peaks are captured when the MR-GSO algorithm is executed (Figures 16(b) and 17(b)).

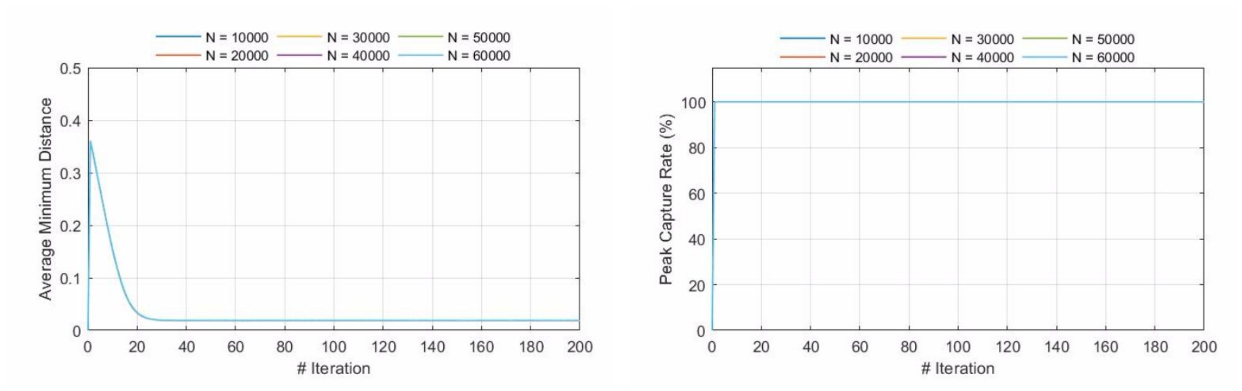


Figure 10 Spark-GSO: Rastrigin (F2) 2-dimensional optimization process, iterations=200, r0=0.5 (a) Average Minimum Distance. (b) Peaks Capture Rate.

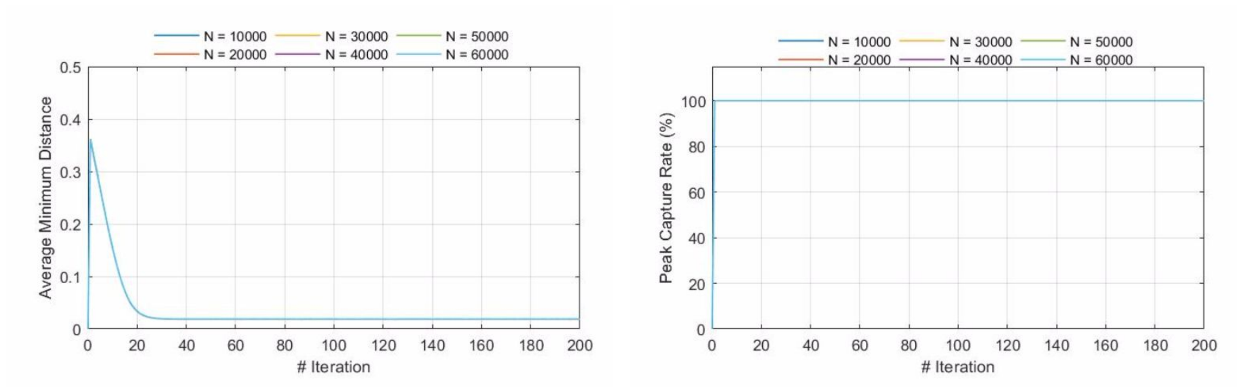


Figure 11 MR-GSO: Rastrigin (F2) 2-dimensional optimization process, iterations=200, r0=0.5 (a) Average Minimum Distance. (b) Peaks Capture Rate.

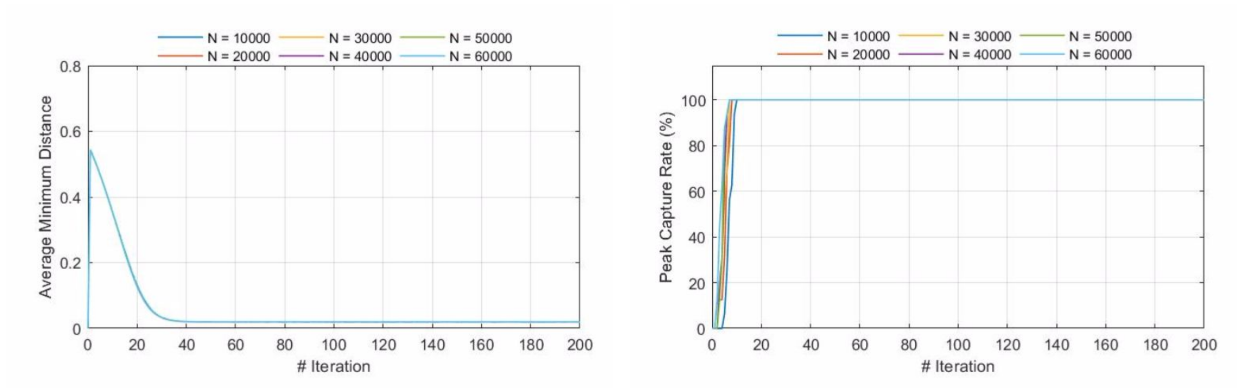


Figure 12 Spark-GSO: Rastrigin (F2) 4-dimensional optimization process, iterations=200, r0=0.5 (a) Average Minimum Distance. (b) Peaks Capture Rate.

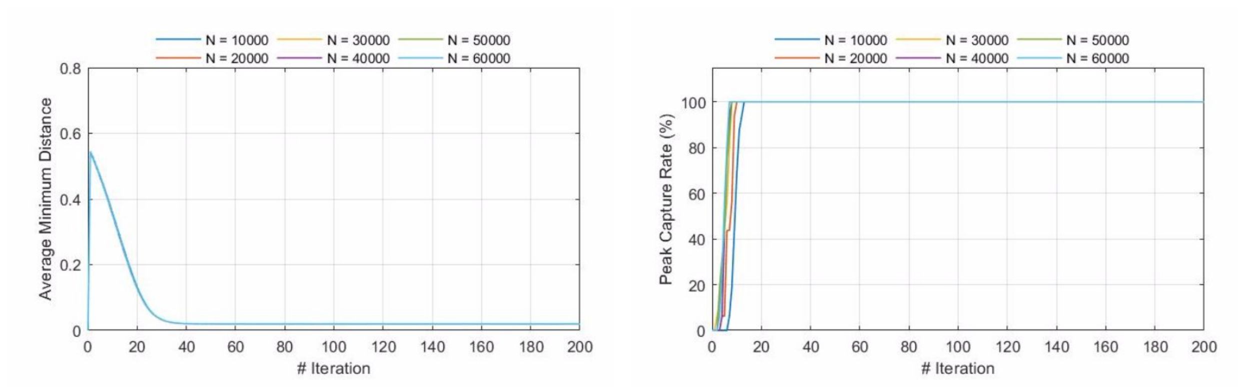


Figure 13 MR-GSO: Rastrigin (F2) 4-dimensional optimization process, iterations=200, r0=0.5 (a) Average Minimum Distance. (b) Peaks Capture Rate.

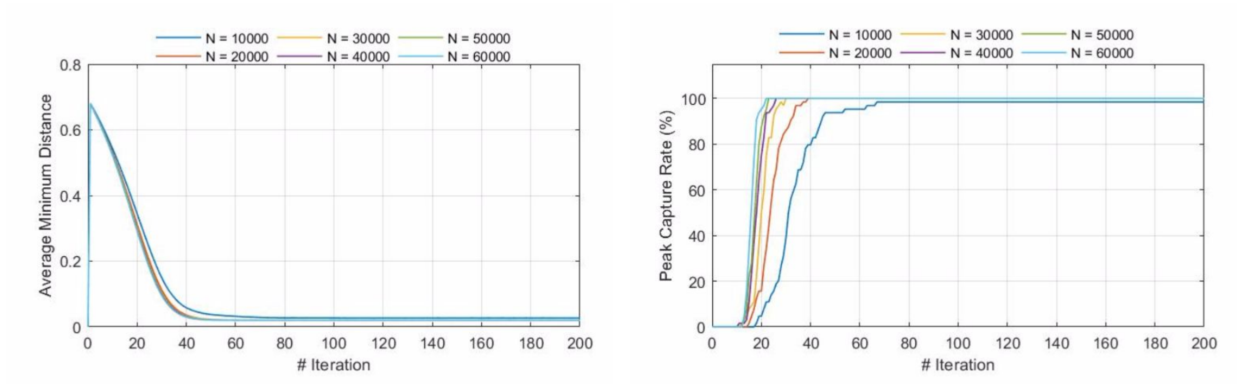


Figure 14 Spark-GSO: Rastrigin (F2) 6-dimensional optimization process, iterations=200, r0=0.5 (a) Average Minimum Distance. (b) Peaks Capture Rate.

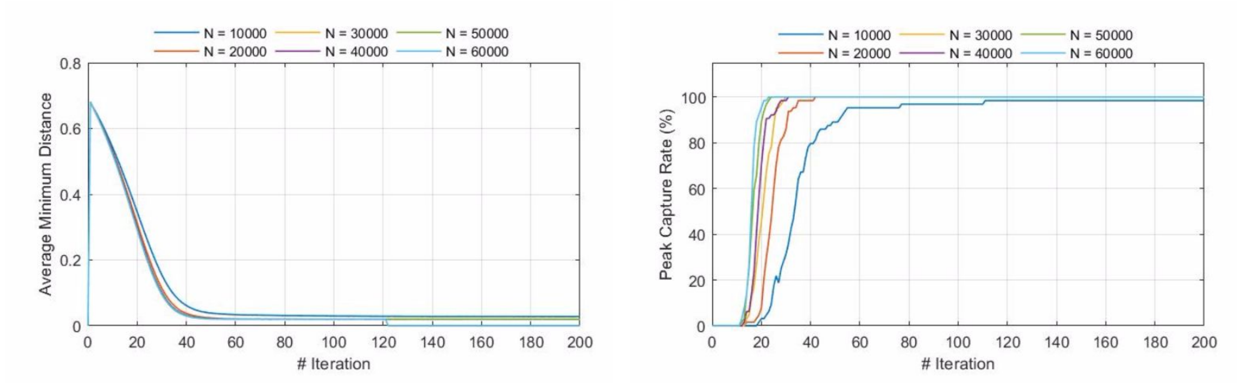


Figure 15 MR-GSO: Rastrigin (F2) 6-dimensional optimization process, iterations=200, r0=0.5 (a) Average Minimum Distance. (b) Peaks Capture Rate.

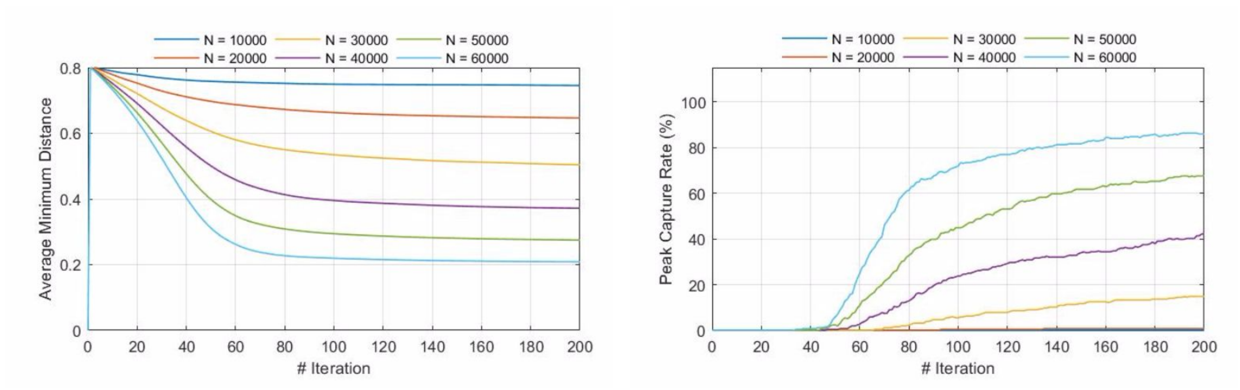


Figure 16 Spark-GSO: Rastrigin (F2) 8-dimensional optimization process, iterations=200, r0=0.5 (a) Average Minimum Distance. (b) Peaks Capture Rate.

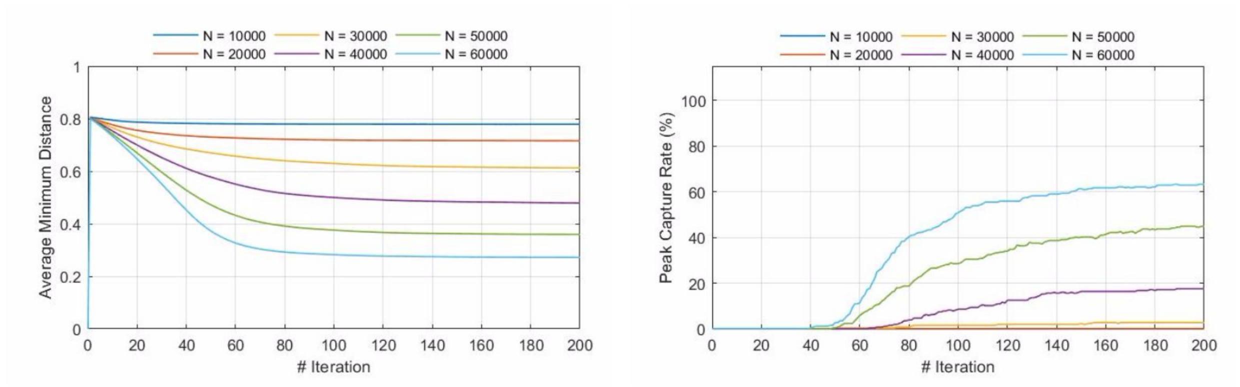
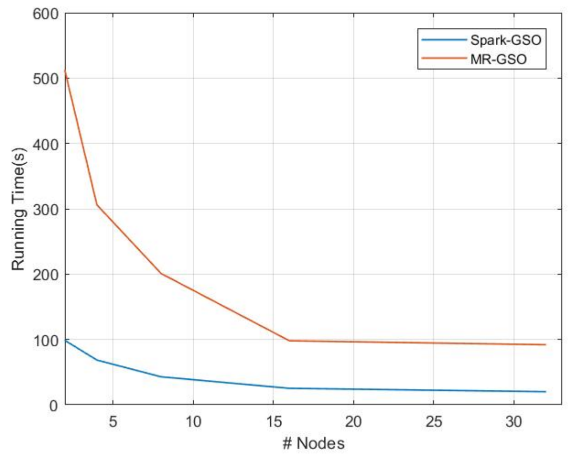


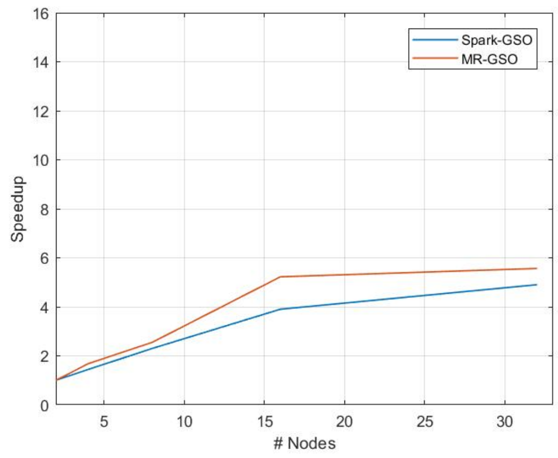
Figure 17 MR-GSO: Rastrigin (F2) 8-dimensional optimization process, iterations=200, r0=0.5 (a) Average Minimum Distance. (b) Peaks Capture Rate.

We have executed both Spark-GSO and MR-GSO on a 2, 4, 8, 16 and 32 node cluster to compare the running times and speedup results. Figure 18 represents the running times and speedup results of Spark-GSO and MR-GSO on various number of nodes in a cluster. As we increase the number of nodes the running times decrease for both Spark-GSO and MR-GSO. Also, we can observe that the running times actually increase when the swarm size increases. Finally, we can conclude that the running times of the Spark-GSO algorithm are less than the ones of the MR-GSO algorithm.

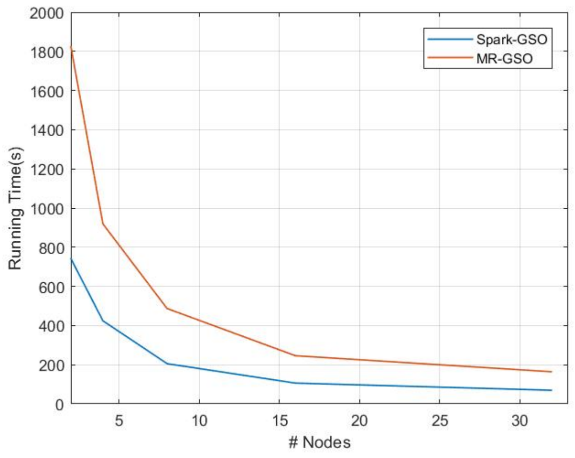
The speedup results for both Spark-GSO and MR-GSO for various swarm sizes executed on various nodes are represented in Figures 18(b), 18(d) and 18(f). As we can see, the speedup of the MR-GSO algorithm is closer to the linear speedup only when 2, 4 and 8 nodes are used when compared to the Spark-GSO algorithms speedup for 100,000 glowworms swarm. The speedup diverges after 8 nodes. For 200,000 glowworms, the speedup for both Spark-GSO and MR-GSO is comparatively closer to the linear speedup until 16 nodes than when N=100,000. When 300,000 glowworms are used, MR-GSO is very close to the linear speedup until 16 nodes and diverges a little after that. But for Spark-GSO, we can see that the divergence is larger than that of the MR-GSO for N=300,000.



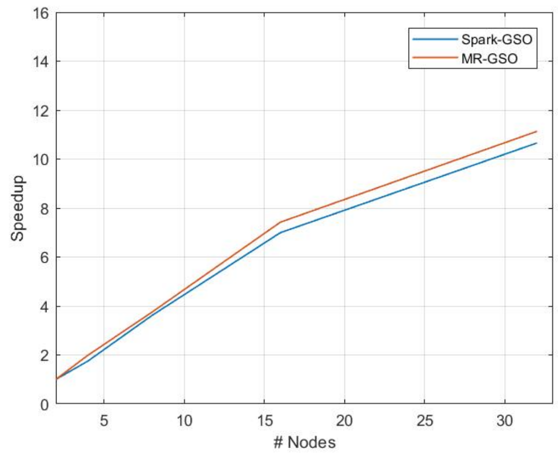
a) Running Time N=100,000



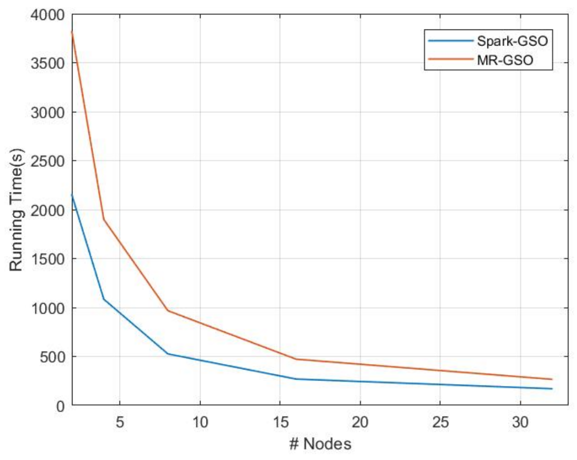
b) Speedup with N=100,000



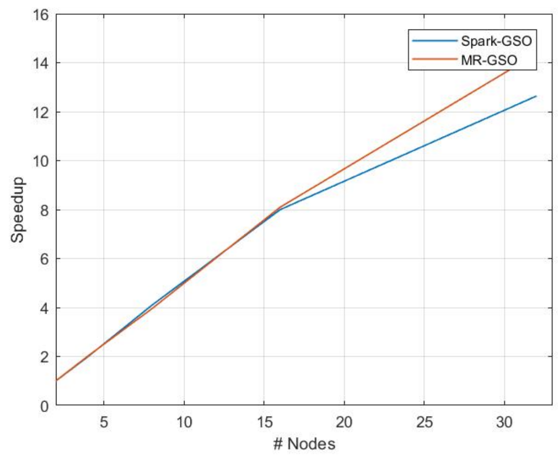
c) Running Time N=200,000



d) Speedup with N=200,000

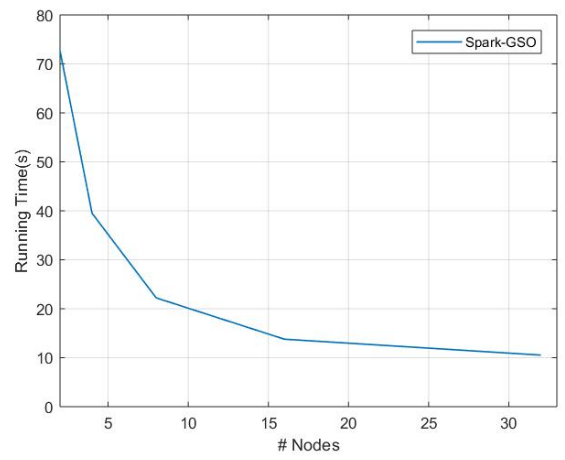


e) Running Time N=300,000

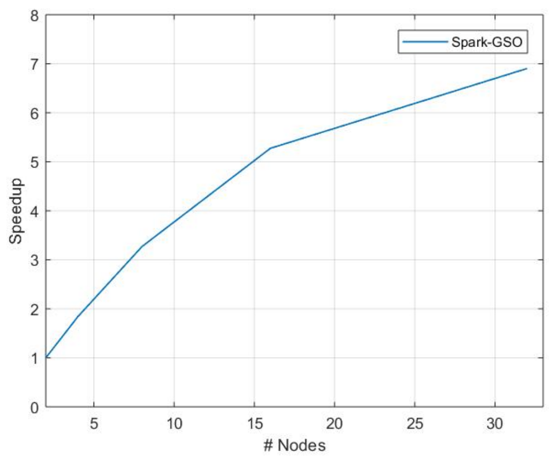


f) Speedup with N=300,000

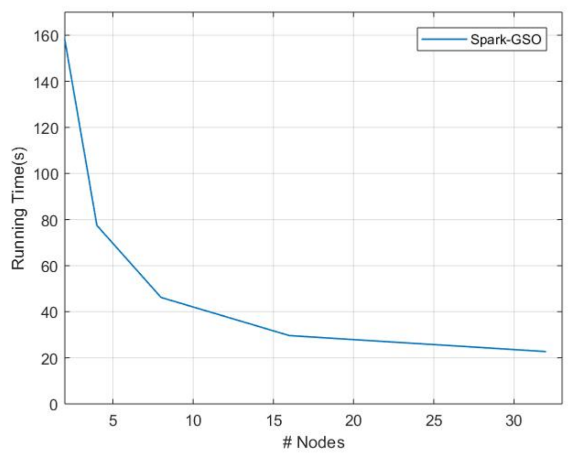
Figure 18 Equal-peaks-B function running time and speedup results for 4-dimensional glowworms



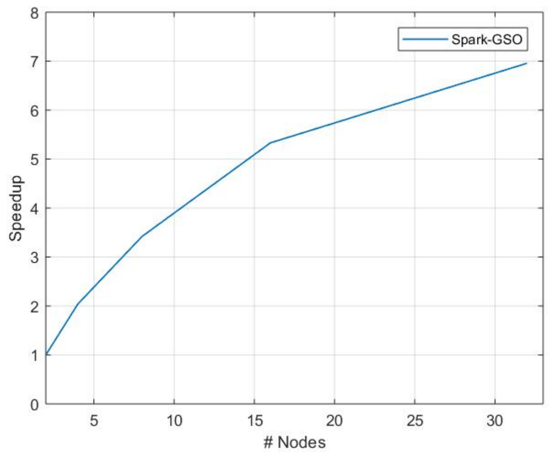
a) Running Time for 1,000 dimensions



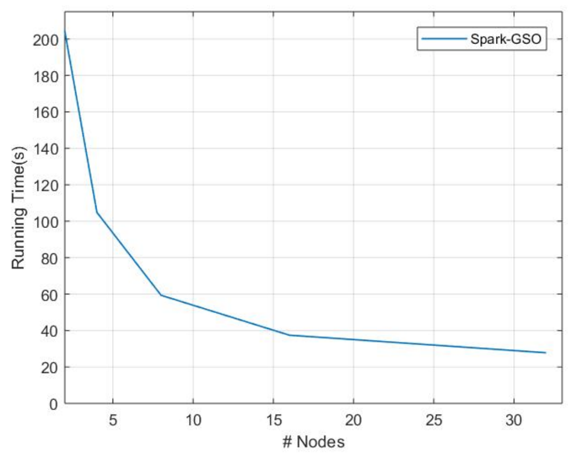
b) Speedup with 1,000 dimensions



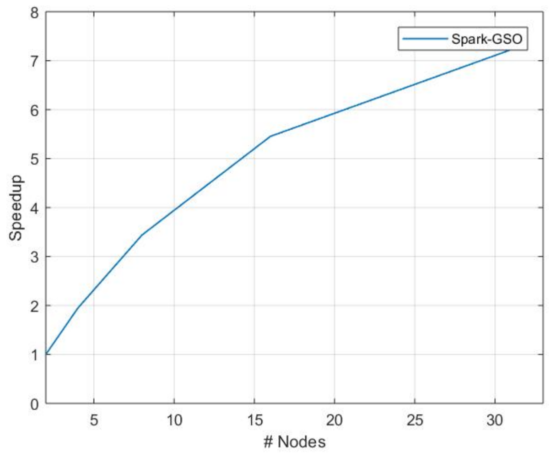
c) Running Time for 2,000 dimensions



d) Speedup with 2,000 dimensions



e) Running Time for 3,000 dimensions



f) Speedup with 3,000 dimensions

Figure 19 Equal-peaks-B running time and speedup results for N=10,000 swarm size

We have executed Spark-GSO for a higher dimensional swarm of size 10,000 to test the scalability of the algorithm on 2, 4, 8, 16 and 32 nodes and obtained running times and speedup results. Figure 19 represents the running times and speedup results of Spark-GSO on various number of nodes with various dimensions and the fixed swarm size of 10,000. As we increase the number of nodes, the running times decrease running Spark-GSO. Also, we can observe that the running times increase when the number of dimensions of the swarm increases (as shown in Figure 19 a) to f)).

CONCLUSION

Many different parallelization frameworks have been introduced in the past. Spark is one such framework that is designed in a way that allows for easy implementation. In order to parallelize an algorithm using the Spark framework one does not need to deal with any parallelization details besides the logic of the algorithm itself.

In previous research work, the Glowworm Swarm Optimization (GSO) algorithm was parallelized using MapReduce (MR-GSO). In this paper, we have parallelized the GSO algorithm using Apache Spark (Spark-GSO) applied to multimodal function optimization. Apache Spark eliminates the read and writing operations of intermediate files onto a hard disk, which MapReduce uses. Furthermore, Spark-GSO parallelizes the algorithm using two transformations and a single action.

For the experimentation, two multimodal benchmark functions were used to evaluate the Spark-GSO algorithm with various sizes of dimensionality (2 to 8) as well as various swarm sizes (10,000 to 60,000). Furthermore, we compared the Spark-GSO results with the ones obtained using the MapReduce-based GSO algorithm. The optimization results, running times, and the speedup were evaluated and compared with the MR-GSO results. The results can be summarized as follows. There is a difference in the convergence of the optimization results comparing the Spark and MapReduce implementations. Spark-GSO converges to the solution in general a little bit faster than MR-GSO, which is especially noticeable for larger dimensions. For both benchmark functions, the optimization results are very similar for 2 and 4 dimensions, but then show for the higher dimensions (6 and 8); most significant for 8 dimensions. In terms of the running time of Spark-GSO and MR-GSO using up to 32 compute nodes, Spark-GSO is expectantly faster than MR-GSO for all swarm sizes tested. The speedup obtained however is better for MR-GSO than Spark-GSO. We have also executed Spark-GSO for higher dimensions (1,000, 2,000 and 3,000) to show that our algorithm is indeed scalable to higher dimensions.

As for future work, the basic RDD operations have been used and implemented to complete the algorithm, however in future we can use the concepts like Data-Frames in Spark to achieve even faster run-times. Furthermore, experiments with even larger dimensionality and population sizes will be conducted.

ACKNOWLEDGMENT

We gratefully acknowledge the support of this work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562.

REFERENCES

Aljarah, I., & Ludwig, S. A. (2013a). A MapReduce based glowworm swarm optimization approach for multimodal functions. In *2013 IEEE Symposium on Swarm Intelligence (SIS)* (pp. 22–31). IEEE. https://doi.org/10.1109/SIS.2013.6615155

Aljarah, I., & Ludwig, S. A. (2013b). A new clustering approach based on Glowworm Swarm Optimization. In *2013 IEEE Congress on Evolutionary Computation* (pp. 2642–2649). IEEE. https://doi.org/10.1109/CEC.2013.6557888

Aljarah, I., & Ludwig, S. A. (2016). A Scalable MapReduce-enabled Glowworm Swarm Optimization Approach for High Dimensional Multimodal Functions. *International Journal of Swarm Intelligence Research*, *7*(1), 32–54. https://doi.org/10.4018/IJSIR.2016010102

Apache Spark @Scale (2017) Apache Spark @Scale: A 60 TB+ production use case from Facebook - The Databricks Blog. Retrieved November 17, 2017, from https://databricks.com/blog/2016/08/31/apache-spark-scale-a-60-tb-production-use-case.html

Apache Spark^{TM} (2017) Apache Spark^{TM} - Lightning-Fast Cluster Computing. Retrieved November 17, 2017, from https://spark.apache.org/

Barrera, J., & Coello, C. A. C. (2009). A Review of Particle Swarm Optimization Methods Used for Multimodal Optimization (pp. 9–37). Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-04225-6_2

Big Data Analytics in the Cloud (2015). Big Data Analytics in the Cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf. *Procedia Computer Science*, *53*, 121–130. https://doi.org/10.1016/J.PROCS.2015.07.286

Dean, J., & Ghemawat, S. (2004). MapReduce: simplified data processing on large clusters. *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. USENIX Association. Retrieved from http://dl.acm.org/citation.cfm?id=1251264

Eiben, A. E. et al (1994). Genetic algorithms with multi-parent recombination. PPSN III: Proceedings of the International Conference on Evolutionary Computation. The Third Conference on Parallel Problem Solving from Nature: 78–87.

Engelbrecht, A. P. (2007). *Computational intelligence : an introduction*. John Wiley & Sons.

Gopalani, S., & Arora, R. (2015). Comparing Apache Spark and Map Reduce with Performance Analysis using K-Means. *International Journal of Computer Applications*, *113*(1), 975–8887. Retrieved from http://search.proquest.com.ezproxy.lib.ndsu.nodak.edu/docview/1672910329/F5DCA63258

EC495DPQ/1?accountid=6766

Grama, A. (2003). *Introduction to parallel computing*. Addison-Wesley.

Ismail, M. A. (2004). Parallel genetic algorithms (PGAs): master slave paradigm approach using MPI. In *E-Tech 2004* (pp. 83–87). IEEE. https://doi.org/10.1109/ETECH.2004.1353848

Jin, C., Vecchiola, C., & Buyya, R. (2008). MRPGA: An Extension of MapReduce for Parallelizing Genetic Algorithms. In *2008 IEEE Fourth International Conference on eScience* (pp. 214–221). IEEE. https://doi.org/10.1109/eScience.2008.78

Krishnanand, K. N., & Ghose, D. (2005). Detection of multiple source locations using a glowworm metaphor with applications to collective robotics. In *Proceedings 2005 IEEE Swarm Intelligence Symposium, 2005. SIS 2005.* (pp. 84–91). IEEE. https://doi.org/10.1109/SIS.2005.1501606

Krishnanand, K. N., & Ghose, D. (2008). Glowworm Swarm Optimization Algorithm for Hazard Sensing in Ubiquitous Environments Using Heterogeneous Agent Swarms. In *Soft Computing Applications in Industry* (pp. 165–187). Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-77465-5_9

Krishnanand, K. N., & Ghose, D. (2009a). Glowworm swarm optimisation: a new method for optimising multi-modal functions. *International Journal of Computational Intelligence Studies*, *1*(1), 93. https://doi.org/10.1504/IJCISTUDIES.2009.025340

Krishnanand, K. N., & Ghose, D. (2009b). Glowworm swarm optimization for simultaneous capture of multiple local optima of multimodal functions. *Swarm Intelligence*, *3*(2), 87–124. https://doi.org/10.1007/s11721-008-0021-5

Li, X., Engelbrecht, A., & Epitropakis, M. G. (2013). Benchmark Functions for CEC'2013 Special Session and Competition on Niching Methods for Multimodal Function Optimization. Retrieved from http://www.epitropakis.co.uk/sites/default/files/pubs/cec2013-niching-benchmark-tech-report.pdf

Liang, J. J., Qu, B. Y., Suganthan, P. N., & Chen, Q. (2014). Problem Definitions and Evaluation Criteria for the CEC 2015 Competition on Learning-based Real-Parameter Single Objective Optimization CEC 2015 Competition on Learning-based Real-Parameter Single Objective Optimization. Retrieved from https://pdfs.semanticscholar.org/9693/b1296203d421d9054d408e226f2e4d5ac15d.pdf

Matei Zaharia, Holden Karau, Andy Konwinski, P. W. (2015). *Learning Spark*. O'Reilly Media. Retrieved from http://shop.oreilly.com/product/0636920028512.do

McNabb, A. W., Monson, C. K., & Seppi, K. D. (2007). Parallel PSO using MapReduce. In *2007 IEEE Congress on Evolutionary Computation* (pp. 7–14). IEEE. https://doi.org/10.1109/CEC.2007.4424448

Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., … Talwalkar, A. (2016). MLlib: Machine Learning in Apache Spark. *Journal of Machine Learning Research*, *17*, 1–7. Retrieved from http://www.jmlr.org/papers/volume17/15-237/15-237.pdf

Miryala, G. (2017). *Journal of Global Research in Computer Science[[Elektronische*

Ressource]] JGRCS. Journal of Global Research in Computer Science(UGC Approved Journal) (Vol. 8). [s.n.]. Retrieved from http://www.jgrcs.info/index.php/jgrcs/article/view/1015

MLlib - Apache Spark. (2017). Retrieved November 16, 2017, from https://spark.apache.org/mllib/

Paduraru, C., Melemciuc, M.-C., & Stefanescu, A. (2017). A distributed implementation using apache spark of a genetic algorithm applied to test data generation. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion on - GECCO '17* (pp. 1857–1863). New York, New York, USA: ACM Press. https://doi.org/10.1145/3067695.3084219

Snir, M. (1998). *MPI--the complete reference*. MIT Press. Retrieved from https://mitpress.mit.edu/books/mpi-complete-reference-0

Spark Streaming - Apache Spark. (2017). Retrieved November 16, 2017, from https://spark.apache.org/streaming/

Stützle, T. (2009). Ant Colony Optimization: Evolutionary Multi-Criterion Optimization (pp. 2–2). https://doi.org/10.1007/978-3-642-01020-0_2

Tan, Q., He, Q., & Shi, Z. (2012). Parallel Max-Min Ant System Using MapReduce (pp. 182–189). Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-30976-2_22

Venter, G., & Sobieszczanski-Sobieski, J. (2006). Parallel Particle Swarm Optimization Algorithm Accelerated by Asynchronous Evaluations. *Journal of Aerospace Computing, Information, and Communication*, *3*(3), 123–137. https://doi.org/10.2514/1.17873

Wong, L.-P., Low, M. Y. H., & Chong, C. S. (2008). A Bee Colony Optimization Algorithm for Traveling Salesman Problem. In *2008 Second Asia International Conference on Modelling & Simulation (AMS)* (pp. 818–823). IEEE. https://doi.org/10.1109/AMS.2008.27

Wu, B., Wu, G., & Yang, M. (2012). A MapReduce based Ant Colony Optimization approach to combinatorial optimization problems. In *2012 8th International Conference on Natural Computation* (pp. 728–732). IEEE. https://doi.org/10.1109/ICNC.2012.6234645

Wu, K., Zhu, Y., Li, Q., & Han, G. (2017). Algorithm and Implementation of Distributed ESN Using Spark Framework and Parallel PSO. *Applied Sciences*, *7*(4), 353. https://doi.org/10.3390/app7040353

Zhou, C., & Chi. (2010). Fast parallelization of differential evolution algorithm using MapReduce. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation - GECCO '10* (p. 1113). New York, New York, USA: ACM Press. https://doi.org/10.1145/1830483.1830689