IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received April 30, 2021, accepted May 12, 2021, date of publication May 19, 2021, date of current version June 2, 2021.
Digital Object Identifier 10.1109/ACCESS.2021.3082026

Parallel and Distributed Implementation of Sine Cosine Algorithm on Apache Spark Platform

MOHAMMAD GH. ALFAILAKAWI [1], **MARYAM ALJAME** [1,2], **AND IMTIAZ AHMAD** [1]
[1]Computer Engineering Department, Kuwait University, Khaldiyah 13060, Kuwait
[2]Educational Software Department, Ministry of Education, Kuwait city 12013, Kuwait
Corresponding author: Mohammad Gh. AlFailakawi (alfailakawi.m@ku.edu.kw)

ABSTRACT The Sine Cosine Algorithm (SCA) has experienced wide spread use in solving optimization problems in many disciplines mainly due to its simplicity and efficiency. However, like many other meta-heuristics, SCA requires considerable amount of compute time when solving large size optimization problems. Therefore, in order to tackle such challenging problems efficiently, this work proposes Spark-SCA, a scalable and parallel implementation of SCA algorithm on Apache Spark distributed framework. Spark-SCA exploits Spark platform native support for iterative algorithms through in-memory computing to speed-up computations when handling large optimization problems. Both the design and implementation details of Spark-SCA are presented herein. The performance of Spark-SCA was compared to standard SCA on different benchmark functions with up to 1,000-dimension as well as three practical engineering design problems. Simulation experiments conducted on Amazon Web Services (AWS) public cloud demonstrated how Spark-SCA outperforms the standard version in terms of solution quality and run time as well as it competitiveness in exploring solution space of complex optimization problems.

INDEX TERMS Apache spark, cluster, Hadoop, meta-heuristics, sine cosine algorithm.

I. INTRODUCTION

Majority of the challenging real-world problems that arise nowadays in many disciplines can be classified as optimization problems. Such complex problems require the algorithm to efficiently and effectively explore their associated search space to find good solutions. Population-based meta-heuristics have been the dominant methods to find optimal or near-optimal solutions to many optimization problems within a reasonable time [1]. These meta-heuristics derive their inspiration from mimicking intelligent processes arising in nature. Meta-heuristics can be divided into evolutionary algorithms (EAs) such as genetic algorithms (GAs), differential evolution (DE); and swarm intelligence algorithms such as particle swarm (PSO), ant colony (ACO), grey wolf (GWO) [2], phylogram analysis (OPA) [3], and cuckoo search (CS) among others [4]. To further improve the performance of meta-heuristics, researchers have applied a variety of techniques such as stochastic operators [5] or hybridization to solve specific optimization problems [6]–[8]. Due to the popularity of meta-heuristics in successfully solving optimization problems, these algorithms are being introduced in engineering and other curricula to equip students with the required skills for the market specifically in the emerging field of machine learning [9].

Since no algorithm can solve all optimization problems as per the ‘‘No Free Lunch’’ theorem [10], researchers have put forward new optimization algorithms for solving problems in diverse fields. The Sine Cosine Algorithm (SCA) [11] is a new population-based algorithm that utilizes the oscillating property of the sine and cosine functions to explore the search space to find a good solution for a given problem. SCA has attracted a widespread usage in solving many practical problems due to its simplicity, flexibility, and effectiveness. SCA was successfully applied to interesting problems such as pairwise global sequence alignment, hydrothermal scheduling, feature selection, medical diagnostic, and CMOS analog circuits optimization among others. For a detailed list of applications, the reader is referred to a recent survey by Mirjalili *et al.* [12]. Due to some inherent weaknesses in SCA for solving certain type of benchmarks, many researchers have started to find ways to enhance SCA's exploitation and exploration capabilities either by introducing new stochastic operators or by hybridizing it with other algorithms.

A memory guided sine cosine algorithm (MG-SCA) was proposed in [13] where a memory matrix of personal best

The associate editor coordinating the review of this manuscript and approving it for publication was Pavlos I. Lazaridis.

solutions is used to guide solution evolution. A combination of four different strategies; Cauchy mutation operator, chaotic local search, mutation and crossover strategies, and opposition based learning, adapted from DE were employed to improve SCA performance [14]. An improved symmetric SCA with adaptive probability selection has been proposed in [15] to enhance SCA exploitation capabilities through horizontally flipped symmetric sine and cosine functions. A multi-core SCA approach that combined three strategies from three other meta-heuristics to enhance exploration capabilities was proposed in [16]. Another recent technique proposed in [17] combines chaotic local search and levy flight operator from cuckoo search with standard SCA to boost its performance. The authors in [18] introduced a multi-group multi-strategy to enhance SCA capabilities. In their approach, the population is partitioned into multiple groups with the same number of individuals and each group executes in parallel for a certain number of iterations using different update strategy and without any communication among groups. After reaching a certain number of iterations, groups communicate with each other to replace the worst individuals by the best one. Experimental results have demonstrated considerable improvement in SCA's exploratory and exploitative properties.

Nevertheless, with the ever-increasing scale, dimensionality and complexity of today's realistic problems, meta-heuristics require enormous amount of time to find good solutions. However, due to their inherent parallelism, population-based meta-heuristics have the potential to greatly benefit from parallel platforms such as Field Programmable Gate Arrays (FPGAs) [19], GPUs [20] as well as distributed platforms such as Apache Hadoop [21] and Apache Spark [22]. The distributed platforms are being preferred over other parallel platforms due to their flexibility, scalability and availability of cloud resources. Therefore, the parallelization of meta-heuristics on emerging distributed frameworks such as Apache Spark offers an interesting opportunity to speed-up computations, handle large optimization problems, or further improve search ability of algorithms. Spark is an emerging throughput-oriented distributed computing framework with enhancement to efficiently support iterative algorithms through in-memory computing [22]. Different meta-heuristic algorithms were parallelized using Spark framework demonstrating good performance for large scale problems. At the present time, traditional meta-heuristics such as GA [23], PSO [24], ACO [25], tabu search (TS) [26] as well as more recent ones such as whale optimization [27] and scatter search [28] have been successfully parallelized using Spark platform. However, a Spark based parallelization of SCA is yet to be implemented which is the topic of this work.

Motivated by the demand for a parallel version of SCA [17], this paper proposes Spark-SCA, a distributed version of the original SCA on Apache Spark framework. Experimental results have demonstrated how Spark-SCA outperformed the standard version of the algorithm in terms solution quality and run time. The main contributions of this work can be summarized as follows:

1) A novel distributed SCA based on Apache Spark is proposed.
2) The performance of Spark-SCA is compared to its serial version on several benchmarks as well as three real engineering problems.
3) The impact of communication cost on the performance of Spark-SCA is analyzed.

The remainder of this paper is organized as follows: Section II provides a brief description of the sine cosine algorithm as well a short overview of Apache Spark platform. Section III discusses the details of Spark-SCA algorithm. Section IV presents results showcasing the performance of the proposed algorithm on unimodal, multimodal, and composite benchmark functions whereas Section V gives the results for three well-known engineering optimization problems. Conclusions drawn and future directions are given in Section VI.

II. BACKGROUND

A. SINE COSINE ALGORITHM (SCA)

Sine Cosine Algorithm (SCA) is a stochastic population-based optimization algorithm proposed by Mirjalili in 2016 [11]. SCA name comes from the mathematical functions sine and cosine. The Sine Cosine Algorithm begins the optimization process by generating a set of random solutions known as the initialization phase. Then, the optimization process starts where an objective function is applied to these solutions to evaluate their quality where the sine and cosine functions are used to modify these solutions to improve their quality in an iterative fashion. This optimization process is repeated until a terminating condition is satisfied. The following equations represent how SCA algorithm modify current solutions to reach possibly better ones:

$$X_i^{t+1} = \begin{cases} X_i^t + r_1 \cdot sin(r_2) \cdot |(r_3)P_i^t - X_i^t| & r_4 < 0.5 \\ X_i^t + r_1 \cdot cos(r_2) \cdot |(r_3)P_i^t - X_i^t| & r_4 \geqslant 0.5 \end{cases} \tag{1}$$

where X_i^t represents the current solution position in dimension i at the t-th iteration. Similarly, P_i^t indicates the destination position in the i-th dimension at iteration t, $||$ is the absolute value and r_4 is a random value $\in [0, 1]$. The parameter r_4 is used to control the switching between using either sine or cosine function to update the solution as indicated in Eq. (1). The remaining random parameters r_i; r_1, r_2, and r_3, are used to determine how solutions characteristic are modified. In particular, parameter r_1 is responsible for determining the direction (region) of movement described mathematically by:

$$r_1 = a - ta/T \tag{2}$$

where a is a constant, t and T represent the current and maximum number of iterations, respectively. The r_2 parameter is used to specify the amount of movement toward/away from the destination. Parameter r_3 is a random weighting

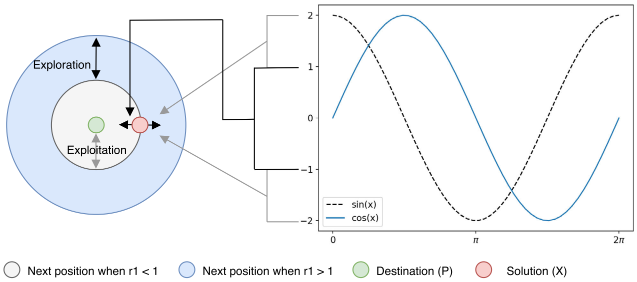

FIGURE 1. Search agents movement according to the impact of Sine and Cosine.

factor that emphasis ($r_3 > 1$) or deemphasize ($r_3 < 1$) the impact of the destination in defining the distance. Algorithm 1 gives the pseudocode of SCA algorithm. Initially in line 1, the algorithm starts with initializing a set of random solutions followed by an evaluation step where the objective function is calculated and the best solution is saved as the destination point (lines 3-4). Then, r_i parameters are updated and the value of the destination is calculated using Eq. (1) to update the current solutions (lines 5-6). These steps, with the exception of step 1, are repeated until the terminating condition is reached (line 7) where the best solution is identified and returned as shown in line 8.

Algorithm 1 SCA Algorithm

1: Initialize a set of search agents (solutions) (X)
2: **do**
3: Evaluate each search agent using the objective function
4: Identify the best solution obtained so far ($P = X^*$)
5: Update r_1, r_2, r_3, and r_4
6: Update position of each search agent using Eq. (1)
7: **while** (t < maximum iterations)
8: Return the best solution obtained as the global optimum

An optimization algorithm should guarantee proper exploration and exploitation of the search space to realize global optimum. Figure 1 illustrates how the sine and cosine functions affect the movement of search agents with respect to the destination in the range $[-2, 2]$. As shown in Figure 1, exploration is identified by the regions $[-2, -1)$ and $(1, 2]$ while exploitation happens between $[-1, 1]$. Figure 1 depicts how the position of a solution is updated to the next random location either inward or outward as compared to the destination. SCA achieves this position update by introducing r_2 in Eq. (1) where r_2 is a random number $\in [0, 2\pi]$.

B. APACHE SPARK

Apache Spark [29] is a distributed in-memory computing framework based on MapReduce paradigm designed for big data processing. Spark was developed by University of California, Berkeley [22]. Spark's in-memory primitives make it an appropriate framework to query data repeatedly without the need for accessing system storage/disk. Therefore, Spark is a well suited framework for iterative processing, batch applications, streaming as well as interactive queries. Spark as a big data processing framework has several configuration parameters that control parallelism, computing resources, compression, and I/O operations [30]. Setting those parameters is a crucial step for performance improvement and resource utilization. For instance, partition tuning is essential as it determines the degree of parallelism. Consequently, setting a sufficient number of partitions leads to better resource utilization [30]. In fact, allocating optimal parameters is an NP-hard problem [31]; hence, many recent studies have emerged to propose parameters tuning methods including [32]–[36].

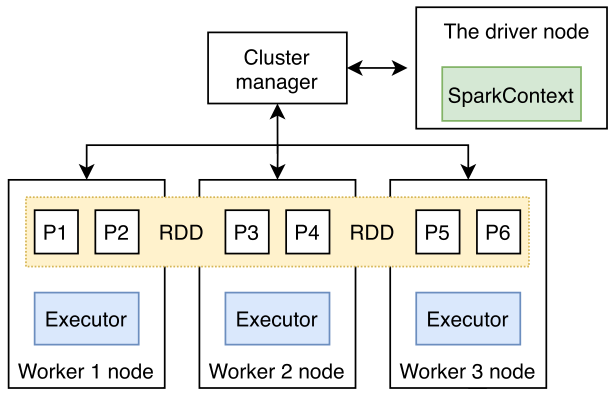

FIGURE 2. Apache Spark architecture.

Figure 2 illustrates Spark architecture which is a master/slave architecture where the driver node is the master and worker nodes are the slaves. The driver node converts a Spark program into multiple tasks and distributes them to worker nodes on the cluster. Each worker node has an executor that executes the tasks assigned to it. One of the responsibilities of SparkContext is to establish a connection to the cluster manager. Spark supports several cluster managers including standalone cluster, Hadoop YARN, Amazon EMR, and Apache Mesos. The Resilient Distributed Dataset (RDD) [37] is the essence of Spark which represents an immutable collection of data partitioned across worker nodes on the cluster as shown in Figure 2. RDDs are created by applying operations on data. There are two types of operations in Spark, namely, transformations and actions. Transformations apply a function on an existing RDD resulting in creating a new RDD. On the other hand, actions return a value to the driver program or store it to a storage system such as Hadoop Distributed File System (HDFS). Each RDD stores its own *lineage*, a set of transformations that has been applied to the RDD. In case of data lost, Spark achieves fault-tolerance by using the lineage to reconstruct lost data and thus evade the need for costly data replication or checkpointing. During parallel operations, Spark uses shared variables across all worker nodes on the cluster. Spark has two types of shared variables, accumulators and broadcast variables. Accumulators are used to aggregate commutative operations such as counters or sum values. On the other hand, broadcast variables are cached read-only variables that are available on each worker node thus allowing efficient data sharing. Spark supports different programming languages namely Scala, Java, Python and R [38].

III. SPARK-SCA ALGORITHM

The Sine Cosine Algorithm is an optimization algorithm that begins with a set of random solutions where it uses an objective function to repeatedly evaluate solutions fitness. In fact, each iteration depends on the previous one which signifies the serial nature of algorithm execution. This paper proposes Spark-SCA, an algorithm that aims to parallelize SCA serial evaluations in order to reduce execution time. Fitness evaluation process is the bottleneck of any optimization algorithm as it requires the evaluation of the whole population. The algorithm's time complexity scales as population size increases, therefore, in this work we employ Apache Spark to divide SCA population into several subpopulations where each subpopulation computes its own fitness independently. Communication between subpopulations is controlled by the algorithm and in our implementation it is a function of the total number of iterations as will be discussed later. Moreover, during subpopulation fitness evaluation, a Spark broadcast variable is used to communicate to other subpopulation best fitness individual (destination point) currently available. Implementation details of the proposed algorithm, Spark-SCA, are discussed next.

Algorithm 2 Spark-SCA

Input: X = population size, N = number of partitions, D = dimension size, fn = objective function, T = maximum number of iterations

Output: best solution (fitness)

1: $[A]$ = sc.parallelize(List.range(1, X), N)
2: [<F, A>] = $[A]$.mapPartitions( ) ▷ evaluate agents using *fn*
3: <F_{best}, A> = [<F, A>].glom().map(minBy(*key*)) .reduce(min(*key*))
4: $\text{F}_{agentBC}$ = sc.broadcast(<F_{best}, A>)
5: **repeat**
6:     Update r_1, r_2, r_3, and r_4
7:     $[A]$ = [<F, A>].map( ) ▷ update agents position using Eq. (1)
8:     [<F, A>] = $[A]$.mapPartitions( ) ▷ evaluate agents using *fn*
9:     <$\text{F}_{bestTmp}$, A> = [<F, A>].glom().map (minBy(*key*)).reduce(min(*key*))
10:     **if** <$\text{F}_{bestTmp}$, A>._1 < ($\text{F}_{agentBC}$.value)._1 **then**
11:         $\text{F}_{agentBC}$.destroy
12:         $\text{F}_{agentBC}$ = sc.broadcast(<$\text{F}_{bestTmp}$, A>)
13:     **end if**
14:     $t = t + 1$
15: **until** $t > T$
16: return best solution (fitness)

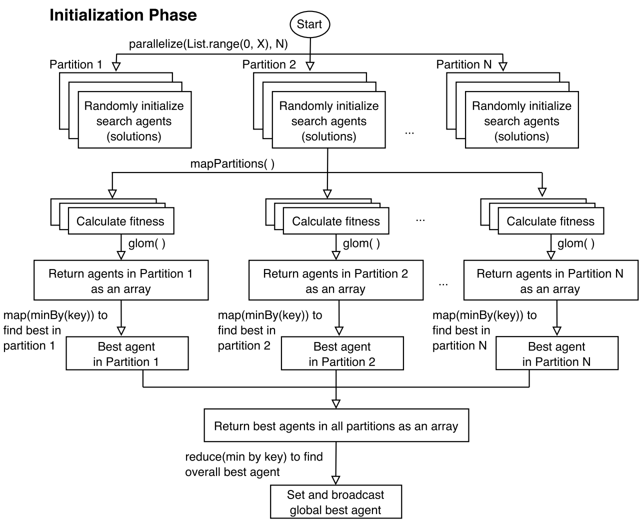

FIGURE 3. Spark-SCA initialization phase.

The pseudocode of Spark-SCA is given in Algorithm 2. Figure 3 shows flowchart of the initialization phase of Spark-SCA. Spark-SCA starts with a population size of X and a number of partitions equal to N. Initialization begins with generating a random set of agents (solutions). Those agents are of dimensions D and they are constrained between a lower and an upper bound of the objective function. During the initialization phase, Spark *parallelize* method is used to divide population X into independent subpopulations over N partitions. As can be seen in Algorithm 2 in line 1, this step produces an RDD $[A]$ that has the following records: $[agent_1, agent_2, \ldots, agent_X]$. On each partition, the fitness of each search agent in the subpopulation is computed in parallel by applying Spark *mapPartitions*. Line 2 shows how fitness computation generates a new RDD [<F, A>] where each record is a key-value pair where value A represents a search agent (value) and its fitness as a key (i.e. F). Subsequently, in line 3, the *glom*() function returns each partition subpopulation as an array and *map*(*minBy*(key)) is used to find the minimum fitness search agent as a key-value pair <F_{best}, A>. Consequently, each partition will find its fittest agent thereby N partitions will generate an array of N minimum fitness-agent [<F_{best}, A>]. Next, *reduce*(*min*(key)) is applied to find the overall best fitness between all partitions and broadcast it to all nodes in the cluster. Finding the best fitness-agent requires a massive amount of shuffling between partitions which is very costly, therefore, Spark's *glom*() function is used in the proposed algorithm to reduce shuffling. This reduction is achieved by finding first the fittest agent in each partition rather than comparing all agents in all partitions. Once the best agent per partition is found, a comparison between best fitness for all partitions is performed in identifying the global best. Such approach for finding best fitness-agent reduces communication overhead significantly.

After the initialization phase, Spark-SCA starts the optimization phase wherein at each iteration the fittest agent for each subpopulation is found. Then, the optimization phase

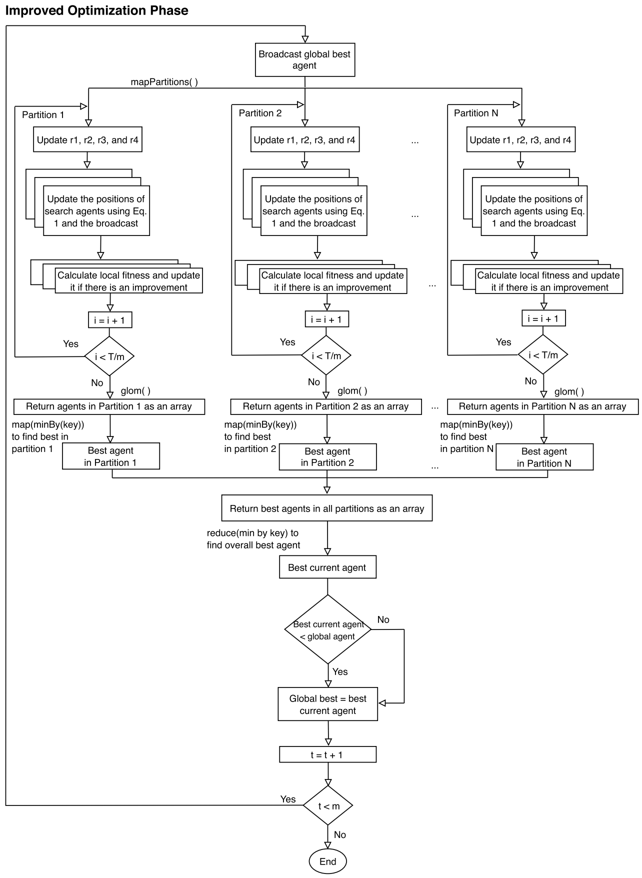

FIGURE 4. Spark-SCA improved optimization phase.

starts where r_i parameters are updated in line 6 and then, in line 7, search agents positions are updated by applying Eq. (1) which utilizes the overall best agent ($F_{agentBC}$) found so far. Line 8 and 9 show how the best fitness-agent for each partition is found in a similar fashion to what was done in the initialization phase. Then, in line 10, the fittest agent found in this iteration is compared to the current best "global" fitness-agent. The agent with the minimum fitness among all partitions is then broadcasted if it was found to be superior as compared to the current global best. The loop counter is then incremented (line 14) and the process is repeated until the maximum number of iterations is reached.

This Spark-SCA implementation suffers from increased communication overhead. As it can be seen in Algorithm 2, the broadcast operation is performed after each iteration and for each subpopulation resulting in a significant amount of communication in the cluster ($N*T$ broadcasts) and thus negatively effecting run-time characteristic of the algorithm. In this next subsection, we present an improved version of Spark-SCA to overcome such deficiency.

A. IMPROVED SPARK-SCA

The proposed improvement to Spark-SCA is limited to the optimization phase of the algorithm. Since the algorithm splits the population between partitions where each subpopulation improves on its own search agents, the enhancement proposed is to limit the number of broadcasts performs by each partition. In other words, instead of broadcasting best agent information on every iteration, Spark-SCA will limit broadcasts to occur only after a certain number of iterations. Algorithm 3 gives the pseudocode of the improved Spark-SCA algorithm and its flowchart is illustrated in Figure 4. The improvement has two loops: a main loop and a partition (inner) loop. In this version of the algorithm, the broadcast operation is only performed m times which is a user defined parameter. This means that each subpopulation will successively operate on improving its own population for a number of iterations equal to T/m times before any broadcast is performed, where T represents the maximum number of iterations.

For example, if T was set to 100 with $m = 2$, then each partition will iterate 50 times to find its best local agent before broadcasting it to the cluster. In Algorithm 3, line 9 shows how the best local fitness for each partition is set before starting the inner loop (lines 9-18) where its set initially to the global best value. During inner loop, F_{local} is updated according to the fittest agent in the partition (i.e. line 12). This is realized by setting $F_{local} = F_{tmp}$ where this will be repeated until the limit for the inner loop is reached. The rest of the algorithm works in a similar manner as before.

B. ILLUSTRATIVE EXAMPLE OF SPARK-SCA

This subsection gives an illustrative example of how Spark-SCA's optimization phase works. The benchmark used in this example is Ackley which is a multimodal benchmark with lower and upper bound in range [−32, 32]. Table 8 lists the mathematical description of Ackley benchmark and its plot is illustrated in Figure 10 in the appendix. For simplicity, population size for this example was set to 12 agents with each agent having a dimension of 2. As shown in Figure 5, the current example has three partitions, the population is distributed among them where each partition gets four agents. From the initialization phase the global best fitness has been set to 2.85. Subsequently, the optimization phase starts with this broadcasted fitness (2.85) and using it as the current best fitness for each partition. As mentioned in subsection III-A the inner loop for each partition will repeat the optimization process for T/m times. Let us demonstrate the optimization process numerically by considering the third partition. During the inner loop, at a specific iteration the third partition has the following agents: [1.19, 0.26], [0.57, 0.34], [0.85, 0.41], and [0.57, 0.20] as shown in the top right corner of Figure 5. Spark-SCA processes this subpopulation by calculating the fitness of its agents and creating fitness-agent key-value pairs. After that, fitness of the different agents are compared to the local best and the local best fitness is updated if appropriate. As depicted in the upper right corner of Figure 5, the first iteration shown has a minimum fitness equals to 3.63 which is greater than the local fitness. Thus, the local fitness is left unchanged and the loop will continue the optimization process. In the next iteration, the parameters are updated and

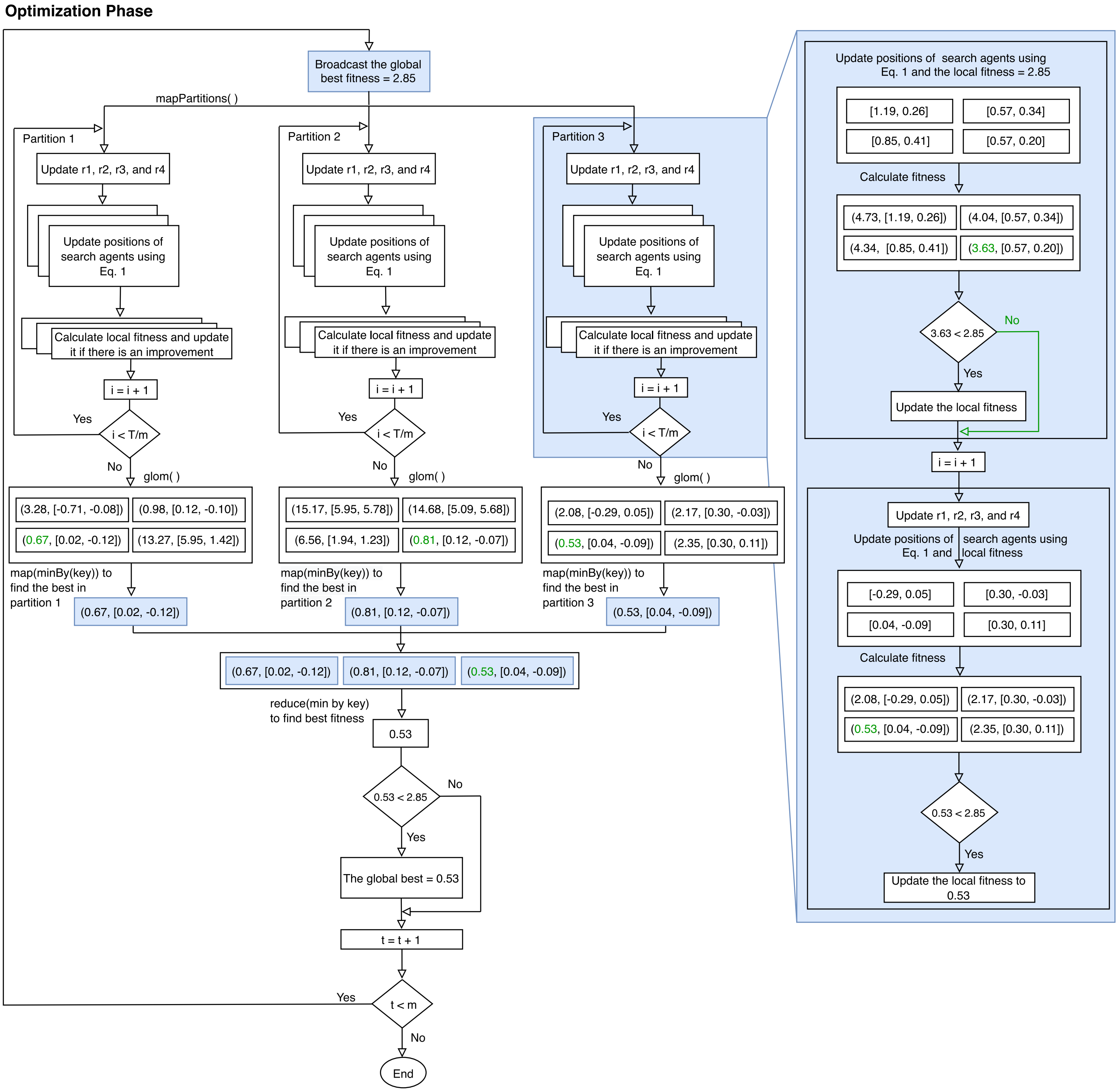

FIGURE 5. **An illustrative example of Spark-SCA optimization phase.**

agents' positions are updated using the current local fitness (2.85) and Eq. (1). Updating the search agents resulted in the following new agents [−0.29, 0.05], [0.30, −0.03], [0.04, −0.09], and [0.30, 0.11] as shown in the lower right corner of Figure 5. The fitness of these agents is calculated and the fittest agent in the case has a fitness of 0.53. Assuming this is the last iteration of the inner loop, the *glom*() operation is used to return these agents as an array to be processed by the main loop of the algorithm. Note that the same process is performed by the other two partitions resulting three different arrays, one for each partition. Then, each partition will find its own best fitness-agent using *map*(*minBy*(key)) transformation. Following that, *reduce*(*min*(key)) function is used to compare the fittest agents from each partition and find the best among all partitions to be compared with the current global best. Since the fittest agent in partitions 1, 2, and 3 has a fitness value of 0.67, 0.81, and 0.53 respectively, the fitness of partition 3 agent is compared to the global best fitness of 2.85. Since the new fitness is better than the current global best, partition 3 agent now becomes the new global best and is broadcasted to all partitions. Then, the loop counter will be incremented and the termination condition is checked. Spark-SCA uses the *glom*() function to reduce communication overhead. To demonstrate let us consider the current example of finding the minimum fitness between 12 agents that are distributed among 3 partitions.

Algorithm 3 Improved Spark-SCA

```
Input: X = population size, N = number of partitions,
       D = dimension size, fn = objective function,
       T = maximum number of iterations,
       m = inner loop iterations
Output: best solution (fitness)
[A] = sc.parallelize(List.range(1, X), N)
[<F, A>] = [A].mapPartitions        ▷ evaluate agents using fn
<F_best, A> = [<F, A>].glom().map(minBy(key)) .reduce(min(key))
F_agentBC = sc.broadcast(<F_best, A>)
repeat
    Update r1, r2, r3, and r4
    [A] = [<F, A>].map( )        ▷ update agent position using Eq. (1)
    [<F, A>] = [A].mapPartitions
    repeat        ▷ inner loop in each partition
        F_local = F_agentBC.value
        for A in partition do
            if F_tmp < F_local then
                F_local = F_tmp
            end if
        end for
        Update r1, r2, r3, and r4
        Update agents position using Eq. (1) and F_local
    until i > T/m
    <F_bestTmp, A> = [<F, A>].glom().map(minBy (key)).reduce(min(key))
    if <F_bestTmp, A>._1 < (F_agentBC.value)._1 then
        F_agentBC.destroy
        F_agentBC = sc.broadcast(<F_bestTmp, A>)
    end if
    t = t + 1
until t > m
return best solution (fitness)
```

To find the minimum fitness in the population, the first step is to find the minimum fitness in each partition. By applying *glom*() which will return all agents in each partition in an array. Then, each partition best fitness is found by applying *map*(*minBy*(key)) on the partition array. After that, all partitions best fitness are returned in one array with three elements one for each partition. Thereafter *reduce*(*min*(key)) is utilized to find the best fitness between all partitions. In fact, Spark *reduce*() operation is an action which requires shuffling which requires a great deal of communication. Hence, rather than shuffling and comparing the whole population (12 agents), only 3 agents are shuffled and thus reducing the number of comparisons to 3 only.

IV. SPARK-SCA EVALUATION

To study the efficiency of Spark-SCA, its performance as compared to the serial SCA was evaluated on nine benchmark functions. The details of these benchmark functions are listed in the appendix. For the sake of fair comparison between the parallel and the serial version, equal number of function evaluations were used in both implementations. Further, experimental results shown are the average values of best fitness/runtime obtained for the various benchmarks for 30 independent runs. The first subsection compares Spark-SCA to the standard version whereas the second subsection evaluates the impact of number of broadcasts as well as number of nodes on algorithm performance.

TABLE 1. Algorithm parameter settings.

Algorithm	Parameter settings	Random variables
Spark-SCA	Population size $X = 96$ Dimension size $D = 50$ Maximum number of iterations $T = 100$ Number of partitions $N = 4$	r_1, r_2, r_3 $r_4 \in [0,1]$
SCA	Population size $X = 32$ Dimension size $D = 50$ Maximum number of iterations $T = 300$	r_1, r_2, r_3 $r_4 \in [0,1]$

A. SPARK-SCA VS. STANDARD SCA

This subsection compares Spark-SCA and the serial version using benchmark functions with dimensions 50, 250, and 1,000. This experiment is conducted to study the impact of benchmarks size on algorithms performance. Both implementations were run on Amazon Elastic MapReduce (EMR) which is one of AWS tools that provides a fully managed big data processing framework on top of Amazon Elastic Compute Cloud (EC2). Both algorithms use EC2 node of type m4.xlarge which has 4 vCPU and 16 Mem (GiB). The serial SCA uses a population size of 32 with maximum number of iterations set to 300. As for Spark-SCA, a population size of 96 was used with maximum number of iterations of 100 resulting in a total number of function evaluations for each implementation to be 9,600. In this experiment, two versions of Spark-SCA were used where the first one uses a single broadcast to return the final answer whereas the other one uses a broadcast operation on every iteration. Table 1 lists the parameter settings for SCA and Spark-SCA with 100 broadcasts (Spark-100) for the case when dimensions size is equal to 50. In Table 2, Spark-1 and Spark-100 are used to represent the different versions where the number of broadcast is 1 and 100, respectively. The SCA column gives the result for serial SCA. Moreover, the table also reports the best fitness as well as speedup as compared to the serial case. Values in bold face represent best value obtained over all cases considered (SCA, Spark-1, Spark-100). Table 2 shows a summary of the results obtained. It is apparent from the table that Spark-100 provides the best fitness for all benchmarks except for composite benchmarks regardless of problem size. We believe for the composite benchmarks, the serial version of the algorithm was able to reach the best fitness due to the large population size in the serial version as compared to Spark implementation which allows the algorithm to effectively escape local minima. Another important observation is that for small size benchmarks (low dimension), run-time

TABLE 2. Spark-SCA vs SCA.

Benchmark	Dim.	Avg. normalized fitness						Speedup	
		Spark-100		Spark-1		SCA		Spark-100	Spark-1
		ave	std	ave	std	ave	std		
Sphere (F1)	50	5.729e-09	3.654e-08	3.944e-4	0.003	0.058	0.023	0.039	0.467
	250	**0.0**	0.0	7.253e-4	0.005	0.215	0.188	0.125	1.447
	1,000	1.849e-07	1.232e-06	0.002	0.011	1.0	1.0	0.318	**1.768**
Schwefel 2.21 (F4)	50	6.163e-05	0.004	0.007	0.345	0.929	0.205	0.04	0.488
	250	**0.0**	0.0	0.002	0.052	0.988	0.056	0.15	1.695
	1,000	3.318e-4	0.021	0.027	1.0	1.0	0.012	0.317	**1.941**
Rosenbrock (F5)	50	**0.0**	0.0	1.617e-07	3.353e-05	0.032	0.324	0.084	0.86
	250	8.864e-09	3.279e-07	2.209e-05	0.005	0.225	0.633	0.203	2.106
	1,000	3.809e-08	1.781e-06	2.108e-4	0.067	1.0	1.0	0.211	**1.927**
Rastrigin (F9)	50	**0.0**	0.0	0.022	0.019	0.245	0.041	0.052	0.985
	250	2.863e-05	3.544e-05	0.089	0.132	0.585	0.213	0.186	3.026
	1,000	6.907e-05	1.227e-4	1.0	1.0	0.976	0.280	0.389	**4.515**
Ackley (F10)	50	4.281e-4	0.034	0.019	0.676	0.997	0.393	0.059	0.685
	250	**0.0**	0.0	0.016	0.403	0.983	0.857	0.18	1.526
	1,000	1.836e-4	0.005	0.011	0.528	1.0	1.0	0.361	**2.932**
Griewank (F11)	50	1.257e-06	5.155e-06	0.002	0.009	0.067	0.025	0.071	0.866
	250	**0.0**	0.0	3.166e-05	9.358e-05	0.318	0.204	0.154	1.361
	1,000	7.348e-06	3.883e-05	0.072	0.417	1.0	1.0	0.375	**2.708**
CF3 (F16)	50	0.018	0.029	1.0	1.0	0.001	0.002	0.036	0.731
	250	0.064	0.180	0.836	0.901	0.001	0.002	0.15	2.417
	1,000	0.027	0.039	0.790	0.828	**0.0**	0.0	0.393	**4.381**
CF4 (F17)	50	0.026	0.047	0.816	0.839	0.021	0.031	0.037	0.733
	250	0.147	0.295	0.938	1.0	**0.0**	0.0	0.145	2.516
	1,000	0.030	0.068	1.0	0.969	0.005	0.007	0.328	**3.755**
CF5 (F18)	50	0.048	0.064	0.817	1.0	1.344e-4	1.292e-4	0.04	0.719
	250	0.019	0.044	1.0	0.979	2.471e-4	3.358e-4	0.136	1.921
	1,000	0.044	0.059	0.873	0.837	**0.0**	0.0	0.34	**4.437**

characteristic of the Spark versions perform poorly even for the case where a single broadcast is used. However, as the size of the problem increases, Spark-1 version outperforms the serial version in term of run time. Moreover, Spark-100 implementation does not provide any speedup as compared to the serial version due to the prohibitive communication cost associated with this implementation of the algorithm. To validate the significance of the results, a non-parametric statistical test named the Wilcoxon rank-sum test is conducted to determine the statistical significance of the results between the proposed algorithm and the original SCA. Table 3 lists the p-values of the Wilcoxon rank-sum test where the desired level of significance p is set to 0.05. In the Wilcoxon rank-sum test, there is a significant difference between the two algorithms when the p-value is less than 0.05 otherwise the difference is negligible. Most of the benchmarks have p-value less than 0.05 which demonstrates that the enhancement in solution quality obtained by the proposed algorithm is statistically significant. From this experiment, it can be concluded that Spark-SCA is only appropriate when dealing with large benchmarks to reap any run-time benefits from parallelizing the algorithm. Moreover, it is crucial to balance communication between worker nodes to find better quality solution but it should be done in such a way that it does not offset speedup gains. This fact is demonstrated in the next section as we study the impact of broadcasts on fitness and run-time characteristics.

TABLE 3. *p*-value results of wilcoxon rank sum test.

Benchmark	Spark-100 vs SCA	Spark-1 vs SCA
Sphere (F1)	0.0495	0.0495
Schwefel 2.21 (F4)	0.0495	0.0495
Rosenbrock (F5)	0.0495	0.0495
Rastrigin (F9)	0.0495	0.5127
Ackley (F10)	0.0495	0.0495
Griewank (F11)	0.0495	0.1266
CF3 (F16)	0.0495	0.0495
CF4 (F17)	0.0495	0.0495
CF5 (F18)	0.0495	0.0495

B. BROADCAST IMPACT ON PERFORMANCE

In this subsection, we study the impact of using different number of broadcasts on three different cluster sizes. Three different EMR cluster sizes of four, eight, and sixteen nodes were used with each cluster having a population size of 96. The same benchmarks used in the previous subsection are used again here but limiting the dimension to be 1,000 and maximum number of iterations equal to 100. The four nodes cluster has one master node and three slaves with 12 partitions, the eight nodes cluster consists of one master node and seven slaves with 28 partitions, and the sixteen nodes cluster contains one master, fifteen slaves and number of partitions equals to 60. Again, these setting were used to unify the number of function evaluations to be 9,600 for all implementations. The node type used in this experiment is m4.xlarge which has 4 vCPU and 16 (GiB) Memory. Table 4 gives the average normalized fitness for the three clusters

TABLE 4. Normalized fitness vs. cluster size/broadcasts.

Benchmark	BC #	4 nodes Avg Norm. Fitness ave	4 nodes std	8 nodes Avg Norm Fitness ave	8 nodes std	16 nodes Avg Norm Fitness ave	16 nodes std
Sphere (F1)	1	1.0	1.0	1.015e-4	1.764e-4	1.558e-05	3.049e-05
	2	4.202e-2	7.332e-2	3.244e-06	3.645e-06	5.424e-09	6.435e-09
	3	5.266e-2	0.101	1.074e-4	2.099e-4	1.334e-07	2.431e-07
	4	0.313	0.575	1.317e-06	1.408e-06	1.009e-08	1.382e-08
	5	9.298e-3	1.341e-2	6.910e-07	1.149e-06	**0.0**	0.0
	100	1.236e-3	2.325e-3	1.000e-06	1.755e-06	2.116e-10	3.592e-10
Schwefel 2.21 (F4)	1	1.0	1.0	1.888e-2	2.019e-2	3.683e-05	2.542e-05
	2	0.104	8.367e-2	6.091e-4	4.481e-4	7.128e-05	5.229e-05
	3	9.984e-4	5.571e-4	6.985e-4	5.590e-4	3.322e-05	2.352e-05
	4	2.635e-2	1.963e-2	1.741e-05	1.701e-05	1.715e-4	1.577e-4
	5	3.489e-2	3.236e-2	2.722e-4	2.782e-4	**0.0**	0.0
	100	4.203e-3	3.266e-3	2.132e-4	1.367e-4	5.105e-06	3.642e-06
Rosenbrock (F5)	1	1.0	1.0	0.914	1.909e-3	0.912	0.0
	2	0.915	6.528e-3	0.869	0.413	0.818	0.631
	3	0.860	0.435	0.771	0.731	0.698	0.845
	4	0.799	0.680	0.816	0.599	0.584	0.917
	5	0.826	0.580	0.771	0.593	0.522	0.946
	100	0.191	0.771	3.734e-2	0.183	**0.0**	0.054
Rastrigin (F9)	1	1.612e-2	1.125e-2	3.009e-4	2.139e-4	2.956e-07	2.434e-07
	2	1.088e-4	9.063e-05	1.487e-06	9.874e-07	4.195e-09	1.293e-09
	3	1.006e-3	8.085e-4	8.077e-05	7.690e-05	1.212e-07	1.207e-07
	4	0.427	0.299	4.328e-4	3.021e-4	2.943e-08	1.500e-08
	5	1.0	1.0	3.048e-05	2.980e-05	**0.0**	0.0
	100	1.096e-3	1.095e-3	7.299e-06	7.055e-06	2.451e-09	2.587e-09
Ackley (F10)	1	1.0	1.0	1.284e-3	1.267e-3	1.532e-06	3.053e-07
	2	1.481e-2	9.921e-3	2.423e-4	1.960e-4	**0.0**	0.0
	3	2.044e-2	2.003e-2	7.849e-4	4.429e-4	4.638e-05	3.411e-05
	4	2.199e-2	1.896e-2	5.816e-4	2.737e-4	7.229e-07	3.645e-07
	5	4.153e-3	2.734e-3	3.061e-3	2.966e-3	2.007e-05	1.979e-05
	100	3.561e-2	1.654e-2	1.294e-4	5.493e-05	5.029e-05	4.492e-05
Griewank (F11)	1	1.0	1.0	3.791e-4	4.572e-4	5.408e-06	7.817e-06
	2	0.222	0.330	6.969e-06	7.898e-06	1.203e-06	1.843e-06
	3	8.401e-4	1.252e-3	1.199e-4	1.748e-4	1.931e-06	2.957e-06
	4	1.096e-3	1.275e-3	1.133e-05	1.186e-05	3.813e-06	5.851e-06
	5	7.696e-2	0.118	4.862e-08	4.855e-08	7.534e-09	1.034e-08
	100	6.295e-06	6.583e-06	2.299e-07	2.158e-07	**0.0**	0.0
CF3 (F16)	1	8.837e-3	1.0	0.444	0.625	0.232	0.316
	2	1.0	0.674	0.381	0.661	0.158	0.182
	3	0.613	0.612	0.231	0.297	8.637e-2	9.002e-2
	4	0.426	0.479	0.134	0.137	7.006e-2	0.111
	5	0.281	0.326	0.109	0.139	4.303e-2	5.656e-2
	100	0.287	0.020	1.252e-3	0.0	**0.0**	3.226e-3
CF4 (F17)	1	1.0	0.834	0.961	0.858	0.500	0.588
	2	0.864	1.0	0.532	0.446	0.266	0.385
	3	0.689	0.638	0.313	0.328	0.220	0.249
	4	0.717	0.718	0.257	0.225	0.181	0.229
	5	0.562	0.551	0.286	0.323	0.167	0.224
	100	1.611e-2	1.905e-2	2.815e-3	4.160e-3	**0.0**	0.0
CF5 (F18)	1	0.976	1.0	1.0	0.825	0.478	0.263
	2	0.831	0.721	0.310	0.118	0.307	0.117
	3	0.406	0.109	0.268	0.105	0.221	9.216e-2
	4	0.592	0.352	0.192	7.599e-2	0.135	7.103e-2
	5	0.328	0.147	0.153	4.837e-2	0.105	4.515e-2
	100	7.383e-2	4.985e-2	**0.0**	0.0	3.529e-2	3.052e-2

with varying number of broadcast operations. It is apparent from the table that the cluster with 16 nodes provides the most fit solution for most benchmarks with the exception of benchmark F18. As a matter of fact, the fitness for the 16 nodes cluster outperforms all other cases regardless of the number of broadcasts. This suggests that the enhanced fitness can be attributed to the fact that the larger cluster benefits from more smaller subpopulations (partitions) working independent of each other which improves solution diversity and thus effectively avoiding local optima.

Another interesting observation was the fact that for half of the benchmarks in the 16 nodes cluster, the best fitness solution was found with either a number of broadcast equals to 5 or 100. This means that it is possible to limit the number of broadcasts and still get excellent solution quality without exhaustive communication cost (compared to

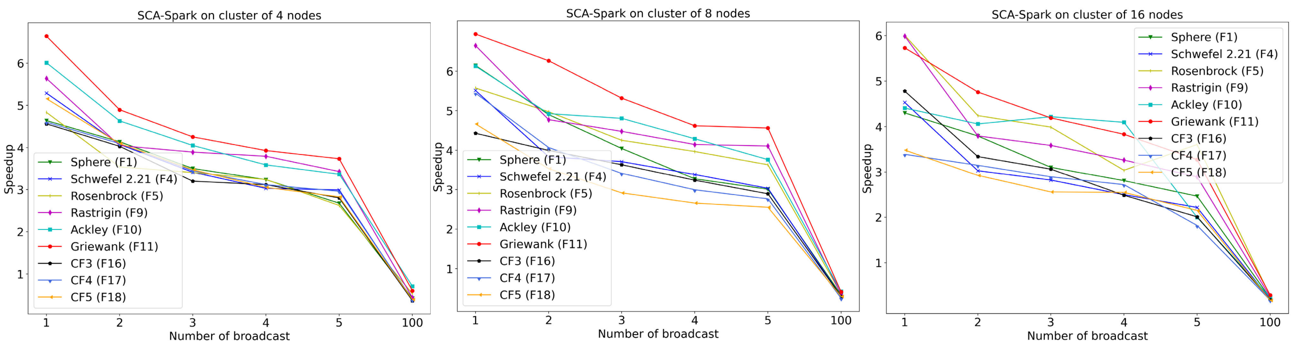

FIGURE 6. **Speedup vs. cluster size.**

100 broadcast case). The advantage of using a limited number of broadcasts can be clearly observed in Figure 6. It is apparent that performing a broadcast after each iteration results in the worst possible run-time performance for all cluster sizes. This means that in order to provide good performance in term of both run-time and fitness, a combination of large size cluster with limited number of broadcasts is the best possible route.

V. ENGINEERING DESIGN PROBLEMS

In the previous section, the performance of Spark-SCA was evaluated by solving several mathematical optimization functions. In this section, Spark-SCA was applied to three constrained engineering design problems namely the welded beam [39], tension/compression spring [40], [41], and pressure vessel [42], [43] to evaluate the performance of the proposed algorithm on real optimization problems. The engineering design problems have different constraints and thus a constraint handling method needs to be employed. Constraints divide the candidate solutions into: feasible and infeasible. The mathematical formulation of a general constrained optimization problem is written as the following:

Minimize $f(\vec{x})$ where $\vec{x} = (x_1, \ldots, x_n)^t \in F \subseteq S \subseteq R^n$

Subject to the following constraints:

$$g(\vec{x}) \leqslant 0 \quad i = 1, \ldots, m$$
$$h(\vec{x}) = 0 \quad j = 1, \ldots, p$$

where $f(\vec{x})$ is the objective function, $\vec{x}$ represents the vector of solutions, n is solution dimension, F is feasible solution region, and S is the complete search space. The constraints are divided into two types, equality and inequality. The number of inequality and equality constraints of the design problem are m and p, respectively [44]. In the literature [45], there are various constraint handling methods such as: penalty functions, repair algorithms, separation of objectives and constraints, hybrid methods, and special operators. Among those methods, the penalty functions is used in this study. According to [45], the penalty functions have different types including adaptive, annealing, co-evolutionary, death, dynamic, and static. For the sake of simplicity, Spark-SCA used the death penalty function to handle constraints. The vector of solutions in the death penalty function is represented as:

$$\vec{x} \in S - F$$

The main advantages of the death penalty function are: low computational cost and its simplicity. On the other hand, one of its limitations is that during the optimization process infeasible solutions are discarded automatically. Thus, the death penalty function is not recommended for solving problems that have dominated infeasible regions.

Results in this section are the average performance of running the algorithms for 30 runs. In a similar fashion to what was done in the earlier section, the number of function evaluations for each algorithm was kept the same. Spark-SCA implementation was tested on EMR Yarn cluster with four Amazon EC2 instances with one master node and three worker nodes. Each EC2 instance was chosen from the general purpose family of type m4.xlarge with 4 vCPU and 16 (GiB) Memory. On the other side, serial SCA implementation was tested on one node of type m4.xlarge.

The welded beam design problem aims to minimize the manufacturing cost. Figure 7 illustrates welded beam problem with four optimization variables namely weld thickness (h), joint length (l), bar height (t), and beam thickness (b). This design problem has four constraints that need to be taken into consideration: shear stress (τ), bending stress (θ), buckling load (P), and end of beam deflection (δ). The mathematical formulation of the welded beam design problem can be described as follows:

Let $\vec{x} = [x_1\ x_2\ x_3\ x_4] = [h\ l\ t\ b]$

Minimize $f(\vec{x}) = 1.10471x_1^2\, x_2 + 0.04811x_3x_4(14.0 + x_2)$

Subject to the following constraints:

$$g_1(\vec{x}) = \tau(\vec{x}) - 13,600 \leqslant 0$$
$$g_2(\vec{x}) = \sigma(\vec{x}) - 30,000 \leqslant 0$$
$$g_3(\vec{x}) = x_1 - x_4 \leqslant 0$$
$$g_4(\vec{x}) = 0.10471x_1^2 + 0.04811x_4x_3(14 + x_2) - 5.0 \leqslant 0$$
$$g_5(\vec{x}) = 0.125 - x_1 \leqslant 0$$
$$g_6(\vec{x}) = \delta(\vec{x}) - 0.25 \leqslant 0$$
$$g_7(\vec{x}) = 6,000 - P(\vec{x}) \leqslant 0$$

TABLE 5. Welded beam results.

Algorithm	# of evaluations		Optimization variables				Cost		Speedup
	Max. iter.	Agents	h	l	t	b	ave	std	
SCA	250	500	0.1902	3.8649	9.7275	0.2094	1.9009	3.918e-2	
Spark-SCA	200	625	0.1956	3.9455	9.1558	0.2119	1.8389	2.382e-2	2.0555
SCA	500	2,000	0.2015	3.5874	9.4239	0.2106	1.8356	3.185e-2	
Spark-SCA	125	8,000	0.2017	3.6851	9.1029	0.2097	1.7891	1.799e-2	4.1504
SCA	600	4,000	0.2017	3.6540	9.2174	0.2115	1.8163	2.333e-2	
Spark-SCA	100	24,000	0.2014	3.6560	9.0687	0.2092	1.7753	1.107e-2	4.5502
SCA	300	10,000	0.1975	3.7664	9.2274	0.2106	1.8201	2.092e-2	
Spark-SCA	50	60,000	0.2011	3.6662	9.0867	0.2084	1.7721	1.513e-2	4.5166
SCA	120	75,000	0.2023	3.6463	9.1081	0.2108	1.7938	2.332e-2	
Spark-SCA	100	90,000	0.2019	3.5989	9.0819	0.2079	1.7607	9.215e-3	4.9319

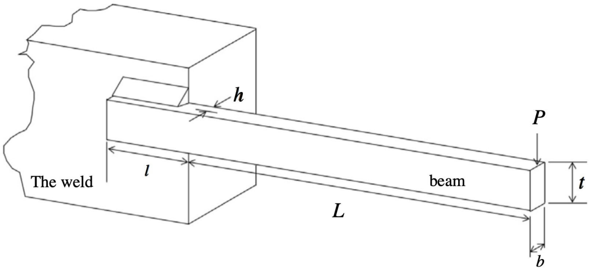

FIGURE 7. Welded beam schematic.

With:

$$\tau(\vec{x}) = \sqrt{(\tau')^2 + (2\tau'\tau'')\frac{x_2}{2R} + (\tau'')^2}$$

$$\tau' = \frac{6,000}{\sqrt{2}x_1x_2}$$

$$M = 6,000(14 + \frac{x_2}{2})$$

$$R = \sqrt{\frac{x_2^2}{4} + (\frac{x_1+x_3}{2})^2}$$

$$J = 2\left\{x_1x_2\sqrt{2}\left[\frac{x_2^2}{12} + (\frac{x_1+x_3}{2})^2\right]\right\}$$

$$\sigma(\vec{x}) = \frac{504,000}{x_4x_3^2}$$

$$\delta(\vec{x}) = \frac{65,856,000}{(30\times10^6)x_4x_3^3}$$

$$P(\vec{x}) = \frac{4.013(30\times10^6)\sqrt{\frac{x_3^2x_4^6}{36}}}{196}\left(1 - \frac{x_3\sqrt{\frac{30\times10^6}{4(12\times10^6)}}}{28}\right)$$

With the following bounds:

$$0.10 \leqslant h, \quad b \leqslant 2.0$$
$$0.10 \leqslant l, \quad t \leqslant 10.0$$

The tension/compression spring design problem consists of three design parameters: wire diameter (d), mean coil diameter (D), and the number of active coils (N) as shown in Figure 8. The objective of this problem is to minimize the weight of the spring by optimizing the aforementioned design parameters. This problem is formulated as the following:

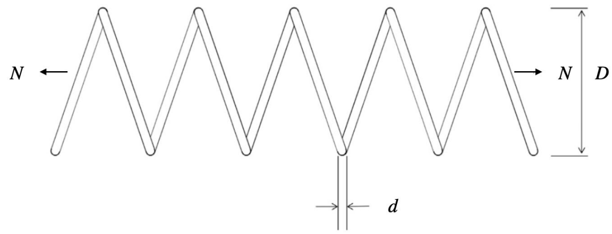

FIGURE 8. Tension/compression spring schematic.

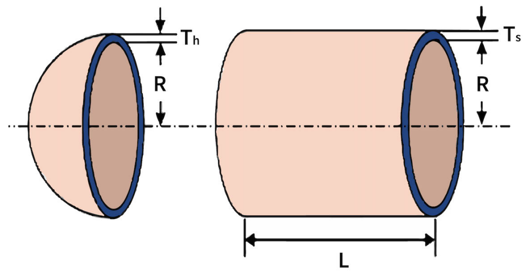

FIGURE 9. Pressure vessel schematics.

Let $\vec{x} = [x_1\ x_2\ x_3] = [d\ D\ N]$
Minimize $f(\vec{x}) = x_1^2\, x_2\, x_3 + 2x_1^2\, x_2$
Subject to the following constraints:

$$g_1(\vec{x}) = 1 - \frac{x_2^3x_3}{71785x_1^4} \leqslant 0$$

$$g_2(\vec{x}) = \frac{4x_2^2 - x_1x_2}{12566(x_2x_1^3 - x_1^4)} + \frac{1}{5108x_1^2} - 1 \leqslant 0$$

$$g_3(\vec{x}) = 1 - \frac{140.45x_1}{x_2^2x_3} \leqslant 0$$

$$g_4(\vec{x}) = \frac{x_1+x_2}{1.5} - 1 \leqslant 0$$

With the following bounds:

$$0.05 \leqslant d \leqslant 2.0$$
$$0.25 \leqslant D \leqslant 1.3$$
$$2.0 \leqslant N \leqslant 15.0$$

As for the pressure vessel problem, the optimization process aims to obtain the optimal parameters that lead to minimum total cost. The total cost includes the cost of material, forming, and welding. As illustrated in Figure 9, the pressure

TABLE 6. Tension/compression spring results.

Algorithm	# of evaluations		Optimization variables			Cost		Speedup
	Max. iter.	Agents	d	D	N	ave	std	
SCA	250	500	0.0501	0.3182	14.1969	0.01290	1.041e-4	
Spark-SCA	200	625	0.0506	0.3299	13.4213	0.01289	1.112e-4	2.4176
SCA	500	2,000	0.0501	0.3196	13.9708	0.01279	3.914e-5	
Spark-SCA	125	8,000	0.0509	0.3382	12.6687	0.01276	1.655e-5	4.2846
SCA	600	4,000	0.0501	0.3205	13.8599	0.01276	2.041e-5	
Spark-SCA	100	24,000	0.0508	0.3351	12.8437	0.01273	3.009e-5	4.8203
SCA	300	10,000	0.0503	0.3243	13.5964	0.01275	2.260e-5	
Spark-SCA	50	60,000	0.0509	0.3382	12.6651	0.01274	2.109e-5	5.1379
SCA	120	75,000	0.0508	0.3362	12.795	0.01274	1.679e-5	
Spark-SCA	100	90,000	0.0514	0.3501	11.8240	0.01271	1.655e-5	5.4513

TABLE 7. SCA vs Spark-SCA optimization results for the pressure vessel design problem.

Algorithm	# of evaluations		Optimization variables				Cost		Speedup
	Max. iter.	Agents	Ts	Th	R	L	ave	std	
SCA	250	500	0.9399	0.5610	45.9361	158.6315	7216.6709	556.116	
Spark-SCA	200	625	0.8499	0.4577	42.7622	178.1989	6500.4553	281.814	2.0705
SCA	500	2,000	0.8443	0.4613	42.3651	181.2957	6452.1224	186.963	
Spark-SCA	125	8,000	0.8142	0.4174	41.5762	186.5731	6139.1930	98.817	3.5110
SCA	600	4,000	0.8046	0.4321	40.9555	197.2977	6261.6008	115.802	
Spark-SCA	100	24,000	0.7978	0.4092	40.9380	194.7225	6082.6517	53.838	3.8277
SCA	300	10,000	0.7981	0.4218	40.8335	197.1327	6156.1648	143.612	
Spark-SCA	50	60,000	0.8013	0.4147	41.2283	190.1168	6065.9492	41.200	3.8753
SCA	120	75,000	0.7945	0.4061	40.5203	199.9025	6098.0664	79.388	
Spark-SCA	100	90,000	0.7985	0.4053	41.1032	191.3584	6026.2446	47.330	3.6140

TABLE 8. Benchmark functions description.

Name	Category	Function	Range	f_{min}
Sphere	Unimodal	$f_1(x) = \sum_{i=1}^{D} x_i^2$	[-100, 100]	0
Schwefel 2.21	Unimodal	$f_4(x) = max_i\{\|x_i\|, 1 \leq i \leq D\}$	[-100, 100]	0
Rosenbrock	Unimodal	$f_5(x) = \sum_{i=1}^{D-1}[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$	[-30, 30]	0
Rastrigin	Multimodal	$f_9(x) = 10D + \sum_{i=1}^{D}[x_i^2 - 10cos(2\pi x_i)]$	[-5.12, 5.12]	0
Ackley	Multimodal	$f_{10}(x) = -20e^{-0.02}\sqrt{D^{-1}\sum_{i=1}^{D} x_i^2} - e^{-D^{-1}}\sum_{i=1}^{D} cos(2\pi x_i) + 20 + e$	[-32, 32]	0
Griewank	Multimodal	$f_{11}(x) = 1 + \frac{1}{4000}\sum_{i=1}^{n} x^2 - \prod_{i=1}^{n} cos(\frac{x_i}{\sqrt{i}})$	[-600, 600]	0
CF3	Composite	$f_{16}(CF3)$ $f_1,\ f_2,\ f_3\ ...,\ f_{10} = Griewank's\ Function$ $[\sigma_1,\ \sigma_2,\ \sigma_3,\ ....,\ \sigma_{10}] = [1,\ 1,\ 1,\ ..,\ 1]$ $[\lambda_1,\ \lambda_2,\ \lambda_3,\ ....,\ \lambda_{10}] = [1,\ 1,\ 1, ..,\ 1]$	[-5, 5]	0
CF4	Composite	$f_{17}(CF4)$ $f_1,\ f_2 = Ackley's\ Function,\ f_3,\ f_4 = Rastrigin's\ Function,$ $f_5,\ f_6 = Weierstrass\ Function,\ f_7,\ f_8 = Griewank's\ Function,\ f_9,\ f_{10} = Sphere\ Function$ $[\sigma_1,\ \sigma_2,\ \sigma_3,\ ....,\ \sigma_{10}] = [1,\ 1,\ 1,\ ..,\ 1]$ $[\lambda_1,\ \lambda_2,\ \lambda_3,\ ....,\ \lambda_{10}] = [5/32,\ 5/32,\ 1,\ 1,\ 5/0.5,\ 5/0.5,\ 5/100,\ 5/100,\ 5/100,\ 5/100]$	[-5, 5]	0
CF5	Composite	$f_{18}(CF5)$ $f_1,\ f_2 = Rastrigin's\ Function,\ f_3,\ f_4 = Weierstrass\ Function,$ $f_5,\ f_6 = Griewank's\ Function,\ f_7,\ f_8 = Ackley's\ Function,\ f_9,\ f_{10} = Sphere\ Function$ $[\sigma_1,\ \sigma_2,\ \sigma_3,\ ....,\ \sigma_{10}] = [1,\ 1,\ 1,\ ..,\ 1]$ $[\lambda_1,\ \lambda_2,\ \lambda_3,\ ....,\ \lambda_{10}] = [1/5,\ 1/5,\ 5/0.5,\ 5/0.5,\ 5/100,\ 5/100,\ 5/32,\ 5/32,\ 5/100,\ 5/100]$	[-5, 5]	0

vessel is composed of a cylindrical vessel with a hemispherical shape head capped at both ends. The pressure vessel design has four optimization variables namely: thickness of the hemispherical head (T_h), cylindrical vessel thickness (T_s), inner radius (R), and cylindrical vessel length excluding both heads (L). According to [42], the pressure vessel optimization problem is mathematically modeled by the following equations:

Let $\vec{x} = [x_1\ x_2\ x_3\ x_4] = [T_s\ T_h\ R\ L]$

$$\text{Minimize } f(\vec{x}) = 0.6224x_1x_3x_4 + 1.7781x_2x_3^2 + 3.1661x_1^2x_4 + 19.84x_1^2x_3$$

Subject to the following constraints:

$$g_1(\vec{x}) = -x_1 + 0.0193x_3 \leqslant 0$$
$$g_2(\vec{x}) = -x_2 + 0.00954x_3 \leqslant 0$$
$$g_3(\vec{x}) = -\pi x_3^2x_4 - \frac{4}{3}\pi x_3^3 + 1296000 \leqslant 0$$
$$g_4(\vec{x}) = x_4 - 240 \leqslant 0$$

With the following bounds:

$$0.0625 \leqslant T_s, \quad T_h \leqslant 6.1875$$
$$10 \leqslant R, \quad L \leqslant 200$$

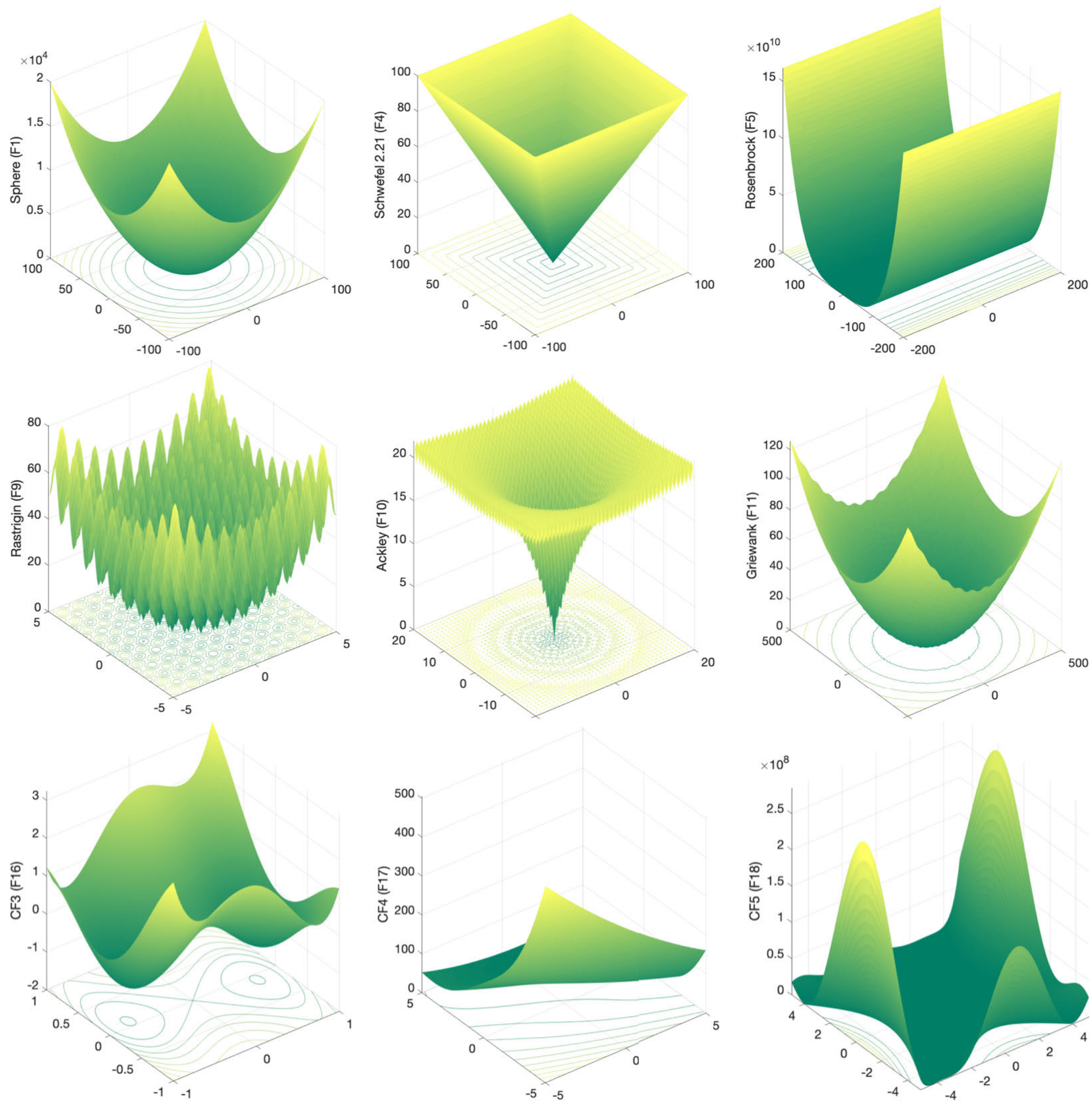

FIGURE 10. Graphical plot of benchmark functions.

Solutions of the welded beam design problem using SCA and Spark-SCA are presented in Table 5. The table specifies the population size and the maximum number of iterations used for each algorithm as well as the various values for the various design variables. Moreover, the table specifies overall cost and run-time for each algorithm as well as speedup obtained for Spark-SCA implementation compared to the serial one. It is apparent from the table that Spark-SCA provides a minimum speedup of 2 times as compared to the serial version and it increases as population size is increased. It can be seen that as the number of agents increases, speedup increases as well indicating the effectiveness of Spark-SCA in distributing computations between the different nodes in the cluster, especially with large number of agents. In term of the minimum cost obtained, Spark-SCA was able to reach a value of 1.7607 as compared to 1.7938 found by the serial version. Despite the fact that the cost difference is not that significant, Spark-SCA is more efficient in term of run-time where it requires only one fifth of the time needed for the serial algorithm.

The results of the compression/tension spring and the pressure vessel design problems are shown in Table 6 and Table 7, respectively. The tables are organized in a similar fashion to the case of the welded beam case. All findings found in the case of the welded beam case can be clearly observed in these cases as well. These experiments clearly show the advantage of using Spark-SCA to solve realistic design problems especially when run-time characteristics are of most importance.

VI. CONCLUSION

In this paper, we presented Spark-SCA, a parallel implementation of the sine cosine algorithm on Spark architecture. Experimental results on various benchmark functions demonstrated that even though Spark implementation may provide considerable advantages in terms of solution quality, run-time performance may suffer if communication cost was not taken into consideration. Moreover, it was shown that a good practice to consider when optimizing for both fitness and run time is to limit broadcast operation especially when the number of nodes is large. It was also observed that even though Spark-SCA was not able to provide good quality solutions for composite benchmarks when small size cluster is used, such shortcoming was alleviated when cluster size was increased resulting in more partitions and hence better exploration of the search space. Additional experiments were conducted on realistic optimization problems in engineering field. The performance of Spark-SCA for all such problems was shown to provide superior performance in term of both run-time as well as solution quality. This proves the appropriateness of using the distributed version of the SCA algorithm when solving complex real life problems. The source codes of Spark-SCA algorithm are publicly available at https://github.com/Maryom/Spark-SCA.

Possible future extension of this work is to study the implementation of different variants of SCA algorithm by incorporating different update strategies such as cauchy mutation operator, chaotic local search mechanism, opposition-based learning strategy and their performance when implemented on Spark platform. Parallel implementation of multi-objective SCA is another fertile area for future exploration.

APPENDIX

All benchmark functions used in this paper are mathematically described in Table 8. The graphical plot of these functions are also shown in Figure 10.

REFERENCES

[1] N. Khanduja and B. Bhushan, "Recent advances and application of metaheuristic algorithms: A survey (2014–2020)," in *Metaheuristic and Evolutionary Computation: Algorithms and Applications*. Cham, Switzerland: Springer, 2021, pp. 207–228, doi: 10.1007/978-981-15-7571-6_10.

[2] R.-E. Precup, R.-C. David, E. M. Petriu, A.-I. Szedlak-Stinean, and C.-A. Bojan-Dragos, "Grey wolf optimizer-based approach to the tuning of pi-fuzzy controllers with a reduced process parametric sensitivity," *IFAC-PapersOnLine*, vol. 49, no. 5, pp. 55–60, 2016.

[3] A. Soares, R. Râbelo, and A. Delbem, "Optimization based on phylogram analysis," *Expert Syst. Appl.*, vol. 78, pp. 32–50, Jul. 2017.

[4] K. Hussain, M. N. M. Salleh, S. Cheng, and Y. Shi, "Metaheuristic research: A comprehensive survey," *Artif. Intell. Rev.*, vol. 52, no. 4, pp. 2191–2233, Dec. 2019.

[5] B. H. Abed-Alguni, "Island-based cuckoo search with highly disruptive polynomial mutation," *Int. J. Artif. Intell.*, vol. 17, no. 1, pp. 57–82, 2019.

[6] M. AlFailakawi, I. Ahmad, L. AlTerkawi, and S. Hamdan, "Depth optimization for topological quantum circuits," *Quantum Inf. Process.*, vol. 14, no. 2, pp. 447–463, Feb. 2015.

[7] H. Zapata, N. Perozo, W. Angulo, and J. Contreras, "A hybrid swarm algorithm for collective construction of 3D structures," *Int. J. Artif. Intell.*, vol. 18, no. 1, pp. 1–18, 2020.

[8] M. G. Alfailakawi, I. Ahmad, and S. Hamdan, "Harmony-search algorithm for 2D nearest neighbor quantum circuits realization," *Expert Syst. Appl.*, vol. 61, pp. 16–27, Nov. 2016.

[9] R.-E. Precup, E.-L. Hedrea, R.-C. Roman, E. M. Petriu, A.-I. Szedlak-Stinean, and C.-A. Bojan-Dragos, "Experiment-based approach to teach optimization techniques," *IEEE Trans. Educ.*, vol. 64, no. 2, pp. 88–94, May 2020.

[10] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE Trans. Evol. Comput.*, vol. 1, no. 1, pp. 67–82, Apr. 1997.

[11] S. Mirjalili, "SCA: A sine cosine algorithm for solving optimization problems," *Knowl.-Based Syst.*, vol. 96, pp. 120–133, Mar. 2016.

[12] S. M. Mirjalili, S. Z. Mirjalili, S. Saremi, and S. Mirjalili, "Sine cosine algorithm: Theory, literature review, and application in designing bend photonic crystal waveguides," in *Nature-Inspired Optimizers: Theories, Literature Reviews and Applications*. Cham, Switzerland: Springer, 2020, pp. 201–217, doi: 10.1007/978-3-030-12127-3_12.

[13] S. Gupta, K. Deep, and A. P. Engelbrecht, "A memory guided sine cosine algorithm for global optimization," *Eng. Appl. Artif. Intell.*, vol. 93, Aug. 2020, Art. no. 103718.

[14] H. Chen, M. Wang, and X. Zhao, "A multi-strategy enhanced sine cosine algorithm for global optimization and constrained practical engineering problems," *Appl. Math. Comput.*, vol. 369, Mar. 2020, Art. no. 124872.

[15] B. Wang, T. Xiang, N. Li, W. He, W. Li, and X. Hei, "A symmetric sine cosine algorithm with adaptive probability selection," *IEEE Access*, vol. 8, pp. 25272–25285, 2020.

[16] W. Zhou, P. Wang, A. A. Heidari, M. Wang, X. Zhao, and H. Chen, "Multi-core sine cosine optimization: Methods and inclusive analysis," *Expert Syst. Appl.*, vol. 164, Feb. 2021, Art. no. 113974.

[17] H. Huang, X. Feng, A. A. Heidari, Y. Xu, M. Wang, G. Liang, H. Chen, and X. Cai, "Rationalized sine cosine optimization with efficient searching patterns," *IEEE Access*, vol. 8, pp. 61471–61490, 2020.

[18] Q. Yang, S.-C. Chu, J.-S. Pan, and C.-M. Chen, "Sine cosine algorithm with multigroup and multistrategy for solving CVRP," *Math. Problems Eng.*, vol. 2020, pp. 1–10, Mar. 2020.

[19] M. G. Alfailakawi, M. El-Shafei, I. Ahmad, and A. Salman, "FPGA-based implementation of cuckoo search," *IET Comput. Digit. Techn.*, vol. 13, no. 1, pp. 28–37, 2019.

[20] M. Essaid, L. Idoumghar, J. Lepagnot, and M. Brévilliers, "GPU parallelization strategies for metaheuristics: A survey," *Int. J. Parallel, Emergent Distrib. Syst.*, vol. 34, no. 5, pp. 497–522, Sep. 2019.

[21] Y. Khalil, M. Alshayeji, and I. Ahmad, "Distributed whale optimization algorithm based on MapReduce," *Concurrency Comput., Pract. Exper.*, vol. 31, no. 1, p. e4872, Jan. 2019.

[22] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[23] H.-C. Lu, F. J. Hwang, and Y.-H. Huang, "Parallel and distributed architecture of genetic algorithm on apache Hadoop and spark," *Appl. Soft Comput.*, vol. 95, Oct. 2020, Art. no. 106497.

[24] J. Al-Sawwa and S. A. Ludwig, "Parallel particle swarm optimization classification algorithm variant implemented with apache spark," *Concurrency Comput., Pract. Exper.*, vol. 32, no. 2, p. e5451, Jan. 2020.

[25] T. Wen, H. Liu, L. Lin, B. Wang, J. Hou, C. Huang, T. Pan, and Y. Du, "Multiswarm artificial bee colony algorithm based on spark cloud computing platform for medical image registration," *Comput. Methods Programs Biomed.*, vol. 192, Aug. 2020, Art. no. 105432.

[26] Y. Lu, B. Cao, C. Rego, and F. Glover, "A tabu search based clustering algorithm and its parallel implementation on spark," *Appl. Soft Comput.*, vol. 63, pp. 97–109, Feb. 2018.

[27] M. AlJame, I. Ahmad, and M. Alfailakawi, "Apache spark implementation of whale optimization algorithm," *Cluster Comput.*, vol. 23, no. 3, pp. 2021–2034, Sep. 2020.

[28] X. C. Pardo, P. Argüeso-Alejandro, P. González, J. R. Banga, and R. Doallo, "Spark implementation of the enhanced scatter search metaheuristic: Methodology and assessment," *Swarm Evol. Comput.*, vol. 59, Dec. 2020, Art. no. 100748.

[29] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," *HotCloud*, vol. 10, no. 10, p. 95, Jun. 2010.

[30] H. Herodotou, Y. Chen, and J. Lu, "A survey on automatic parameter tuning for big data processing systems," *ACM Comput. Surv.*, vol. 53, no. 2, pp. 1–37, Jul. 2020.

[31] T. Weise, "Global optimization algorithms-theory and application," Self-Published Thomas Weise, Tech. Rep., 2009. [Online]. Available: http://www.it-weise.de/

[32] A. Gounaris and J. Torres, "A methodology for spark parameter tuning," *Big Data Res.*, vol. 11, pp. 22–32, Mar. 2018.
[33] A. J. Awan, M. Brorsson, V. Vlassov, and E. Ayguade, "How data volume affects spark based data analytics on a scale-up server," in *Big Data Benchmarks, Performance Optimization, and Emerging Hardware*, J. Zhan, R. Han, and R. V. Zicari, Eds. Cham, Switzerland: Springer, 2016, pp. 81–92.
[34] T. B. G. Perez, W. Chen, R. Ji, L. Liu, and X. Zhou, "PETS: Bottleneck-aware spark tuning with parameter ensembles," in *Proc. 27th Int. Conf. Comput. Commun. Netw. (ICCCN)*, Jul. 2018, pp. 1–9.
[35] N. Nguyen, M. M. H. Khan, and K. Wang, "Towards automatic tuning of apache spark configuration," in *Proc. IEEE 11th Int. Conf. Cloud Comput. (CLOUD)*, Jul. 2018, pp. 417–425.
[36] Z. Yu, Z. Bei, and X. Qian, "Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing," in *Proc. 23rd Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, Mar. 2018, pp. 564–577.
[37] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2012, pp. 15–28.
[38] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, *Learning Spark: Lightning-Fast Big Data Analysis*. Sebastopol, CA, USA: O'Reilly Media, 2015.
[39] K. M. Ragsdell and D. T. Phillips, "Optimal design of a class of welded structures using geometric programming," *J. Eng. Ind.*, vol. 98, no. 3, pp. 1021–1025, Aug. 1976, doi: 10.1115/1.3438995.
[40] J. S. Arora, *Introduction to Optimum Design*, 4th ed. Boston, MA, USA: Elsevier, 2017.
[41] A. D. Belegundu and J. S. Arora, "A study of mathematical programming methods for structural optimization. Part I: Theory," *Int. J. Numer. Methods Eng.*, vol. 21, no. 9, pp. 1583–1599, Sep. 1985. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.1620210904
[42] B. K. Kannan and S. N. Kramer, "An augmented Lagrange multiplier based method for mixed integer discrete continuous optimization and its applications to mechanical design," *J. Mech. Des.*, vol. 116, no. 2, pp. 405–411, Jun. 1994. [Online]. Available: https://ci.nii.ac.jp/naid/80007711575/en/
[43] D. R. Moss, *Pressure Vessel Design Manual*. Amsterdam, The Netherlands: Elsevier, 2004.
[44] T. Bäck, F. Hoffmeister, and H.-P. Schwefel, "A survey of evolution strategies," in *Proc. 4th Int. Conf. Genet. Algorithms*. San Mateo, CA, USA: Morgan Kaufmann, 1991, pp. 2–9.
[45] C. A. C. Coello, "Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: A survey of the state of the art," *Comput. Methods Appl. Mech. Eng.*, vol. 191, nos. 11–12, pp. 1245–1287, Jan. 2002.

MOHAMMAD GH. ALFAILAKAWI received the B.S. degree in electrical engineering and computer engineering from the University of Missouri, Columbia, MO, USA, in 1996, and the M.S. and Ph.D. degrees in electrical engineering from the University of Wisconsin–Madison, in 1999 and 2002, respectively. From 2012 to 2015, he has served as the Vice Dean of Academic Affairs for the College of Computing Sciences and Engineering. He was the Director of the Engineering Training and Alumni Center, College of Engineering and Petroleum, Kuwait University, from 2009 to 2012. He is currently an Associate Professor with the Computer Engineering Department, Kuwait University, where he teaches courses in logic design, embedded systems, computer architecture and organization, and testing and fault-tolerant computing. He is also the Chairman of the Computer Engineering Department, College of Engineering and Petroleum, Kuwait University. His current research interests include nonvolatile memory technology and test, defect-based testing, reversible circuit optimization, and computational optimization.

MARYAM ALJAME received the B.Sc. and M.Sc. degrees from the Computer Engineering Department, Kuwait University, in 2012 and 2018, respectively. She is currently an iOS Developer with the Ministry of Education. Her research interests include machine learning, bioinformatics, and parallel and distributed computing.

IMTIAZ AHMAD received the B.Sc. degree in electrical engineering from the University of Engineering and Technology, Lahore, Pakistan, in 1984, the M.Sc. degree in electrical engineering from the King Fahd University of Petroleum & Minerals, Dhahran, Saudi Arabia, in 1988, and the Ph.D. degree in computer engineering from Syracuse University, Syracuse, NY, USA, in 1992.

Since 1992, he has been with the Department of Computer Engineering, Kuwait University, Kuwait, where he is currently a Professor. His research interests include the design automation of digital systems, parallel and distributed computing, machine learning, and software-defined networks.